

Christophe Blaess

Développement système sous Linux

Ordonnancement multitâche,
gestion mémoire, communications,
programmation réseau

4^e édition

EYROLLES

Développement système sous Linux

4^e édition

Tirer le meilleur parti de l'environnement Linux

La possibilité de consulter les sources du système, de la bibliothèque glibc et de la plupart des applications qui tournent sur cet environnement représente une richesse inestimable aussi bien pour les passionnés qui souhaitent intervenir sur le noyau, que pour les développeurs curieux de comprendre comment fonctionnent les programmes qu'ils utilisent quotidiennement.

Nombreuses sont les entreprises qui ont compris aujourd'hui tout le parti qu'elles pouvaient tirer de cette ouverture des sources, gage de fiabilité et de pérennité, sans parler de l'extraordinaire niveau de compétences disponible au sein d'une communauté de programmeurs aguerris au contact du code des meilleurs développeurs open source.

Un ouvrage conçu pour les programmeurs Linux et Unix les plus exigeants

Sans équivalent en langue française, l'ouvrage de Christophe Blaess constitue une référence complète du développement système sous Linux, y compris dans les aspects les plus avancés de la gestion des processus, des threads ou de la mémoire. Les programmeurs travaillant sous d'autres environnements Unix apprécieront tout particulièrement l'attachement de l'auteur au respect des standards (C Ansi, glibc, Posix...), garant d'une bonne portabilité des applications. Cette quatrième édition entièrement actualisée apporte une vue différente sur la programmation système en ajoutant de nombreux scripts en Python qui complètent les exemples en langage C.

À qui s'adresse cet ouvrage ?

- Aux développeurs concernés par les aspects système de la programmation sous Linux et Unix
- Aux administrateurs système en charge de la gestion d'un parc Linux et/ou Unix
- Aux étudiants en informatique (1^{er} et 2^e cycles universitaires, écoles d'ingénieurs, etc.)

Sur le site www.editions-eyrolles.com

- Téléchargez le code source des exemples
- Consultez les mises à jour et compléments

Diplômé de l'Esigelec et titulaire d'un DEA de l'université de Caen, **Christophe Blaess** est un expert de Linux dans l'industrie. Il conduit de nombreux projets et réalise des prestations d'ingénierie et de conseil dans différents domaines liés à Linux : télévision numérique, informatique médicale, ingénierie aérienne embarquée, traitement radar... Soucieux de partager ses connaissances et son savoir-faire, il dispense des cours sur Linux embarqué en école d'ingénieurs et anime depuis plusieurs années des formations professionnelles (Linux temps réel et embarqué, écriture de drivers et programmation noyau Linux...) dans de nombreux centres de formation, en particulier avec la société Logilin qu'il a créée en 2004.

Sommaire

Concepts et outils • Les processus • Exécution d'un programme • Environnement et ligne de commande • Fin d'un programme • Déroulement et aspects avancés des Pthreads • Fonctions horaires • Sommeil des processus et contrôle des ressources • Ordonnancements sous Linux • Gestion classique des signaux • Gestion portable des signaux • Signaux temps réel • Gestion de la mémoire du processus • Gestion avancée de la mémoire • Utilisation des blocs mémoire et des chaînes • Tris, recherches et structuration des données • Routines avancées de traitement des blocs mémoire • Types de données et conversions • Entrées-sorties simplifiées • Flux de données • Descripteurs de fichiers • Communications classiques entre processus • Communications avec les IPC • Entrées-sorties avancées • Programmation réseau • Utilisation des sockets • Accès au contenu des répertoires • Attributs des fichiers • Accès aux informations du système • Internationalisation • Gestion du terminal.

www.editions-eyrolles.com

Développement système sous

Linux

Ordonnancement multitâche,
gestion mémoire, communications,
programmation réseau

Dans la collection *Les guides de formation Tsoft*

J.-F. BOUCHAUDY. – **Linux Administration. Tome 1 : les bases de l'administration système.**
N°14082, 3^e édition, 2014, 690 pages.

J.-F. BOUCHAUDY. – **Linux Administration. Tome 2 : administration système avancée.**
N°12882, 2^e édition, 2011, 504 pages.

J.-F. BOUCHAUDY. – **Linux Administration. Tome 3 : sécuriser un serveur Linux.**
N°13462, 2^e édition, 2012, 520 pages.

J.-F. BOUCHAUDY. – **Linux Administration. Tome 4 : installer et configurer des services Web, mail et FTP.**
N°13790, 2^e édition, 2013, 420 pages.

Autres ouvrages

C. BLAESS. – **Solutions temps réel sous Linux.**
N°14208, 2^e édition, 2015, 300 pages.

C. BLAESS. – **Shells Linux et Unix par la pratique.**
N°13579, 2^e édition, 2012, 296 pages.

I. HURBAIN, E. DREYFUS. – **Mémento Unix/Linux**
N°13306, 2^e édition, 2011, 14 pages.

Christophe Blaess

Développement système sous

Linux

Ordonnancement multitâche,
gestion mémoire, communications,
programmation réseau

4^e édition

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans l'autorisation de l'Éditeur ou du Centre Français d'exploitation du droit de copie, 20, rue des Grands Augustins, 75006 Paris.

© Groupe Eyrolles, 2016, ISBN : 978-2-212-14207-5

Avant-propos

La dynamique des logiciels libres en général et du système Gnu/Linux en particulier a commencé à prendre de l'importance il y a une vingtaine d'années, et son succès ne s'est pas démenti depuis. Le scepticisme que l'on rencontrait parfois à l'encontre de Linux a fortement diminué, et on trouve de plus en plus d'applications professionnelles de ce système, notamment dans les domaines industriels et scientifiques, ainsi que pour les services réseau.

Ce succès dans le monde professionnel démontre qu'outre la gratuité et la liberté de ce système, il s'agit en réalité d'un phénomène technologique particulièrement intéressant. La conception même du noyau Linux ainsi que celle de tout l'environnement qui l'accompagne sont des éléments passionnants pour le programmeur. La possibilité de consulter les sources du système d'exploitation, de la bibliothèque C ou de la plupart des applications représente une richesse inestimable non seulement pour les passionnés qui désirent intervenir sur le noyau, mais également pour les développeurs curieux de comprendre les mécanismes intervenant dans les programmes qu'ils utilisent.

Dans cet ouvrage, j'aimerais communiquer le plaisir que j'éprouve depuis plusieurs années à travailler quotidiennement avec un système Linux. Je me suis trouvé professionnellement dans divers environnements industriels utilisant initialement des systèmes Unix classiques. Le basculement vers des PC fonctionnant sous Linux nous a permis de multiplier le nombre de postes de travail et d'enrichir nos systèmes en créant des stations dédiées à des tâches précises (filtrage et diffusion de données, postes de supervision...), tout en conservant une homogénéité dans les systèmes d'exploitation des machines utilisées.

Depuis une dizaine d'années, mon orientation professionnelle m'a conduit plus particulièrement vers les systèmes Linux embarqués et temps réel, ainsi que vers l'écriture de drivers personnalisés pour Linux, domaines passionnants et dont les applications sont chaque jour plus nombreuses.

Ce livre est consacré à Linux en tant que noyau, mais également à la bibliothèque Gnu *Glibc*, qui lui offre toute sa puissance applicative. On considérera que le lecteur est à l'aise avec le langage C et avec les commandes élémentaires d'utilisation du système Linux. Dans les programmes fournis en exemple, l'effort a porté sur la lisibilité du code source plutôt que sur l'élégance du codage. Les ouvrages d'initiation au langage C sont nombreux ; on conseillera l'indispensable [Kernighan 1994], ainsi que l'excellent cours [Cassagne 1998] disponible librement sur Internet.

Cette nouvelle édition s'enrichit d'exemples en Python, langage particulièrement actif et dynamique depuis une dizaine d'années. Mon but est de montrer que la plupart des tâches « bas niveau » qu'on confie habituellement au langage C peuvent très bien être réalisées par des scripts Python.

L'écriture de « beaux » scripts Python repose habituellement sur une programmation orientée objet pour laquelle ce langage a été conçu. Je ne l'ai pas utilisée ici, préférant montrer des petits scripts élémentaires se rapprochant au maximum des exemples en C.

Le premier chapitre présentera rapidement les concepts et les outils nécessaires au développement sous Linux. Les utilitaires ne seront pas détaillés en profondeur, on se reportera aux documentations les accompagnant (pages de manuels, fichiers info, etc.).

Nous aborderons ensuite la programmation proprement dite avec Linux et la *Glibc*. Nous pouvons distinguer trois thématiques successives.

- Les chapitres 2 à 13 sont plus particulièrement orientés vers l'exécution des programmes. Nous y verrons les processus, les *threads*, l'ordonnancement des tâches et les signaux.
- Nous nous consacrerons ensuite à la mémoire, tant au niveau des mécanismes d'allocation et de libération que de l'utilisation effective des blocs ou des chaînes de caractères. Cette partie recouvrira les chapitres 14 à 19 et couvrira des traitements avancés sur les blocs de mémoire, comme les expressions régulières ou le cryptage DES.
- Nous aurons enfin une série de chapitres consacrés aux fichiers et aux communications entre applications (mécanismes IPC, socket réseau, etc.) ainsi que les éléments liés au système de fichiers et aux terminaux.

On remarquera que j'accorde une importance assez grande à l'appartenance d'une fonction aux normes logicielles courantes. C'est une garantie de portabilité des applications. Le standard C Ansi (qu'on devrait d'ailleurs plutôt nommer *Iso C*) est important au niveau de la syntaxe d'écriture des applications. La norme Posix (*Posix.1*, et ses extensions *Posix.1b* – temps réel et *Posix.1c* – threads) a longtemps fait figure de référence dans le domaine Unix, accompagnée d'un autre standard : *Unix 98*.

Le standard qui s'impose de nos jours est une fusion de Posix et de la norme *Unix 98*, ayant évoluées ensembles pour donner naissance à SUSv4 (*Single Unix Specifications version 4*). Non seulement ce document décrit des fonctionnalités bien respectées sur les systèmes Unix en général et Linux en particulier, mais il est en outre disponible gratuitement sur Internet, sur le site de l'association Open Group (www.opengroup.org). De nombreux lecteurs des éditions précédentes m'ont écrit pour me communiquer leurs observations et me faire part de leurs remarques, je les en remercie de tout cœur.

Ris-Orangis, avril 2016

Christophe@Blaess.fr

<http://www.blaess.fr/christophe/>

Table des matières

CHAPITRE 1

Concepts et outils..... 1

Généralités sur le développement sous Linux 1

Outils de développement 4

Eclipse 5

NetBeans 6

Programmation en Python 7

Éditeurs de texte 8

Compilateur, éditeur de liens 10

Débogueur, profileur 13

Traitement du code source 20

Utilitaires divers 23

Construction d'application 25

Distribution du logiciel 26

Bibliothèques supplémentaires pour le développement 29

Interface utilisateur en mode texte 29

Développement sous X-Window 30

Les environnements KDE et Gnome 30

Conclusion 31

CHAPITRE 2

Les processus 33

Principe des processus 33

Identification par le PID 36

Identification de l'utilisateur correspondant au processus 40

Identification du groupe d'utilisateurs du processus 48

Identification du groupe de processus 52

Identification de session 56

Capacités d'un processus 59

Conclusion 63

CHAPITRE 3

Exécution d'un programme	65
Lancement d'un nouveau programme	65
Causes d'échec de lancement d'un programme	74
Fonctions simplifiées pour exécuter un sous-programme	77
Conclusion	87

CHAPITRE 4

Environnement et ligne de commande	89
Variables d'environnement	90
Variables d'environnement couramment utilisées	98
Arguments en ligne de commande	101
Options simples – SUSv4	103
Options longues – Gnu	105
Sous-options	109
Exemple complet d'accès à l'environnement	110
Conclusion	117

CHAPITRE 5

Fin d'un programme.....	119
Terminaison d'un programme	119
Terminaison normale d'un processus	119
Terminaison anormale d'un processus	124
Exécution automatique de routines de terminaison	127
Attendre la fin d'un processus fils	132
Signaler une erreur	143
Conclusion	152

CHAPITRE 6

Déroulement des Pthreads	153
Présentation	154
Implémentation	154
Création de threads	156
Passage d'argument à la création d'un thread	160
Partage d'espace mémoire	163
Fin d'un thread	165
Élimination d'un thread	168
Récupération de la valeur de retour	169
Détachement des threads	170
Attributs des threads	174

Synchronisation entre threads	176
Les mutex	177
Verrous R/W locks	181
Conclusion	182
CHAPITRE 7	
Aspects avancés des Pthreads	183
Annulation d'un thread	183
Fonctions de nettoyage	187
Variables conditions	190
Types de mutex	195
Taille de la pile	200
Appel de fork()	201
Données globales privées	202
Conclusion	204
CHAPITRE 8	
Fonctions horaires	205
Horodatage et type time_t	206
Lecture de l'heure	208
Configuration de l'heure système	214
Conversions, affichages de dates et d'heures	215
Calcul d'intervalles	230
Fuseau horaire	231
Conclusion	234
CHAPITRE 9	
Sommeil des processus et contrôle des ressources	235
Endormir un processus	235
Utilisation des temporisations Unix	242
Timers temps réel	247
Notifications par descripteur	251
Suivre l'exécution d'un processus	252
Obtenir des statistiques sur un processus	256
Limiter les ressources consommées par un processus	260
Conclusion	268
CHAPITRE 10	
Ordonnancements sous Linux	269
États d'une tâche	269
Fonctionnement multitâche, priorités	274

Modification de la priorité d'un autre processus	280
Systèmes multiprocesseurs, migrations	282
Consultation du processeur utilisé	283
Choix des processeurs autorisés pour une tâche	284
Ordonnancements temps réel	288
Ordonnancement sous algorithme FIFO	290
Ordonnancement sous algorithme RR	292
Ordonnancement sous algorithme OTHER	292
Récapitulation	293
Temps Réel ?	293
Modification de la politique d'ordonnancement	294
Conclusion	300

CHAPITRE 11

Gestion classique des signaux..... 301

Généralités	301
Liste des signaux sous Linux	304
Signaux SIGABRT et SIGIOT	304
Signaux SIGALRM, SIGVTALRM et SIGPROF	305
Signaux SIGBUS et SIGSEGV	306
Signaux SIGCHLD et SIGCLD	306
Signaux SIGFPE et SIGSTKFLT	307
Signal SIGHUP	307
Signal SIGILL	308
Signal SIGINT	309
Signaux SIGIO et SIGPOLL	309
Signal SIGKILL	310
Signal SIGPIPE	310
Signal SIGQUIT	311
Signaux SIGSTOP, SIGCONT, et SIGTSTP	311
Signal SIGTERM	311
Signal SIGTRAP	312
Signaux SIGTTIN et SIGTTOU	312
Signal SIGURG	313
Signaux SIGUSR1 et SIGUSR2	313
Signal SIGWINCH	313
Signaux SIGXCPU et SIGXFSZ	314
Signaux temps réel	314
Émission d'un signal sous Linux	317
Délivrance des signaux	319
Réception des signaux avec l'appel système signal()	322
Conclusion	333

CHAPITRE 12

Gestion portable des signaux..... 335

Réception des signaux avec sigaction()	335
Configuration des ensembles de signaux	339
Exemples d'utilisation de sigaction()	340
Blocage des signaux	347
Attente d'un signal	352
Écriture correcte d'un gestionnaire de signaux	354
Utilisation d'un saut non local	357
Un signal particulier : l'alarme	360
Conclusion	364

CHAPITRE 13

Signaux temps réel 365

Caractéristiques des signaux temps réel	366
Nombre de signaux temps réel	366
Empilement des signaux bloqués	367
Délivrance prioritaire des signaux	368
Informations supplémentaires fournies au gestionnaire	369
Émission d'un signal temps réel	369
Traitement rapide des signaux temps réel	378
Conclusion	382

CHAPITRE 14

Gestion de la mémoire du processus..... 383

Allocation et libération de mémoire	383
Utilisation de malloc()	384
Utilisation de calloc()	391
Utilisation de realloc()	394
Utilisation de free()	395
Règles de bonne conduite pour l'allocation et la libération de mémoire	396
Désallocation automatique avec alloca()	399
Débogage des allocations mémoire	402
Configuration de l'algorithme utilisé par malloc()	406
Suivi des allocations et des libérations	407
Surveillance automatique des zones allouées	410
Fonctions d'encadrement personnalisées	413
Utilisation de Valgrind	414
Conclusion	420

CHAPITRE 15

Gestion avancée de la mémoire 421

Verrouillage de pages en mémoire	421
Projection d'un fichier sur une zone mémoire	425
Protection de l'accès à la mémoire	439
Conclusion	444

CHAPITRE 16

Utilisation des blocs mémoire et des chaînes 445

Manipulation de blocs de mémoire	446
Mesures, copies et comparaisons de chaînes	452
Caractères accentués et codage UTF-8	464
Recherches dans une zone de mémoire ou dans une chaîne	471
Recherche dans un bloc de mémoire	471
Recherche de caractères dans une chaîne	473
Recherche de sous-chaînes	474
Analyse lexicale	478
Conclusion	482

CHAPITRE 17

Tris, recherches et structuration des données 483

Fonctions de comparaison	483
Recherche linéaire, données non triées	486
Recherches dichotomiques dans une table ordonnée	492
Manipulation, exploration et parcours d'un arbre binaire	499
Gestion d'une table de hachage	505
Récapitulatif sur les méthodes d'accès aux données	512
Conclusion	514

CHAPITRE 18

Routines avancées de traitement des blocs mémoire..... 515

Utilisation des expressions rationnelles	515
Cryptage de données	524
Cryptage élémentaire	524
Cryptage simple et mots de passe	525
Cryptage de blocs de mémoire avec DES	529
Conclusion	534

CHAPITRE 19

Types de données et conversions 535

Types de données génériques	535
Types de tailles définies	537
Catégories de caractères	537
Conversions entre catégories de caractères	541
Conversions entre différents types	543
Types et conversions mathématiques	552
Nombres complexes	552
Conversions de réels en entiers	554
Infinis et erreurs	557
Représentation des réels en virgule flottante	560
Conclusion	561

CHAPITRE 20

Entrées-sorties simplifiées 563

Flux standard d'un processus	563
Écritures dans un flux	567
Écritures formatées	567
Autres fonctions d'écriture formatée	576
Écritures simples de caractères ou de chaînes	580
Saisie de caractères	584
Réinjection de caractère	589
Saisie de chaînes de caractères	591
Lectures formatées depuis un flux	598
Conclusion	609

CHAPITRE 21

Flux de données..... 611

Différences entre flux et descripteurs	611
Ouverture et fermeture d'un flux	613
Ouverture normale d'un flux	613
Fermeture d'un flux	616
Présentation des buffers associés aux flux	617
Ouvertures particulières de flux	619
Lectures et écritures dans un flux	622
Positionnement dans un flux	626
Positionnement classique	627
Positionnement compatible Unix 98	629
Problèmes de portabilité	633

Paramétrage des buffers associés à un flux	634
Type de buffers	634
Modification du type et de la taille du buffer	636
État d'un flux	640
Conclusion	642

CHAPITRE 22

Descripteurs de fichiers 643

Ouverture et fermeture d'un descripteur de fichier	643
Lecture ou écriture sur un descripteur de fichier	654
Primitives de lecture	654
Primitives d'écriture	658
Positionnement dans un descripteur de fichier	666
Manipulation et duplication de descripteurs	668
Duplication de descripteur	672
Accès aux attributs du descripteur	672
Attributs du fichier	675
Verrouillage d'un descripteur	677
Autre méthode de verrouillage	685
Conclusion	686

CHAPITRE 23

Communications classiques entre processus 687

Les tubes	688
Les tubes nommés	702
Conclusion	708

CHAPITRE 24

Communications avec les IPC 709

Communications avec les IPC Posix	710
Files de messages	710
Mémoire partagée	714
Sémaphores	720
Administration des ressources existantes	724
Les mécanismes IPC Système V	725
Obtention d'une clé	725
Ouverture de l'IPC	726
Contrôle et paramétrage	727
Files de messages	727
Mémoire partagée	731
Conclusion	741

CHAPITRE 25

Entrées-sorties avancées.....	743
Entrées-sorties non bloquantes	743
Multiplexage d'entrées-sorties	750
Attente d'événements – Multiplexage d'entrées	750
Distribution de données – Multiplexage de sorties	760
Entrées-sorties asynchrones	763
Asynchronisme utilisant fcntl()	763
Asynchronisme compatible Posix.1b	764
Écritures synchronisées	776
Conclusion	780

CHAPITRE 26

Programmation réseau	781
Réseaux et couches de communication	781
Résolution de nom	786
Services et numéros de ports	792
Ordre des octets	795
Conclusion	798

CHAPITRE 27

Utilisation des sockets	799
Concept de socket	799
Création d'une socket	800
Affectation d'adresse	803
Mode connecté et mode non connecté	807
Attente de connexions	808
Demander une connexion	814
Fermeture d'une socket	819
Recevoir ou envoyer des données	822
Accès aux options des sockets	829
Programmation d'un démon ou utilisation de inetd	835
Conclusion	839

CHAPITRE 28

Accès au contenu des répertoires	841
Lecture du contenu d'un répertoire	842
Changement de répertoire de travail	847
Changement de répertoire racine	853
Création et suppression de répertoire	857

Suppression ou déplacement de fichiers	859
Fichiers temporaires	863
Recherche de noms de fichiers	865
Correspondance simple d'un nom de fichier	865
Recherche sur un répertoire total	868
Développement complet à la manière d'un shell	872
Descente récursive de répertoires	878
Conclusion	881

CHAPITRE 29

Attributs des fichiers..... 883

Informations associées à un fichier	883
Autorisations d'accès	888
Propriétaire et groupe d'un fichier	890
Taille du fichier	891
Horodatages d'un fichier	894
Liens physiques	895
Liens symboliques	897
Nœud générique du système de fichiers	901
Masque de création de fichier	905
Surveillance du système de fichiers	906
Conclusion	910

CHAPITRE 30

Accès aux informations du système 911

Groupes et utilisateurs	911
Fichier des groupes	912
Fichier des utilisateurs	915
Fichier des interpréteurs shell	918
Nom d'hôte et de domaine	918
Nom d'hôte	918
Nom de domaine	920
Identifiant d'hôte	920
Informations sur le noyau	921
Identification du noyau	921
Informations sur l'état du noyau	922
Système de fichiers	924
Caractéristiques des systèmes de fichiers	925
Informations sur un système de fichiers	931
Montage et démontage des partitions	933

Journalisation	934
Journal utmp	934
Fonctions X/Open	939
Journal wtmp	939
Journal syslog	941
Conclusion	945
CHAPITRE 31	
Internationalisation	947
Principe	948
Catégories de localisations disponibles	948
Traduction de messages	952
Catalogues de messages gérés par catgets()	953
Catalogues de messages Gnu GetText	958
Configuration de la localisation	963
Localisation et fonctions de bibliothèques	966
Localisation et fonctions personnelles	972
Conclusion	979
CHAPITRE 32	
Gestion du terminal	981
Définition des terminaux	981
Configuration d'un terminal	983
Membre c_iflag de la structure termios	987
Membre c_oflag de la structure termios	988
Membre c_cflag de la structure termios	989
Membre c_lflag de la structure termios	990
Membre c_cc[] de la structure termios	991
Basculement du terminal en mode brut	994
Connexion à distance sur une socket	999
Utilisation d'un pseudo-terminal	1002
Configuration d'un port série RS-232	1012
Conclusion	1021
Bibliographie	1023
Standards	1023
Livres et articles	1023
Index.....	1027

1

Concepts et outils

Ce premier chapitre a pour but de présenter les principes généraux de la programmation sous Linux, ainsi que les outils disponibles pour réaliser des applications. Nous insisterons sur les outils livrés automatiquement avec les distributions Linux courantes, mais il en existe de nombreux autres développés de manière indépendante, et facilement disponibles sur Internet. Certains utilitaires spécifiques seront présentés dans les chapitres qui les concernent (ordonnancement, gestion mémoire, etc.).

Nous nous concentrerons sur le développement en C « pur », mais nous verrons aussi des utilitaires et des bibliothèques permettant d'étendre les possibilités de la bibliothèque Glibc. Nous parlerons également de Python car nous verrons qu'il est possible de réaliser des scripts avec ce langage fonctionnellement équivalents aux programmes C interagissant avec le système.

Nous ne présenterons pas le détail des commandes permettant de manipuler les outils décrits mais plutôt leurs rôles, pour bien comprendre comment s'organise le processus de développement système sous Linux.

Généralités sur le développement sous Linux

Dans une machine fonctionnant sous Linux, de nombreuses couches logicielles sont empilées, chacune fournissant des services aux autres. Il est important de comprendre comment fonctionne ce modèle pour savoir où une application viendra s'intégrer.

La base du système est le noyau, qui est le seul élément à porter véritablement le nom « Linux ». Le noyau est souvent imaginé comme une sorte de logiciel mystérieux fonctionnant en arrière-plan pour surveiller les applications des utilisateurs, mais il s'agit avant tout d'un ensemble cohérent de routines fournissant des services aux applications, en s'assurant de conserver l'intégrité du système. Pour le développeur, le noyau est surtout une interface entre son application, qui peut être exécutée par n'importe quel utilisateur, et la machine physique dont la manipulation directe doit être supervisée par un dispositif privilégié.

Le noyau fournit donc des points d'entrée, qu'on nomme « appels-système », et que le programmeur invoque comme des sous-routines offrant des services variés. Par exemple l'appel système `write()` permet d'écrire des données dans un fichier. L'application appelante n'a pas besoin de savoir sur quel type de système de fichiers (*ext4*, *ext3*, *reiserfs*, *vfat*...) l'écriture se fera. L'envoi des données peut même avoir lieu de manière transparente dans un tube de communication entre applications ou vers un client distant connecté par réseau. Seul le noyau s'occupera de la basse besogne consistant à piloter les contrôleurs de disque, gérer la mémoire ou coordonner le fonctionnement des périphériques comme les cartes réseau.

Il existe plusieurs centaines d'appels système sous Linux. Ils effectuent des tâches très variées, allant de l'allocation mémoire aux entrées-sorties directes sur un périphérique, en passant par la gestion du système de fichiers, le lancement d'applications ou la communication réseau.

L'utilisation des appels système est en principe suffisante pour écrire n'importe quelle application sous Linux. Toutefois, ce genre de développement serait particulièrement fastidieux, et la portabilité du logiciel résultant serait loin d'être assurée. Les systèmes Unix compatibles avec la norme SUSv4 offrent normalement un jeu d'appels système commun, assurant ainsi une garantie de compatibilité minimale. Néanmoins, cet ensemble commun est loin d'être suffisant dès qu'on dépasse le stade d'une application triviale.

La norme *Single Unix Specifications version 4* (dite « SUSv4 ») décrit l'environnement d'utilisation et l'interface de programmation que doivent implémenter les systèmes qui veulent correspondre au modèle Unix tel que l'Open Group le conçoit. On peut obtenir gratuitement toutes ces informations à partir du site www.opengroup.org.

Il existe donc une couche supérieure avec des fonctions qui viennent compléter les appels système, permettant ainsi un développement plus facile et assurant également une meilleure portabilité des applications vers les environnements non SUSv4. Cette interface est constituée par la bibliothèque C.

Cette bibliothèque regroupe des fonctionnalités complémentaires de celles qui sont assurées par le noyau, par exemple toutes les fonctions mathématiques (le noyau n'uti-

lise jamais les nombres réels). La bibliothèque C permet aussi d'encapsuler les appels système dans des routines de plus haut niveau, qui sont donc plus aisément portables d'une machine à l'autre. Nous verrons par exemple que les descripteurs de fichiers fournis par l'interface du noyau restent limités à l'univers Unix, alors que les flux de données qui les encadrent sont portables sur tout système implémentant une bibliothèque ISO C, tout en ajoutant d'ailleurs des fonctionnalités importantes. Les routines proposées par la bibliothèque C (par exemple `malloc()` et toutes les fonctions d'allocation mémoire) sont aussi un moyen de faciliter la tâche du programmeur en offrant une interface de haut niveau pour des appels système plus ardu, comme `sbrk()`.

Il y a eu plusieurs bibliothèques C successivement utilisées sous Linux. Les versions 1 à 4 de la `libc` Linux étaient principalement destinées aux programmes exécutables utilisant le format `a.out`. La version 5 de la `libc` a représenté une étape importante puisque le standard exécutable est devenu le format `elf`, beaucoup plus performant que le précédent. À partir de la version 2.0 du noyau Linux, toutes les distributions ont basculé vers une autre version de bibliothèque, la `Glibc`, issue du projet Gnu. Elle est parfois nommée – abusivement – `libc 6`. Au moment de la rédaction de ce texte, la version utilisée de la `Glibc` est la 2.9, mais elle est évolue régulièrement. Toutefois, les fonctionnalités que nous étudierons ici resteront normalement immuables pendant longtemps.

Pour connaître votre version de bibliothèque C, exécutez depuis le *shell* le fichier `/lib/libc.so.6`. De plus, le numéro de noyau est visible avec la commande `uname -a`.

La bibliothèque `Glibc 2` est très performante. Elle se conforme de manière précise aux standards actuels comme `SUSv4`, tout en offrant des extensions personnelles innovantes. Le développeur sous Linux dispose donc d'un environnement de qualité, permettant aussi bien l'écriture d'applications portables que l'utilisation d'extensions Gnu performantes. La disponibilité du code source de la `Glibc 2` rend également possible la transposition d'une particularité Gnu vers un autre système en cas de portage du logiciel.

En fait, la richesse de la bibliothèque `Glibc` rend plus facile le portage d'une application depuis un autre environnement vers Linux que dans le sens inverse. Le défaut – résultant de cette richesse – est la taille de la bibliothèque C. Elle est souvent trop volumineuse pour être employée dans des environnements restreints et des systèmes embarqués. On lui préfère alors d'autres bibliothèques C (comme `musl` ou `uClibc`), moins riches mais sensiblement plus légères.

Le langage Python se présente sous la forme d'un interpréteur qui exécute des scripts en s'appuyant entre autres sur la bibliothèque `Glibc`.

Les fonctions de la bibliothèque Glibc et les appels système représentent un ensemble minimal de fonctionnalités indispensables pour le développement d'applications. Ils sont pourtant très limités en termes d'interface utilisateur. Aussi plusieurs bibliothèques de fonctions ont-elles été créées pour rendre le dialogue avec l'utilisateur plus convivial. Ces bibliothèques sortent du cadre de ce livre, mais nous en citerons quelques-unes à la fin de ce chapitre.

Le programmeur retiendra donc que nous décrirons ici deux types de fonctions, les appels système, implémentés par le noyau et offrant un accès de bas niveau aux fonctionnalités du système, et les routines de bibliothèques, qui peuvent compléter les possibilités du noyau, mais aussi l'encadrer pour le rendre plus simple et plus portable. L'invocation d'un appel système est une opération assez coûteuse, car il est nécessaire d'assurer une commutation du processus en mode noyau avec toutes les manipulations que cela impose sur les registres du processeur. L'appel d'une fonction de bibliothèque au contraire est un mécanisme léger, équivalent à l'appel d'une sous-routine du programme (sauf bien entendu quand la fonction de bibliothèque invoque elle-même un appel système).

Pour obtenir plus de précisions sur le fonctionnement du noyau Linux, on pourra se reporter à [LOVE 2003] *Linux Kernel Development*, ou directement aux fichiers source de Linux

Pour des détails sur l'implémentation des systèmes Unix, l'ouvrage [Bach 1989] *Conception du système Unix* est un grand classique, ainsi que [Tanenbaum 1997] *Operating Systems, Design and Implementation*.

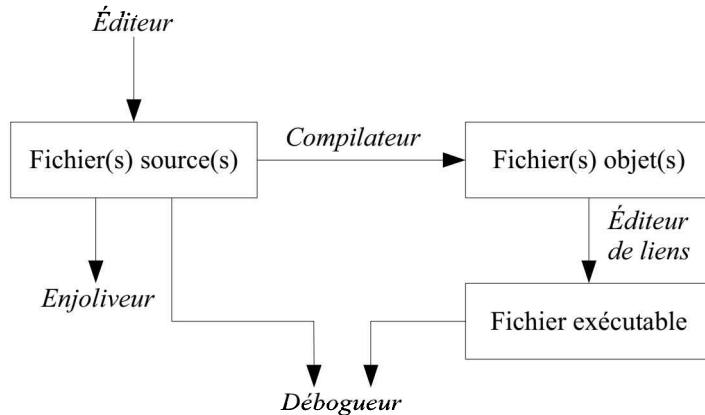
Outils de développement

Le développement en C sous Linux, comme sous la plupart des autres systèmes d'exploitation, met en œuvre principalement cinq types d'utilitaires.

- L'éditeur de texte, qui est à l'origine de tout le processus de développement applicatif. Il nous permet de créer et de modifier les fichiers source.
- Le compilateur, qui permet de passer d'un fichier source à un fichier objet. Cette transformation se fait en réalité en plusieurs étapes grâce à différents composants (préprocesseur C, compilateur, assembleur), mais nous n'avons pas besoin de les distinguer ici.
- L'éditeur de liens, qui assure le regroupement des fichiers objet provenant des différents modules et les associe avec les bibliothèques utilisées pour l'application. Nous obtenons ici un fichier exécutable.

- Le débogueur, qui peut alors permettre l'exécution pas à pas du code, l'examen des variables internes, etc. Pour cela, il a besoin du fichier exécutable et du code source.
- Notons également l'emploi éventuel d'utilitaires annexes travaillant à partir du code source, comme les indexeurs, les enjoliveurs de code, les outils de documentation automatique, etc.

Figure 1–1
Processus de développement en C

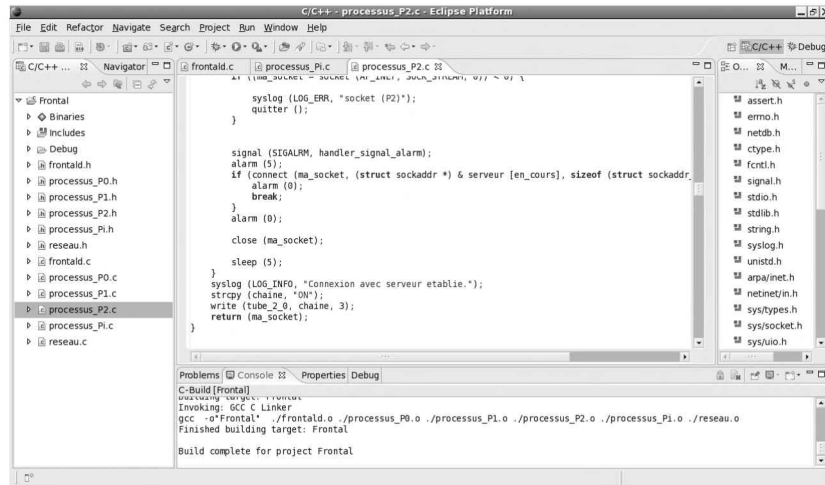


Deux écoles de programmeurs coexistent sous Linux (et Unix en général) : ceux qui préfèrent disposer d'un environnement intégrant tous les outils de développement, depuis l'éditeur de texte jusqu'au débogueur, et ceux qui utilisent plutôt les différents utilitaires de manière séparée, configurant manuellement un fichier Makefile pour recompiler leur application sur un terminal Xterm, tandis que leur éditeur préféré s'exécute dans une autre fenêtre. Dans cet ouvrage, nous considérerons surtout la situation d'un développeur préférant lancer lui-même ses outils en ligne de commande. Toutefois, il est intéressant de s'arrêter un instant sur les environnements de développement intégrés les plus répandus aujourd'hui dans le monde Linux.

Eclipse

Eclipse est un environnement de développement intégré écrit en langage Java. Conçu à l'origine par IBM pour l'écriture d'applications en Java, il fut étendu à de nombreux autres langages par l'intermédiaire de greffons (*plug-ins*) comme le CDT (C/C++ Development Tool). L'avantage principal d'Eclipse – et de tous les environnements présentés ci-après – est de regrouper ainsi les différents outils de développement avec une interface commune et homogène. En outre, Eclipse permet d'améliorer grandement l'efficacité des sessions de débogage, comme nous le verrons plus loin.

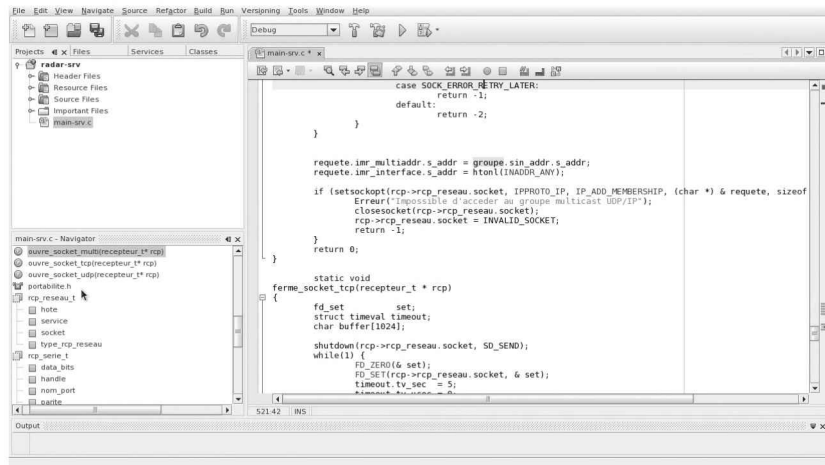
Figure 1-2
L'environnement
de développement Eclipse



Il existe plusieurs ouvrages documentant le fonctionnement d'Eclipse, ainsi que de nombreux tutoriaux disponibles sur Internet. Des adresses sont indiquées en annexe de ce livre.

NetBeans

Figure 1-3
NetBeans



NetBeans est le concurrent d'Eclipse développé à l'origine par Sun Microsystems (qui trouvait le nom et le logo d'Eclipse un peu trop provocateurs à l'encontre d'une entreprise nommée « Sun » !) pour le développement en Java et étendu par la suite pour d'autres langages.

Il existe d'autres environnements de développement intégrés, comme Code::Blocks, Geany ou Kdevelop. Leurs fonctionnalités sont globalement équivalentes, et chacun pourra choisir en fonction de ses propres goûts, son historique et son environnement de travail.

Pour les programmeurs préférant utiliser les outils séparément, nous allons détailler leurs rôles respectifs. Nous décrirons certaines options, mais de nombreuses précisions supplémentaires pourront être trouvées dans les pages de manuel (commande `man`) ou dans la documentation accessible avec la commande `info`.

Programmation en Python

Le Python, pour sa part, est un langage interprété, ce qui signifie qu'on fournit directement le fichier source (qu'on appelle plutôt un « script ») à un programme qui l'analyse et exécute directement ses opérations. Ce programme est nommé un « interpréteur ». Il existe de nombreux autres langages interprétés, les plus célèbres étant par exemple Perl, Tcl, Ruby, Java, etc.

Le travail d'interprétation étant effectué dynamiquement, l'exécution est plus lente qu'avec un programme compilé (l'analyse étant réalisée au préalable). Néanmoins, les interpréteurs modernes utilisent diverses optimisations qui offrent des performances d'exécution tout à fait acceptables.

Pour qu'un script Python fonctionne, il suffit donc de saisir son code dans un éditeur de texte, puis d'invoquer la commande `python` en lui passant le nom du script en argument. On peut même simplifier le travail en ajoutant une première ligne (nommée *shebang*) ayant l'aspect suivant :

```
| #! /usr/bin/python
```

Le script est ensuite rendu exécutable avec la commande :

```
$ chmod +x nom-du-script
```

Dès lors, on peut lancer l'exécution du script directement, comme on lance un exécutable après compilation :

```
| $ ./nom-du-script
```

La ligne *shebang* (identifiée par les deux premiers caractères du fichier) permet de connaître l'interpréteur à invoquer pour analyser et exécuter le script.

La version actuelle du langage Python est 3.5 ; c'est celle qui est considérée dans ce livre. Néanmoins, il existe encore de nombreux systèmes avec des versions anté-

rieures. Il y a eu de gros changements entre les versions 2.x et 3.x ; c'est donc pour assurer une compatibilité avec les versions précédentes qu'on rencontrera les directives `from __future__` dans de nombreux scripts, essentiellement pour indiquer qu'on doit dorénavant considérer `print` comme une fonction.

Éditeurs de texte

L'éditeur de texte est probablement la fenêtre de l'écran que le développeur regarde le plus. Il passe la majeure partie de son temps à saisir, relire, modifier son code, et il est essentiel de maîtriser parfaitement les commandes de base pour le déplacement, les fonctions de copier-coller et le basculement rapide entre plusieurs fichiers source.

Chaque programmeur a généralement son éditeur fétiche, dont il connaît les possibilités, et qu'il essaye au maximum d'adapter à ses préférences. Il existe deux champions de l'édition de texte sous Unix, Vi d'une part et Emacs de l'autre. Ces deux logiciels ne sont pas du tout équivalents, mais ont chacun leurs partisans et leurs détracteurs.

Vi et Emacs

Emacs est théoriquement un éditeur de texte, mais des possibilités d'extension par l'intermédiaire de scripts Lisp en ont fait une énorme machine capable d'offrir l'essentiel des commandes dont un développeur peut rêver.

Vi est beaucoup plus léger, il offre nettement moins de fonctionnalités et de possibilités d'extensions que Emacs. Les avantages de Vi sont sa disponibilité sur toutes les plates-formes Unix et la possibilité de l'utiliser même sur un système très réduit pour réparer des fichiers de configuration. La version utilisée sous Linux est nommée `vim` (mais un alias permet de le lancer en tapant simplement `vi` sur le clavier).

Figure 1–4
Vi sous X11

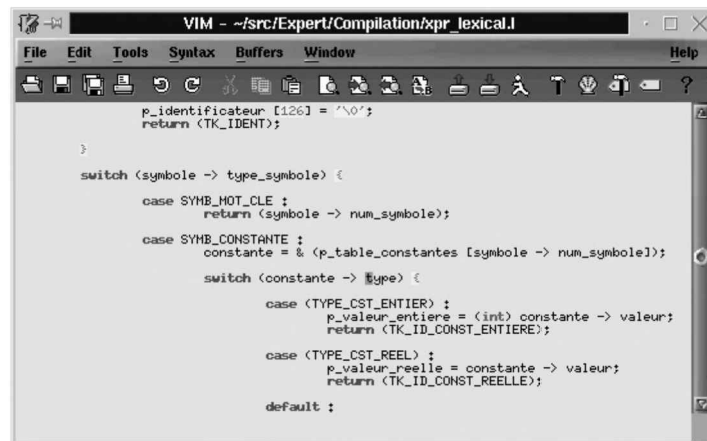
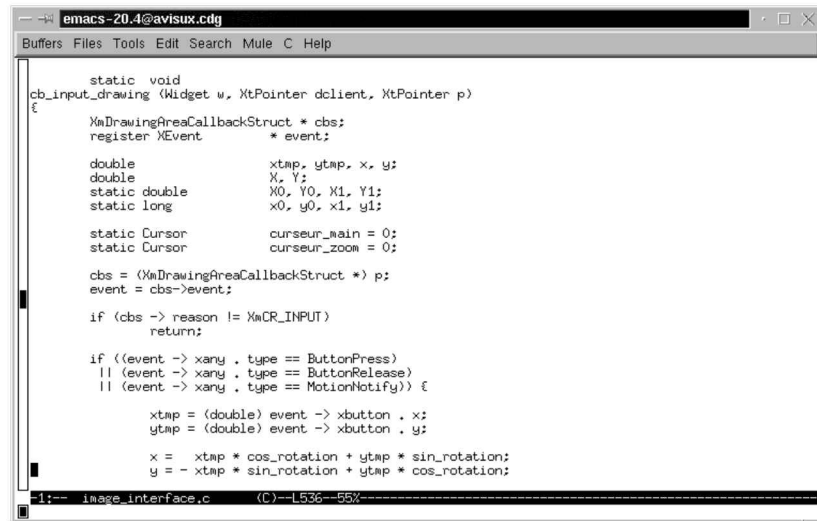


Figure 1-5
Emacs sous X11



Vi et Emacs peuvent fonctionner sur un terminal texte, mais ils sont largement plus simples à utiliser dans leur version fenêtrée X11. L'un comme l'autre nécessitent un apprentissage. Il existe de nombreux ouvrages et didacticiels pour l'utilisation de ces éditeurs performants.

Éditeurs Gnome ou KDE

Les deux environnements graphiques les plus en vogue actuellement, Gnome et KDE, proposent tous deux un éditeur de texte parfaitement incorporé dans l'ensemble des applications fournies. Malheureusement, ces éditeurs ne sont pas vraiment très appropriés pour le programmeur.

Ils sont bien adaptés pour le dialogue avec le reste de l'environnement (ouverture automatique de documents depuis le gestionnaire de fichiers, accès aux données par un glissement des icônes, etc.).

En contrepartie, il leur manque les possibilités les plus appréciables pour un développeur, comme le basculement alternatif entre deux fichiers, la création de macros rapides pour répéter des commandes de formatage sur un bloc complet de texte, ou la possibilité de scinder la fenêtre en deux pour éditer une routine tout en jetant un coup d'œil sur la définition des structures en début de fichier.

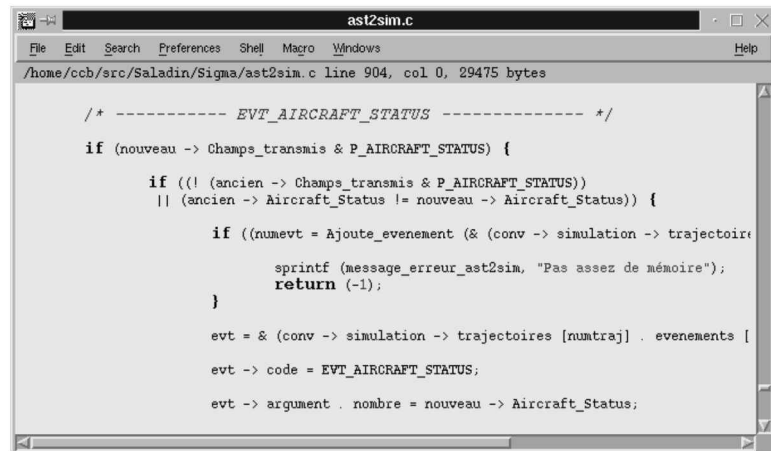
On les utilisera donc plutôt comme des outils d'appoint mais rarement pour travailler longuement sur une application.

Nedit

Comme il est impossible de citer tous les éditeurs disponibles sous Linux, je n'en mentionnerai qu'un seul, que je trouve parfaitement adapté aux besoins du développeur. L'éditeur Nedit est très intuitif et ne nécessite aucun apprentissage. La lecture de sa documentation permet toutefois de découvrir une puissance surprenante, tant dans la création de macros que dans le lancement de commandes externes (`make`, `spell`, `man...`), ou la manipulation de blocs de texte entiers.

Nedit est disponible – sous forme de code source ou précompilé – pour la plupart des Unix, mais n'est pas toujours installé à l'origine. L'essentiel des distributions Linux l'incluent sur leurs CD-Rom d'installation.

Figure 1–6
Nedit



Compilateur, éditeur de liens

Le compilateur C utilisé sous Linux est `gcc` (*Gnu Compiler Collection*). On peut également l'invoquer sous le nom `cc`, comme c'est l'usage sous Unix, ou `g++` si on compile du code C++.

Le compilateur s'occupe de regrouper les appels aux sous-éléments utilisés durant la compilation.

- 1 Le préprocesseur, nommé `cpp`, gère toutes les directives `#define`, `#ifdef`, `#include...` du code source.
- 2 Le compilateur C proprement dit, nommé `cc1` ou `cc1plus` si on compile en utilisant la commande `g++` (voire `cc1obj` si on utilise le dialecte Objective-C). Le compilateur transforme le code source prétraité en fichier contenant le code assembleur. Il est donc possible d'examiner en détail le code engendré, voire d'optimiser manuellement certains passages cruciaux (bien que ce soit rarement utile).

3 L'assembleur `as` fournit des fichiers objet.

4 L'éditeur de liens, nommé `ld`, assure le regroupement des fichiers objet et des bibliothèques pour fournir enfin le fichier exécutable.

Les différents outils intermédiaires invoqués par `gcc` se trouvent dans un répertoire situé dans l'arborescence en dessous de `/usr/lib/gcc-lib/`. On ne s'étonnera donc pas de ne pas les trouver dans le chemin de recherche `PATH` habituel du shell.

On notera que `gcc` est un outil très complet, disponible sur de nombreuses plates-formes Unix, et permettant la compilation croisée, où le code pour la plate-forme cible est produit sur un environnement de développement généralement plus puissant, même si les systèmes d'exploitation et les processeurs sont différents.

Le compilateur `gcc` utilise des conventions sur les suffixes des fichiers pour savoir quel utilitaire invoquer lors des différentes phases de compilation. Ces conventions sont les suivantes.

Suffixe	Produit par	Rôle
<code>.c</code>	Programmeur	Fichier source C, sera transmis à <code>cpp</code> , puis à <code>cc1</code> .
<code>.cc</code> ou <code>.C</code>	Programmeur	Fichier source C++, sera transmis à <code>cpp</code> , puis à <code>cc1plus</code> .
<code>.m</code>	Programmeur	Fichier source Objective C, sera transmis à <code>cpp</code> , puis à <code>cc1obj</code> .
<code>.h</code>	Programmeur	Fichier d'en-tête inclus dans les sources concernées. Considéré comme du C ou du C++ en fonction du compilateur invoqué (<code>gcc</code> ou <code>g++</code>).
<code>.i</code>	<code>cpp</code>	Fichier C prétraité par <code>cpp</code> , sera transmis à <code>cc1</code> .
<code>.ii</code>	<code>cpp</code>	Fichier C++ prétraité par <code>cpp</code> , sera transmis à <code>cc1plus</code> .
<code>.s</code> ou <code>.S</code>	<code>cc1</code> , <code>cc1plus</code> , <code>cc1obj</code>	Fichier d'assemblage fourni par l'un des compilateurs <code>cc1</code> , va être transmis à l'assembleur <code>as</code> .
<code>.o</code>	<code>as</code>	Fichier objet obtenu après l'assemblage, prêt à être transmis à l'éditeur de liens <code>ld</code> pour fournir un exécutable.
<code>.a</code>	<code>ar</code>	Fichier de bibliothèque que l'éditeur de liens peut lier avec les fichiers objet pour créer l'exécutable.

En général, seules les trois premières lignes de ce tableau concernent le programmeur, car tous les autres fichiers sont transmis automatiquement à l'utilitaire adéquat. Dans le cadre de ce livre, nous ne nous intéresserons qu'aux fichiers C, même si les fonctions de bibliothèques et les appels système étudiés peuvent très bien être employés en C++.

La plupart du temps, on invoque simplement `gcc` en lui fournissant le ou les noms des fichiers source, et éventuellement le nom du fichier exécutable de sortie, et il assure toute la transformation nécessaire. Si aucun nom de fichier exécutable n'est indiqué, `gcc` en créera un, nommé `a.out`. Cela est simplement une tradition historique sous Unix, même si le fichier est en réalité au format actuel `elf`.

L'invocation de `gcc` se fait donc avec les arguments suivants.

- Les noms des fichiers C à compiler ou les noms des fichiers objet à lier. On peut en effet procéder en plusieurs étapes pour compiler les différents modules d'un projet, retardant l'édition des liens jusqu'au moment où tous les fichiers objet seront disponibles.
- Éventuellement des définitions de macros pour le préprocesseur, précédées de l'option `-D`. Par exemple `-D_XOPEN_SOURCE=500` est équivalent à une directive `#define _XOPEN_SOURCE 500` avant l'inclusion de tout fichier d'en-tête.
- Éventuellement le chemin de recherche des fichiers d'en-tête (en plus de `/usr/include`), précédé de l'option `-I`. Ceci est utile par exemple lors du développement sous X-Window, en ajoutant `-I/usr/include/X11`.
- Éventuellement le chemin de recherche des bibliothèques supplémentaires (en plus de `/lib` et `/usr/lib`), précédé de l'option `-L`. Comme pour l'option précédente on utilise surtout ceci pour le développement sous X11, avec par exemple `-L/usr/lib/X11`.
- Le nom d'une bibliothèque supplémentaire à utiliser lors de l'édition des liens, précédé du préfixe `-l`. Il s'agit bien du nom de la bibliothèque, et pas du fichier. Par exemple la commande `-lm` permet d'inclure le fichier `libm.so` indispensable pour les fonctions mathématiques. De même, `-lcrypt` permet d'utiliser la bibliothèque `libcrypt.so` contenant les fonctions de chiffrement DES ou encore `-lrt` intègre les fonctionnalités temps réel.
- On peut préciser le nom du fichier exécutable, précédé de l'option `-o`.

Enfin, plusieurs options simples peuvent être utilisées, les plus courantes sont les suivantes.

Option	Argument	But
<code>-E</code>		Arrêter la compilation après le passage du préprocesseur, avant le compilateur.
<code>-S</code>		Arrêter la compilation après le passage du compilateur, avant l'assembleur.
<code>-c</code>		Arrêter la compilation après l'assemblage, laissant les fichiers objet disponibles.
<code>-W</code>	Avertissement	Valider les avertissements (<i>warnings</i>) décrits en arguments. Il en existe une multitude, mais l'option la plus couramment utilisée est <code>-Wall</code> , pour activer tous les avertissements.
<code>-pedantic</code>		Le compilateur fournit des avertissements encore plus rigoureux qu'avec <code>-Wall</code> , principalement orientés sur la portabilité du code.
<code>-g</code>		Inclure dans le code exécutable les informations nécessaires pour utiliser le débogueur. Cette option est généralement conservée jusqu'au basculement du produit en code de distribution.
<code>-O</code>	0 à 3	Optimiser le code engendré. Le niveau d'optimisation est indiqué en argument (0 = aucune). Il est déconseillé d'utiliser simultanément l'optimisation et le débogage.

Les combinaisons d'options les plus couramment utilisées sont donc :

```
$ gcc -Wall -g fichier1.c -c  
$ gcc -Wall -g fichier2.c -c
```

qui permettent d'obtenir deux fichiers exécutables qu'on regroupe ensuite ainsi :

```
$ gcc fichier1.o fichier2.o -o resultat
```

On peut aussi effectuer directement la compilation et l'édition des liens :

```
$ gcc -Wall -g fichier1.c fichier2.c -o resultat
```

Lorsque le code a atteint la maturité nécessaire pour basculer en version de distribution, on peut utiliser :

```
$ gcc -Wall -DNDEBUG -O2 fichier1.c fichier2.c -o resultat
```

La constante `NDEBUG` sert, nous le verrons dans un chapitre ultérieur, à éliminer tous le code de débogage incorporé explicitement dans le fichier source.

Les options permettant d'ajuster le comportement de `gcc` sont tellement nombreuses que l'on pourrait y consacrer un ouvrage complet. D'autant plus que `gcc` permet le développement croisé, c'est-à-dire la compilation sur une machine d'une application destinée à une autre plate-forme. Cela est particulièrement précieux pour la mise au point de programmes destinés à des systèmes embarqués par exemple, ne disposant pas nécessairement des ressources nécessaires au fonctionnement des outils de développement.

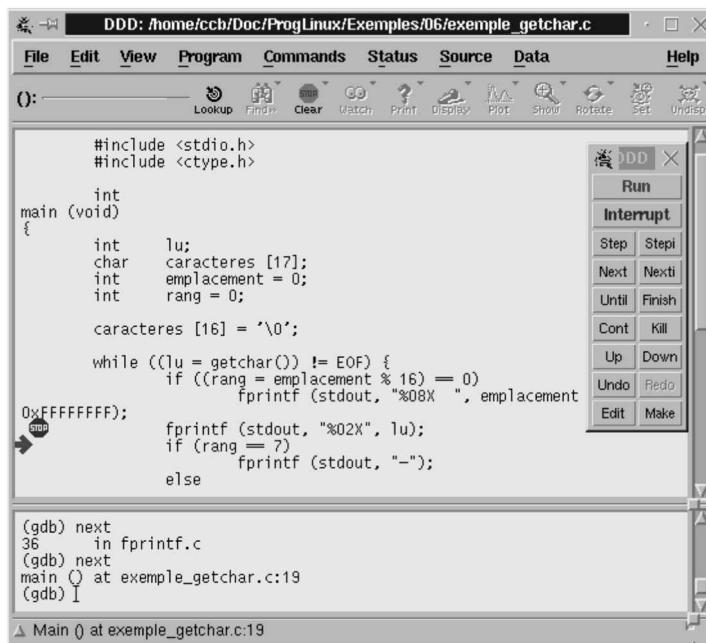
La plupart du temps nous ne nous soucierons pas de la ligne de commande utilisée pour compiler les applications, car elle se trouve incorporée directement dans le fichier de configuration du constructeur d'application `make` que nous verrons plus bas.

Débogueur, profileur

Lorsqu'une application a été compilée avec l'option `-g`, il est possible de l'exécuter sous le contrôle d'un débogueur. L'outil utilisé sous Linux est nommé `gdb` (*Gnu Debugger*). Cet utilitaire fonctionne en ligne de commande, avec une interface assez rébarbative.

Plusieurs frontaux graphiques pour gdb sont également disponibles, comme ddd (*Data Display Debugger*), plus agréable visuellement.

Figure 1-7
Utilisation de ddd



On peut trouver une autre interface graphique pour gdb dans l'environnement Eclipse dont nous parlerons plus loin. L'avantage de cet environnement pour le développement est la facilité de correction du code, recompilation et démarrage d'un nouveau débogage.

Le débogage d'une application pas à pas est un processus important lors de la mise au point d'un logiciel, mais ce n'est pas la seule utilisation de gdb et de ses frontaux. Lorsqu'un processus exécute certaines opérations interdites (écriture dans une zone non autorisée, tentative d'utilisation d'instruction illégale...) le noyau lui envoie un signal pour le tuer. Sous certaines conditions, l'arrêt de processus s'accompagne de la création d'un fichier *core*¹ sur le disque, représentant l'image de l'espace mémoire du processus au moment de l'arrêt, y compris le code exécutable. Le débogueur gdb est capable d'examiner ce fichier, afin de procéder à l'autopsie du processus tué. Cette analyse post-mortem est particulièrement précieuse lors de la mise au point d'un

1. Le terme « core » fait référence au noyau de fer doux se trouvant dans les tores de ferrite utilisés comme mémoire centrale sur les premières machines de l'informatique moderne. La technologie a largement évolué, mais le vocabulaire traditionnel a été conservé.

logiciel pour détecter où se produit un dysfonctionnement apparemment intempestif. De plus, gdb est également capable de déboguer un processus déjà en cours de fonctionnement !

Notons que gdb peut servir au débogage « croisé » : le débogueur fonctionne sur la station de développement tandis qu'un petit démon nommé gdbserver tourne sur la plate-forme cible (en général un système embarqué) reliée par réseau à la station de développement. On peut ainsi déboguer à distance un programme dans son environnement d'exécution définitif, même s'il s'agit d'un système embarqué.

Dans l'informatique « de terrain », il arrive parfois de devoir analyser d'urgence les circonstances d'arrêt d'un programme au moyen de son fichier core. Ce genre d'intervention peut avoir lieu à distance, par une connexion réseau, ou par une liaison modem vers la machine où l'application était censée fonctionner de manière sûre. Dans ces situations frénétiques, il est inutile d'essayer de lancer les interfaces graphiques encadrant le débogueur, et il est nécessaire de savoir utiliser gdb en ligne de commande.

On invoque généralement le débogueur gdb en lui fournissant en premier argument le nom du fichier exécutable. Au besoin, on peut fournir ensuite le nom d'un fichier core obtenu avec le même programme.

Lors de son invocation, gdb affiche un message de présentation, puis passe en attente de commande avec un message d'invite (gdb). Pour se documenter en détail sur son fonctionnement, on tapera « help ». Le débogueur proposera alors une série de thèmes que l'on peut approfondir. Les commandes les plus courantes sont les suivantes.

Commande	Rôle
list	Afficher le listing du code source.
run [argument]	Lancer le programme, qui s'exécutera jusqu'au prochain point d'arrêt.
break <ligne>	Insérer un point d'arrêt sur la ligne dont le numéro est fourni.
step	Avancer d'un pas, en entrant au besoin dans le détail des sous-routines.
next	Avancer jusqu'à la prochaine instruction, en exécutant les sous-routines sans s'arrêter.
cont	Continuer l'exécution du programme jusqu'au prochain point d'arrêt.
print <variable>	Afficher le contenu de la variable indiquée.
backtrace	Afficher le contenu de la pile, avec les invocations imbriquées des routines.
quit	Quitter le débogueur.

Il existe de très nombreuses autres commandes, comme attach <PID> qui permet de déboguer un programme déjà en cours d'exécution. Pour tout cela, on se reportera par exemple à la documentation en ligne `info` sur gdb.

Voici un exemple de session de débogage sur un exemple très simple, copié du chapitre 4.

```
$ gdb ./exemple-argv
GNU gdb Fedora (6.8-32.fc10)
Copyright (C) 2008 Free Software Foundation, Inc.
[...]
This GDB was configured as "i386-redhat-linux-gnu"...
```

Nous commençons par demander un aperçu du listing du programme :

```
(gdb) list
1 // -----
2 // exemple-argv.c
3 // Fichier d'exemple du livre "Developpement Systeme sous Linux"
4 // (C) 2000-2010 - Christophe BLAESS -Christophe.Blaess@Logilin.fr
5 // http://www.logilin.fr
6 // -----
7
8 #include <stdio.h>
9
10 int main (int argc, char * argv[])
(gdb)      (Entrée)
11 {
12     int i;
13
14     fprintf(stdout, "%s a reçu en argument :\n", argv[0]);
15     for (i = 1; i < argc; i++)
16         fprintf(stdout, " %s\n", argv[i]);
17     return 0;
18 }
(gdb)
```

Nous plaçons un point d'arrêt sur la première ligne de la fonction main :

```
(gdb) break main
Breakpoint 1 at 0x80483f8: file exemple-argv.c, line 14.
```


Nous indiquons les arguments en ligne de commande, puis nous démarrons le programme :

```
(gdb) set args un deux trois
(gdb) run
Starting program: /home/ccb/ProgLinux/Exemples/chapitre-01/exemple-argv
un deux trois
Breakpoint 1, main (argc=4, argv=0xbffff3b4) at exemple-argv.c:14
14     fprintf(stdout, "%s a reçu en argument :\n", argv[0]);
```

Le programme s'étant arrêté, nous pouvons examiner ses variables, puis le faire avancer d'une ligne de code :

```
(gdb) print argv[1]
$1 = 0xbffff594 "un"
(gdb) next
/home/cpb/ProgLinux/Exemples/chapitre-01/exemple-argv a reçu en argument :
15     for (i = 1; i < argc; i ++)
```

Nous plaçons un nouveau point d'arrêt en sortie de boucle, avant de demander au programme de continuer son exécution :

```
(gdb) break 17
Breakpoint 2 at 0x8048459: file exemple-argv.c, line 17
(gdb) cont
Continuing.
    un
    deux
    trois

Breakpoint 2, main (argc=4, argv=0xbffff3b4) at exemple-argv.c:17
17     return 0;
```

Le programme est arrivé sur le nouveau point d'arrêt, nous pouvons le continuer en pas à pas :

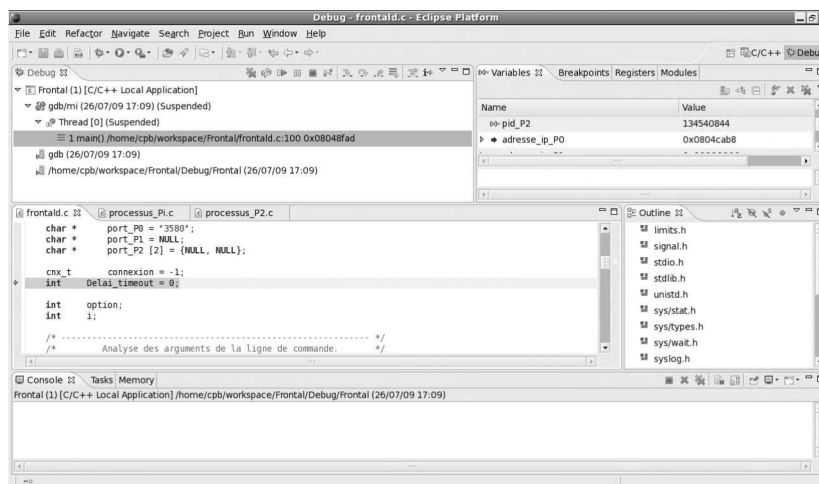
```
(gdb) next
14     }
(gdb) cont
Program exited normally.
```

Nous quittons à présent gdb :

```
(gdb) quit
$
```

On peut également utiliser à profit les environnements intégrés du type Eclipse pour le débogage. L'avantage par rapport à ddd est de pouvoir corriger et recompiler instantanément le code, et le relancer avec les mêmes points d'arrêt que précédemment.

Figure 1–8
Débogage sous Eclipse



Il existe un autre outil important dans la phase de mise au point : le profileur. Cet utilitaire observe le déroulement de l'application, et enregistre dans un fichier les temps de présence dans chaque routine du programme. Il est alors facile d'analyser les goulots d'étranglement dans lesquels le logiciel passe le plus clair de son temps. Ceci permet de cibler efficacement les portions de l'application qui auront besoin d'être optimisées. Bien entendu ceci ne concerne pas tous les logiciels, loin de là, puisque la plupart des applications passent l'essentiel de leur temps à attendre les ordres de l'utilisateur. Toutefois, il convient que chaque opération effectuée par le programme se déroule dans des délais raisonnables, et une simple modification d'algorithme, ou de structure de données, peut parfois permettre de réduire considérablement le temps d'attente de l'utilisateur. Ceci a pour effet de rendre l'ensemble de l'application plus dynamique à ses yeux et améliore la perception qualitative de l'ensemble du logiciel.

Valgrind

Il existe un outil très puissant pour le diagnostic et le débogage – entre autres – des problèmes liés à la mémoire (fuites mémoire et débordements de *buffer* principalement). Il s'agit de `valgrind`. Ce programme fait fonctionner l'application à tester en examinant les effets de chaque instruction et de chaque appel système. Il permet généralement de détecter de nombreuses faiblesses dans des applications qui semblaient a priori fonctionner normalement.

Je présenterai un exemple d'utilisation de `Valgrind` dans le chapitre 14 traitant des allocations mémoire.

Gprof

L'outil de profilage Gnu s'appelle `gprof`. Il fonctionne en analysant le fichier `gmon.out` qui est créé automatiquement lors du déroulement du processus, s'il a été compilé avec l'option `-pg` de `gcc`. Les informations fournies par `gprof` sont variées, mais permettent de découvrir les points où le programme passe l'essentiel de son temps.

On compile donc le programme à profiler ainsi :

```
$ cc -Wall -pg programme.c -o programme
```

On l'exécute alors normalement :

```
$ ./programme  
$
```

Un fichier `gmon.out` est alors créé, que l'on examine à l'aide de la commande `gprof` :

```
$ gprof programme gmon.out | less
```

L'utilitaire `gprof` étant assez bavard, il est conseillé de rediriger sa sortie standard vers un programme de pagination comme `more` ou `less`. Les résultats et les statistiques obtenus sont expliqués en clair dans le texte affiché par `gprof`.

Un autre outil de suivi du programme s'appelle `strace`. Il s'agit d'un logiciel permettant de détecter tous les appels système invoqués par un processus. Il observe l'interface entre le processus et le noyau, et mémorise tous les appels, avec leurs arguments. On l'utilise simplement en l'invoquant avec le nom du programme à lancer en argument.

```
$ strace ./programme
```

Les résultats sont présentés sur la sortie d'erreur, (que l'on peut rediriger dans un fichier). Une multitude d'appels système insoupçonnés apparaissent alors, principalement en ce qui concerne les allocations mémoire du processus.

Dans la série des utilitaires permettant d'analyser le code exécutable ou les fichiers objets, il faut également mentionner `nm` qui permet de lister le contenu d'un fichier objet, avec ses différents symboles privés ou externes, les routines, les variables, etc. Pour cela il faut bien entendu que la table des symboles du fichier objet soit disponible. Cette table n'étant plus utile lorsqu'un exécutable est sorti de la phase de débogage, on peut la supprimer en utilisant `strip`. Cet utilitaire permet de diminuer la taille du fichier exécutable (attention à ne pas l'employer sur une bibliothèque partagée !).

Enfin, citons `objdump` qui permet de récupérer beaucoup d'informations en provenance d'un fichier objet, comme son désassemblage, le contenu des variables initialisées, etc.

Traitement du code source

Il existe toute une classe d'outils d'aide au développement qui permettent des interventions sur le fichier source. Ces utilitaires sont aussi variés que l'analyseur de code, les outils de mise en forme ou de statistiques, sans oublier les applications de manipulation de fichiers de texte, qui peuvent parfaitement s'appliquer à des fichiers sources.

Vérificateur de code

L'outil `Lint` est un grand classique de la programmation sous Unix, et son implémentation actuelle sous Linux se nomme `splint` (*Secure Programming Lint*). Le but de cet utilitaire est d'analyser un code source C qui se compile correctement, pour rechercher d'éventuelles erreurs sémantiques dans le programme. L'appel de `splint` peut donc être vu comme une sorte d'extension aux options `-Wall` et `-W` de `gcc`.

L'invocation se fait tout simplement en appelant `splint` suivi du nom du fichier source. On peut bien sûr ajouter des options, permettant de configurer la tolérance de `splint` vis-à-vis des constructions sujettes à caution. Il y a environ 600 options différentes, décrites dans la page d'aide accessible avec « `splint -help flags all` ».

L'invocation de `splint` avec ses options par défaut peut parfois être déprimante. Je ne crois pas qu'il y ait un seul exemple de ce livre qui soit accepté tel quel par `splint` sans déclencher au moins une page d'avertissements. Dans la plupart des cas le problème provient d'ailleurs des bibliothèques système, et il est nécessaire de relâcher la contrainte avec des options ajoutées en ligne de commande. On peut aussi insérer des commentaires spéciaux dans le corps du programme (du type `/*@null@/`) qui indiqueront à `splint` que la construction en question est volontaire, et qu'elle ne doit pas déclencher d'avertissement.

Cet outil est donc très utile pour rechercher tous les points litigieux d'une application. J'ai plutôt tendance à l'employer en fin de développement, pour vérifier un code source avant le passage en phase de test, plutôt que de l'utiliser quotidiennement durant la programmation. Je considère la session de vérification à l'aide de `splint` comme une étape à part entière, à laquelle il faut consacrer du temps, du courage et de la patience, afin d'éliminer dès que possible les bogues éventuels.

Mise en forme

Il existe un outil Unix nommé `indent`, dont une version Gnu est disponible sous Linux. Cet utilitaire est un enjoliveur de code. Ceci signifie qu'il est capable de prendre un fichier source C, et de le remettre en forme automatiquement en fonction de certaines conventions précisées par des options.

On l'utilise souvent pour des projets développés en commun par plusieurs équipes de programmeurs. Avant de valider les modifications apportées à un fichier, et l'insérer dans l'arborescence des sources maîtresses, on invoque `indent` pour le formater suivant les conventions adoptées par l'ensemble des développeurs. De même, lorsqu'un programmeur extrait un fichier pour le modifier, il peut appeler `indent` avec les options qui correspondent à ses préférences.

La documentation de `indent`, décrit une soixantaine d'options différentes, mais trois d'entre-elles sont principalement utiles, `-gnu` qui convertit le fichier aux normes de codage Gnu, `-kr` qui correspond à la présentation utilisée par Kernighan et Ritchie dans leur ouvrage [Kernighan 1994]. Il existe aussi `-orig` pour avoir le comportement de l'utilitaire `indent` original, c'est à dire le style Berkeley. Le programme suivant va être converti dans ces trois formats :

```
hello.c :
#include <stdio.h>

int main (int argc, char * argv [])
{
    int i;
    fprintf (stdout, "Hello world ! ");
    if (argc > 1)
    {
        fprintf (stdout, ": ");
        /* Parcours et affichage des arguments */
        for (i = 1; i < argc; i++) fprintf (stdout, "%s ", argv [i]);
    }
    fprintf (stdout, "\n");
    return (0);
}
```

Nous demandons une mise en forme dans le style Gnu :

```
$ indent -gnu hello.c -o hello.2.c
$ cat hello.2.c
#include <stdio.h>

int
main (int argc, char *argv[])
{
    int i;
    fprintf (stdout, "Hello world ! ");
    if (argc > 1)
    {
        fprintf (stdout, ": ");
        /* Parcours et affichage des arguments */
        for (i = 1; i < argc; i++)
            fprintf (stdout, "%s ", argv[i]);
    }
    fprintf (stdout, "\n");
    return (0);
}
$
```

Voyons la conversion en style Kernighan et Ritchie :

```
$ indent -kr hello.c -o hello.3.c
$ cat hello.3.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    fprintf(stdout, "Hello world ! ");
    if (argc > 1) {
        fprintf(stdout, ": ");
        /* Parcours et affichage des arguments */
        for (i = 1; i < argc; i++)
            fprintf(stdout, "%s ", argv[i]);
    }
    fprintf(stdout, "\n");
    return (0);
}
$
```

Et finalement le style Berkeley original :

```
$ indent -orig hello.c -o hello.4.c
$ cat hello.4.c
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int            i;
    fprintf(stdout, "Hello world ! ");
    if (argc > 1) {
        fprintf(stdout, ": ");
        /*
         * Parcours et affichage des arguments
         */
        for (i = 1; i < argc; i++)
            fprintf(stdout, "%s ", argv[i]);
    }
    fprintf(stdout, "\n");
    return (0);
}
$
```

Chaque programmeur peut ainsi utiliser ses propres habitudes de mise en forme, indépendamment des autres membres de son équipe.

Utilitaires divers

L'outil `grep` est essentiel pour un programmeur, car il permet de rechercher une chaîne de caractères dans un ensemble de fichiers. Il est fréquent d'avoir à retrouver le fichier où une routine est définie, ou l'emplacement de la déclaration d'une structure par exemple. De même, on a souvent besoin de rechercher à quel endroit un programme affiche un message d'erreur avant de s'arrêter. Pour toutes ces utilisations `grep` est parfaitement adapté. Sa page de manuel documente ces nombreuses fonctionnalités, et l'emploi des expressions régulières pour préciser le motif à rechercher.

Lorsque l'on désire retrouver une chaîne de caractères dans toute une arborescence, il faut le coupler à l'utilitaire `find`, en employant la commande `xargs` pour les relier. Voici à titre d'exemple la recherche d'une constante symbolique (`ICMPV6_ECHO_REQUEST` en l'occurrence) dans tous les fichiers source du noyau Linux :

```
$ cd /usr/src/linux
$ find . -type f | xargs grep ICMPV6_ECHO_REQUEST
./net/ipv6/icmp.c: else if (type >= ICMPV6_ECHO_REQUEST &&
./net/ipv6/icmp.c: (&icmpv6_statistics.Icmp6InEchos)[type-
ICMPV6_ECHO_REQUEST]++;
./net/ipv6/icmp.c:         case ICMPV6_ECHO_REQUEST:
./include/linux/icmpv6.h:#define ICMPV6_ECHO_REQUEST      128
$
```

La commande `find` recherche tous les fichiers réguliers (`-type f`) de manière récursive à partir du répertoire en cours (`.`), et envoie les résultats à `xargs`. Cet utilitaire les regroupe en une liste d'arguments qu'il transmet à `grep` pour y rechercher la chaîne demandée.

L'importance de `grep` pour un développeur est telle que les éditeurs de texte contiennent souvent un appel direct à cet utilitaire depuis une option de menu.

Lorsqu'on développe un projet sur plusieurs machines simultanément, on est souvent amené à vérifier si un fichier a été modifié et, si c'est le cas, dans quelle mesure. Ceci peut être obtenu à l'aide de l'utilitaire `diff`. Il compare intelligemment deux fichiers et indique les portions modifiées entre les deux. Cet instrument est très utile lorsqu'on reprend un projet après quelque temps et qu'on ne se rappelle plus quelle version est la bonne.

Par exemple, nous pouvons comparer les programmes `hello.3.c` (version Kernighan et Ritchie) et `hello.4.c` (version Berkeley) pour trouver leurs différences :

```
$ diff hello.3.c hello.4.c
3c3,4
< int main(int argc, char *argv[])
---
> int
> main(int argc, char *argv[])
5c6
<     int i;
---
>     int             i;
9c10,12
<     /* Parcours et affichage des arguments */
---
>     /*
>      * Parcours et affichage des arguments
>      */
$
```


Ici, `diff` nous indique une différence à la ligne 3 du premier fichier, qui se transforme en lignes 3 et 4 du second, puis une seconde variation à la ligne 5 de l'un et 6 de l'autre, ainsi qu'une dernière différence à la ligne 9, qui se transforme en 10, 11 et 12 de l'autre. On le voit, la comparaison est intelligente, `diff` essayant de se resynchroniser le plus vite possible lorsqu'il rencontre une différence. Toutefois, lorsque l'envergure d'une application augmente et que le nombre de développeurs s'accroît, il est préférable d'employer un système de contrôle de version comme `cv`s.

L'outil `diff` est aussi très utilisé dans le monde du logiciel libre et de Linux en particulier, pour créer des fichiers de différences qu'on transmet ensuite à l'utilitaire `patch`. Ces fichiers sont beaucoup moins volumineux que les fichiers source complets.

Construction d'application

Dès qu'une application s'appuie sur plusieurs modules indépendants – plusieurs fichiers source C –, il est indispensable d'envisager d'utiliser les mécanismes de compilation séparée. Ainsi, chaque fichier C est compilé en fichier objet `.o` indépendamment des autres modules (grâce à l'option `-c` de `gcc`), et finalement on regroupe tous les fichiers objet ensemble lors de l'édition des liens (assurée également par `gcc`).

L'avantage de ce système réside dans le fait qu'une modification apportée à un fichier source ne réclame plus qu'une seule compilation et une édition des liens au lieu de nécessiter la compilation de tous les modules du projet. Ceci est déjà très appréciable en langage C, mais devient réellement indispensable en C++, où les phases de compilation sont très longues, notamment à cause du volume des fichiers d'en-tête.

Pour ne pas être obligé de recompiler un programme source non modifié, on fait appel à l'utilitaire `make`. Celui-ci compare les dates de modification des fichiers source et cibles pour évaluer les tâches à réaliser. Il est aidé en cela par un fichier de configuration nommé « `Makefile` » (ou `makefile`, voire `GNUmakefile`), qu'on conserve dans le même répertoire que les fichiers source. Ce fichier est constitué par une série de règles du type :

```
cible : dépendances
      commandes
```

La cible indique le but désiré, par exemple le nom du fichier exécutable. Les dépendances mentionnent tous les fichiers dont la règle a besoin pour s'exécuter, et les commandes précisent comment obtenir la cible à partir des dépendances. Par exemple, on peut avoir :

```
mon_programme : interface.o calcul.o centre.o
      cc -o mon_programme interface.o c calcul.o centre.o
```

Lorsque make est appelé, il vérifie l'heure de modification de la cible et celle des dépendances, et peut ainsi décider de refaire l'édition des liens. Si un fichier de dépendance est absent, make recherchera une règle pour le créer, par exemple :

```
interface.o : interface.c interface.h commun.h  
cc -Wall -c interface.c
```

Ce système est à première vue plutôt simple, mais la syntaxe même des fichiers Makefile est assez pénible, car il suffit d'insérer un espace en début de ligne de commande, à la place d'une tabulation, pour que make refuse le fichier. Par ailleurs, il existe un certain nombre de règles implicites que make connaît, par exemple comment obtenir un fichier .o à partir d'un .c. Pour obtenir des détails sur les fichiers Makefile, on consultera donc la documentation Gnu.

Comme la création d'un Makefile peut être laborieuse, on emploie parfois des utilitaires supplémentaires, imake ou xmkmf, qui utilisent un fichier Imakefile pour créer le ou les fichiers Makefile de l'arborescence des sources. La syntaxe des fichiers Imakefile est décrite dans la page de manuel de imake.

Notons que les environnements de développement intégrés comme Eclipse savent généralement créer automatiquement les fichiers Makefile des projets que l'on construit avec eux. En outre, Eclipse permet à l'utilisateur de gérer lui-même son propre Makefile dans le cas de dépendances complexes entre projets.

Une autre possibilité pour créer automatiquement les fichiers Makefile adaptés lors d'un portage de logiciel est d'utiliser les outils Gnuautomake et autoconf (voir à ce sujet la documentation info automake).

Distribution du logiciel

La distribution d'un logiciel sous Linux peut se faire de plusieurs manières. Tout dépend d'abord du contenu à diffuser. S'il s'agit d'un logiciel libre, le plus important est de fournir les sources du programme ainsi que la documentation dans un format le plus portable possible sur d'autres Unix. Le point le plus important ici sera de laisser l'entière liberté au destinataire pour choisir l'endroit où il placera les fichiers sur son système, l'emplacement des données de configuration, etc. On pourra consulter le document *Linux Software-Release-Practice-HOWTO*, qui contient de nombreux conseils pour la distribution de logiciels libres.

S'il s'agit de la distribution d'une application commerciale fournie uniquement sous forme binaire, le souci majeur sera plutôt de simplifier l'installation du produit, quitte à imposer certaines restrictions concernant les emplacements de l'application et des fichiers de configuration.

Pour simplifier l'installation du logiciel, il est possible de créer un script qui se charge de toute la mise en place des fichiers. Toutefois ce script devra être lancé depuis un support de distribution (CD ou clé USB), ce qui peut nécessiter une intervention manuelle de l'administrateur pour autoriser l'exécution des programmes sur un support extractible ou une copie du script dans le répertoire de l'utilisateur avant le lancement. Il est donc souvent plus simple de fournir une simple archive tar ou un paquetage rpm, et de laisser l'utilisateur les décompacter lui-même.

Archive classique

L'utilitaire tar (*Tape Archiver*) est employé dans le monde Unix depuis longtemps pour regrouper plusieurs fichiers en un seul paquet. À l'origine, cet outil servait surtout à copier le contenu d'un répertoire sur une bande de sauvegarde. De nos jours, on l'utilise pour créer une archive – un gros fichier – regroupant tout le contenu d'une arborescence de fichiers source.

Les conventions veulent que la distribution de l'arborescence des sources d'un projet se fasse en incluant son répertoire de départ. Par exemple si une application est développée dans le répertoire `~/src/mon_appli/` et ses sous-répertoires, il faudra que l'archive soit organisée pour qu'en la décompactant l'utilisateur se trouve avec un répertoire `mon_appli/` et ses descendants. Pour créer une telle archive, on procède ainsi :

```
$ cd ~/src
$ tar -cf mon_appli.tar mon_appli/
```

Le fichier `mon_appli.tar` contient alors toute l'archive. Pour le décompresser, on peut effectuer :

```
$ cp mon_appli.tar ~/tmp
$ cd ~/tmp
$ tar -xf mon_appli.tar
$ ls
mon_appli.tar
mon_appli/
$
```

La commande « c » de tar sert à créer une archive, alors que « x » sert à extraire son contenu. Le « f » précise que l'archive est un fichier dont le nom est indiqué à la suite (et pas l'entrée ou la sortie standard). On peut aussi ajouter la commande « z », pour indiquer que l'archive doit être (dé)compressée en invoquant gzip, ou « j » pour la (dé)compresser avec bzip2.

Lorsqu'on désire fournir un fichier d'installation regroupant un exécutable, à placer par exemple dans `/usr/local/bin`, et des données se trouvant dans `/usr/local/lib/...`, ainsi qu'un fichier d'initialisation globale dans `/etc`, l'emploi de `tar` est toujours possible mais moins commode. Dans ce cas, il faut créer l'archive à partir de la racine du système de fichiers en indiquant uniquement les fichiers à incorporer. L'extraction sur le système de l'utilisateur devra aussi être réalisée à partir de la racine du système de fichiers (par `root`).

Dans ces conditions, les paquetages `rpm` représentent une bonne alternative.

Paquetage à la manière Red Hat

L'utilitaire `rpm` (*Red Hat Package Manager*) n'est pas du tout limité à cette distribution. Les paquetages `.rpm` sont en réalité supportés plus ou moins directement par l'essentiel des grandes distributions Linux actuelles.

Le principe de ces paquetages est d'incorporer non seulement les fichiers, mais aussi des informations sur les options de compilation, les dépendances par rapport à d'autres éléments du système (bibliothèques, utilitaires...), ainsi que la documentation des logiciels. Ces paquetages permettent naturellement d'intégrer au besoin le code source de l'application.

La création d'un paquetage nécessite un peu plus d'attention que l'utilisation de `tar`, car il faut passer par un fichier intermédiaire de spécifications. En revanche, l'utilisation au niveau de l'administrateur qui installe le produit est très simple. Il a facilement accès à de nombreuses possibilités, en voici quelques exemples :

- Installation ou mise à jour d'un nouveau paquetage :

```
$ rpm -U paquet.rpm
```

- Mise à jour uniquement si une ancienne version était déjà installée :

```
$ rpm -F paquet.rpm
```

- Suppression d'un paquetage :

```
$ rpm -e paquet
```

- Recherche du paquetage contenant un fichier donné :

```
$ rpm -qf /usr/local/bin/fichier
```

- Liste de tous les paquets installés et recherche de ceux qui ont un nom donné :

```
$ rpm -qa | grep nom
```

La page de manuel de rpm est assez complète, et il existe de surcroît un document *RPM-HOWTO* aidant à la création de paquetages.

Bibliothèques supplémentaires pour le développement

En fait, la bibliothèque C seule ne permet pas de construire d'application très évoluée, ou alors au prix d'un effort de codage démesuré et peu portable. Les limitations de l'interface utilisateur nous empêchent de dépasser le stade des utilitaires du type « filtre » qu'on rencontre sous Unix (tr, grep, wc...). Pour aller plus loin dans l'ergonomie d'une application, il est indispensable de recourir aux services de bibliothèques supplémentaires.

Celles-ci se présentent sous forme de logiciels libres, disponibles sur la majorité des systèmes Linux.

Interface utilisateur en mode texte

La première interface disponible pour améliorer l'ergonomie d'un programme en mode texte est la bibliothèque Gnu Readline, conçue pour faciliter la saisie de texte. Lorsqu'un programme fait appel aux routines de cette bibliothèque, l'utilisateur peut corriger facilement la ligne de saisie, en se déplaçant en arrière ou en avant, en modifiant les caractères déjà entrés, en utilisant même des possibilités de complétion du texte ou d'historique des lignes saisies.

Il est possible de configurer les touches associées à chaque action par l'intermédiaire d'un fichier d'initialisation, qui peut même accepter des directives conditionnelles en fonction du type de terminal sur lequel l'utilisateur se trouve. La bibliothèque Readline est par exemple employée par le shell *Bash*.

Pour l'affichage des résultats d'un programme en mode texte, il est conseillé d'employer la bibliothèque ncurses. Il s'agit d'un ensemble de fonctions permettant d'accéder de manière portable aux diverses fonctionnalités qu'on peut attendre d'un écran de texte, comme le positionnement du curseur, l'accès aux couleurs, les manipulations de fenêtres, de panneaux, de menus...

La bibliothèque ncurses disponible sous Linux est libre et compatible avec la bibliothèque curses, décrite par les spécifications SUSv4, présente sur l'essentiel des Unix commerciaux.

Non seulement ncurses nous fournit des fonctionnalités gérant tous les types de terminaux de manière transparente, mais en plus la portabilité du programme sur d'autres environnements Unix est assurée. On comprendra que de nombreuses applications y fassent appel.

Développement sous X-Window

La programmation d'applications graphiques sous X-Window peut parfois devenir un véritable défi, en fonction de la portabilité désirée pour le logiciel.

Le développement sous X-Window est organisé en couches logicielles successives. Au bas de l'ensemble se trouve la bibliothèque Xlib. Cette bibliothèque offre les fonctionnalités élémentaires en termes de dessin (tracé de polygones, de cercles, de texte, etc.), de fenêtrage et de récupération d'événements produits par la souris ou le clavier. La notion de fenêtrage est ici réduite à sa plus simple expression, puisqu'il s'agit uniquement de zones rectangulaires sur l'écran, sans matérialisation visible (pas de bordure).

L'appel des fonctions de la Xlib est indispensable dès qu'on utilise des primitives graphiques de dessin. En revanche, si on veut disposer ne serait-ce que d'un bouton à cliquer, il faut le dessiner entièrement avec ses contours, son texte, éventuellement la couleur de fond et les ombrages. Naturellement, une bibliothèque prend en charge ce travail et offre des composants graphiques élémentaires (les *widgets*).

Les fonctionnalités proposées par la couche nommée « Xt » ne sont toujours pas suffisantes, car celle-ci ne fait que définir des classes génériques d'objets graphiques et n'en offre pas d'implémentation esthétique.

Pour obtenir une bonne interface graphique, il faut donc utiliser une couche supplémentaire. Le standard le plus employé dans le domaine industriel est la bibliothèque Qt.

Les environnements KDE et Gnome

Les deux environnements homogènes les plus répandus sous Linux sont KDE (*K Desktop Environment*) et Gnome (*Gnu Network Model Environment*). L'un comme l'autre possèdent une interface de programmation très évoluée, rendant plus facile le développement de logiciels graphiques.

Ces environnements sont parfaitement appropriés pour la mise en œuvre de logiciels – libres ou commerciaux – pour Linux. Toutefois la portabilité vers d'autres Unix est sensiblement amoindrie.

L'environnement Gnome est construit autour de la bibliothèque graphique GTK (*Gimp Toolkit*), initialement développée, comme son nom l'indique, pour l'utilitaire graphique Gimp. La programmation sous Kde repose sur la bibliothèque Qt. Il existe de nombreux documents sur la programmation sous KDE ou sous Gnome sur le Web.

Conclusion

Ce chapitre nous aura permis de faire le point sur les outils disponibles pour le développeur dans l'environnement Linux/Gnu.

De nombreux livres, magazines et sites web décrivent l'installation et l'utilisation d'une station Linux, et il existe une aide en ligne très riche pour la mise en œuvre des outils de développement GNU. Il est important de se familiariser avec l'environnement de programmation et de savoir naviguer avec aisance entre les utilitaires disponibles.

2

Les processus

L'un des premiers choix de conception que devra faire le développeur d'une application multitâche sera : « processus ou *threads* ? ». Doit-il faire reposer son système sur un ensemble de processus distincts, chacun étant totalement indépendant des autres, ou sur des threads s'exécutant dans un espace mémoire commun ?

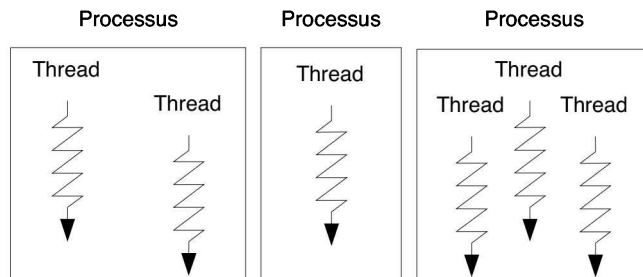
Chacune de ces approches présente des avantages et des inconvénients. Nous allons commencer par nous intéresser à la première d'entre elles. Ce chapitre introduira la notion de processus, ainsi que les différents identifiants qui y sont associés, leurs significations et leurs utilisations dans le système.

Le fonctionnement multitâche classique sous Unix repose sur la notion de processus, ceci depuis le début des années soixante-dix. Les threads ne sont apparus que beaucoup plus récemment, et se sont généralisés depuis la fin des années quatre-vingt-dix. Avec Linux, les threads n'ont été disponibles de manière efficace que depuis 1999 (version 2.2 du noyau).

Principe des processus

Les processus s'exécutent dans des espaces mémoire distincts. Ceci est très important : chaque processus dispose d'une zone de mémoire totalement indépendante et protégée des autres processus. Et au sein de l'espace virtuel de mémoire attribué à un processus peuvent s'exécuter un ou plusieurs threads.

Figure 2-1
Processus et threads



La figure 2-1 nous présente un système dans lequel trois processus cohabitent, dans trois espaces virtuels de mémoires distincts. Dans l'espace mémoire du processus de gauche, deux threads s'exécutent simultanément, partageant ainsi leurs données. Le processus central ne comporte qu'un seul thread, c'est le cas de la plupart des processus sous Linux. Enfin le processus de droite englobe trois threads concurrents.

Le fonctionnement de la mémoire, telle qu'elle est gérée par le noyau, sera abordé plus en détail dans le chapitre 14. Indiquons rapidement qu'un composant spécial (la MMU « *Memory Management Unit* »), intégré dans les processeurs modernes permet de présenter une vue abstraite de la mémoire en ne rendant accessibles que les portions explicitement projetées par le noyau. Ainsi lorsqu'un processus s'exécute, il ne peut atteindre que les zones de mémoire qui lui appartiennent. Il n'a aucun moyen d'accéder aux zones réservées aux autres processus ; et inversement, il sait qu'aucun autre processus ne peut venir modifier sa mémoire à son insu. Nous aurons l'occasion d'expérimenter cela un peu plus loin.

À un instant donné, une tâche peut, comme nous le verrons plus loin, se trouver dans divers états. Le noyau du système d'exploitation est chargé de réguler leur exécution afin de garantir à l'utilisateur un comportement multitâche performant. Le noyau fournit un mécanisme de régulation des tâches qu'on nomme « ordonnancement » (en anglais *scheduling*). Cela assure la répartition équitable de l'accès au microprocesseur par les divers processus et threads concurrents.

Sur une machine uni-processeur (et uni-cœur), il n'y a qu'une seule tâche qui s'exécute effectivement à un instant donné. Le noyau assure une commutation régulière entre tous les processus présents sur le système pour garantir un fonctionnement multitâche. Sur une machine multiprocesseur, le principe est le même, à la différence que plusieurs tâches – mais rarement toutes – peuvent s'exécuter réellement en parallèle.

On peut imaginer un processus comme un programme en cours d'exécution. Cette représentation est très imparfaite car une application peut non seulement utiliser plusieurs processus concurrents, mais un unique processus peut également lancer l'exé-

cution d'un nouveau programme, en remplaçant entièrement le code et les données du programme précédent.

On peut examiner la liste des processus présents sur le système à l'aide de la commande `ps`, et plus particulièrement avec ses options `ax`, qui nous permettent de voir les processus endormis, et ceux qui appartiennent aux autres utilisateurs. On voit alors, même sur un système apparemment au repos, une bonne cinquantaine de processus plus ou moins actifs :

```
$ ps ax
 1 ?      Ss      0:03 init [5]
 2 ?      S<      0:00 [kthreadd]0,18cm
 3 ?      S<      0:02 [migration/0]
 4 ?      S<      0:02 [ksoftirqd/0]
 5 ?      S<      0:00 [watchdog/0]
 6 ?      S<      0:02 [migration/1]
 7 ?      S<      0:00 [ksoftirqd/1]
 8 ?      S<      0:00 [watchdog/1]
[...]
2117 ?    Ss      0:00 crond
2131 ?    Ss      0:00 /usr/sbin/atd
2153 ?    Ss      0:00 avahi-daemon: running [linux.local]
2154 ?    Ss      0:00 avahi-daemon: chroot helper
2171 ?    Ss      1:17 cupsd
[...]
2337 tty1 Ss+      0:00 /sbin/mingetty tty1
2338 tty2 Ss+      0:00 /sbin/mingetty tty2
2339 tty3 Ss+      0:00 /sbin/mingetty tty3
2340 tty4 Ss+      0:00 /sbin/mingetty tty4
[...]
3904 ?    Ss      0:00 kdeinit Running...
3907 ?    S       0:00 dcopserver --nosid
3909 ?    S       0:00 klauncher [kdeinit] --new-startup
3911 ?    S       4:51 kded --new-startup
[...]
6789 ?    R       0:00 konsole
6790 pts/1 Ss      0:00 /bin/bash
6825 pts/1 R+     0:00 ps ax
$
```

La commande `ps` affiche plusieurs colonnes dont la signification ne nous importe pas pour le moment. Retenons simplement que nous voyons en dernière colonne l'intitulé complet de la commande qui a démarré le processus, et en première colonne un numéro d'identification qu'on nomme *PID*.

Identification par le PID

Le premier processus du système (nommé `init`) ainsi que quelques autres sont créés directement par le noyau au démarrage. La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel système `fork()`, qui va dupliquer le processus appelant. Au retour de cet appel système, deux processus identiques continueront d'exécuter le code à la suite de `fork()`. La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original, qu'on nomme traditionnellement le processus *père*, et la nouvelle copie, le processus *fil*s.

L'appel système `fork()` est déclaré dans `<unistd.h>`, ainsi :

```
| pid_t fork (void);
```

En Python, cet appel système est proposé par le module `os` comme nous le verrons plus loin.

Les deux processus pouvant être distingués par leur numéro d'identification PID (*Process IDentifier*), il est possible d'exécuter deux codes différents au retour de l'appel système `fork()`. Par exemple, le processus fils peut demander à charger le code d'un autre programme exécutable se trouvant sur le disque. C'est exactement ce que fait un *shell* habituellement.

Pour connaître son propre identifiant PID, on utilise l'appel système `getpid()`, qui ne prend pas d'argument et renvoie une valeur de type `pid_t`. Il s'agit, bien entendu, du PID du processus appelant. Cet appel système, déclaré dans `<unistd.h>` (ou dans le module `os` de Python), est l'un des rares qui n'échouent jamais :

```
| pid_t getpid (void);
```

Ce numéro de PID est celui que nous avons vu affiché en première colonne de la commande `ps`. Sous Linux, le type `pid_t` est un entier sur 32 bits, mais ce n'est pas le cas pour tous les Unix. Pour assurer une bonne portabilité lors de l'affichage d'un PID, nous utiliserons la conversion `%ld` de `printf()`, et nous ferons explicitement une conversion de type en `long int` ainsi :

```
| fprintf(stdout, "Mon PID est : %ld\n", (long) getpid());
```

La distinction entre processus père et fils peut se faire directement au retour de l'appel `fork()`. Celui-ci, en effet, renvoie une valeur de type `pid_t`, qui vaut zéro si on se trouve dans le processus fils, est négative en cas d'erreur, et correspond au PID du fils si on se trouve dans le processus père.

Voici en effet un point important : dans la plupart des applications courantes, la création d'un processus fils a pour but de faire dialoguer deux parties indépendantes du programme (à l'aide de signaux, de tubes, de mémoire partagée...). Le processus fils peut aisément accéder au PID de son père (noté *PPID* pour *Parent PID*) grâce à l'appel système `getppid()`, déclaré dans `<unistd.h>` :

```
| pid_t getppid (void);
```

Cette routine se comporte comme `getpid()`, mais renvoie le PID du père du processus appelant. En revanche, le processus père ne peut connaître le numéro du nouveau processus créé qu'au moment du retour du `fork()`.

En réalité, un processus pourrait établir la liste de ses fils en analysant le PPID de tous les processus en cours d'exécution, par exemple, à l'aide du pseudo-système de fichiers `/proc`, mais il est quand même beaucoup plus simple de mémoriser la valeur de retour de `fork()`.

On peut examiner la hiérarchie des processus en cours sur le système avec le champ PPID de la commande `ps axj` :

```
$ ps axj
PPID  PID PGID  SID TTY TPGID STAT UID  TIME COMMAND
    0    1    1    1 ?      -1 Ss   0   0:03 init [5]
    0    2    0    0 ?      -1 S<   0   0:00 [kthreadd]
    2    3    0    0 ?      -1 S<   0   0:02 [migration/0]
    2    4    0    0 ?      -1 S<   0   0:02 [ksoftirqd/0]
[...]
    1 2117 2117 2117 ?      -1 Ss   0   0:00 crond
    1 2131 2131 2131 ?      -1 Ss   0   0:00 /usr/sbin/atd
    1 2171 2171 2171 ?      -1 Ss   0   1:17 cupsd
    1 2180 2180 2180 ?      -1 Ss  68   2:39 hald
[...]
    1 3904 3904 3904 ?      -1 Ss 500   0:00 kdeinit Running...
    1 3907 3904 3904 ?      -1 S   500   0:00 dcopserver --nosid
[...]
 3904 6789 3904 3904 ?      -1 R   500   0:00 konsole
 6789 6790 6790 6790 pts/1 7533 Ss 500   0:00 /bin/bash
 6790 7533 7533 6790 pts/1 7533 R+ 500   0:00 ps axj
$
```

On voit que `init` n'a pas de père (`PPID = 0`), mais qu'un grand nombre de processus héritent de lui. On peut observer également une filiation directe `kdeinit (3904) – .konsole (6789) – bash (6790) – ps (7533)`.

Lorsqu'un processus est créé par `fork()`, il dispose d'une copie des données de son père, mais également de l'environnement de celui-ci et d'un certain nombre d'autres éléments (table des descripteurs de fichiers, etc.). On parle alors d'*héritage* du père.

Notons que, sous Linux, l'appel système `fork()` est très économe car il utilise une méthode de « copie sur écriture » (*copy-on-write*). Cela signifie que toutes les données qui doivent être dupliquées pour chaque processus (descripteurs de fichier, mémoire allouée...) ne seront pas immédiatement recopiées. Tant qu'aucun des deux processus n'a modifié des informations dans ces pages mémoire, il n'y en a qu'un seul exemplaire sur le système. Par contre, dès que l'un des processus réalise une écriture dans la zone concernée, le noyau assure la véritable duplication des données. Une création de processus par `fork()` n'a donc qu'un coût faible en termes de ressources système.

En cas d'erreur, `fork()` renvoie la valeur -1, et la variable globale `errno` contient le code d'erreur, défini dans `<errno.h>`. Ce code d'erreur peut être soit `ENOMEM`, qui indique que le noyau n'a plus assez de mémoire disponible pour créer un nouveau processus, soit `EAGAIN`, qui signale que le système n'a plus de place libre dans sa table des processus, mais qu'il y en aura probablement sous peu. Un processus est donc autorisé à réitérer sa demande de duplication lorsqu'il a obtenu un code d'erreur `EAGAIN`.

Voici à présent un exemple de création d'un processus fils par l'appel système `fork()`.

```
exemple-fork.c :
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/wait.h>

int main (void)
{
    pid_t pid_fils;

    do {
        pid_fils = fork();
    } while ((pid_fils == -1) && (errno == EAGAIN));

    if (pid_fils == -1) {
        fprintf(stderr, "fork() impossible, errno=%d\n", errno);
        return 1;
    }
    if (pid_fils == 0) { /* Processus fils */
        printf(stdout, "Fils : PID=%ld, PPID=%ld\n",
            (long)getpid(), (long)getppid());
        return 0;
    } else { /* Processus père */
```

```
        fprintf(stdout, "Pere : PID=%ld, PPID=%ld, PID fils=%ld\n",  
                  (long)getpid(), (long)getppid(), (long)pid_fils);  
        wait(NULL);  
        return 0;  
    }  
}
```

Lors de son exécution, ce programme fournit le résultat suivant :

```
$ echo $$  
6790  
$ ./exemple-fork  
Fils : PID=8234, PPID=8233  
Pere : PID=8233, PPID=6790, PID fils=8234  
$
```

La variable spéciale \$\$ du shell correspond à son PID. Ceci nous permet de voir que le PPID du processus père correspond au shell.

Dans notre exemple, l'appel système `fork()` boucle si le noyau n'a plus assez de place dans sa table interne pour créer un nouveau processus. Dans ce cas, le système est déjà probablement dans une situation assez critique, et il n'est pas utile de gâcher des ressources CPU en effectuant une boucle hystérique sur `fork()`. Il serait préférable d'introduire un délai d'attente dans notre code pour ne pas réitérer notre demande immédiatement, et attendre ainsi pendant quelques secondes que le système revienne dans un état plus calme. Nous verrons des moyens d'endormir le processus dans le chapitre 9.

Comme nous l'avons précisé plus haut, l'appel `fork()` existe également en Python, par l'intermédiaire du module `os`, ainsi que `getpid()` et `getppid()`. En cas d'erreurs dans `fork()`, l'exception `OSError` est levée.

Voici un exemple qui réalise le même travail que le code précédent :

```
exemple-fork.py :  
#!/usr/bin/python  
from __future__ import print_function  
import os  
  
try:  
    child_pid = os.fork()  
except OSError:  
    print("Error in fork()")  
    sys.exit(-1)
```

```
if (child_pid == 0):  
    print("Child: PID =", os.getpid(),  
          " PPID =", os.getppid())  
else:  
    print("Parent: PID =", os.getpid(),  
          " PPID =", os.getppid(),  
          " Child =", child_pid)  
os.wait()
```

Le résultat donne :

```
$ ./exemple-fork.py  
Parent: PID = 8617   PPID = 6790   Child = 8618  
Child: PID = 8618   PPID = 8617
```

On remarquera que nous avons introduit un appel système `wait(NULL)` à la fin du code du père (`os.wait` avec Python). Nous en reparlerons ultérieurement, mais on peut d'ores et déjà noter que cela permet d'attendre la fin de l'exécution du fils. Si nous n'avions pas employé cet appel système, le processus père aurait pu se terminer avant son fils, redonnant la main au shell qui aurait alors affiché son symbole d'invite (\$) avant que le fils n'ait imprimé ses informations. Voici ce qu'on aurait pu observer :

```
$ ./exemple-fork  
Pere : PID=4344, PPID=4284, PID fils=4345  
$ Fils : PID=4345, PPID=4344
```

Lors d'un `fork()`, le processus père reprend son exécution avant son fils. Ceci est configurable grâce au pseudo-fichier `sched_child_runs_first` présent dans `/proc/sys/kernel/`. Il contient par défaut la valeur zéro, indiquant que le père redémarre avant son fils. En écrivant 1 dans ce fichier, on inverse le comportement.

Identification de l'utilisateur correspondant au processus

À l'opposé des systèmes mono-utilisateurs, un système Unix est particulièrement orienté vers l'identification de ses utilisateurs. Toute activité entreprise par un utilisateur est soumise à des contrôles stricts quant aux permissions qui lui sont attribuées. Pour cela, chaque processus s'exécute sous une identité précise. Dans la plupart des cas, il s'agit de l'identité de l'utilisateur qui a invoqué le processus et qui est définie par une valeur numérique : l'*UID* (*User Identifier*). Dans certaines situations que nous examinerons plus bas, il est nécessaire pour le processus de changer d'identité.

La commande « `id` » du shell affiche les identifiants et les groupes auxquels appartient l'utilisateur.

Il existe trois identifiants d'utilisateur par processus : l'*UID réel*, l'*UID effectif*, et l'*UID sauvé*. L'*UID réel* est celui de l'utilisateur ayant lancé le programme. L'*UID effectif* est celui qui correspond aux privilèges accordés au processus. L'*UID sauvé* est une copie de l'ancien UID effectif lorsque celui-ci est modifié par le processus.

L'essentiel des ressources sous Unix (données, périphériques...) s'exprime sous forme de nœuds du système de fichiers. Lors d'une tentative d'accès à un fichier, le noyau effectue des vérifications d'autorisation en prenant en compte l'UID effectif du processus appelant. Généralement, cet UID effectif est le même que l'UID réel (celui de la personne ayant invoqué le processus). C'est le cas de toutes les applications classiques ne nécessitant pas de privilège particulier, par exemple les commandes Unix classiques (`ls`, `cp`, `mv`...) qui s'exécutent sous l'identité de leur utilisateur, laissant au noyau le soin de vérifier les permissions d'accès.

Certaines applications peuvent toutefois avoir besoin – souvent ponctuellement – d'autorisations spéciales, tout en étant invoquées par n'importe quel utilisateur. L'exemple le plus évident est `su`, qui permet de changer d'identité, mais on peut en citer beaucoup d'autres, comme `mount`, qui peut autoriser sous Linux tout utilisateur à monter des systèmes de fichiers provenant d'un CD-Rom ou d'une clé USB, par exemple. Il y a également les applications utilisant des couches basses des protocoles réseau comme `ping`. Dans ce cas, il faut que le processus garde son UID réel pour savoir qui agit, mais il dispose d'un UID effectif lui garantissant une liberté suffisante sur le système pour accéder aux ressources désirées.

Les appels système `getuid()` et `geteuid()` permettent respectivement d'obtenir l'UID réel et l'UID effectif du processus appelant. Ils sont déclarés dans `<unistd.h>`, ainsi :

```
uid_t getuid(void);  
uid_t geteuid(void);
```

Le type `uid_t` correspondant au retour des fonctions `getuid()` et `geteuid()` est défini dans `<sys/types.h>`. Il s'agit d'un entier non signé de 32 bits. Nous utiliserons la conversion `%u` pour `printf()`, pour garantir une bonne portabilité.

L'UID effectif est différent de l'UID réel lorsque le fichier exécutable dispose d'un attribut particulier permettant au processus de changer d'identité au démarrage du programme. Considérons par exemple le programme suivant.

```
exemple-getuid.c :  
#include <stdio.h>  
#include <unistd.h>
```

```
int main(void)
{
    fprintf(stdout, " UID reel = %u, UID effectif = %u\n",
              getuid(), geteuid());
    return 0;
}
```

Quand on compile ce programme, on obtient un fichier exécutable, qu'on lance ensuite :

```
$ ls -ln exemple-getuid*
-rwxrwxr-x 1 500 500 7634 mai 26 10:23 exemple-getuid
-rw-rw-r-- 1 500 500 162 mai 26 10:23 exemple-getuid.c
$ ./exemple-getuid
  UID reel = 500, UID effectif = 500
$
```

Le comportement est pour l'instant parfaitement normal. Imaginons maintenant que l'administrateur root passe par là, s'attribue le fichier exécutable et lui ajoute le bit « *Set-UID* » à l'aide de la commande `chmod`. Lorsqu'un utilisateur va maintenant exécuter `exemple-getuid`, le système va lui fournir l'UID effectif du propriétaire du fichier, à savoir root (qui a toujours l'UID0 par définition) :

```
$ su
Mot de passe :
# chown root.root exemple-getuid
# chmod u+s exemple-getuid
# ls -ln exemple-getuid*
-rwsrwxr-x 1 0 0 7634 mai 26 10:23 exemple-getuid
-rw-rw-r-- 1 500 500 162 mai 26 10:23 exemple-getuid.c
# exit
exit
$ ./exemple-getuid
  UID reel = 500, UID effectif = 0
$
```

Nous voyons l'attribut *Set-UID* indiqué par la lettre « s » dans les autorisations d'accès. L'UID réel est conservé à des fins d'identification éventuelle au sein du processus.

Notre processus ayant l'UID effectif de root en a tous les privilèges. Vous pouvez en avoir le cœur net en lui faisant, par exemple, créer un nouveau fichier dans le répertoire `/etc`. Si vous n'avez pas les privilèges root sur votre système, vous pouvez néanmoins effectuer les tests en accord avec un autre utilisateur qui copiera votre exécutable dans son répertoire personnel (pour en prendre possession) et lui ajoutera le bit *Set-UID*.

Il existe plusieurs appels système permettant à un processus de modifier son UID. Il ne peut toutefois s'agir que de perdre des privilèges, éventuellement d'en retrouver des anciens, mais jamais d'en gagner. Imaginons une application qui ait besoin d'utiliser des timers très précis. L'idéal est de passer cette tâche en ordonnancement temps réel, comme nous le verrons dans le chapitre 10. Pour ce faire, elle doit disposer des privilèges d'administrateur, même si elle est lancée par un utilisateur quelconque. On active donc le bit *Set-UID* de l'exécutable en lui donnant l'appartenance à root.

Toutefois, il lui faut également sauvegarder sur disque des informations n'appartenant qu'à l'utilisateur ayant lancé l'application (par exemple, sa configuration ou des données reçues). Pour ce faire, le programme ne doit créer des fichiers que dans des endroits où l'utilisateur est autorisé à le faire. Plutôt que de vérifier toutes les autorisations d'accès, il est plus simple de perdre ses privilèges root pour reprendre l'identité de l'utilisateur original. Nous reviendrons à plusieurs reprises sur ce mécanisme.

Sous Linux (contrairement à d'autres systèmes de type Unix comme Solaris), le bit *Set-UID* est ignoré pour les scripts que l'on lance avec la ligne shebang « `#!/ ...` » pour des raisons de sécurité. Voici en effet un programme Python qui réalise le même travail que l'exemple précédent :

```
exemple-getuid.py :
#!/usr/bin/python
from __future__ import print_function
import os

print("Real UID =", os.getuid(),
      " Effective UID =" os.geteuid())
```

Il s'exécute correctement, mais nous remarquons que le bit *Set-UID* n'a pas d'effet.

```
$ ls -ln exemple-getuid.py
-rwxrwxr-x 1 500 500 100 janv. 19 05:56 exemple-getuid.py
$ ./exemple-getuid.py
Real UID = 500 Effective UID = 500
# su
# chown root.root exemple-getuid.py
# chmod u+s exemple-getuid.py
# ls -ln exemple-getuid.py
-rwsrwxr-x 1 0 0 100 janv. 19 05:56 exemple-getuid.py
# exit
$ ./exemple-getuid.py
Real UID = 500 Effective UID = 500
```

Le troisième type d'UID d'un processus est l'UID *sauvé*. Il s'agit d'une copie de l'ancien UID effectif lorsque celui-ci est modifié par l'un des appels décrits plus bas. Cette copie est effectuée automatiquement par le noyau. Un processus peut toujours demander à changer son UID effectif ou son UID réel pour prendre la valeur de l'UID sauvé. Il est également possible de prendre en UID effectif la valeur de l'UID réel, et inversement.

Un processus avec le bit *Set-UID* positionné démarre donc avec un UID effectif différent de celui de l'utilisateur qui l'a invoqué. Quand il désire effectuer une opération non privilégiée, il peut demander à remplacer son UID effectif par l'UID réel. Une copie de l'UID effectif est conservée dans l'UID sauvé. Il pourra donc à tout moment demander à remplacer à nouveau son UID effectif par son UID sauvé.

Pour cela, il existe – pour des raisons historiques – plusieurs appels système permettant sous Linux de modifier son UID : `setuid()`, `seteuid()` et `setreuid()` sont définis par SUSv4 ; `setresuid()` est spécifique à Linux.

Les trois premiers appels système sont déclarés dans `<unistd.h>`, ainsi :

```
int setuid (uid_t uid_effectif);
int seteuid(uid_t uid_effectif);
int setreuid (uid_t uid_reel, uid_t uid_effectif);
```

Ils permettent de modifier un ou plusieurs UID du processus appelant, renvoyant 0 s'ils réussissent, ou -1 en cas d'échec.

Nous allons voir le comportement d'un programme *Set-UID* qui abandonne temporairement ses privilèges pour disposer des permissions de l'utilisateur l'ayant invoqué, puis qui reprend à nouveau ses autorisations originales. Notez bien que, dans cette première version, la récupération de l'ancienne identité ne fonctionne pas si le programme appartient à root. Ceci est clairement défini dans l'implémentation de `setuid()`. Les développeurs de Linux préviennent bien qu'en cas de mécontentement, il faut s'en prendre au comité Posix, qui est responsable de cette règle. Nous verrons immédiatement après une version utilisant `setreuid()`, qui fonctionne dans tous les cas de figure.

```
exemple-setuid.c :
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    uid_t uid_reel;
    uid_t uid_eff;
```

```
uid_reel = getuid();
uid_eff = geteuid();
fprintf(stdout, " UID-R = %u, UID-E = %u\n",
        getuid(), geteuid());
fprintf(stdout, " setuid(%d) = %d\n",
        uid_reel, setuid(uid_reel));
fprintf(stdout, " UID-R = %u, UID-E = %u\n",
        getuid(), geteuid());
fprintf(stdout, " setuid(%d) = %d\n",
        uid_eff, setuid(uid_eff));
fprintf(stdout, " UID-R = %u, UID-E = %u\n",
        getuid(), geteuid());
return 0;
}
```

L'exécution du programme (copié par un autre utilisateur, et avec le bit *Set-UID* positionné) donne :

```
$ ls -ln exemple-setuid
-rwsrwxr-x 1 501 501 8032 mai 26 10:39 exemple-setuid
$ ./exemple-setuid
UID-R = 500, UID-E = 501
setuid(500) = 0
UID-R = 500, UID-E = 500
setuid(501) = 0
UID-R = 500, UID-E = 501
$
```

Le programme évolue donc entre les privilèges de l'utilisateur 501 (le propriétaire du fichier exécutable avec bit *Set-UID*) et 500 (l'utilisateur qui l'a lancé).

Si on tente la même opération avec un programme *Set-UIDroot*, il ne pourra plus reprendre ses privilèges, car lorsque `setuid()` est invoqué par un utilisateur ayant un UID effectif nul (root), il écrase également l'UID sauvé pour empêcher le retour en arrière.

Les appels système `setuid()`, `seteuid()`, `setreuid()` et `setresuid()` existent dans le module `os` de Python. Néanmoins, le bit *Set-UID* n'ayant pas d'effet sur les scripts, leur intérêt est limité. On trouvera dans le répertoire d'exemples de ce chapitre l'implémentation équivalente en Python des codes C, mais nous ne les détaillons pas ici.

Voici maintenant une variante utilisant l'appel système `setreuid()`. Comme on peut s'en douter, il permet de fixer les deux UID en une seule fois. Si l'un des deux UID vaut `-1`, il n'est pas changé. Cet appel système à longtemps été réservé aux Unix

BSD, mais il est dorénavant défini par SUSv4 et ne devrait pas poser de problèmes de portabilité sur les systèmes récents.

```
exemple-setreuid.c :
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    uid_t uid_reel;
    uid_t uid_eff;

    uid_reel = getuid();
    uid_eff = geteuid();
    fprintf(stdout, "UID-R = %u, UID-E = %u\n",
            getuid(), geteuid());
    fprintf(stdout, "setreuid(-1, %d) = %d\n", uid_reel,
            setreuid(-1, uid_reel));
    fprintf(stdout, "UID-R = %u, UID-E = %u\n",
            getuid(), geteuid());
    fprintf(stdout, "setreuid(-1, %d) = %d\n", uid_eff,
            setreuid(-1, uid_eff));
    fprintf(stdout, "UID-R = %u, UID-E = %u\n",
            getuid(), geteuid());
    fprintf(stdout, "setreuid(%d, -1) = %d\n", uid_eff,
            setreuid(uid_eff, -1));
    fprintf(stdout, "UID-R = %u, UID-E = %u\n",
            getuid(), geteuid());
    return 0;
}
```

En voici l'exécution, après passage en *Set-UID root* :

```
$ ls -ln exemple-setreuid
-rwsrwxr-x 1 0 0 8184 mai 26 10:51 exemple-setreuid
$ ./exemple-setreuid
UID-R = 500, UID-E = 0
setreuid(-1, 500) = 0
UID-R = 500, UID-E = 500
setreuid(-1, 0) = 0
UID-R = 500, UID-E = 0
setreuid(0, -1) = 0
UID-R = 0, UID-E = 0
$
```

Cette fois-ci, le changement fonctionne parfaitement, même avec un UID effectif nul.

Enfin, il est possible – mais c’est une option spécifique à Linux – de modifier également l’UID sauvé, principalement pour empêcher le retour en arrière comme le fait `setuid()`, avec l’appel système `setresuid()`. En raison de la spécificité de cet appel système, il faut définir la constante `_GNU_SOURCE` avant d’inclure le fichier d’en-tête `<unistd.h>`.

```
int setresuid (uid_t  reel, uid_t  effectif, uid_t  sauve);
int getresuid (uid_t * reel, uid_t * effectif, uid_t * sauve);
```

exemple-setresuid.c :
#define _GNU_SOURCE

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void)
{
    uid_t uid_R, uid_E, uid_S;

    getresuid(& uid_R, & uid_E, & uid_S);
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E, uid_S);
    printf("setresuid(-1, %u, %u)=%d\n",
           uid_E, uid_R, setresuid(-1, uid_E, uid_R));
    getresuid(& uid_R, & uid_E, & uid_S);
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E, uid_S);
    printf("setresuid(-1, %u, -1)=%d\n",
           uid_S, setresuid(-1, uid_S, -1));
    getresuid(& uid_R, & uid_E, & uid_S);
    printf("UID-R=%u, UID-E=%u, UID-S=%u\n", uid_R, uid_E, uid_S);
    return 0;
}
```

L’exécution est intéressante si le programme est installé *Set-UID root* :

```
$ ls -ln exemple-setresuid
-rwsrwxr-x 1 0 0 6923 mai 26 10:56 exemple-setresuid
$ ./exemple-setresuid
UID-R=500, UID-E=0, UID-S=0
setresuid(-1, 0, 500)=0
UID-R=500, UID-E=0, UID-S=500
setresuid(-1, 500, -1)=0
UID-R=500, UID-E=500, UID-S=500
$
```

Identification du groupe d'utilisateurs du processus

Chaque utilisateur du système appartient à un ou plusieurs groupes. Ces derniers sont définis dans le fichier `/etc/groups`. Un processus fait donc également partie des groupes de l'utilisateur qui l'a lancé. Comme nous l'avons vu avec les UID, un processus dispose donc de plusieurs *GID* (*Group IDentifier*) réel, effectif, sauvé, ainsi que de GID supplémentaires si l'utilisateur qui a lancé le processus appartient à plusieurs groupes.

Attention

Il ne faut pas confondre les groupes d'utilisateurs auxquels un processus appartient, et qui dépendent de la personne qui lance le processus et éventuellement des attributs *Set-GID* du fichier exécutable, avec les groupes de processus, qui permettent principalement d'envoyer des signaux à des ensembles de processus. Un processus appartient donc à deux types de groupes qui n'ont rien à voir les uns avec les autres.

Le GID *réel* correspond au groupe principal de l'utilisateur ayant lancé le programme (celui qui est mentionné dans `/etc/passwd`).

Le GID *effectif* peut être différent du GID réel si le fichier exécutable dispose de l'attribut *Set-GID* (`chmod g+s`). C'est le GID effectif qui est utilisé par le noyau pour vérifier les autorisations d'accès aux fichiers.

La lecture de ces GID se fait symétriquement à celle des UID avec les appels système `getgid()` et `getegid()`. La modification (sous réserve d'avoir les autorisations nécessaires) peut se faire à l'aide des appels `setgid()`, `setegid()` et `setregid()`. Les prototypes de ces fonctions sont présents dans `<unistd.h>`, le type `gid_t` étant défini dans `<sys/types.h>` :

```
gid_t getgid (void);
gid_t getegid(void);
int  setgid (gid_t egid);
int  setegid(gid_t egid);
int  setregid (gid_t rgid, gid_t egid);
```

Les deux premières fonctions renvoient le GID demandé, les deux dernières renvoient 0 si elle réussissent et -1 en cas d'échec.

L'ensemble complet des groupes auxquels appartient un utilisateur est indiqué dans `/etc/groups` (en fait, c'est une table inversée puisqu'on y trouve la liste des utilisateurs appartenant à chaque groupe). Un processus peut obtenir cette liste en utilisant l'appel système `getgroups()` :

```
int getgroups (int taille, gid_t liste []);
```

Celui-ci prend deux arguments, une dimension et une table. Le premier argument indique la taille (en nombre d'entrées) de la table fournie en second argument.

L'appel système va remplir le tableau avec la liste des GID supplémentaires du processus. Si le tableau est trop petit, `getgroups()` échoue (renvoie -1 et remplit `errno`), sauf si la taille est nulle ; auquel cas, il renvoie le nombre de groupes supplémentaires du processus. La manière correcte d'utiliser `getgroups()` est donc la suivante.

```
exemple-getgroups.c :
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

int main (void)
{
    int taille;
    gid_t * table_gid = NULL;
    int i;

    if ((taille = getgroups(0, NULL)) < 0) {
        fprintf(stderr, "Erreur getgroups, errno = %d\n", errno);
        return 1;
    }

    if ((table_gid = calloc(taille, sizeof(gid_t))) == NULL) {
        fprintf(stderr, "Erreur calloc, errno = %d\n", errno);
        return 1;
    }

    if (getgroups(taille, table_gid) < 0) {
        fprintf(stderr, "Erreur getgroups, errno = %d\n", errno);
        return 1;
    }
    for (i = 0; i < taille; i++)
        fprintf(stdout, "%u ", table_gid[i]);
    fprintf(stdout, "\n");

    free(table_gid);
    return 0;
}
```

La fonction `calloc()` qui permet d'allouer de la mémoire pour un tableau est étudiée ultérieurement. L'exécution donne :

```
$ ./exemple-getgroups
500 100
$
```

L'implémentation du même programme en Python est beaucoup plus simple du fait de la gestion directe par ce langage des listes d'éléments :

```
exemple-getgroups.py :
#!/usr/bin/python
from __future__ import print_function
import os

for g in os.getgroups():
    print(g, end=' ')
print()
```

Il est possible de fixer sa liste de groupes supplémentaires. La fonction `setgroups()` n'est néanmoins utilisable que par `root` (ou un processus dont le fichier exécutable est *Set-UID root*)¹. Contrairement à `getgroups()`, le prototype est inclus dans le fichier `<grp.h>` :

```
int setgroups (size_t taille, const gid_t * table);
```

Il faut définir la constante symbolique `_BSD_SOURCE` pour avoir accès à cette fonction.

```
exemple-setgroups.c :
#define _BSD_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <grp.h>

int main (int argc, char * argv [])
{
    gid_t * table_gid = NULL;
    int i;
    int taille;

    if (argc < 2) {
        fprintf(stderr, "Usage %s GID ...\n", argv[0]);
        return 1;
    }
    if ((table_gid = calloc(argc - 1, sizeof(gid_t))) == NULL) {
        fprintf(stderr, "Erreur calloc, errno = %d\n", errno);
        return 1;
    }
}
```

1. En réalité, depuis Linux 2.2, il suffit que le processus ait la capacité `CAP_SETGID` comme nous le verrons en fin de chapitre.

```

for (i = 1; i < argc ; i ++)
    if (sscanf(argv[i], "%u", &(table_gid[i - 1])) != 1) {
        fprintf(stderr, "GID invalide : %s\n", argv[i]);
        return 1;
    }
if (setgroups(i - 1, table_gid) < 0) {
    fprintf(stderr, "Erreur setgroups, errno = %d\n", errno);
    return 1;
}
free(table_gid);

/* Passons maintenant à la vérification des groupes */
fprintf(stdout, "Verification : ");
/*
 * ...
 * Même code que la fonction main() de exemple-getgroups.c
 * ...
 */
}

```

Ce programme ne fonctionne que s'il est *Set-UID root* ; c'est pourquoi l'implémentation en Python (disponible dans le répertoire d'exemples de ce chapitre) n'est pas présentée ici.

```

$ ls -ln exemple-setgroups
-rwxrwxr-x 1 500 500 9303 mai 26 11:39 exemple-setgroups
$ ./exemple-setgroups
Usage ./exemple-setgroups GID ...
$ ./exemple-setgroups 501 502
Erreur setgroups, errno = 1
$ su
Mot de passe :
# chown root.root exemple-setgroups
# chmod u+s exemple-setgroups
# exit
exit
$ ls -ln exemple-setgroups
-rwsrwxr-x 1 0 0 9303 mai 26 11:39 exemple-setgroups
$ ./exemple-setgroups 501 502
Verification : 501 502
$

```

Pour un processus *Set-UID root*, le principal intérêt de la modification de la liste des groupes auxquels appartient un processus est de pouvoir ajouter un groupe spécial (donnant par exemple un droit de lecture et d'écriture sur un fichier spécial de périphé-

rique) à sa liste, et de changer ensuite son UID effectif pour continuer à s'exécuter sous l'identité de l'utilisateur, tout en gardant le droit d'agir sur ledit périphérique.

Tout comme nous l'avons vu plus haut avec les UID, il existe sous Linux un GID *sauvé* pour chaque processus. Cela permet de modifier son GID effectif (en reprenant temporairement l'identité réelle), puis de retrouver le GID effectif original (qui était probablement fourni par le bit *Set-GID*). Pour accéder aux GID sauvés, deux appels système, `setresgid()` et `getresgid()`, sont disponibles lorsque la constante `_GNU_SOURCE` est définie avant d'inclure `<unistd.h>` :

```
int setresgid (gid_t  reel, uid_t  effectif, uid_t  sauve);
int getresgid (gid_t * reel, uid_t * effectif, uid_t * sauve);
```

Le programme `exemple-setresgid.c` est une copie de `exemple-setresuid.c` dans lequel on a changé toutes les occurrences de `uid` en `gid`. En voici un exemple d'exécution après sa transformation en programme *Set-GID root* :

```
$ ls -ln exemple-setresgid
-rwsrwxr-x 1 0 0 6923 mai 26 11:47 exemple-setresgid
$ ./exemple-setresgid
GID-R=500, GID-E=0, GID-S=0
setresgid(-1, 0, 500)=0
GID-R=500, GID-E=0, GID-S=500
setresgid(-1, 500, -1)=0
GID-R=500, GID-E=500, GID-S=500
$
```

Identification du groupe de processus

Les processus sont organisés en *groupes*. Rappelons qu'il ne faut pas confondre les *groupes de processus* avec les *groupes d'utilisateurs* que nous venons de voir, auxquels appartiennent les processus. Les groupes de processus permettent l'envoi global de *signaux* à un ensemble de processus. Ce concept, tout comme l'identificateur de session que nous verrons immédiatement à la suite, sert surtout aux interpréteurs de commandes – les shells – pour implémenter le contrôle des jobs. La prise en considération des groupes de processus dans les applications classiques est rare.

Pour savoir à quel groupe appartient un processus donné, on utilise l'appel système `getpgid()` déclaré dans `<unistd.h>` :

```
pid_t getpgid (pid_t pid);
```

Celui-ci prend en argument le PID du processus visé et renvoie son numéro de groupe, ou -1 si le processus mentionné n'existe pas. Avec la bibliothèque Glibc, `getpgid()` n'est défini dans `<unistd.h>` que si la constante symbolique `_GNU_SOURCE` est déclarée avant l'inclusion.

exemple-getpgid.c :

```
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (int argc, char * argv[])
{
    int i;
    long int pid;
    long int pgid;

    if (argc == 1) {
        fprintf(stdout, "%d : %d\n", getpid(), getpgid(0));
        return 0;
    }
    for (i = 1; i < argc; i++)
        if (sscanf(argv[i], "%ld", & pid) != 1) {
            fprintf(stderr, "PID invalide : %s\n", argv[i]);
        } else {
            pgid = (long) getpgid((pid_t) pid);
            if (pgid == -1)
                fprintf(stderr, "%ld inexistant\n", pid);
            else
                fprintf(stderr, "%ld : %ld\n", pid, pgid);
        }
    return 0;
}
```

Ce programme permet de consulter les groupes de n'importe quels processus, « 0 » signifiant « processus appelant ».

```
$ ps a
  PID TTY          STAT TIME COMMAND
[...]
```

PID	TTY	STAT	TIME	COMMAND
17585	pts/3	Ss	0:00	/bin/bash
17624	pts/3	S+	0:00	man getpgid
17627	pts/3	S+	0:00	sh -c (cd /usr/share/man/fr && (echo
17628	pts/3	S+	0:00	sh -c (cd /usr/share/man/fr && (echo
17635	pts/3	S+	0:00	/usr/bin/less -is

```
$ ./exemple-getpgid 17585 17624 17627 17628 17635
17585 : 17585
17624 : 17624
17627 : 17624
17628 : 17624
17635 : 17624
$
```

Un groupe a été créé au lancement du processus 17624 (man), et il comprend tous les descendants (mise en forme et affichage de la page). Le processus dont le PID est identique au numéro de groupe est nommé *leader* du groupe. Un groupe n'a pas nécessairement de *leader*, celui-ci pouvant se terminer alors que ses descendants continuent de s'exécuter.

On peut implémenter le même exemple en Python :

```
exemple-getpgid.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys

if len(sys.argv) == 1:
    print(os.getpid(), ":", os.getpgid(0))
else:
    sys.argv.pop(0)
    for pid in sys.argv:
        try:
            print(pid, ":", os.getpgid(int(pid)))
        except ValueError:
            print("Invalid PID:", pid)
            pass
        except OSError:
            print("Unknown PID:", pid)
            pass
```

Il existe un appel système `getpgrp()`, qui ne prend pas d'argument et renvoie le numéro de groupe du processus appelant, exactement comme `getpgid(0)`. Attention toutefois, la portabilité de cet appel système n'est pas assurée, certaines versions d'Unix l'implémentant comme un synonyme exact de `getpgid()`.

```
exemple-getpgrp.py :
#!/usr/bin/python
from __future__ import print_function
import os
```

```
print(os.getpid(), ":", os.getpgrp())
```

Ou en C :

```
exemple-getpgrp.c :
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (int argc, char * argv[])
{
    fprintf(stdout, "%ld : %ld\n",
            (long) getpid(), (long) getpgrp());
    return 0;
}
```

```
$ ./exemple-getpgrp
18398 : 18398
$
```

La plupart des applications n'ont pas à se préoccuper de leur groupe de processus, mais cela peut parfois être indispensable lorsqu'on désire envoyer un signal à tous les descendants d'un processus père. Les interpréteurs de commandes, ou les programmes qui lancent des applications diverses (gestionnaires de fenêtres X11, gestionnaires de fichiers...), doivent pouvoir tuer tous les descendants directs d'un processus fils. Cela peut aussi être nécessaire si l'application crée de nombreux processus fils (par exemple à chaque demande de connexion pour un démon serveur réseau) et désire pouvoir se terminer complètement en une seule fois.

Un processus peut modifier son propre identifiant de groupe ou celui de l'un de ses descendants grâce à l'appel système `setpgid()` :

```
int setpgid (pid_t pid, pid_t pgid);
```

Le premier argument correspond au PID du processus à modifier. Si cet argument est nul, on considère qu'il s'agit du processus appelant. Le second argument indique le nouveau numéro de groupe pour le processus concerné. Si le second argument est égal au premier ou s'il est nul, le processus devient *leader* de son groupe.

L'appel système échoue si le processus visé n'est ni le processus appelant ni l'un de ses descendants. Par ailleurs, un processus ne peut plus modifier le groupe de l'un de ses

descendants si celui-ci a effectué un appel à l'une des fonctions de la famille `exec()`. Généralement, les interpréteurs de commandes utilisent la procédure suivante :

- Le shell exécute un `fork()`. Le processus père garde le résultat dans une variable `pid_fils`.
- Le processus fils demande à devenir *leader* de son groupe en invoquant `setpgid(0,0)`.
- De manière redondante, le processus père réclame que son fils devienne *leader* de son groupe, cela pour éviter tout problème de concurrence d'exécution. Le père exécute donc `setpgid(pid_fils, pid_fils)`.
- Le père peut alors attendre, par exemple, la fin de l'exécution du fils avec `waitpid()`.
- Le fils appelle une fonction de la famille `exec()` pour lancer la commande désirée, nous verrons ces fonctions dans le prochain chapitre.

Le shell pourra alors contrôler l'ensemble des processus appartenant au groupe du fils en leur envoyant des signaux (`STOP`, `CONT`, `TERM...`). Le double appel à `setpgid()` dans le père et le fils est nécessaire car nous devons être sûr que la modification soit réalisée avant que le fils appelle `exec()` (donc l'invocation dans le processus père seul n'est pas suffisante) et avant que le père ne commence à lui envoyer des signaux (symétriquement l'appel dans le processus fils n'est pas suffisant).

Il existe un appel système `setpgrp()`, qui sert directement à créer un groupe de processus et à en devenir leader. Il s'agit d'un synonyme de `setpgid(0,0)`. Attention là encore à la portabilité de cet appel système, car sous BSD il s'agit d'un synonyme de `setpgid()` utilisant donc deux arguments.

Identification de session

Il existe finalement un dernier regroupement de processus, les *sessions*, qui réunissent divers groupes de processus. Les sessions sont très liées à la notion de *terminal de contrôle* des processus. Il n'y a guère que les shells ou les gestionnaires de fenêtres pour les environnements graphiques qui ont besoin de gérer les sessions. Une exception toutefois : les applications qui s'exécutent sous forme de *démon* doivent accomplir quelques formalités concernant leur session. C'est donc principalement ce point de vue qui nous importera ici.

Généralement, une session est attachée à un terminal de contrôle, celui qui a servi à la connexion de l'utilisateur. Avec l'évolution des systèmes, les terminaux de contrôle sont souvent des pseudo-terminaux virtuels gérés par les systèmes graphiques de fenêtrage ou par les pilotes de connexion réseau. Au sein d'une session, un groupe de processus est en *avant-plan* ; il reçoit directement les données saisies sur le clavier du terminal, et peut afficher ses informations de sortie sur l'écran de celui-ci. Les autres

groupes de processus de la session s'exécutent en arrière-plan. Leur interaction avec le terminal sera étudiée ultérieurement dans le chapitre sur les signaux.

Pour créer une nouvelle session, un processus ne doit pas être *leader* de son groupe. En effet, la création de la session passe par une étape de constitution d'un nouveau groupe de processus prenant l'identifiant du processus appelant. Il est indispensable que cet identifiant ne soit pas encore attribué à un groupe qui pourrait contenir éventuellement d'autres processus.

La création d'une session s'effectue par l'appel système `setsid()`, déclaré dans `<unistd.h>` :

```
| pid_t setsid (void);
```

Il renvoie le nouvel identifiant de session, de type `pid_t`. Lors de cet appel, un nouveau groupe est créé, il ne contient que le processus appelant (qui en est donc le leader). Puis, une nouvelle session est créée, ne contenant pour le moment que ce groupe. Cette session ne dispose pas de terminal de contrôle. Elle devra en récupérer un explicitement si elle le désire. Les descendants du processus leader se trouveront, bien entendu, dans cette nouvelle session.

Un point de détail reste à préciser. Pour être sûr que le processus initial n'est pas leader de son groupe, on utilise généralement l'astuce suivante :

- Un processus père exécute un `fork()`, suivi d'un `exit()`.
- Le processus fils se trouvant dans le même groupe que son père ne risque pas d'être leader, et peut donc tranquillement invoquer `setsid()`.

La fonction `getsid()` prend en argument un PID et renvoie l'identifiant de la session, c'est-à-dire le PID du processus *leader* :

```
| pid_t getsid (pid_t pid);
```

Cet appel système n'est déclaré dans `<unistd.h>` que si la constante `_GNU_SOURCE` est définie avant son inclusion. Cette fonction n'échoue que si le PID transmis ne correspond à aucun processus existant. Comme d'habitude, `getsid(0)` renvoie l'identifiant du processus appelant. Bien que définie dans SUSv4, cette fonction n'est pas portable sur tous les systèmes BSD.

```
| exemple-getsid.c :
```

```
| #define _GNU_SOURCE  
| #include <stdio.h>  
| #include <unistd.h>  
| #include <sys/types.h>
```

```

int main (int argc, char * argv[])
{
    int    i;
    long int pid;
    long int sid;

    if (argc == 1) {
        fprintf(stdout, "%d : %d\n", getpid(), getsid(0));
        return 0;
    }
    for (i = 1; i < argc; i++)
        if (sscanf(argv[i], "%ld", & pid) != 1) {
            fprintf(stderr, "PID invalide : %s\n", argv[i]);
        } else {
            sid = (long) getsid((pid_t)pid);
            if (sid == -1)
                fprintf(stderr, "%ld inexistant\n", pid);
            else
                fprintf(stderr, "%ld : %ld\n", pid, sid);
        }
    return 0;
}

```

L'implémentation en Python :

```

exemple-getsid.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys

if len(sys.argv) == 1:
    print(os.getpid(), ":", os.getsid(0))
else:
    sys.argv.pop(0)
    for pid in sys.argv:
        try:
            print(pid, ":" os.getsid(int(pid)))
        except ValueError:
            print("Invalid PID:", pid)
            pass
        except OSError:
            print("Unknown PID:", pid)
            pass

```

```
$ ps x
  PID TTY          STAT TIME COMMAND
[...]  
3904 ?            Ss     0:00 kdeinit Running...  
[...]  
3921 ?            S       0:09 kdesktop  
[...]  
3943 ?            S       0:00 kmix -session  
[...]  
5726 ?            S       0:07 konqueror --silent  
6789 ?            R       0:11 konsole  
6790 pts/1        Ss     0:00 /bin/bash  
[...]  
$ ./exemple-getsid 0 3921 3943 5726 6789  
0 : 6790  
3921 : 3904  
3943 : 3904  
5726 : 3904  
6789 : 3904  
$
```

Nous voyons que le processus en cours appartient à la session de son interpréteur de commandes (/bin/bash) et que les applications graphiques dépendent de l'environnement KDE.

L'interaction entre un processus et un terminal s'effectue donc par l'intermédiaire de plusieurs indirections :

- Le processus appartient toujours à un groupe.
- Le groupe appartient dans son intégralité à une session.
- La session peut – éventuellement – avoir un terminal de contrôle.
- Le terminal connaît le numéro du groupe de processus en avant-plan.

C'est en général le *leader* de session (le shell) qui assure le basculement en avant-plan ou en arrière-plan des groupes de processus de sa session, en utilisant les fonctions de dialogue avec le terminal, `tcgetpgrp()` et `tcsetpgrp()`. Ces fonctions seront analysées ultérieurement.

Capacités d'un processus

La toute-puissance d'un processus exécuté sous l'UID effectif root peut être limitée. Une application dispose d'un jeu de *capacités* permettant de définir ce que le processus peut faire sur le système. Cela est défini dans le document *Posix.1e* (anciennement *Posix.6*).

Les capacités d'un processus correspondent à des privilèges, aussi les applications courantes ont-elles des ensembles de capacités vides. En dotant un programme d'un jeu restreint de privilèges (par exemple, pour modifier sa propre priorité d'ordonnancement, on lui accorde une puissance suffisante pour accomplir son travail, tout en évitant tout problème de sécurité qui pourrait survenir si le programme était détourné de son utilisation normale. Ainsi, même si une faille de sécurité existe dans l'application, et si elle est découverte par un utilisateur malintentionné, celui-ci ne pourra exploiter que le privilège accordé au programme et pas d'autres capacités dangereuses réservées habituellement à root (par exemple, pour insérer un module personnel dans le noyau).

Un processus dispose de trois ensembles de capacités :

- L'ensemble des capacités *effectives* est celui qui est utilisé à un instant donné pour vérifier les autorisations du processus. Cet ensemble joue un rôle similaire à celui de l'UID effectif, qui n'est pas nécessairement égal à l'UID réel, mais est utilisé pour les permissions d'accès aux fichiers.
- L'ensemble des capacités *transmissibles* est celui qui sera hérité lors d'un appel système `exec()`. Notons que l'appel `fork()` ne modifie pas les ensembles de capacités ; le fils a les mêmes privilèges que son père.
- L'ensemble des capacités *possibles* est une réserve de privilèges. Un processus peut copier une capacité depuis cet ensemble vers n'importe lequel des deux autres. C'est en fait cet ensemble qui représente la véritable limite des possibilités d'une application.

Une application a le droit de réaliser les opérations suivantes sur ses capacités :

- On peut mettre dans l'ensemble effectif ou l'ensemble transmissible n'importe quelle capacité.
- On peut supprimer une capacité de n'importe quel ensemble.

Un fichier exécutable dispose également en théorie des mêmes trois ensembles. Toutefois, les systèmes de fichier actuels ne permettent pas encore le support pour toutes ces données. Aussi un fichier exécutable *Set-UIDroot* est-il automatiquement lancé avec ses ensembles de capacités effectives et possibles remplis. Un fichier exécutable normal démarre avec des ensembles effectif et possible égaux à l'ensemble transmissible du processus qui l'a lancé. Dans tous les cas, l'ensemble transmissible n'est pas modifié durant l'appel système `exec()`.

Les capacités présentes dans le noyau Linux sont définies dans `<linux/capability.h>`. En voici une description, les astérisques signalant les capacités mentionnées dans le document Posix.1e.

Nom	Signification
CAP_CHOWN (*)	Possibilité de modifier le propriétaire ou le groupe d'un fichier.
CAP_DAC_OVERRIDE (*)	Accès complet sur tous les fichiers et les répertoires.

Nom	Signification
CAP_DAC_READ_SEARCH (*)	Accès en lecture ou exécution sur tous les fichiers et répertoires.
CAP_FOWNER (*)	Possibilité d'agir à notre gré sur un fichier ne nous appartenant pas, sauf pour les cas où CAP_FSETID est nécessaire.
CAP_FSETID (*)	Possibilité de modifier les bits <i>Set-UID</i> ou <i>Set-GID</i> d'un fichier ne nous appartenant pas.
CAP_IPC_LOCK	Autorisation de verrouiller des segments de mémoire partagée et de bloquer des pages en mémoire avec <code>mlock()</code> .
CAP_IPC_OWNER	Accès aux communications entre processus sans passer par les autorisations d'accès.
CAP_KILL (*)	Possibilité d'envoyer un signal à un processus ne nous appartenant pas.
CAP_LEASE	Autorisation d'établir un bail sur un fichier pour être averti par un signal si un autre processus ouvre ce fichier.
CAP_MKNOD	Création d'un fichier spécial représentant un périphérique.
CAP_LINUX_IMMUTABLE	Modification d'attributs spéciaux des fichiers.
CAP_NET_ADMIN	Possibilité d'effectuer de nombreuses tâches administratives concernant le réseau, les interfaces, les tables de routage, etc.
CAP_NET_BIND_SERVICE	Autorisation d'accéder à un port privilégié sur le réseau (numéro de port inférieur à 1024).
CAP_NET_BROADCAST	Autorisation d'émettre des données en <i>broadcast</i> et de s'inscrire à un groupe <i>multicast</i> .
CAP_NET_RAW	Possibilité d'utiliser des sockets réseau de type <i>raw</i> .
CAP_SETGID (*)	Autorisation de manipuler le bit <i>Set-GID</i> et de s'ajouter des groupes supplémentaires.
CAP_SETPCAP	Possibilité de transférer nos capacités à un autre processus (impossible à utiliser).
CAP_SETUID (*)	Autorisation de manipuler les bits <i>Set-UID</i> et <i>Set-GID</i> d'un fichier nous appartenant.
CAP_SYS_ADMIN	Possibilité de réaliser de nombreuses opérations de configuration concernant le système proprement dit.
CAP_SYS_BOOT	Autorisation d'arrêter et de redémarrer la machine.
CAP_SYS_CHROOT	Possibilité d'utiliser l'appel système <code>chroot()</code> .
CAP_SYS_MODULE	Autorisation d'insérer ou de retirer des modules de code dans le noyau.
CAP_SYS_NICE	Possibilité de modifier sa priorité d'ordonnancement, ou de basculer en fonctionnement temps réel.
CAP_SYS_PACCT	Mise en service de la comptabilité des processus.
CAP_SYS_PTRACE	Possibilité de suivre l'exécution de n'importe quel processus.
CAP_SYS_RAWIO	Accès aux ports d'entrée-sortie de la machine.

Nom	Signification
CAP_SYS_RESOURCE	Possibilité de modifier plusieurs limitations concernant les ressources du système.
CAP_SYS_TIME	Mise à l'heure de l'horloge système.
CAP_SYS_TTY_CONFIG	Autorisation de configurer les consoles.

Lorsque nous examinerons une fonction privilégiée, nous indiquerons quelle capacité est nécessaire pour s'en acquitter. En revanche, nous n'allons pas détailler le moyen de configurer les permissions d'un processus, car l'interface du noyau est sujette aux changements. Il existe deux appels système, `capset()` et `capget()`, permettant de configurer les ensembles de permissions d'un processus. Toutefois, ils ne sont ni portables ni même garantis d'exister dans les noyaux futurs.

Pour agir sur les privilèges d'une application, il faut employer la bibliothèque `libcap`, qui n'est pas toujours installée dans les distributions courantes. Cette bibliothèque fournit non seulement des fonctions Posix.1e pour modifier les permissions, mais également des utilitaires permettant, par exemple, de lancer une application avec un jeu restreint de privilèges.

Attention

Veillez à ne pas confondre la bibliothèque *libcap* qui gère les capacités des processus et la bibliothèque *libpcap* qui sert à capturer des paquets sur le réseau.

La segmentation des privilèges habituellement réservés à root est une chose importante pour l'avenir de Linux. Cela permet non seulement à un administrateur de déléguer certaines tâches à des utilisateurs de confiance (par exemple en leur fournissant un shell possédant la capacité `CAP_SYS_BOOT` pour pouvoir arrêter l'ordinateur), mais la sécurité du système est aussi augmentée. Une application ayant besoin de quelques privilèges bien ciblés ne disposera pas de la toute-puissance de root. Ainsi, un serveur X11 ayant besoin d'accéder à la mémoire vidéo aura la capacité `CAP_SYS_RAWIO`, mais ne pourra pas aller écrire dans n'importe quel fichier système. De même, un logiciel d'extraction de pistes audio depuis un CD, comme l'application `cdda2wav`, aura le privilège `CAP_SYS_NICE` car il lui faudra passer sur un ordonnancement temps réel, mais il n'aura pas d'autres autorisations particulières.

Si un pirate découvre une faille de sécurité lui permettant de faire exécuter le code de son choix sous l'UID effectif de l'application, il n'aura toutefois que le privilège du processus initial. Dans les deux exemples indiqués précédemment, il pourra perturber l'affichage grâce à l'accès à la mémoire vidéo, ou bloquer le système en faisant boucler un processus de haute priorité temps réel. Dans un cas comme dans l'autre, cela ne présente aucun intérêt pour lui. Il ne pourra modifier aucun fichier système

(pas d'ajout d'utilisateur, par exemple) ni agir sur le réseau pour se dissimuler en préparant l'attaque d'un autre système. Ses possibilités sont largement restreintes.

Malheureusement, il n'existe pas encore d'interface homogène au niveau des systèmes de fichiers pour configurer les capacités des processus. Des évolutions dans ce sens apparaissent, comme le système *SeLinux* (*Security Enhanced Linux*), mais elles sont encore peu répandues.

Conclusion

Dans ce chapitre, nous avons essayé de définir la notion de processus, la manière d'en créer, et les différents identifiants qui peuvent y être attachés. Une application classique n'a pas souvent l'occasion de manipuler ses UID, GID, etc. Cela devient indispensable toutefois si l'accès à des ressources privilégiées qui doivent être offertes à tout utilisateur est nécessaire. L'application doit savoir perdre temporairement ses privilèges, quitte à les récupérer ultérieurement. De même, certains programmes ayant un dialogue important avec leurs descendants seront amenés à gérer des groupes de processus. Bien entendu, tout ceci est également nécessaire lors de la création de processus démons, comme nous le verrons dans la partie consacrée à la programmation réseau.

Une présentation détaillée des permissions associées aux processus se trouve dans [Bach 1989] *Conception du système Unix*. Nous avons également abordé les principes des capacités Posix.1e qui permettent d'améliorer la sécurité d'une application nécessitant des privilèges. Il faut toutefois être conscient que l'implémentation actuelle de ces capacités est loin d'être aussi riche que ce que propose Posix.1e.

Dans les chapitres suivants, nous étudierons l'exécution d'un nouveau programme, avec création d'un nouveau processus ou non, ainsi que l'environnement de ce processus, c'est-à-dire la vision qu'il a du système, et que nous pouvons configurer avant son démarrage. Enfin nous nous intéresserons à la fin d'un processus, les conditions de sa terminaison, et leur suivi.

3

Exécution d'un programme

Ce chapitre va être principalement consacré aux débuts d'un processus. Tout d'abord, nous examinerons les méthodes utilisables pour lancer un nouveau programme, ainsi que les mécanismes sous-jacents, qui peuvent conduire à un échec du démarrage. Nous nous intéresserons ensuite à des fonctions simplifiées, permettant d'utiliser une application indépendante comme une sous-routine de notre logiciel.

Lancement d'un nouveau programme

Nous avons déjà vu que le seul moyen de créer un nouveau processus dans le système est d'invoquer `fork()`, qui duplique le processus appelant. Mais lors de cette duplication, le processus est identique à son père. Seul le code de retour de `fork()` peut servir à exécuter des actions différentes. Nous allons à présent voir comment charger un nouveau fichier exécutable en mémoire.

La seule façon d'exécuter un nouveau programme est d'appeler l'une des fonctions de la famille `exec()`. Nous verrons toutefois qu'il existe les fonctions `popen()` et `system()`, qui permettent d'exécuter une autre application mais en s'appuyant sur `fork()` et `exec()`.

L'appel de l'une des fonctions `exec()` permet de remplacer l'espace mémoire du processus appelant par le code et les données de la nouvelle application. Ces fonctions ne reviennent qu'en cas d'erreur, sinon le processus appelant est entièrement remplacé.

On parle couramment de l'appel système `exec()` sous forme générique, mais en fait il n'existe aucune routine ayant ce nom. Simplement, il y a six variantes nommées `exec()`, `execle()`, `execvp()`, `execv()`, `execve()` et `execvp()`. Ces fonctions permettent de lancer une application. Les différences portent sur la manière de transmettre les arguments et l'environnement, et sur la méthode pour accéder au programme à lancer. Il n'existe sous Linux qu'un seul véritable appel système dans cette famille de fonctions : `execve()`. Les autres fonctions sont implémentées dans la bibliothèque C à partir de cet appel système.

Les fonctions dont le suffixe commence par un `l` utilisent une liste d'arguments à transmettre de nombre variable, tandis que celles qui débutent par un `v` emploient un tableau à la manière du vecteur `argv[]`.

Les fonctions se terminant par un `e` transmettent l'environnement dans un tableau `envp[]` explicitement passé dans les arguments de la fonction, alors que les autres utilisent la variable globale `environ` (`os.environ` en Python).

Les fonctions se finissant par un `p` utilisent la variable d'environnement `PATH` pour rechercher le répertoire dans lequel se situe l'application à lancer, alors que les autres nécessitent un chemin d'accès complet. La variable `PATH` est déclarée dans l'environnement comme étant une liste de répertoires séparés par des deux-points. On utilise typiquement une affectation du genre :

```
| PATH=/usr/bin:/bin:/usr/local/bin:/usr/sbin:/sbin
```

Il est préférable de placer en tête de `PATH` les répertoires dans lesquels se trouvent les applications les plus utilisées afin d'accélérer la recherche. Certains ajoutent à leur `PATH` un répertoire simplement composé d'un point, représentant le répertoire en cours. Cela peut entraîner une faille de sécurité, surtout si ce répertoire `.` n'est pas placé en dernier dans l'ordre de recherche. Il vaut mieux ne pas le mettre dans le `PATH` et utiliser explicitement une commande :

```
$ ./mon-programme
```

pour lancer une application qui se trouve dans le répertoire courant.

Quand `execvp()` ou `execvp()` rencontrent, lors de leur parcours des répertoires du `PATH`, un fichier exécutable du nom attendu, ils tentent de le charger. S'il ne s'agit pas d'un fichier binaire mais d'un fichier de texte commençant par une ligne du type :

```
| #! /bin/interpreteur
```

le programme indiqué (interpréteur) est chargé, et le fichier lui est transmis sur son entrée standard. Il s'agit souvent de `/bin/sh`, qui permet de lancer des scripts shell, mais on peut trouver d'autres fichiers à interpréter (`/bin/awk`, `/usr/bin/perl`, `/usr/bin/python...`). Nous verrons une invocation de script shell plus loin.

Si l'appel `exec()` réussit, il ne revient pas, sinon il renvoie `-1`, et `errno` contient un code expliquant les raisons de l'échec. Celles-ci sont détaillées dans la page de manuel `execve(2)`.

Le prototype de `execve()` est le suivant :

```
int execve (const char * appli, const char * argv[],  
            const char * envp[]);
```

La chaîne `appli` doit contenir le chemin d'accès au programme à lancer à partir du répertoire de travail en cours ou à partir de la racine du système de fichiers s'il commence par un *slash* `"/"`.

Le tableau `argv[]` contient des chaînes de caractères correspondant aux arguments qu'on trouve habituellement sur la ligne de commande.

La première chaîne `argv[0]` doit contenir le nom de l'application à lancer (sans chemin d'accès). Ceci peut parfois être utilisé pour des applications qui modifient leur comportement en fonction du nom sous lequel elles sont invoquées. Par exemple, `/bin/gzip` sert à compresser des fichiers. Il est également utilisé pour décompresser des fichiers si on lui transmet l'option `-d` ou si on l'invoque sous le nom `gunzip`. Pour ce faire, il analyse `argv[0]`. Dans la plupart des distributions Linux, il existe d'ailleurs un lien physique nommé `/bin/gunzip` sur le même fichier que `/bin/gzip`.

Le troisième argument est un tableau de chaînes déclarant les variables d'environnement (sur lesquelles nous reviendrons plus en détail dans le prochain chapitre).

On peut éventuellement utiliser la variable externe globale `environ` si on désire transmettre le même environnement au programme à lancer. Dans la majorité des applications, il est toutefois important de mettre en place un environnement cohérent, grâce aux fonctions que nous étudierons dans le prochain chapitre. Ceci est particulièrement nécessaire dans les applications susceptibles d'être installées *Set-UID root*.

Les tableaux `argv[]` et `envp[]` doivent se terminer par des pointeurs `NULL`.

Pour montrer l'utilisation de `execve()`, nous allons invoquer le shell, en lui passant la commande « `echo $SHLVL` ». Le shell nous affichera alors la valeur de cette variable d'environnement. `bash` comme `tcsh` indiquent dans cette variable le nombre d'invocations successives du shell qui sont « empilées ». Voici un exemple sous `bash` :

```
$ echo $SHLVL
1
$ sh
$ echo $SHLVL
2
$ sh
$ echo $SHLVL
3
$ exit
$ echo $SHLVL
2
$ exit
$ echo $SHLVL
1
$
```

Notre programme exécutera donc simplement cette commande en lui transmettant son propre environnement. On notera que la commande « `echo $SHLVL` » doit être transmise en un seul argument, comme on le ferait sur la ligne de commande :

```
$ sh -c "echo $SHLVL"
2
$
```

(L'option `-c` demande au shell d'exécuter l'argument suivant, puis de se terminer.)

```
exemple-execve.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

extern char ** environ;

int main (void)
{
    char * argv[] = {"sh", "-c", "echo $SHLVL", (char *) NULL };

    fprintf(stdout, "Je lance /bin/sh -c \"echo $SHLVL\" :\n");

    execve("/bin/sh", argv, environ);

    fprintf(stdout, "Echec : erreur = %d\n", errno);
    return 0;
}
```

Ou en Python :

```
exemple-execve.py:
#!/usr/bin/python
from __future__ import print_function
import os

arg = [ 'sh', '-c', 'echo $SHLVL' ]
print("Running: /bin/sh -c 'echo $SHLVL'");
try:
    os.execve('/bin/sh', arg, os.environ)
except OSError:
    print("Failed")
```

Voici un exemple d'exécution sous bash :

```
$ echo $SHLVL
1
$ ./exemple-execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$ sh
$ ./exemple-execve
Je lance /bin/sh -c "echo $SHLVL" :
3
$ exit
$ ./exemple-execve
Je lance /bin/sh -c "echo $SHLVL" :
2
$
```

Bien entendu, le programme ayant lancé un nouveau shell pour exécuter la commande, le niveau d'imbrication est incrémenté par rapport à la variable d'environnement, consultée directement avec « echo \$SHLVL ».

La fonction `execv()` dispose du prototype suivant :

```
| int execv (const char * application, const char * argv []);
```

Elle fonctionne comme `execve()`, mais l'environnement est directement transmis par l'intermédiaire de la variable externe `environ`, sans avoir besoin d'être passé explicitement en argument durant l'appel. Nous détaillerons le contenu de l'environnement dans le prochain chapitre.

La fonction `execvp()` utilise un prototype semblable à celui de `execv()`, mais elle se sert de la variable d'environnement `PATH` pour rechercher l'application. Nous allons en voir un exemple, qui exécute simplement la commande `ls`.

```
exemple-execvp.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(void)
{
    char * argv[] = { "ls", "-l", "-n", (char *) NULL };

    execvp("ls", argv);

    fprintf(stderr, "Erreur %d\n", errno);
    return 1;
}
```

Ou encore :

```
exemple-execvp.py :
#!/usr/bin/python
from __future__ import print_function
import os

args = [ "ls", "-l", "-n" ]
try:
    os.execvp(args[0], args)
except OSError:
    print("Error, unable to run", args[0])
```

Lorsqu'on exécute cette application, celle-ci recherche `ls` dans les répertoires de la variable d'environnement `PATH`. Ainsi, en modifiant cette variable pour éliminer le répertoire contenant `ls`, `execvp()` échoue.

```
$ which ls
/bin/ls
$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/home/cpb/bin
$ ./exemple-execvp
total 28
-rwxrwxr-x 1 500 500 8125 mai 26 15:21 exemple-execve
-rw-rw-r-- 1 500 500 353 mai 26 15:21 exemple-execve.c
```

```
-rwxrwxr-x 1 500 500 7731 mai 26 15:27 exemple-execvp
-rw-rw-r-- 1 500 500 232 mai 26 15:27 exemple-execvp.c
-rw-r--r-- 1 500 500 200 mai 26 15:27 Makefile
$ PATH=/usr/bin
$ ./exemple-execvp
Erreur 2
$ PATH=$PATH:/bin
$ ./exemple-execvp
total 28
-rwxrwxr-x 1 500 500 8125 mai 26 15:21 exemple-execve
-rw-rw-r-- 1 500 500 353 mai 26 15:21 exemple-execve.c
-rwxrwxr-x 1 500 500 7731 mai 26 15:27 exemple-execvp
-rw-rw-r-- 1 500 500 232 mai 26 15:27 exemple-execvp.c
-rw-r--r-- 1 500 500 200 mai 26 15:27 Makefile
$
```

La fonction `execvp()` permet de lancer une application qui sera recherchée dans les répertoires mentionnés dans la variable d'environnement `PATH`, en fournissant les arguments sous la forme d'une liste variable terminée par un pointeur `NULL`. Le prototype de `execvp()` est le suivant :

```
| int execvp (const char * application, const char * arg, ...);
```

Cette présentation est plus facile à utiliser que `execvp()` lorsqu'on a un nombre précis d'arguments connus à l'avance. Si les arguments à transmettre sont définis dynamiquement durant le déroulement du programme, il est plus simple d'utiliser un tableau comme avec `execvp()`.

Voici un exemple de programme qui se rappelle lui-même en incrémentant un compteur transmis en argument. Il utilise `argv[0]` pour connaître son nom ; l'argument `argv[1]` contient alors le compteur qu'on incrémente jusqu'à 5 au maximum avant de relancer le même programme.

```
exemple-execvp.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char * argv [])
{
    char compteur[2];
    int i;

    i = 0;
```

```

    if (argc == 2)
        sscanf(argv[1], "%d", & i);

    if (i < 5) {
        i ++;
        sprintf(compteur, "%d", i);

        fprintf(stdout, "execvp(%s, %s, %s, NULL)\n",
                argv[0], argv[0], compteur);
        execvp(argv[0], argv[0], compteur, (char *) NULL);
    }
    return 0;
}

```

```

$ ./exemple_execvp
execvp(/exemple_execvp, ./exemple-execvp, 1, NULL)
execvp(/exemple_execvp, ./exemple-execvp, 2, NULL)
execvp(/exemple_execvp, ./exemple-execvp, 3, NULL)
execvp(/exemple_execvp, ./exemple-execvp, 4, NULL)
execvp(/exemple_execvp, ./exemple-execvp, 5, NULL)
$

```

En Python, il est également nécessaire de fournir séparément l'exécutable et les différents arguments de la ligne de commande.

```

exemple-execvp.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys

i = 0
if len(sys.argv) == 2:
    try:
        i = int(sys.argv[1])
    except ValueError:
        print("Invalid value", sys.argv[1])
        sys.exit(1)
if i < 5:
    print("Counter =", (i+1))
    try:
        os.execvp(sys.argv[0], sys.argv[0], '{0}'.format(i+1))
    except OSError:
        print("Error, unable to run", sys.argv[0])

```


La fonction `exec1()` est identique à `exec1p()`, mais il faut indiquer le chemin d'accès complet, sans recherche dans `PATH`. La fonction `exec1e()` utilise le prototype suivant :

```
int exec1e (const char * app, const char * arg,...,
            const char * envp[]);
```

dans lequel on fournit un tableau explicite pour l'environnement désiré, comme avec `execve()`.

Récapitulons les caractéristiques des six fonctions de la famille `exec()`.

Fonction	Exécutable	Arguments	Environnement
<code>execv()</code>	Chemin complet	Tableau <code>argv[]</code>	Variable externe globale
<code>execve()</code>	Chemin complet	Tableau <code>argv[]</code>	Tableau <code>envp[]</code>
<code>execvp()</code>	Recherche avec <code>PATH</code>	Tableau <code>argv[]</code>	Variable externe globale
<code>exec1()</code>	Chemin complet	Liste terminée par <code>NULL</code>	Variable externe globale
<code>exec1e()</code>	Chemin complet	Liste terminée par <code>NULL</code>	Tableau <code>envp[]</code>
<code>exec1p()</code>	Recherche avec <code>PATH</code>	Liste terminée par <code>NULL</code>	Variable externe globale

Lorsqu'un processus exécute un appel `exec()` et que celui-ci réussit, le nouveau programme remplace totalement l'ancien. Les segments de données, de code, de pile sont réinitialisés. En conséquence, les variables allouées en mémoire sont automatiquement libérées. Les chaînes d'environnement et d'argument sont copiées ; on peut donc utiliser n'importe quel genre de variables (statiques ou allouées dynamiquement, locales ou globales) pour transmettre les arguments de l'appel `exec()`.

L'ancien programme transmet automatiquement au nouveau programme :

- 1 Les *PID* et *PPID*, *PGID* et *SID*. Il n'y a donc pas de création de nouveau processus.
- 2 Les identifiants *UID* et *GID*, sauf si le fichier exécutable du nouveau programme est *Set-UID* ou *Set-GID*. Dans ce cas, seuls les *UID* ou *GID* réels sont conservés, les identifiants effectifs étant mis à jour.
- 3 Le masque des signaux bloqués, et les signaux en attente.
- 4 La liste des signaux ignorés. Un signal ayant un gestionnaire installé reprend son comportement par défaut. Nous discuterons de ce point dans le second volume de ce livre.
- 5 Les descripteurs de fichiers ouverts ainsi que leurs éventuels verrous, sauf si le fichier dispose de l'attribut « *close-on-exec* » ; dans ce cas, il est refermé.

En revanche :

- 1 Les temps d'exécution associés au processus ne sont pas remis à zéro.
- 2 Les privilèges du nouveau programme sont identiques aux privilèges précédents si le fichier exécutable n'est pas (*Set-UID*)

Causes d'échec de lancement d'un programme

Nous avons dit que lorsque l'appel `exec()` réussit, il ne revient pas. Lorsque le programme lancé se finit par `exit()`, `abort()` ou `return` depuis la fonction `main()`, le processus est terminé. Par conséquent, lorsque `exec()` revient dans le processus appelant, une erreur s'est produite. Il est important d'analyser alors le contenu de la variable globale `errno` afin d'expliquer le problème à l'utilisateur. Le détail en est fourni dans la page de manuel de l'appel `exec()` considéré. Voyons les types d'erreurs pouvant se produire :

- Le fichier n'existe pas, n'est pas exécutable, le processus appelant n'a pas les autorisations nécessaires, ou l'interpréteur requis n'est pas accessible : `EACCES`, `EPERM`, `ENOEXEC`, `ENOENT`, `ENOTDIR`, `EINVAL`, `EISDIR`, `ELIBBAD`, `ENAMETOOLONG`, `ELOOP`. Le programme doit alors détailler l'erreur avant de proposer à l'utilisateur une nouvelle tentative d'exécution.
- Le fichier est trop gros, la mémoire manque, ou un problème d'ouverture de fichier se pose : `E2BIG`, `ENOMEM`, `EIO`, `ENFILE`, `EMFILE`. On peut considérer cela comme une erreur critique, où le programme doit s'arrêter, après avoir expliqué le problème à l'utilisateur.
- Un pointeur est invalide : `EFAULT`. Il s'agit d'un bogue de programmation.
- Le fichier est déjà ouvert en écriture : `ETXTBSY`.

Pour pouvoir détailler un peu cette dernière erreur, nous devons nous intéresser quelques instants à la méthode employée par Linux pour gérer la mémoire virtuelle. On retrouvera une explication plus détaillée dans le chapitre 14.

L'espace mémoire dont dispose un processus est découpé en pages. Ces pages mesurent 4 Ko sur les systèmes les plus courants mais peuvent varier suivant les architectures des machines. Les processus ont l'impression d'avoir un espace d'adressage linéaire et continu, mais en réalité le noyau peut déplacer les pages à son gré dans la mémoire physique du système. Une collaboration entre le noyau et le processeur permet d'assurer automatiquement la traduction d'adresse nécessaire lors d'un accès mémoire.

Une page peut également ne pas se trouver en mémoire, mais résider sur le disque. Lorsque le processus tente d'y accéder, le processeur déclenche une « faute de page » et le noyau charge à ce moment la page désirée. Cela permet d'économiser la mémoire physique vraiment disponible.

Parallèlement, lorsque le noyau a besoin de trouver de la place en mémoire, il élimine une ou plusieurs pages qui ont peu de chances d'être utilisées dans un avenir proche. Si la page à supprimer a été modifiée par le processus, il est nécessaire de la sauvegarder sur le disque. Le noyau utilise alors la zone de swap. Si, au contraire, la page n'a pas été changée depuis son premier chargement sur le disque, on peut l'éliminer sans problème, le noyau sait où la retrouver.

Nous découvrons là une grande force de cette gestion mémoire : le code exécutable d'un programme, n'étant jamais modifié par le processus, n'a pas besoin d'être chargé entièrement en permanence. Le noyau peut relire sur le disque les pages de code nécessaires au fur et à mesure de l'exécution du programme. Il faut donc s'assurer qu'aucun autre processus ne risque de modifier le fichier exécutable. Pour cela, le noyau le verrouille, et toute tentative d'ouverture en écriture d'un fichier en cours d'exécution se soldera par un échec.

Un scénario classique pour un développeur met en avant ce phénomène : on utilise simultanément plusieurs consoles virtuelles ou plusieurs *Xterm*, en répartissant l'éditeur de texte sur une fenêtre, le compilateur sur une seconde, et le lancement du programme en cours de travail sur la troisième. Cela permet de relancer la compilation en utilisant simplement la touche de rappel de l'historique du shell, et de redémarrer le programme développé de la même manière dans une autre fenêtre. On apporte une modification au programme, et on oublie de le quitter avant de relancer la compilation. Le compilateur échouera alors en indiquant qu'il ne peut pas écrire sur un fichier exécutable en cours d'utilisation.

De la même façon, il n'est pas possible de lancer un programme dont le fichier est déjà ouvert en écriture par un autre processus. Dans ce cas, l'erreur ETXTBSY se produit. Il est bon dans ce cas de prévenir l'utilisateur. Le message peut même lui indiquer de se reporter à la commande *fuser* pour savoir quel processus a ouvert le fichier en question.

Nous allons mettre en lumière ce principe dans le programme suivant, `exemple-execv`, qui tente – vainement – d'ouvrir en écriture son propre fichier exécutable. Il ouvre ensuite en mode d'ajout en fin de fichier `exemple-execvp` que nous avons créé plus haut. Le fait d'ouvrir ce fichier en mode d'ajout évite de détruire les informations qu'il contient. Il tente alors de l'exécuter.

```
exemple-execv.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, char * argv[])
{
    int fd;

    char * nv_argv[] = { "./exemple-execvp", (char *) NULL };

    fprintf(stdout, "Essai d'ouverture de %s\n", argv[0]);
```

```
if ((fd = open(argv[0], O_WRONLY | O_APPEND)) < 0) {
    if (errno != ETXTBSY) {
        fprintf(stdout, " impossible, errno %d\n", errno);
        exit(1);
    }
    fprintf(stdout, " echec ETXTBSY, fichier deja utilise \n");
}

fprintf(stdout, "Ouverture exemple-execvp en ecriture \n");

if ((fd = open("exemple-execvp", O_WRONLY | O_APPEND)) < 0) {
    fprintf(stdout, " impossible, errno %d\n", errno);
    exit(1);
}

fprintf(stdout, " Ok\n");
fprintf(stdout, "Essai d'executer exemple-execvp\n");
execv("./exemple-execvp", nv_argv);

if (errno == ETXTBSY)
    fprintf(stdout, " echec ETXTBSY fichier deja utilise\n");
else
    fprintf(stdout, " errno = %d\n", errno);
return 1;
}
```

Comme on pouvait s'y attendre, le programme n'arrive pas à ouvrir en écriture un fichier en cours d'exécution ni à lancer un programme dont le fichier est ouvert.

```
$ ./exemple-execv
Essai d'ouverture de ./exemple-execv
echec ETXTBSY, fichier deja utilise
Ouverture exemple-execvp en ecriture
Ok
Essai d'executer exemple-execvp
echec ETXTBSY fichier deja utilise
$
```

Cette expérience fonctionnerait différemment avec un script Python, car au démarrage l'interpréteur charge tout le contenu du script sans passer par le mécanisme de projection en mémoire. Le fichier script n'est donc pas verrouillé pendant l'exécution et pourrait très bien être modifié.

Fonctions simplifiées pour exécuter un sous-programme

Il y a de nombreux cas où on désire lancer une commande externe au programme, sans pour autant remplacer le processus en cours. On peut par exemple avoir une application principale qui lance des sous-programmes indépendants, ou désirer faire appel à une commande système. Dans ce dernier cas, on peut classiquement invoquer la commande `mail` pour transmettre un message à l'utilisateur, à l'administrateur, ou envoyer un rapport de bogue au concepteur du programme.

Pour cela, nous disposons de la fonction `system()` et de la paire `popen()` / `pclose()`, qui sont implémentées dans la bibliothèque C en invoquant `fork()` et `exec()` selon les besoins. Ces fonctions sont disponibles en Python et nous les examinerons ci-après, néanmoins il convient de mentionner que les fonctionnalités offertes par le module `subprocess` sont plus adaptées pour réaliser ces tâches indépendamment du système d'exploitation.

La fonction `system()` est déclarée ainsi dans `<stdlib.h>` :

```
| int system (const char * commande);
```

Cette fonction invoque le shell en lui transmettant la commande fournie, puis revient après la fin de l'exécution. Pour ce faire, il faut exécuter un `fork()`, puis le processus lance la commande en appelant le shell « `/bin/sh -c commande` », tandis que le processus père attend la fin de son fils. Si l'invocation du shell échoue, `system()` renvoie 127. Si une autre erreur se produit, elle renvoie -1, sinon elle renvoie la valeur de retour de la commande exécutée.

En voici un exemple en Python.

```
| exemple-system.py :  
| #! /usr/bin/python  
| import os;  
  
| os.system("ls -l")
```

À l'exécution, notre script lance bien la commande `ls` avec l'argument `-l` :

```
$ ./exemple-system.py  
total 60  
-rw-r--r-- 1 cpb cpb 708 janv. 11 13:32 exemple-exec1p.c  
-rwxrwxr-x 1 cpb cpb 336 janv. 28 13:00 exemple-exec1p.py  
-rw-r--r-- 1 cpb cpb 1259 janv. 16 19:06 exemple-execv.c  
[...]  
-rw-r--r-- 1 cpb cpb 405 janv. 11 13:32 exemple-system.c
```

```
-rwxr-xr-x 1 cpb cpb 53 janv. 28 17:00 exemple-system.py
-rwxr-xr-x 1 cpb cpb 351 janv. 11 13:32 ls
-rw-r--r-- 1 cpb cpb 359 janv. 17 09:45 Makefile
```

Une manière simplifiée d'implémenter `system()` pourrait être la suivante :

```
int notre_system(const char * commande)
{
    char * argv[4];
    int retour;
    pid_t pid;

    if ((pid = fork()) < 0)
        /* Erreur dans fork */
        return -1;

    if (pid == 0) {
        /* Processus fils */
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = commande;
        argv[3] = (char *) NULL;
        execv("/bin/sh", argv);
        /* Execv a échoué */
        exit(127);
    }

    /* Processus père */
    /* Attente de la fin du processus fils */
    while (waitpid(pid, & retour, 0) < 0)
        if (errno != EINTR)
            return -1;
    return retour;
}
```

Attention : la fonction `system()` représente une énorme faille de sécurité dans toute application installée *Set-UID*. Ceci ne concerne que les programmes exécutables binaires (compilés), pas les scripts puisque dans leur cas, le bit *Set-UID* n'est pas pris en considération. Voyons le programme simple suivant :

```
exemple-system.c :
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
```

```
system("ls");  
return 0;  
}
```

Le programme ne fait que demander au shell d'exécuter "ls". Pourtant, si on l'installe *Set-UID root*, il s'agit d'une faille de sécurité. En effet, lorsque le shell recherche la commande "ls", il parcourt les répertoires mentionnés dans la variable d'environnement PATH. Celle-ci est héritée du processus père et peut donc être configurée par l'utilisateur pour inclure en premier le répertoire ".". Le shell exécutera alors de préférence la commande "ls" qui se trouve dans le répertoire en cours. Il suffit que l'utilisateur crée un *shell script* exécutable, et le tour est joué. Voyons un exemple, en créant le *shell script* suivant :

```
ls :  
#!/bin/sh  
echo faux ls  
echo qui lance un shell  
sh
```

Examinons l'exécution suivante :

```
$ ./exemple-system  
exemple-execlp  exemple-execv.c  exemple-execvp  
exemple-system.c exemple-execlp.c exemple-execve  
exemple-execvp.c exemple-execv  exemple-execve.c  
exemple-system  ls  
$ export PATH=.:$PATH  
$ ./exemple-system  
faux ls  
qui lance un shell  
sh-3.2 $ exit  
$
```

Tout d'abord, le programme s'exécute normalement et invoque « sh -c ls », qui trouve ls dans le répertoire /bin comme d'habitude. Ensuite, nous modifions notre PATH pour y placer en première position le répertoire en cours. À ce moment, le shell exécutera notre "ls" piégé qui lance un shell. Jusque-là, rien d'inquiétant. Mais imaginons maintenant que le programme soit *Set-UIDroot*. C'est ce que nous configurons avant de revenir en utilisateur normal.

```
$ su
Password:
# chown root.root exemple-system
# chmod +s exemple-system
# exit
$
```

À ce moment, l'exécution du programme lance le "ls" piégé avec l'identité de root !

```
$ ./exemple-system
faux ls
qui lance un shell
#
```

Comme nous avons inclus dans notre script une invocation de shell, nous nous retrouvons avec un shell connecté sous root ! Il ne faut pas s'imaginer que le fait de forcer la variable d'environnement PATH dans le programme aurait résolu le problème. D'autres failles de sécurité classiques existent, notamment en faussant la variable d'environnement IFS qui permet au shell de séparer ses arguments (normalement des espaces, des tabulations, etc.).

Il ne faut donc jamais employer la fonction `system()` dans un programme *Set-UID* (ou *Set-GID*). On peut utiliser à la place les fonctions `exec()`, qui ne parcourent pas les répertoires du PATH. Le vrai danger avec `system()` est qu'il appelle le shell au lieu de lancer la commande directement.

La véritable version de `system()`, présente dans la Glibc, est légèrement plus complexe puisqu'elle gère les signaux SIGINT et SIGQUIT (en les ignorant) et SIGCHLD (en le bloquant).

En théorie, le fait de transmettre une commande NULL sert à vérifier la présence du shell/bin/sh. Normalement, `system()` doit renvoyer une valeur non nulle s'il est bien là. En pratique, sous Linux, la vérification n'a pas lieu, la Glibc considère que /bin/sh appartient au minimum vital d'un système Unix.

Après avoir bien compris que la fonction `system()` ne doit jamais être employée dans un programme *Set-UID* ou *Set-GID*, rien n'empêche de l'utiliser dans des applications simples ne nécessitant pas de privilèges. L'exemple que nous invoquons précédemment concernant l'appel de l'utilitaire `mail` est pourtant difficile à utiliser avec la fonction `system()`, car il faudrait d'abord créer un fichier contenant le message, puis lancer `mail` avec une redirection d'entrée.

Pour cela, il est plus pratique d'utiliser la fonction `popen()`, qui permet de lancer un programme à la manière de `system()`, mais en fournissant un des flux d'entrée ou de sortie standard pour dialoguer avec le programme appelant.

Le prototype de cette fonction, dans `<stdio.h>`, est le suivant :

```
| FILE * popen (const char * commande, const char * mode);
```

La commande est exécutée comme avec `system()` en invoquant `fork()` et `exec()`, mais, de plus, le flux d'entrée ou de sortie standard de la commande est renvoyé au processus appelant. La chaîne de caractères `mode` doit contenir soit « r » (*read*), si on souhaite lire les données de la sortie standard de la commande dans le flux renvoyé, « w » (*write*) si on préfère écrire sur son entrée standard. Le flux renvoyé par la fonction `popen()` est tout à fait compatible avec les fonctions d'entrée-sortie classiques telles `fprintf()`, `fscanf()`, `fread()` ou `fwrite()`. Par contre, le flux doit toujours être refermé en utilisant la fonction `pclose()` à la place de `fclose()`.

Lorsqu'on appelle `pclose()`, cette fonction attend que le processus exécutant la commande se termine, puis renvoie son code de retour.

Voici un exemple simple dans lequel nous avons exécuté la commande "mail", suivie de notre nom d'utilisateur obtenu avec `getlogin()`. La commande est exécutée en redirigeant son flux d'entrée standard. Nous pouvons donc écrire notre message tranquillement par une série de `fprintf()`.

```
| exemple-popen-1.c :  
| #include <stdio.h>  
| #include <stdlib.h>  
| #include <string.h>  
| #include <unistd.h>  
| #include <errno.h>  
  
| int main(void)  
| {  
|     FILE * message;  
|     char * commande;  
  
|     if ((commande = malloc(strlen(getlogin()) + 6)) == NULL) {  
|         fprintf(stderr, "Erreur malloc %d\n", errno);  
|         exit(1);  
|     }  
  
|     strcpy(commande, "mail ");  
|     strcat(commande, getlogin());  
  
|     if ((message = popen(commande, "w")) == NULL) {  
|         fprintf(stderr, " Erreur popen %d \n", errno);  
|         exit(1);  
|     }  
| }
```

```
fprintf(message, "Ceci est un message \n");  
fprintf(message, "emis par moi-meme\n");  
pclose(message);  
return 0;  
}
```

Lorsqu'il est lancé, ce programme émet bien le mail prévu. On notera que `popen()` effectue, comme `system()`, un `exec1()` de `/bin/sh -c commande`. Cette commande est donc recherchée dans les répertoires mentionnés dans le `PATH`.

Une autre application classique de `popen()`, utilisant l'entrée standard de la commande exécutée, est d'invoquer le programme indiqué dans la variable d'environnement `PAGER`, ou si elle n'existe pas, `less` ou `more`. Ces utilitaires affichent page par page les données qu'on leur envoie, en s'occupant de gérer la taille de l'écran (`less` permet même de revenir en arrière). C'est un moyen simple et élégant de fournir beaucoup de texte à l'utilisateur en lui laissant la possibilité de le consulter à sa guise.

Notre second exemple va lire la sortie standard de la commande exécutée. C'est une méthode généralement utilisée pour récupérer les résultats d'une application complémentaire ou pour invoquer une commande système qui fournit des données difficiles à obtenir directement (`who`, `ps`, `last`, `netstat`...).

Nous allons ici invoquer la commande `ifconfig` en lui demandant l'état de l'interface réseau `eth0`. Si celle-ci est activée, `ifconfig` renvoie une sortie du genre :

```
eth0 Lien encap:Ethernet HWaddr 00:50:04:8C:7A:ED  
  inet adr:172.16.15.16 Bcast:172.16.255.255 Masque:255.255.0.0  
  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1  
  Paquets Reçus:0 erreurs:0 jetés:0 débordements:7395 trames:0  
  Paquets transmis:29667 erreurs:0 jetés:0 débordements:0  
  collisions:7395 lg file transmission:100  
  Interruption:3 Adresse de base:0x200
```

Si `eth0` est désactivée, on obtient :

```
eth0 Lien encap:Ethernet HWaddr 00:50:04:8C:7A:ED  
  inet adr:172.16.15.16 Bcast:172.16.255.255 Masque:255.255.0.0  
  BROADCAST MULTICAST MTU:1500 Metric:1  
  Paquets Reçus:0 erreurs:0 jetés:0 débordements:11730 trames:0  
  Paquets transmis:47058 erreurs:0 jetés:0 débordements:0  
  collisions:11730 lg file transmission:100  
  Interruption:3 Adresse de base:0x200
```

(Remarquez la différence dans la troisième ligne, UP dans un cas, et pas dans l'autre.) Si l'interface n'existe pas, `ifconfig` ne renvoie rien sur sa sortie standard, mais écrit un message :

```
eth0: erreur lors de la recherche d'infos sur l'interface:
Périphérique non trouvé
```

sur sa sortie d'erreur.

Notre programme va donc lancer la commande et rechercher si une ligne de la sortie standard commence par UP. Si c'est le cas, il indique que l'interface est active. S'il ne trouve pas cette chaîne de caractères ou si la commande ne renvoie aucune donnée sur sa sortie standard, il considère l'interface comme étant inactive.

```
exemple-popen-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

int main(void)
{
    FILE * sortie;
    char ligne [128];
    char etat [128];

    if ((sortie = popen("/sbin/ifconfig eth0", "r")) == NULL) {
        fprintf(stderr, " Erreur popen %d \n", errno);
        exit(1);
    }
    while (fgets(ligne, 127, sortie) != NULL) {
        if (sscanf(ligne, "%s", etat) == 1)
            if (strcmp(etat, "UP") == 0) {
                fprintf(stderr, "interface eth0 en marche \n");
                pclose(sortie);
                return 0;
            }
    }
    fprintf(stdout, "interface eth0 inactive \n");
    pclose(sortie);
    return 0;
}
```

Cet exemple (un peu artificiel, convenons-en) montre quand même l'utilité d'invoquer une commande système et d'en récupérer aisément les informations. Encore

une fois, insistons sur le manque de sécurité qu'offre `popen()` pour un programme susceptible d'être installé *Set-UID* ou *Set-GID*.

Un dernier exemple concernant `popen()` nous permet d'invoquer un script associé à l'application principale. Ce script est écrit en Python et s'appuie sur la bibliothèque graphique TkInter pour offrir une boîte de saisie configurable. Il utilise les chaînes de caractères transmises en argument en ligne de commande :

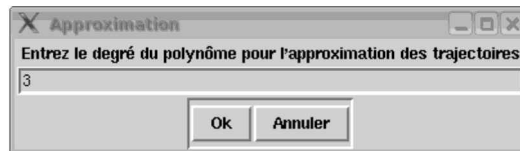
- Le premier argument correspond au nom de la boîte de saisie (le titre de la fenêtre).
- Le second argument est le libellé affiché pour questionner l'utilisateur.
- Le troisième argument (éventuel) est la valeur par défaut pour la zone de saisie.

En invoquant ainsi ce script dans un *Xterm* :

```
$ ./exemple-popen-3.py Approximation "Entrez le degré du polynôme  
pour l'approximation des trajectoires" 3
```

La fenêtre suivante apparaît.

Figure 3-1
Fenêtre de saisie en Python



Lorsqu'on appuie sur le bouton Ok, la valeur saisie est affichée sur la sortie standard.

Le script Python est volontairement simplifié ; il ne traite aucun cas d'erreur.

```
exemple-popen-3.py :  
#!/usr/bin/python  
from __future__ import print_function  
import Tkinter  
import sys  
  
class DialogBox:  
    def __init__(self, window, title, label, default_value):  
        self.window = window  
        self.window.title(title)  
        Tkinter.Label(self.window, text=label).pack(padx=5, pady=5)  
        self.entry = Tkinter.Entry(self.window)  
        self.entry.pack(padx=5, pady=5)  
        self.entry.insert(0, default_value)  
        self.default_value = default_value
```

```
frame = Tkinter.Frame(self.window)
frame.pack()
bt = Tkinter.Button(frame, text="Ok",
                    command=self.bt_ok)
bt.pack(padx=5, pady=5, side=Tkinter.LEFT)
bt = Tkinter.Button(frame, text="Cancel",
                    command=self.bt_cancel)
bt.pack(padx=5, pady=5, side=Tkinter.LEFT)

def bt_ok(self):
    print(self.entry.get())
    self.window.destroy()

def bt_cancel(self):
    print(self.default_value)
    self.window.destroy()

if len(sys.argv) < 4:
    print("usage: " + sys.argv[0] + " title label default")
    sys.exit(1)

root_window = Tkinter.Tk()
dialog_box = DialogBox(root_window, sys.argv[1], sys.argv[2],
                        sys.argv[3])
root_window.wait_window(dialog_box.window)
```

Notre programme C va invoquer le script et traiter quelques cas d'échec, notamment en testant le code de retour de `pclose()`. Si une erreur se produit, on effectue la saisie à partir de l'entrée standard du processus. Ceci permet d'utiliser le même programme dans un environnement *X-Window* avec une boîte de dialogue ou sur une console texte avec une saisie classique.

La ligne de commande que `popen()` invoque est la suivante :

```
./exemple-popen-3.tk py Saisie "Entrez votre nom" nom_login 2> /dev/null
```

dans laquelle `nom_login` est obtenu par la commande `getlogin()`.

On redirige la sortie d'erreur standard vers `/dev/null` afin d'éviter les éventuels messages d'erreur de Python si on se trouve sur une console texte (on suppose que le shell `/bin/sh` utilisé par `popen()` est du type Bourne, ce qui est normalement le cas sous Linux). La chaîne *"Entrez votre nom"* est encadrée par des guillemets pour qu'elle ne constitue qu'un seul argument de la ligne de commande.

Voici le programme C qui invoque le script décrit précédemment :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (void) {
    FILE * saisie;
    char * login;
    char  nom [128];
    char  commande [128];

    if ((login = getlogin()) == NULL)
        strcpy(nom, "\\");
    else
        strcpy(nom, login);
    sprintf(commande, "./exemple-popen-3.py "
        "Saisie "
        "\\\"Entrez votre nom\\" "
        "%s 2>/dev/null", nom);
    exemple-popen-3.c :
    if ((saisie = popen(commande , "r")) == NULL) {
        // Le script est, par exemple, introuvable
        // On va essayer de lire sur stdin.
        fprintf(stdout, "Entrez votre nom : ");
        if (fscanf(stdin, "%s", nom) != 1) {
            // La lecture sur stdin échoue...
            // On utilise une valeur par défaut.
            strcpy(nom, getlogin());
        }
        fprintf(stdout, "Nom saisi : %s\\n", nom);
        return 0;
    }

    if (fscanf(saisie, "%s", nom) != 1) {
        if (pclose(saisie) != 0) {
            // Le script a échoué pour une raison quelconque.
            // On recommence la saisie sur stdin.
            fprintf(stdout, "Entrez votre nom : ");
            if (fscanf(stdin, "%s", nom) != 1) {
                // La lecture sur stdin échoue...
                // On utilise une valeur par défaut.
                strcpy(nom, getlogin());
            }
        } else {
            // L'utilisateur a cliqué sur Annuler. Il faut
            // abandonner l'opération en cours.
            fprintf(stdout, "Pas de nom fourni - abandon\\n");
        }
    }
}
```

```
        return 1;
    }
    } else {
        pclose(saisie);
    }
    fprintf(stdout, "Nom saisi : %s\n", nom);
    return 0;
}
```

Conclusion

Ce chapitre nous a permis de découvrir plusieurs méthodes pour lancer une application. Les mécanismes à base de `exec()` permettent de remplacer totalement le programme en cours par un autre qui est exécutable, tandis que les fonctions `system()` et `popen()/pclose()` servent plutôt à utiliser une autre application comme sous-programme de la première.

Dans le prochain chapitre, nous nous intéresserons à l'environnement d'un processus, et aux configurations possibles à la création du processus ou lors de l'exécution d'un nouveau programme.

4

Environnement et ligne de commande

Une application peut être exécutée, sous Linux, dans des contextes très différents. Il existe une multitude de types de terminaux, et le répertoire personnel de l'utilisateur peut se trouver à n'importe quel endroit du système de fichiers (par exemple, les utilisateurs spéciaux `mail`, `mysql` ou `apache`). De plus, la plupart des applications permettent une configuration de leur interface en fonction des préférences de l'utilisateur.

Il est donc souvent nécessaire d'avoir accès à différents paramètres de l'environnement dans lequel s'exécute un programme. Pour cela, les systèmes Linux offrent une manière assez élégante de transmettre aux applications des informations relatives aussi bien au système en général (type de système d'exploitation, nom de l'hôte...) qu'à l'utilisateur lui-même (emplacement du répertoire personnel, langage utilisé, fichier contenant le courrier en attente), voire aux paramètres n'ayant trait qu'à la session en cours (type de terminal...).

Nous allons voir d'abord les moyens d'accéder aux variables d'environnement, ainsi qu'une liste des variables les plus couramment utilisées. Nous étudierons ensuite l'accès aux arguments en ligne de commande, comprenant aussi bien les options simples, à la manière `SUSv4`, que les options longues *Gnu*. Enfin, nous observerons un exemple complet de paramétrage d'une application en fonction de son environnement d'exécution.

Variables d'environnement

Les variables d'environnement sont définies sous la forme de chaînes de caractères contenant des affectations du type `NOM=VALEUR`. Ces variables sont accessibles aux processus, tant dans les programmes en langage C que dans les scripts shell, par exemple. Lors de la duplication d'un processus avec un `fork()`, le fils hérite d'une copie des variables d'environnement de son père. Un processus peut modifier, créer ou détruire des variables de son propre environnement, et donc de celui des processus fils à venir, mais en aucun cas il ne peut intervenir sur l'environnement de son père.

Un certain nombre de variables sont automatiquement initialisées par le système lors de la connexion de l'utilisateur. D'autres sont mises en place par les fichiers d'initialisation du shell, d'autres enfin peuvent être utilisées temporairement dans des scripts shell avant de lancer une application.

Lorsqu'un programme C démarre, son environnement est automatiquement copié dans un tableau de chaînes de caractères. Ce tableau est disponible dans la variable globale `environ`, à déclarer ainsi en début de programme (elle n'est pas déclarée dans les fichiers d'en-tête courants) :

```
| extern char ** environ;
```

Ce tableau contient des chaînes de caractères terminées par un caractère nul, et se finit lui-même par un pointeur nul. Chaque chaîne a la forme `NOM=VALEUR`, comme nous l'avons précisé. Voici un exemple de balayage de l'environnement.

```
| exemple-envIRON.c :  
| #include <stdio.h>  
  
| extern char ** environ;  
  
| int main (void)  
| {  
|     int i = 0;  
  
|     for (i = 0; environ[i] != NULL; i ++)  
|         fprintf(stdout, "%d : %s\n", i, environ[i]);  
  
|     return 0;  
| }
```

Voici un exemple d'exécution (raccourci) :

```
$ ./exemple-environ
0 : SSH_AGENT_PID=18628
1 : HOSTNAME=vega
2 : DESKTOP_STARTUP_ID=
3 : TERM=xterm
4 : SHELL=/bin/bash
5 : HISTSIZE=1000
6 : KDE_NO_IPV6=1
7 : GTK_RC_FILES=/etc/gtk/gtkrc:/home/cpb/.gtkrc-1.2-gnome2
8 : WINDOWID=58722617
9 : USER=cpb
[...]
24 : LANG=fr_FR.UTF-8
25 : SHLV=1
26 : HOME=/home/cpb
27 : LOGNAME=cpb
28 : CVS_RSH=ssh
29 : _=./exemple_environ
```

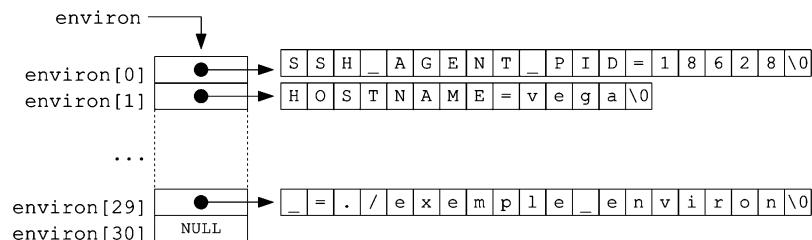
L'interpréteur Python fournit un dictionnaire `os.environ` dont on peut parcourir les clés pour obtenir un résultat similaire.

```
exemple-environ.py :
#!/usr/bin/python
from __future__ import print_function
import os

for v in os.environ:
    print(v + "=" + os.environ[v])
```

La représentation en mémoire du tableau `environ` du langage C est celle de la figure 4-1.

Figure 4-1
Variables d'environnement
d'un processus



Notons que le tableau d'environnement est également fourni comme troisième argument à la fonction `main()`, comme les options de ligne de commande `argc` et `argv`, que nous verrons plus bas. La norme SUSv4 recommande, pour des raisons de portabilité, d'éviter d'utiliser cette possibilité et de lui préférer la variable globale `environ`. Voici toutefois un exemple qui fonctionne parfaitement sous Linux.

```
exemple-environ-2.c :
#include <stdio.h>

int main (int argc, char * argv[], char * envp[])
{
    int i = 0;

    for (i = 0; envp[i] != NULL; i++)
        fprintf(stdout, "%d : %s\n", i, envp[i]);

    return 0;
}
```

On peut parfois avoir besoin de balayer le tableau `environ`, mais c'est assez rare, car les applications ne s'intéressent généralement qu'à un certain nombre de variables bien précises. En Python, l'accès aux variables d'environnement est très simple car il s'agit d'une structure classique de dictionnaire. En C, c'est un peu plus compliqué, mais des fonctions de bibliothèque donnent accès aux variables d'environnement afin de pouvoir en ajouter, en détruire, ou en consulter le contenu.

Précisons tout de suite que les chaînes de caractères attendues par les routines de la bibliothèque sont de la forme `NOM=VALEUR`, où il ne doit pas y avoir d'espace avant le signe égal (=). En fait, un espace présent à cet endroit serait considéré comme faisant partie du nom de la variable. Notons également que la différenciation entre minuscules et majuscules est prise en compte dans les noms de variables. Les variables d'environnement ont des noms traditionnellement écrits en majuscules (bien que cela ne soit aucunement une obligation), et une chaîne `Home=...` n'est pas considérée comme étant équivalente à `HOME=...`

La routine `getenv()` est déclarée dans `<stdlib.h>`, ainsi :

```
char * getenv (const char * nom);
```

Elle permet de rechercher une variable d'environnement. On lui donne le nom de la variable désirée, et elle renvoie un pointeur sur la chaîne de caractères suivant immédiatement le signe = dans l'affectation `NOM=VALEUR`. Si la variable n'est pas trouvée, la routine renvoie un pointeur `NULL`.

Avec la Glibc, cette routine renvoie directement un pointeur sur la chaîne de l'environnement du processus. Elle n'effectue pas de copie de la chaîne d'environnement. Aussi, toute modification apportée sur la chaîne renvoyée affectera directement l'environnement du processus comme si on modifiait la variable globale `environ`. D'autres implémentations peuvent préférer recopier le contenu de la chaîne dans une variable globale statique, écrasée à chaque appel.

Précisons bien que la norme SUSv4 interdit toute modification sur le pointeur renvoyé et indique qu'il faut faire une copie de la chaîne renvoyée si on désire la réutiliser par la suite.

Remarquons également que cette fonction n'est pas nécessairement réentrante, ni sûre dans un contexte *multithread*. Nous verrons dans le chapitre 6 les mécanismes de protection (*mutex* par exemple) permettant de l'utiliser en toute sécurité.

Une variable peut être définie sans avoir de valeur (`NOM=`). Dans ce cas, la routine `getenv()` renverra un pointeur sur une chaîne vide.

```
exemple-getenv.c :
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char * argv[])
{
    int    i;
    char * variable;

    if (argc == 1) {
        fprintf(stderr, "Utilisation : %s variable...\n", argv[0]);
        return 1;
    }
    for (i = 1; i < argc; i++) {
        variable = getenv(argv[i]);
        if (variable == NULL)
            fprintf(stdout, "%s : non definie\n", argv[i]);
        else
            fprintf(stdout, "%s : %s\n", argv[i], variable);
    }
    return 0;
}
```

Ce programme permet de tester la valeur des variables d'environnement dont on lui transmet le nom sur la ligne de commande. Nous étudierons plus loin le fonctionnement des arguments `argc` et `argv`.

```
$ ./exemple_getenv HOME LANG SHELL INEXISTANTE
HOME : /home/cpb
LANG : fr_FR.UTF-8
SHELL : /bin/bash
INEXISTANTE : non definie
$
```

L'équivalent en Python est plus concis :

```
/exemple-getenv.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys

sys.argv.pop(0)
for var in sys.argv:
    try:
        print(var + "=" + os.environ[var])
    except:
        print(var, "non definie")
```

Pour tester nos programmes, il est intéressant de voir comment remplir les variables d'environnement au niveau du shell. Cela dépend bien entendu du type d'interpréteur de commandes utilisé. Certains shells font une différence entre leurs propres variables (qu'on utilise pour stocker des informations dans les scripts) et les variables de l'environnement qui seront transmises aux processus fils. Voici les syntaxes pour les principaux interpréteurs de commandes utilisés sous Linux.

Avec les shells *bash* ou *ksh* :

- Assignment d'une variable du shell : `NOM=VALEUR`
- Visualisation d'une variable du shell : `echo $NOM`
- Visualisation de toutes les variables définies (internes au shell et environnement) :
`set`
- Visualisation des variables d'environnement : `env`
- Exportation de la variable vers l'environnement :
`export NOM`
ou directement : `export NOM=VALEUR`
- Destruction d'une variable : `unset NOM`
- Avec le shell *tcsch* :
- Assignment d'une variable pour le shell uniquement : `set NOM=VALEUR`
- Assignment d'une variable de l'environnement : `setenv NOM VALEUR`

- Visualisation de la valeur d'une variable de l'environnement : `printenv NOM`
- Destruction d'une variable d'environnement : `unsetenv NOM`

Les exemples que nous donnerons seront réalisés avec *bash*, mais on pourra facilement les transformer pour d'autres shells.

```
$ ESSAI=UN
$ ./exemple-getenv.py ESSAI
ESSAI non definie
$ export ESSAI
$ ./exemple-getenv.py ESSAI
ESSAI=UN
$ unset ESSAI
$ ./exemple-getenv.py ESSAI
ESSAI non definie
$ export ESSAI=DEUX
$ export VIDE=
$ ./exemple-getenv.py ESSAI VIDE
ESSAI=DEUX
VIDE=
```

Les routines `putenv()` et `setenv()` servent à créer une variable d'environnement ou à en modifier le contenu. Elles sont toutes deux déclarées dans `<stdlib.h>` :

```
| int putenv (const char * chaine);
| int setenv (const char * nom, const char * valeur, int ecraser);
```

La fonction `putenv()` ne prend qu'un seul argument, une chaîne du type `NOM=VALEUR`, et fait appel à `setenv()` après avoir séparé les deux éléments de l'affectation.

La routine `setenv()` prend trois arguments : les deux premiers sont les chaînes `NOM` et `VALEUR`, et le troisième est un entier indiquant si la variable doit être écrasée dans le cas où elle existe déjà. Le fait d'utiliser un troisième argument nul permet de configurer, en début d'application, des valeurs par défaut, qui ne seront prises en compte que si la variable n'est pas déjà remplie.

Ces deux routines renvoient zéro si elles réussissent, ou -1 s'il n'y a pas assez de mémoire pour créer la nouvelle variable.

La routine `unsetenv()` permet de supprimer une variable :

```
| void unsetenv (const char * nom);
```

Cette routine recherche la variable dont le nom lui est transmis, l'efface si elle la trouve, et ne renvoie rien.

Un effet de bord – discutable – de la fonction `putenv()`, fournie par la bibliothèque Glibc, est le suivant : si la chaîne transmise à `putenv()` ne contient pas de signe égal (=), cette dernière est considérée comme le nom d'une variable, qui est alors supprimée de l'environnement en invoquant `unsetenv()`. En Python, pour supprimer une variable du tableau `os.environ`, on utilisera le mot-clé `del`.

Les routines `getenv()`, `setenv()` et `unsetenv()` de la bibliothèque Glibc balayent le tableau d'environnement pour rechercher la variable désirée en utilisant la fonction `strncmp()`. Elles sont donc sensibles, comme nous l'avons déjà précisé, aux différences entre majuscules et minuscules dans les noms de variables.

Notons l'existence, avec Glibc, d'une routine `clearenv()`, déclarée dans `<stdlib.h>`. Cette routine n'a finalement pas été définie dans la norme SUSv4 et reste donc d'une portabilité limitée. Elle sert à effacer totalement l'environnement du processus appelant (ce qui présente vraiment peu d'intérêt pour une application classique).

Les modifications apportées par un programme C ne jouent que dans son environnement – et celui de ses futurs et éventuels descendants –, mais pas dans celui de son processus père (le shell). Pour visualiser l'action des routines décrites précédemment, nous devons donc écrire un programme un peu plus long que d'habitude.

```
exemple-putenv.c :
#include <stdio.h>
#include <stdlib.h>

void recherche_variable (char * nom);

int main (void)
{
    fprintf(stdout, "\n--- test de putenv() --- \n");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI=UN\");\n");
    putenv("ESSAI=UN");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI=\");\n");
    putenv("ESSAI=");
    recherche_variable("ESSAI");
    fprintf(stdout, "putenv(\"ESSAI\"); equivaut à unsetenv()\n");
    putenv("ESSAI");
    recherche_variable("ESSAI");

    fprintf(stdout, "\n--- test de setenv() --- \n");
    recherche_variable("ESSAI");
    fprintf(stdout, "setenv(\"ESSAI\", \"DEUX\", 1);\n");
```



```

setenv("ESSAI", "DEUX", 1);
recherche_variabile("ESSAI");
fprintf(stdout, "setenv(\"ESSAI\", \"TROIS\", 1);\n");
setenv("ESSAI", "TROIS", 1);
recherche_variabile("ESSAI");
fprintf(stdout, "setenv(\"ESSAI\", \"QUATRE\", 0);"
           " ecrasement de valeur non autorise\n");
setenv("ESSAI", "QUATRE", 0);
recherche_variabile("ESSAI");

fprintf(stdout, "\n-- test de unsetenv() -- \n");
recherche_variabile("ESSAI");
fprintf(stdout, "unsetenv(\"ESSAI\");\n");
unsetenv("ESSAI");
recherche_variabile("ESSAI");

return 0;
}

void recherche_variabile (char * nom)
{
    char * valeur;
    fprintf(stdout, "    variable %s ", nom);
    valeur = getenv(nom);
    if (valeur == NULL)
        fprintf(stdout, "inexistante\n");
    else
        fprintf(stdout, "= %s\n", valeur);
}

```

Et voici un exemple d'exécution :

```

$ ./exemple_putenv

--- test de putenv() ---
    variable ESSAI inexistante
putenv("ESSAI=UN");
    variable ESSAI = UN
putenv("ESSAI=");
    variable ESSAI =
putenv("ESSAI"); équivaut à unsetenv()
    variable ESSAI inexistante

--- test de setenv() ---
    variable ESSAI inexistante
setenv("ESSAI", "DEUX", 1);
    variable ESSAI = DEUX

```

```
setenv("ESSAI", "TROIS", 1);
    variable ESSAI = TROIS
setenv("ESSAI", "QUATRE", 0); écrasement de valeur non autorisé
    variable ESSAI = TROIS

-- test de unsetenv() --
    variable ESSAI = TROIS
unsetenv("ESSAI");
    variable ESSAI inexistante
$
```

Le script Python suivant est beaucoup plus court mais il met en évidence les mêmes éléments : recherche, ajout, écrasement et suppression de variable d'environnement.

```
exemple-putenv.py :
#!/usr/bin/python
from __future__ import print_function
import os

def recherche_variable(var):
    try:
        print(" variable", var + "=" + os.environ[var])
    except:
        print(" variable", var, "non definie")

recherche_variable("ESSAI")
print('os.environ["ESSAI"] = "UN"')
os.environ["ESSAI"] = "UN"
recherche_variable("ESSAI")
print('os.environ["ESSAI"] = "DEUX"')
os.environ["ESSAI"] = "DEUX"
recherche_variable("ESSAI")
print('del os.environ["ESSAI"]')
del os.environ["ESSAI"]
recherche_variable("ESSAI")
```

Variables d'environnement couramment utilisées

Un certain nombre de variables d'environnement sont toujours disponibles sur les machines Linux et peuvent être employées par les applications désirant s'informer sur le système dans lequel elles s'exécutent. Pour voir comment l'environnement des processus est constitué, il est intéressant de suivre leur héritage depuis le démarrage du système.

À tout seigneur tout honneur, le noyau lui-même commence par remplir l'environnement du processus initial (qui deviendra ensuite *init*) avec les chaînes suivantes :

```
HOME=/  
TERM=linux
```

Le noyau recherche le fichier `init` dans les emplacements successifs suivants : `/sbin/init`, `/etc/init` et `/bin/init`. Puis, il le lance.

Ce programme `init` configure plusieurs variables d'environnement comme :

```
PATH=/usr/local/sbin:/sbin:/bin:/usr/sbin:/usr/bin  
CONSOLE=périphérique console
```

Suivant les systèmes Linux, il existe deux approches différentes pour le boot. Sur les systèmes utilisant un `init` de type Système V (les distributions traditionnelles, les systèmes embarqués, etc.), ce dernier lancera les scripts de démarrage des différents services du système. Sur les systèmes récents (orientés vers le poste de travail), `init` fait partie du démon `systemd` qui englobe de nombreux services.

Quel que soit le mécanisme de connexion (console texte, graphique, distant), on peut supposer que l'utilitaire d'authentification (`/bin/login` en général) va remplir :

```
HOME=répertoire de l'utilisateur (lu dans /etc/passwd)  
SHELL=shell de connexion (idem)  
TERM=linux (inchangé)  
PATH=/usr/bin:/bin  
MAIL=emplacement du fichier de boîte à lettres de l'utilisateur  
USER=nom de l'utilisateur
```

Le programme `/bin/login` lance ensuite le shell choisi par l'utilisateur dans le fichier `/etc/passwd`. Le shell configure lui-même un certain nombre de variables d'environnement dépendant de l'interpréteur. Enfin, il lit certains fichiers d'initialisation pouvant eux-mêmes contenir des affectations de variables d'environnement. Ces fichiers peuvent être généraux pour le système (par exemple, `/etc/profile`) ou spécifiques à l'utilisateur (`~/.profile`). Leurs noms peuvent également varier en fonction du shell utilisé.

En plus des variables d'environnement « classiques » que nous allons voir ci-après, une application peut très bien faire varier son comportement en fonction de variables qui lui sont tout à fait propres. Une application nommée par exemple `foo` peut rechercher ses fichiers de configuration dans le répertoire signalé dans la variable `FOODIR`, et créer ses fichiers temporaires dans le répertoire indiqué dans la variable `FOOTMP`. Bien entendu, si ces variables n'existent pas, l'application devra prévoir des

valeurs par défaut. Il sera alors plus facile pour l'utilisateur de se créer un script shell de lancement de l'application (par exemple avec *Bash*) :

```
#!/bin/sh

export FOODIR=/usr/local/lib/foo/
export FOOCFG=$HOME/.foo/
export FOOTMP=/tmp/foo/

exec /usr/local/bin/foo
```

Les variables d'environnement les plus couramment utilisées sont les suivantes :

- HOME contient le répertoire personnel de l'utilisateur.
- PATH indique la liste des répertoires où on recherche les fichiers exécutables. Ces répertoires sont séparés par des deux-points ': '.
- PWD correspond au répertoire de travail du shell lors du lancement de l'application.
- LANG indique la localisation choisie par l'utilisateur, complétée par les variables LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME. Nous détaillerons leurs rôles plus loin.
- LOGNAME et/ou USER contiennent le nom de l'utilisateur.
- TERM correspond au type de terminal utilisé.
- SHELL indique le shell de connexion de l'utilisateur.

D'autres variables sont plutôt liées au comportement de certaines routines de bibliothèque, comme :

- TMPDIR est analysée par les routines `tempnam()`, `tmpnam()`, `tmpfile()`, etc.
- POSIXLY_CORRECT modifie le comportement de certaines routines pour qu'elles soient strictement conformes à la norme Posix (SUSv4). Ainsi `getopt()`, que nous verrons plus bas, agit différemment suivant que la variable est définie ou non avec les arguments qu'elle rencontre sur la ligne de commande et qui ne représentent pas des options valides.
- MALLOC_XXX représente toute une famille de fonctions permettant de contrôler le comportement des routines d'allocation mémoire du type `malloc()`.
- TZ correspond au fuseau horaire et modifie le comportement de `tzset()`.

Bien entendu, le comportement de nombreuses routines est influencé par les variables de localisation LC_XXX.

Lorsqu'une application utilise les variables d'environnement pour adapter son comportement, il est très fortement recommandé de bien documenter l'utilisation qu'elle en fait (dans la section *Environnement* de sa page de manuel, par exemple).

Arguments en ligne de commande

Les programmes en langage C reçoivent traditionnellement, dans un tableau de chaînes de caractères, les arguments qui leur sont transmis sur leur ligne de commande. Le nombre d'éléments de ce tableau est passé en premier argument de la fonction `main()`, et le tableau est transmis en second argument. Ces deux éléments sont habituellement notés `argc` (*args count*, nombre d'arguments) et `argv` (*args values*, valeurs des arguments). En Python, il s'agit d'une table accessible dans `sys.argv`.

Quel que soit le langage, un programme reçoit en première position du tableau `argv` (donc à la position 0) son propre nom de fichier exécutable (ou script). Lorsqu'on souhaite, en Python, parcourir l'ensemble des arguments, il sera généralement plus simple d'éliminer cette entrée de la table en utilisant la syntaxe `sys.argv.pop(0)`.

Lorsqu'une application est lancée par un shell, la ligne de commande est analysée et découpée en arguments en utilisant comme séparateurs certains caractères spéciaux. Par exemple, avec *Bash* la liste de ces caractères est conservée dans la variable d'environnement `IFS` et contient l'espace, la tabulation et le retour chariot.

Une application peut donc parcourir sa ligne de commande.

```
exemple-argv.c :
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i;

    fprintf(stdout, "%s a reçu en argument :\n", argv[0]);
    for (i = 1; i < argc; i++)
        fprintf(stdout, " %s\n", argv[i]);
    return 0;
}
```

ou

```
exemple-argv.py :
#!/usr/bin/python
from __future__ import print_function
import sys

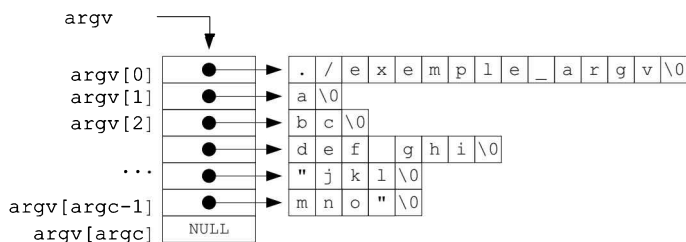
print(sys.argv[0], "a reçu en argument :")
sys.argv.pop(0)
for var in sys.argv:
    print(" ", var)
```

Voici un exemple d'exécution, montrant que le shell a considéré comme un argument unique l'ensemble "def ghi", y compris l'espace, grâce à la protection qu'offraient les guillemets, mais que celle-ci est supprimée lorsqu'on fait précéder les caractères d'un antislash (\) et qu'ils deviennent alors des caractères normaux :

```
$ ./exemple_argv a bc "def ghi" \"jkl mno\"
./exemple_argv a reçu en argument :
a
bc
def ghi
"jkl
mno"
$
```

Par convention, le tableau `argv[]` contient (`argc+1`) éléments, le dernier étant un pointeur NULL.

Figure 4-2
Arguments en ligne de
commande du processus



Certains programmes peuvent parfaitement se contenter d'analyser ainsi leur ligne de commande, surtout si on ne doit y trouver qu'un nombre fixe d'arguments (par exemple, uniquement un nom de fichier à traiter), et si aucune option n'est prévue pour modifier le déroulement du processus.

Toutefois, la plupart des applications permettent à l'utilisateur d'indiquer des options en ligne de commande et de fournir de surcroît des arguments qui ne sont pas des options. Nous faisons ici la distinction entre les options du type `-v -r -f`, etc., qu'on trouve dans la plupart des utilitaires Unix, et les autres arguments, comme les noms de fichiers à copier pour la commande `cp`. La bibliothèque Glibc offre des fonctions puissantes pour l'analyse automatique de la ligne de commande afin d'en extraire les arguments qui représentent des options.

Signalons aussi que certaines options prennent elles-mêmes un argument. Par exemple, l'option `-S` de la version *Gnu* de `cp` réclame un argument représentant le suffixe à utiliser pour conserver une copie de secours des fichiers écrasés.

Options simples – SUSv4

Les options, à la manière SUSv4, sont précédées d'un tiret (-), et sont représentées par un caractère alphanumérique simple. On peut toutefois regrouper plusieurs options à la suite du même tiret (par exemple, -a -b -c équivalent à -abc). Si une option nécessite un argument, elle peut en être séparée ou non par un espace (-a fichier équivaut à -afichier).

L'option spéciale « -- » (deux tirets) sert à indiquer la fin de la liste des options. Tous les arguments à la suite ne seront pas considérés comme des options. On peut ainsi se débarrasser d'un fichier nommé « -f » avec la commande `rm - -f`.

Un tiret isolé n'est pas considéré comme une option. Il est transmis au programme comme un argument non option.

Normalement, l'utilisateur doit fournir d'abord les options sur sa ligne de commande, et ensuite uniquement les autres arguments. Toutefois, la bibliothèque Glibc réordonne au besoin les arguments de la ligne de commande.

Pour lire aisément les options fournies à une application, la bibliothèque C offre la fonction `getopt()` et les variables globales `optind`, `opterr`, `optopt` et `optarg`, déclarées dans `<unistd.h>` :

```
int getopt (int argc, const char * argv [], const char * options);
extern int optind;
extern int opterr;
extern int optopt;
extern char * optarg;
```

On transmet à la fonction `getopt()` les arguments `argc` et `argv` qu'on a reçus dans la fonction `main()`, ainsi qu'une chaîne de caractères indiquant les options reconnues par le programme. À chaque invocation de la fonction, celle-ci nous renverra le caractère correspondant à l'option en cours, et la variable globale externe `optarg` pointera vers l'éventuel argument de la fonction. Lorsque toutes les options auront été parcourues, `getopt()` nous renverra -1, et la variable externe `optind` contiendra le rang du premier élément de `argv[]` qui ne soit pas une option.

Si `getopt()` rencontre un caractère d'option non reconnu, elle affiche un message sur le flux `stderr`. Si la variable externe globale `opterr` ne contient pas 0, elle copie le caractère inconnu dans la variable globale externe `optopt` et renvoie le caractère '?'.

La chaîne de caractères qu'on transmet en troisième argument à `getopt()` contient la liste de tous les caractères d'option reconnus. Si une option prend un argument, on fait suivre le caractère d'un deux-points ':'.

Voici un premier exemple d'analyse des options en ligne de commande, dans lequel le programme reconnaît les options a, b, X, Y seules, et l'option -c suivie d'un argument.

Si un caractère d'option n'est pas reconnu, nous gérerons nous-mêmes l'affichage d'un message d'erreur. Enfin, une fois terminée l'analyse des options, nous afficherons un à un les arguments restants (qui pourraient représenter par exemple des noms de fichiers à traiter).

```
exemple-getopt.c :
#include <stdio.h>
#include <unistd.h>

int main (int argc, char * argv [])
{
    char * liste_options = "abc:XY";
    int option;

    opterr = 0; /* Pas de message d'erreur automatique */

    while ((option = getopt(argc, argv, liste_options)) != -1) {
        switch (option) {
            case 'a' :
                fprintf(stdout, "Option a\n");
                break;
            case 'b' :
                fprintf(stdout, "Option b\n");
                break;
            case 'c' :
                fprintf(stdout, "Option c %s\n", optarg);
                break;
            case 'X' :
            case 'Y' :
                fprintf(stdout, "Option %c\n", option);
                break;
            case '?' :
                fprintf(stderr, "Option %c fausse\n", optopt);
                break;
        }
    }

    if (optind != argc) {
        fprintf(stdout, "Arguments restants :\n");
        while (optind != argc)
            fprintf(stdout, " %s\n", argv[optind ++]);
    }
    return 0;
}
```


Voici un exemple d'exécution regroupant une bonne partie des fonctionnalités disponibles avec `getopt()`:

```
$ ./exemple_getopt -abd -c 12 -XY suite et fin
Option a
Option b
Option d fausse
Option c 12
Option X
Option Y
Arguments restants :
    suite
    et
    fin
$
```

La variable externe globale `optarg`, qu'on utilise pour accéder à l'argument de certaines options, est en réalité un pointeur, de type `char *`, dirigé vers l'élément de `argv[]` qui correspond à la valeur désirée. Il n'est donc pas nécessaire de copier la chaîne de caractères si on désire l'utiliser plus tard ; on peut directement copier la valeur du pointeur, puisque le tableau `argv[]` ne doit plus varier après l'invocation de `getopt()`. Nous verrons un exemple plus concret d'utilisation de cette chaîne de caractères dans le programme nommé `exemple_options.c`, fourni à la fin de ce chapitre.

Il est possible d'utiliser les mêmes fonctionnalités en Python grâce au module `getopt`. Toutefois, il existe un module nommé `argparse` plus simple d'usage et plus performant.

Options longues – Gnu

Les applications issues du projet *Gnu* ont ajouté un autre type d'options qui ont été incorporées dans les routines d'analyse de la ligne de commande : les options longues. Il s'agit d'options commençant par deux tirets « `--` », et dont le libellé est exprimé par des mots complets. Par exemple, la version *Gnu* de `ls` accepte l'option longue `--numeric-uid-gid` de manière équivalente à `-n`.

Bien entendu, ces options ne sont pas prévues pour être utilisées quotidiennement en ligne de commande. Peu d'utilisateurs préfèrent saisir :

```
$ ln --symbolic --force foo bar
```

à la place de :

```
$ ln -sf foo bar
```

Par contre, ces options longues sont très commodes lorsqu'elles sont utilisées dans un script shell, où elles permettent d'autodocumenter les arguments fournis à une commande peu utilisée.

Les options longues peuvent bien entendu accepter des arguments, qui s'écrivent aussi bien :

```
| --option valeur
```

que :

```
| --option=valeur
```

Une option longue peut être abrégée tant qu'il n'y a pas d'ambiguïté avec d'autres options de la même commande. La bibliothèque Glibc offre des routines d'analyse des options longues assez semblables à la routine `getopt()` ; il s'agit de `getopt_long()` et de `getopt_long_only()`. Ces routines sont déclarées dans le fichier d'en-tête `<getopt.h>` et non dans `<unistd.h>`. La fonction `getopt_long()` a le prototype suivant :

```
| int getopt_long (int argc, char * argv [],  
|                  const char * optsring,  
|                  const struct option * longopts,  
|                  int * longindex);
```

Attention toutefois aux problèmes de portabilité : même si elle n'existe pas sur tous les systèmes, la routine `getopt()` est définie par SUSv4 et est donc très répandue sous Unix. Par contre, les options longues (et même le fichier d'en-tête `<getopt.h>`) sont des extensions *Gnu* largement moins courantes. Si une application doit être portable sous plusieurs systèmes Unix, il est conseillé d'encadrer les portions de code spécifiques aux options longues par des directives `#ifdef` / `#endif` permettant à la compilation de basculer au choix avec ou sans options longues.

La routine `getopt_long()` prend `argc` et `argv[]` en premiers arguments comme `getopt()`. Ensuite, on lui transmet également une chaîne de caractères contenant les options courtes, exactement comme `getopt()`. Puis viennent deux arguments supplémentaires : un tableau d'objets de type `struct option`, et un pointeur sur un entier. La structure `struct option` est définie dans le fichier d'en-tête `<getopt.h>`, voir tableau ci-après.

Nom	Type	Signification
name	char *	Nom de l'option longue.
has_arg	int	L'option réclame-t-elle un argument supplémentaire ?
flag	int	Manière de renvoyer la valeur ci-dessous.
val	int	Valeur à renvoyer quand l'option est trouvée.

Chaque élément du tableau `longopts` contient une option longue, le dernier élément devant être obligatoirement rempli avec des zéros.

Le premier champ comprend simplement le nom de l'option. C'est une chaîne de caractères classique, terminée par un caractère nul. Le second champ indique si l'option doit être suivie par un argument. Il y a trois possibilités, décrites par des constantes symboliques dans le fichier `<getopt.h>` :

- `no_argument (0)` : l'option ne prend pas d'argument.
- `required_argument (1)` : l'option prend toujours un argument.
- `optional_argument (2)` : l'argument est éventuel.

Le troisième champ est plus compliqué. S'il est `NULL` (c'est le cas le plus courant), l'appel à `getopt_long()` renverra, lorsqu'il trouvera l'option, la valeur indiquée dans le champ `val`. Ce principe est donc assez semblable à celui qu'on a déjà vu pour `getopt()`, et il est même habituel de mettre dans le champ `val` le caractère correspondant à l'option courte équivalente, afin d'avoir un traitement `switch/case` unique. Dans le cas où ce troisième champ (`flag`) n'est pas `NULL`, il faut le faire pointer vers une variable de type `int`, par exemple une variable déclarée dans la fonction `main()`, dans laquelle `getopt_long()` écrira la valeur contenue dans le champ `val` si l'option est rencontrée. Dans un tel cas, `getopt_long()` renvoie 0.

Lorsque `getopt_long()` rencontre une option courte (contenue dans la chaîne `optstring`), elle se comporte exactement comme `getopt()`. Lorsqu'elle rencontre une option longue, elle remplit la variable pointée par `longindex` avec l'indice de l'option en question dans le tableau `longopts`. Comme pour les options courtes, les arguments éventuels sont transmis par le pointeur global `optarg`. Celui-ci est `NULL` si l'option n'a pas d'argument (ce qui sert dans le cas d'arguments optionnels).

Pour remplir le tableau `longopts` que nous devons fournir à `getopt_long()`, il est pratique d'utiliser l'initialisation automatique d'une variable statique de la fonction `main()`. Nous allons écrire un petit programme (qu'on peut imaginer comme un lecteur de fichiers vidéo) acceptant les options suivantes :

- `--debut` ou `-d`, suivie d'une valeur numérique entière ;
- `--fin` ou `-f`, suivie d'une valeur numérique entière ;
- `--rapide` ;
- `--lent`.

Les deux dernières options serviront à mettre directement à jour une variable interne du programme, en utilisant un champ `flag` non `NULL`. Nous ne traitons pas dans ce programme les arguments autres que les options (une fois que `getopt_long()` renvoie `-1`), et nous laissons à cette routine le soin d'afficher un message d'erreur en cas d'option non reconnue.

```
exemple-getopt-long.c :
#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>

int    vitesse_lecture = 0;
/* -1 = lent, 0 = normal, 1 = rapide */

int main (int argc, char * argv[])
{
    char * optstring = "d:f:";
    struct option longopts[] = {

        /* name    has_arg  flag                                val */
        { "debut",   1,      NULL,                                'd' },
        { "fin",     1,      NULL,                                'f' },
        { "rapide",   0,      &vitesse_lecture,                    1 },
        { "lent",     0,      &vitesse_lecture,                    -1 },
        /* Le dernier élément doit être nul */
        { NULL,       0,      NULL,                                0 },
    };

    int    longindex;
    int    option;
    int    debut = 0;
    int    fin = 999;

    while ((option = getopt_long(argc, argv,
                                optstring, longopts, &longindex)) != -1) {

        switch (option) {

            case 'd' :
                if (sscanf(optarg, "%d", &debut) != 1) {
                    fprintf(stderr, "Erreur pour debut\n");
                }
                break;
            case 'f' :
                if (sscanf(optarg, "%d", &fin) != 1) {
                    fprintf(stderr, "Erreur pour fin\n");
                }
                break;
        }
    }
}
```

```
        case 0 :
            /* Vitesse_lecture traitée automatiquement */
            break;
        case '?' :
            /* On a laissé opterr à 1 */
            break;
    }
}
fprintf(stdout, "Vitesse %d, debut %d, fin %d\n",
        vitesse_lecture, debut, fin);
return 0;
}
```

En voici un exemple d'exécution :

```
$ ./exemple-getopt-long --rapide -d 4 --fin 25
Vitesse 1, debut 4, fin 25
$
```

Il existe également avec la Glibc une routine `getopt_long_only()` fonctionnant comme `getopt_long()`, à la différence que même une option commençant par un seul tiret (-) est considérée d'abord comme une option longue puis, en cas d'échec, comme une option courte. Cela signifie que `-ab` sera d'abord considérée comme équivalant à `--ab` (donc comme une abréviation de `--abort`) avant d'être traitée comme la succession d'options simples « `-a -b` ». Cet usage peut induire l'utilisateur en erreur, et cette routine me semble peu recommandable...

Sous-options

L'argument qu'on fournit à une option peut parfois nécessiter lui-même une analyse pour être séparé en sous-options. La bibliothèque C fournit dans `<stdlib.h>` une fonction ayant ce rôle : `getsubopt()`. La déclaration n'est présente dans le fichier d'en-tête que si la constante symbolique `_XOPEN_SOURCE` est définie et contient la valeur 500, ou si la constante `_GNU_SOURCE` est définie.

L'exemple classique d'utilisation de cette fonction est l'option `-o` de la commande `mount`. Cette option est suivie de n'importe quelle liste de sous-options séparées par des virgules, certaines pouvant prendre une valeur (par exemple `-o async,noexec,bs=512`).

Le prototype de `getsubopt()` est le suivant :

```
int getsubopt (char ** option, const char * const * tokens,
              char ** value);
```

Cette routine n'est appelée que lorsqu'on se trouve dans le case correspondant à l'option à analyser de nouveau (par -o pour mount). Il faut transmettre un pointeur en premier argument sur un pointeur contenant la sous-option. En d'autres termes, on crée un pointeur `char * subopt` qu'on fait pointer sur la chaîne à analyser (`subopt = optarg`), et on transmet `&subopt` à la fonction. Celle-ci avancera ce pointeur d'une sous-option à chaque appel, jusqu'à ce qu'il arrive sur le caractère nul de fin de `optarg`.

Le second argument est un tableau contenant des chaînes de caractères correspondant aux sous-options. Le dernier élément de ce tableau doit être un pointeur NULL.

Enfin, on transmet en dernier argument l'adresse d'un pointeur de chaîne de caractères. Lorsque la routine rencontre une sous-option suivie d'un signe égal '=', elle renseigne ce pointeur de manière à l'amener au début de la valeur. Elle inscrit également un caractère nul pour marquer la fin de la valeur. Si aucune valeur n'est disponible, value est rempli avec NULL.

Si une sous-option est reconnue, son index dans la table `tokens` est renvoyé. Sinon, `getsubopt()` renvoie -1. Un exemple de code permettant l'analyse d'une sous-option sera fourni dans le programme `exemple_options.c` décrit ci-après.

Exemple complet d'accès à l'environnement

Nous allons voir un exemple de code permettant de regrouper l'ensemble des fonctionnalités d'accès à l'environnement que nous avons vues dans ce chapitre. Nous allons imaginer qu'il s'agit d'une application se connectant par exemple sur un serveur *TCP/IP*, comme nous aurons l'occasion d'en étudier plus loin.

Notre application doit fournir tout d'abord des valeurs par défaut pour tous les éléments paramétrables. Ces valeurs sont établies à la compilation du programme. Toutefois, on les regroupe toutes ensemble afin que l'administrateur du système puisse, s'il le désire, recompiler l'application avec de nouvelles valeurs par défaut.

Ensuite, nous essaierons d'obtenir des informations en provenance des variables d'environnement. Celles-ci peuvent être renseignées par l'administrateur système (par exemple dans `/etc/profile`) ou par l'utilisateur (dans `~/.profile` ou dans un script shell de lancement de l'application).

Puis, nous analyserons la ligne de commande. Il est en effet important que les options fournies manuellement par l'utilisateur aient la priorité sur celles qui ont été choisies pour l'ensemble du système.

Voyons la liste des éléments dont nous allons permettre le paramétrage.

- Adresse réseau du serveur à contacter

Il s'agit ici d'une adresse IP numérique ou d'un nom d'hôte. Nous nous contenterons d'obtenir cette adresse dans une chaîne de caractères et de laisser à la suite de l'application les tâches de conversion nécessaires. Nous ne ferons aucune gestion d'erreur sur cette chaîne, nous arrangeant simplement pour qu'elle ne soit pas vide.

Par défaut, la valeur sera `localhost`. On pourra modifier l'adresse en utilisant la variable d'environnement `OPT_ADR`. Les options `-a` et `--adresse`, suivies d'une chaîne de caractères, permettront une dernière configuration.

- Port TCP à utiliser pour joindre le serveur

Le port TCP sur lequel nous désirons contacter le serveur peut être indiqué soit sous forme numérique, soit sous forme symbolique, en utilisant un nom décrit dans le fichier `/etc/services`. Nous considérerons donc qu'il s'agit d'une chaîne de caractères, que le reste de l'application se chargera de convertir en numéro de port effectif.

Par défaut, nous prendrons une valeur arbitraire de 4000, mais nous pourrions modifier cette valeur en utilisant la variable d'environnement `OPT_SRV`, ou l'une des options `-p` ou `--port`, suivie d'une chaîne de caractères.

- Options pour la connexion

Afin de donner un exemple d'utilisation de la fonction `getsubopt()`, nous allons permettre la transmission d'une liste de sous-options séparées par des virgules, en utilisant l'option `-o` ou `--option` de la ligne de commande :

`auto / nonauto` : il s'agit par exemple de tentative de reconnexion automatique au serveur en cas d'échec de transmission. Ce paramètre est également configurable en définissant (ou non) la variable d'environnement `OPT_AUTO`. Par défaut, le choix est `nonauto`.

`delai=<duree>` : il s'agit du temps d'attente en secondes entre deux tentatives de reconnexion au serveur. Cette valeur vaut 4 secondes par défaut, mais peut aussi être modifiée par la variable d'environnement `OPT_DELAI`.

- Affichage de l'aide

Une option `-h` ou `--help` permettra d'obtenir un rappel de la syntaxe de l'application.

- Arguments autres que les options

Le programme peut être invoqué avec d'autres arguments à la suite des options, par exemple des noms de fichiers à transférer, l'identité de l'utilisateur sur la machine distante, etc. Ces arguments seront affichés par notre application à la suite des options.

Pour lire les sous-options introduites par l'option `-o`, une routine séparée est utilisée, principalement pour éviter des niveaux d'indentation excessifs et inesthétiques en imbriquant deux boucles `while` et deux `switch-case`.

Enfin, pour augmenter la portabilité de notre exemple, nous allons encadrer tout ce qui concerne les options longues *Gnu* par des directives `#ifdef`-`#else`-`#endif`. Ainsi, la recompilation sera possible sur pratiquement tous les systèmes Unix, à l'exception peut-être de la routine `getsubopt()`. Pour compiler l'application avec les options longues, sous Linux par exemple, il suffira d'inclure une option `-DOPTIONS_LONGUES` sur la ligne de commande de `gcc` (ou dans un fichier `Makefile`). Sur un système où la bibliothèque C n'offre pas la routine `getopt_long()`, il suffira de ne pas définir cette constante symbolique pour permettre la compilation.

```
exemple-options.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#ifdef OPTIONS_LONGUES
#include <getopt.h>
#endif

/* Définition des valeurs par défaut.      */
/* (Pourraient être regroupées dans un .h) */
#define ADRESSE_SERVEUR_DEFAULT "localhost"
#define PORT_SERVEUR_DEFAULT    "4000"
#define CONNEXION_AUTO_DEFAULT  0
#define DELAI_CONNEXION_DEFAULT 4

void    sous_options      (char * ssopt, int * cnx_auto, int * delai);
void    suite_application (char * adresse_serveur,
                           char * port_serveur,
                           int    connexion_auto,
                           int    delai_reconnexion,
                           int    argc,
                           char * argv []);
void    affiche_aide      (char * nom_programme);

int main (int argc, char * argv[])
{
    /*
     * Copie des chaînes d'environnement.
     * Il n'est pas indispensable sous Linux d'en faire une
     * copie, mais c'est une bonne habitude pour assurer la
     * portabilité du programme.
     */
    char * opt_adr  = NULL;
    char * opt_srv  = NULL;
```



```
int    opt_delai = 0;
char * retour_getenv;
/* Variables contenant les valeurs effectives
 * des paramètres. */
static char *  adresse_serveur = ADRESSE_SERVEUR_DEFAULT;
static char *  port_serveur    = PORT_SERVEUR_DEFAULT;
int           connexion_auto = CONNEXION_AUTO_DEFAULT;
int           delai_connexion = DELAI_CONNEXION_DEFAULT;

int    option;
/* Lecture des variables d'environnement, on code ici
 * "en dur" le nom des variables, mais on pourrait
 * les regrouper (par #define) en tête de fichier.
 */
retour_getenv = getenv("OPT_ADR");
if ((retour_getenv != NULL) && (strlen(retour_getenv) != 0)) {
    opt_adr = malloc (strlen(retour_getenv) + 1);
    if (opt_adr != NULL) {
        strcpy(opt_adr, retour_getenv);
        adresse_serveur = opt_adr;
    } else {
        perror("malloc");
        exit(1);
    }
}
retour_getenv = getenv("OPT_SRV");
if ((retour_getenv != NULL) && (strlen(retour_getenv) != 0)) {
    opt_srv = malloc (strlen(retour_getenv) + 1);
    if (opt_srv != NULL) {
        strcpy(opt_srv, retour_getenv);
        port_serveur = opt_srv;
    } else {
        perror("malloc");
        exit(1);
    }
}
retour_getenv = getenv("OPT_AUTO");
/* Il suffit que la variable existe dans
 * l'environnement, sa valeur ne compte pas. */
if (retour_getenv != NULL)
    connexion_auto = 1;
retour_getenv = getenv("OPT_DELAI");
if (retour_getenv != NULL)
    if (sscanf(retour_getenv, "%d", & opt_delai) == 1)
        delai_connexion = opt_delai;
/* On va passer maintenant a la lecture des options
 * en ligne de commande. */
opterr = 1;
```

```

while (1) {
    #ifdef OPTIONS_LONGUES
        int index = 0;
        static struct option longopts[] = {
            { "adresse", 1, NULL, 'a' },
            { "port", 1, NULL, 'p' },
            { "option", 1, NULL, 'o' },
            { "help", 0, NULL, 'h' },
            { NULL, 0, NULL, 0 }
        };
        option = getopt_long(argc, argv, "a:p:o:h",
                               longopts, & index);
    #else
        option = getopt(argc, argv, "a:p:o:h");
    #endif
    if (option == -1)
        break;

    switch (option) {
        case 'a' :
            /* On libère une éventuelle copie de chaîne
             * d'environnement équivalente. */
            if (opt_adr != NULL)
                free(opt_adr);
            opt_adr = NULL;
            adresse_serveur = optarg;
            break;
        case 'p' :
            /* idem */
            if (opt_srv != NULL)
                free(opt_srv);
            opt_srv = NULL;
            port_serveur = optarg;
            break;
        case 'o' :
            /* On va analyser les sous-options */
            sous_options(optarg,
                          & connexion_auto,
                          & delai_connexion);

            break;
        case 'h' :
            affiche_aide(argv[0]);
            exit(0);
        default :
            break;
    }
}
suite_application(adresse_serveur, port_serveur,
                  connexion_auto, delai_connexion,
                  argc - optind, & (argv[optind]));

```

```

    return 0;
}

void sous_options (char * ssopt, int * cnx_auto, int * delai)
{
    int    subopt;
    char * chaine = ssopt;
    char * value = NULL;
    int    val_delai;

    char * tokens[] = {
        "auto", "nonauto", "delai", NULL
    };
    while ((subopt = getsubopt(& chaine, tokens,
                              & value)) != -1) {
        switch (subopt) {
            case 0 : /* Auto */
                * cnx_auto = 1;
                break;
            case 1 : /* Nonauto */
                * cnx_auto = 0;
                break;
            case 2 : /* Délai=... */
                if (value == NULL) {
                    fprintf(stderr, "délai attendu\n");
                    break;
                }
                if (sscanf (value, "%d", & val_delai) != 1) {
                    fprintf(stderr, "délai invalide\n");
                    break;
                }
                * delai = val_delai;
                break;
        }
    }
}

/* La suite de l'application ne fait qu'afficher
 * les options et les arguments supplémentaires */
void suite_application (char * adr_serveur,
                        char * port_serveur,
                        int    cnx_auto,
                        int    delai_cnx,
                        int    argc,
                        char * argv[])
{
    int    i;
    printf("Serveur : %s - %s\n", adr_serveur, port_serveur);
    printf("Connexion auto : %s\n", cnx_auto ? "oui":"non");
    printf("Delai : %d\n", delai_cnx);
}

```

```

    printf("Arguments supplémentaires : ");
    for (i = 0; i < argc; i++)
        printf("%s - ", argv[i]);
    printf("\n");
}

void affiche_aide (char * nom_prog)
{
    printf("Syntaxe : %s [options] [fichiers...]\n", nom_prog);
    printf("Options :\n");
#ifdef OPTIONS_LONGUES
    printf(" --help\n");
#endif
    printf(" -h                Cet ecran d'aide \n");
#ifdef OPTIONS_LONGUES
    printf(" --adresse <serveur> \n");
#endif
    printf(" -a <serveur>        Adresse IP du serveur \n");
#ifdef OPTIONS_LONGUES
    printf(" --port <numero_port> \n");
#endif
    printf(" -p <num_port>       Numero de port TCP \n");
#ifdef OPTIONS_LONGUES
    printf(" --option [sous_options]\n");
#endif
    printf(" -o [sous_options]   \n");
    printf("Sous-options :\n");
    printf(" auto / nonauto     Connexion automatique \n");
    printf(" delai=<sec>         Delai entre deux connexions \n");
}

```

Voici plusieurs exemples d'utilisation, ainsi que la ligne de commande à utiliser pour définir les constantes nécessaires lors de la compilation :

```

$ cc -D_GNU_SOURCE -DOPTIONS_LONGUES exemple-options.c -o exemple-
options
$ ./exemple-options
Serveur : localhost - 4000
Connexion auto : non
Delai : 4
Arguments supplémentaires :
$ export OPT_ADR="172.16.15.1"
$ ./exemple-options
Serveur : 172.16.15.1 - 4000
Connexion auto : non
Delai : 4

```

```
Arguments supplementaires :  
$ export OPT_SRV="5000"  
$ ./exemple-options --adresse "127.0.0.1"  
Serveur : 127.0.0.1 - 5000  
Connexion auto : non  
Delai : 4  
Arguments supplementaires :  
$ export OPT_AUTO=  
$ ./exemple-options -p 6000 -odelai=5  
Serveur : 172.16.15.1 - 6000  
Connexion auto : oui  
Delai : 5  
Arguments supplementaires :  
$ ./exemple-options -p 6000 -odelai=5,nonauto et d autres arguments  
Serveur : 172.16.15.1 - 6000  
Connexion auto : non  
Delai : 5  
Arguments supplementaires : et - d - autres - arguments  
$
```

Conclusion

Nous voici donc en possession d'un squelette complet de programme capable d'accéder à son environnement et permettant un paramétrage à plusieurs niveaux :

- à la compilation, par l'administrateur système, grâce aux valeurs par défaut ;
- globalement pour toutes les exécutions, par l'administrateur ou l'utilisateur, grâce aux variables d'environnement ;
- lors d'une exécution particulière grâce aux options en ligne de commande.

Il est important, pour une application un tant soit peu complète, de permettre ainsi à l'utilisateur et à l'administrateur système de configurer son comportement à divers niveaux.

5

Fin d'un programme

Dans ce chapitre, nous allons étudier tout d'abord les moyens de mettre fin à l'exécution d'un programme. Nous verrons ensuite des méthodes permettant d'enregistrer des routines qui seront automatiquement exécutées avant de quitter l'application. Nous nous pencherons sur l'attente de la fin d'un processus fils et la récupération de son état de terminaison, puis nous examinerons les moyens de signaler une erreur à l'utilisateur, même si celle-ci ne conduit pas nécessairement à l'arrêt du programme.

Terminaison d'un programme

Un processus peut se terminer volontairement ou involontairement. Dans le premier cas, l'application est abandonnée à la demande de l'utilisateur, ou la tâche à accomplir est finie. Dans le second cas, soit un dysfonctionnement est découvert, si sérieux qu'il ne permet pas au programme de continuer son travail, soit l'utilisateur décide d'interrompre prématurément l'exécution du processus (en pressant *Contrôle-C* dans le terminal par exemple). Les terminaisons involontaires se matérialiseront toujours par l'arrivée d'un signal fatal qui tue brusquement le processus.

Terminaison normale d'un processus

Un programme dispose de plusieurs possibilités pour se finir volontairement. La plus simple est de revenir de la fonction `main()` en renvoyant un compte rendu d'exécution sous forme de valeur entière. Cette valeur est lue par le processus père, qui peut en tirer les

conséquences adéquates. Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échec sont indiqués par des codes de retour non nuls (et qui peuvent être documentés avec l'application). Cela permet d'écrire des scripts shell robustes, qui vérifient le bon fonctionnement de chaque commande employée. Dans la plupart des cas, on ne teste que la nullité du code de retour. Lorsque le processus est arrêté à cause d'un signal, le shell modifie le code de retour (*Bash* ajoute 128, par exemple). Il est donc conseillé de n'utiliser que des valeurs comprises entre 0 et 127.

Lorsqu'un processus lancé par le shell se termine, son code de retour est disponible dans la variable spéciale `$?`. Suivant les applications ce code aura différentes significations. Prenons par exemple l'utilitaire `grep`. Nous allons l'invoquer de manière à ce qu'il cherche – et trouve – la chaîne de caractères `root` dans le fichier `/etc/passwd`.

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
$ echo $?
0
$
```

Le code de retour est nul, indiquant la réussite. À présent, donnons-lui à chercher une chaîne ne se trouvant pas dans le fichier :

```
$ grep abcdefg /etc/passwd
$ echo $?
1
$
```

La valeur 1 signifie donc « j'ai fait mon travail correctement, mais je n'ai pas trouvé la chaîne ». Demandons-lui maintenant de consulter un fichier inexistant :

```
$ grep root /etc/inexistant
grep: /etc/inexistant: Aucun fichier ou répertoire de ce type
$ echo $?
2
$
```

Le code de retour 2 a donc une signification différente : « je n'ai pas pu faire mon travail, la demande est invalide ». Tous ces codes (documentés dans la page de manuel de `grep`), sont renvoyés par le processus quand il se termine normalement – de son plein gré. Toutefois, si le processus est interrompu prématurément et se termine anormalement, le shell renseigne la variable spéciale `$?` en fonction du numéro du signal ayant tué le pro-

cessus. Par exemple, nous invoquons la commande `sleep` pour un sommeil de 30 secondes, et nous l'interrompons pendant ce temps en pressant *Contrôle-C* :

```
$ sleep 30
      (Ctrl-C)
$ echo $$
130
$
```

La valeur 130 correspond à $128 + 2$; ce numéro étant celui du signal `SIGINT`, comme on le voit dans la page de manuel `signal(7)`, correspondant à la pression de la touche d'interruption (*Ctrl-C* par défaut).

Si seuls la réussite ou l'échec du programme importent (si le processus père n'essaye pas de détailler les raisons de l'échec), il est possible d'employer les constantes symboliques `EXIT_SUCCESS` ou `EXIT_FAILURE` définies dans `<stdlib.h>`. Ceci a l'avantage d'adapter automatiquement le comportement du programme, même sur les systèmes non conformes à `SUSv4`, où ces constantes ne sont pas nécessairement 0 et 1.

Une autre façon de terminer un programme normalement est d'utiliser la fonction `exit()` en C, ou `sys.exit()` en Python.

```
| void exit (int code);
```

On lui transmet en argument le code de retour pour le processus père. L'effet est strictement égal à celui d'un retour depuis la fonction `main()`, à la différence que `exit()` peut être invoquée depuis n'importe quelle partie du programme (notamment depuis les routines de traitement d'erreur).

Lorsqu'on utilise uniquement une terminaison avec `exit()` dans un programme, le compilateur se plaint que la fin de la fonction `main()` est atteinte alors qu'aucune valeur n'a été renvoyée.

```
| exemple-exit-1.c :
| #include <stdlib.h>
|
| void sortie (void);
|
| int main (void)
| {
|     sortie();
| }
|
| void sortie(void)
| {
|     exit(EXIT_FAILURE);
| }
```

déclenche à la compilation l'avertissement suivant :

```
$ cc -Wall exemple-exit-1.c -o exemple-exit-1
exemple-exit-1.c: In function `main':
exemple-exit-1.c:9: warning: control reaches end of non-void function
$
```

(Si nous avons directement mis `exit()` dans la fonction `main()`, le compilateur l'aurait reconnu et aurait supprimé cet avertissement.)

Pour éviter ce message, on peut être tenté de déclarer `main()` comme une fonction de type `void`. Sous Linux, cela ne pose pas de problème, mais un tel programme pourrait ne pas être portable sur d'autres systèmes qui exigent que `main()` renvoie une valeur. D'ailleurs, le compilateur `gcc` avertit que `main()` doit normalement être de type `int`.

```
exemple-exit-2.c :
#include <stdlib.h>

void main (void)
{
    exit(EXIT_SUCCESS);
}
```

déclenche un avertissement :

```
$ cc -Wall exemple-exit-2.c -o exemple-exit-2
exemple-exit-2.c:5: warning: return type of `main' is not `int'
$
```

Ayons donc comme règle de bonne conduite – ou plutôt de bonne lisibilité – de toujours déclarer `main()` comme étant de type `int`, et ajoutons systématiquement un `return 0` ou `return EXIT_SUCCESS` à la fin de cette routine. C'est une bonne habitude à prendre, même si nous sortons toujours du programme en invoquant `exit()`.

Lorsqu'un processus se termine normalement, en revenant de `main()` ou en invoquant `exit()`, la bibliothèque C effectue les opérations suivantes :

- Elle appelle toutes les fonctions qui ont été enregistrées à l'aide des routines `atexit()` et `on_exit()`, que nous verrons dans la prochaine section.
- Elle ferme tous les flux d'entrée-sortie, en écrivant effectivement toutes les données qui étaient en attente dans les buffers.
- Elle supprime les fichiers créés par la fonction `tmpfile()`.
- Elle invoque l'appel système `_exit()` qui terminera le processus.

L'appel système `_exit()` exécute – pour ce qui concerne le programmeur applicatif – les tâches suivantes :

- Il ferme les descripteurs de fichiers (transférant les données aux périphériques).
- Les processus fils sont adoptés par le processus 1 (*init*), qui lira leur code de retour dès qu'ils se finiront pour éviter qu'ils ne restent à l'état *zombie* de manière prolongée.
- Le processus père reçoit un signal `SIGCHLD`.
- Selon certaines conditions et si le processus est leader de session, le signal `SIGHUP` peut être envoyé à tous les processus en avant-plan sur le terminal de la session.
- Si le processus est *leader* de son groupe et s'il y a des processus stoppés dans celui-ci, tous les membres du groupe à présent orphelins reçoivent `SIGHUP` et `SIGCONT`.
- Le système se livre également à des tâches de libération des ressources verrouillées, de comptabilisation éventuelle des processus, etc. Le détail de ces opérations n'est pas d'une grande importance pour une application classique, considérons simplement que l'exécution du processus est terminée, et que ses ressources sont libérées.

Le processus devient alors un *zombie*, c'est-à-dire qu'il attend que son processus père lise son code de retour. Si le processus père ignore explicitement `SIGCHLD`, le noyau effectue automatiquement cette lecture. Si le processus père s'est déjà terminé, *init* adopte temporairement le zombie, juste le temps de lire son code de retour. Une fois cette lecture effectuée, le processus est éliminé de la liste des tâches sur le système.

Le fait d'invoquer `_exit()` à la place de `exit()` peut être utile dans certaines circonstances :

- Lorsqu'on utilise un partage des fichiers entre le processus père et le processus fils, par exemple en employant `__clone()` à la place de `fork()`. À ce moment-là, on évite de fermer les flux d'entrée-sortie, car le processus père peut encore avoir besoin de ces fichiers. Ce cas est assez rare dans des applications courantes.
- On peut enregistrer des routines pour qu'elles soient automatiquement exécutées lors de la sortie du programme par `exit()` ou `return` depuis `main()`. Ces fonctions servent généralement à faire du « ménage » ou à signaler explicitement la fin d'une transaction sur une connexion réseau. Le fait d'appeler directement `_exit()` empêchera l'exécution de ces routines.

Si on utilise `_exit()`, il ne faut pas oublier de fermer proprement tous les fichiers pour être sûr que les données temporairement en *buffer* soient écrites entièrement. De même, les éventuels fichiers temporaires créés par `tmpfile()` ne sont pas détruits automatiquement. Notons que Python nous propose bien un appel `os._exit()` même si son usage est déconseillé.

Terminaison anormale d'un processus

Un programme peut également se terminer de manière anormale, involontaire. Ceci est le cas par exemple lorsqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé. Ces actions déclenchent un signal qui, par défaut, arrête le processus en créant un fichier d'image mémoire *core*. Nous en parlerons plus longuement dans les chapitres traitant des signaux.

Une manière « propre » d'interrompre anormalement un programme (par exemple lorsqu'un bogue est découvert) est d'invoquer la fonction `abort()` en C ou `os.abort()` en Python :

```
| void abort (void);
```

Celle-ci envoie immédiatement au processus le signal `SIGABRT`, en le débloquent s'il le faut, et en rétablissant le comportement par défaut si le signal est ignoré. Il est possible de traiter ce signal en installant un gestionnaire pour effectuer quelques tâches de nettoyage avant de finir le programme.

Le problème de la fonction `abort()` ou des arrêts dus à des signaux est qu'il est difficile de déterminer ensuite à quel endroit du programme le dysfonctionnement a eu lieu. Il est possible d'autopsier le fichier *core* (à condition d'avoir inclus les informations de débogage lors de la compilation avec l'option `-g` de `gcc`), mais c'est une tâche parfois ardue. Une autre manière de détecter automatiquement les bogues est d'utiliser systématiquement la fonction `assert()` dans les parties critiques du programme. Elle est généralement implémentée sous forme d'une macro, définie dans `<assert.h>`, qui évalue l'expression qu'on lui transmet en argument. Si l'expression est vraie, elle ne fait rien. Par contre, si elle est fausse, `assert()` arrête le programme après avoir écrit un message sur la sortie d'erreur standard, indiquant le fichier source concerné, la ligne de code et le texte de l'assertion ayant échoué. Il est alors très facile de se reporter au point décrit pour rechercher le bogue.

Le langage Python propose directement un mot-clé `assert` que l'on emploie en le faisant suivre d'une condition – qui est censée être vraie – et d'un message à afficher dans le cas où elle serait finalement fausse. Dans ce cas, une exception `AssertionError` est également levée.

La macro `assert()` agit en surveillant perpétuellement que les conditions prévues pour l'exécution du code soient respectées. Voici un exemple où nous faisons volontairement échouer la seconde assertion.

```
| exemple-assert.c :  
| #include <assert.h>  
| #include <stdio.h>
```

```
#include <stdlib.h>

void fonction_reussissant (int i);
void fonction_echouant (int i);

int main (void)
{
    fonction_reussissant(5);
    fonction_echouant(5);

    return EXIT_SUCCESS;
}

void fonction_reussissant (int i)
{
    /* Cette fonction nécessite que i soit positif */
    assert(i >= 0);
    fprintf(stdout, "Ok, i est positif \n");
}

void fonction_echouant (int i)
{
    /* Cette fonction nécessite que i soit négatif */
    assert(i <= 0);
    fprintf(stdout, "Ok, i est négatif \n");
}
```

Un script équivalent en Python est présent dans les sources des exemples de ce livre. Lors de l'exécution, la première assertion passe, et le message est écrit sur stdout, mais la seconde assertion échoue, et `assert()` affiche alors le détail du problème sur stderr :

```
$ ./exemple_assert
Ok, i est positif
exemple-assert: exemple-assert.c:34: fonction_echouant: l'assertion
`i <= 0' a échoué.
Aborted (core dumped)
$
```

Nous voyons alors le grand intérêt de `assert()` : elle nous indique le nom du programme exécutable, le fichier source concerné, le numéro de la ligne, le nom de la fonction, et le texte intégral de l'assertion ayant engendré l'arrêt. De plus, elle déclenche la création d'un fichier core pouvant servir à analyser plus en détail l'état des données au moment de l'échec.

Notre exemple est assez artificiel car nous avons utilisé une macro `assert()` pour vérifier des conditions qui auraient très bien pu être analysées par une structure `if/else`

renvoyant un code d'erreur. En fait, `assert()` ne doit être utilisée que pour des circonstances ne devant jamais se produire durant l'exécution normale du programme.

En effet, lorsque la phase de débogage est terminée, on supprime toutes les assertions en définissant une constante symbolique spéciale (`NDEBUG`) à la compilation. Cela permet de gagner en efficacité en éliminant tous ces tests qui sont dorénavant inutiles.

Attention ! `assert()` étant définie comme une macro qui évalue son argument, il faut éviter totalement tous les effets de bord dans l'expression transmise. On n'utilisera donc jamais des formulations du genre : `assert(++i < 5)` ou `assert(ptr[i++] != NULL)`, car la partie assignation de ces expressions disparaît lorsqu'on passe en code de production.

Il est bon d'utiliser systématiquement `assert()` pour vérifier les arguments d'entrée d'une routine lorsqu'ils doivent, dans tous les cas, se situer dans une plage de valeurs données (taille supérieure à 0, pointeur non `NULL`...). Ceci permet d'ailleurs de documenter automatiquement les contraintes sur les arguments attendus, une ligne :

```
| assert(ptr != NULL);
```

étant aussi parlante et plus efficace qu'un commentaire :

```
| /* On suppose que le pointeur n'est jamais NULL */
```

qui risque de ne pas être mis à jour en cas de modification du code.

On notera qu'il est possible de définir avec `#define` ou de supprimer avec `#undef` la constante `NDEBUG` dans le corps même d'un module, en ré-incluant `<assert.h>` à la suite. La macro `assert()` sera lors validée ou ignorée jusqu'à la prochaine modification. Ceci permet de n'activer le débogage que dans des portions restreintes du logiciel, ou au contraire d'exclure des fonctions qui ont été totalement validées.

Face à un test précis, il est parfois difficile de décider s'il faut l'incorporer dans une assertion ou dans une vérification plus classique avec un message sur `stderr`. La règle est que le code définitif qui sera soumis à l'utilisateur ne pourra en aucun cas faire échouer une assertion. La fonction `assert()` est un outil de débogage et pas une méthode de sortie sur erreur. C'est pourquoi, si une demande d'allocation mémoire échoue, la gestion d'erreur doit être effectuée par le programme directement, car ici ce n'est pas un bogue mais un problème de ressource indisponible temporairement. Cependant, une routine de traitement de chaîne de caractères peut refuser systématiquement un pointeur `NULL` en entrée. Dans ce cas, une assertion échouant permet de rechercher (à l'aide du fichier `core` créé) la routine appelante ayant transmis un mauvais argument.

Lorsqu'on désire basculer en code définitif pour l'utilisateur, on peut inclure à la compilation la constante `NDEBUG` sur la ligne de commande de `gcc` :

```
$ cc -Wall -DNDEBUG programme.c -o programme
```

ou dans le corps du fichier C, avant l'inclusion de `<assert.h>` :

```
#define NDEBUG  
#include <assert.h>
```

Cette dernière méthode permet de faire basculer indépendamment en mode de développement ou de production les divers modules du projet.

Notons finalement que `assert()` est implémentée dans la Glibc en appelant `abort()`, ce qui signale une terminaison anormale au processus père, comme nous en verrons le détail dans les routines `wait()`, `waitpid()`, etc.

Exécution automatique de routines de terminaison

Il est possible, grâce aux routines `atexit()` et `on_exit()` de faire enregistrer des fonctions qui seront automatiquement invoquées lorsque le processus se terminera normalement, c'est-à-dire par un retour de la fonction `main()` ou par un appel de la fonction `exit()`. Ces routines peuvent être utiles dans plusieurs cas :

- effacer des fichiers temporaires ou au contraire enregistrer les préférences de l'utilisateur, ou l'historique des actions effectuées ;
- enregistrer sur disque les structures d'une base de données maintenue en mémoire ;
- libérer les verrous sur les fichiers ou bases de données partagés ;
- signaler la fin du processus au démon de journalisation du système (*syslog*) ;
- restaurer l'état initial du terminal ;
- terminer un dialogue réseau proprement en suivant un protocole complet, plutôt que de simplement couper la connexion...

Il est toujours possible d'appeler explicitement ces routines avant de quitter l'application, mais l'avantage de ce mécanisme d'invocation automatique est double :

- On peut quitter le programme depuis plusieurs endroits en appelant `exit()`, ou revenir de la fonction `main()` sans avoir à se soucier des routines de terminaison. Elles seront appelées systématiquement quel que soit le cas.
- Lorsqu'on définit une bibliothèque de fonctions pouvant être réutilisées dans plusieurs programmes, et que celle-ci nécessite un traitement final avant la fin du processus (par exemple pour l'un des cas cités précédemment), il est plus sûr d'enregistrer avec `atexit()` la routine désirée plutôt que de demander au programmeur qui utilisera la bibliothèque d'appeler une fonction finale.

Le prototype de `atexit()` est déclaré dans `<stdlib.h>`, ainsi :

```
int atexit (void * routine (void));
```

En d'autres termes, on doit lui transmettre un pointeur sur une routine de type :

```
void routine_terminaison (void);
```

Lorsqu'elle réussit, `atexit()` renvoie 0. Sinon, elle renvoie une valeur non nulle. La norme C *Ansi* indique qu'on peut enregistrer au minimum 32 fonctions. La Glibc ne fixe pas de limites, en allouant dynamiquement les structures de données nécessaires à la mémorisation.

En Python, il existe un module `atexit` proposant la méthode `atexit.register()` à qui l'on passe en argument la fonction de nettoyage et ses éventuels paramètres. On peut également l'utiliser comme un décorateur. Ainsi,

```
import atexit

def cleanup_function():
    ...
    ...

atexit.register(cleanup_function)
```

est équivalent à :

```
import atexit

@atexit.register
def cleanup_function():
    ...
    ...
```

Les fonctions mémorisées avec `atexit()` sont invoquées, en sortie, dans l'ordre inverse de leur enregistrement. Une fonction enregistrée deux fois est invoquée deux fois. Il n'y a pas de possibilité de « déprogrammer » une fonction mémorisée. La meilleure solution pour désactiver une routine de terminaison est d'utiliser une variable globale que la routine consultera pour savoir si elle doit agir ou non.

Lorsqu'on appelle la fonction `exit()` depuis l'intérieur d'une routine de terminaison, elle n'a pas d'effet (en particulier, le programme ne boucle pas sur cette routine). Par contre, si on invoque l'appel système `_exit()`, la sortie est immédiate, sans appeler les éventuelles autres routines de terminaison. Les routines de terminaison sont invoquées avant la fermeture systématique des fichiers ouverts et l'effacement des fichiers temporaires fournis par `tmpfile()`. Il est donc possible de les utiliser encore dans les routines de terminaison.

Voici un exemple d'utilisation de la fonction `atexit()`. Nous appellerons trois routines, et la deuxième sera enregistrée deux fois. On voit également que l'appel système `exit()` n'a pas d'effet lorsqu'il est appelé depuis l'intérieur d'une routine de terminaison.

```
exemple-atexit.c :
#include <stdio.h>
#include <stdlib.h>

void sortie_1 (void);
void sortie_2 (void);
void sortie_3 (void);

int main (void)
{
    if (atexit(sortie_3) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_3()\n");
    if (atexit(sortie_2) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_2()\n");
    if (atexit(sortie_2) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_2()\n");
    if (atexit(sortie_1) != 0)
        fprintf(stderr, "Impossible d'enregistrer sortie_1()\n");
    fprintf(stdout, "Allez... on quitte en revenant de main()\n");
    return EXIT_SUCCESS;
}

void sortie_1 (void)
{
    fprintf(stdout, "Sortie_1 : appelle exit()\n");
    exit(EXIT_SUCCESS);
}

void sortie_2 (void)
{
    fprintf(stdout, "Sortie_2\n");
}

void sortie_3 (void)
{
    fprintf(stdout, "Sortie_3\n");
}
```

Et voici un équivalent en Python :

```
./exemple-atexit.py :
#!/usr/bin/python
from __future__ import print_function
import atexit
import sys
```

```
def sortie_1():
    print("Sortie_1: calling sys.exit()")
    sys.exit(0)

def sortie_2():
    print("Sortie_2")

def sortie_3():
    print("Sortie_3")

atexit.register(sortie_3)
atexit.register(sortie_2)
atexit.register(sortie_2)
atexit.register(sortie_1)
print("Normal termination at end of script.")
```

L'exécution a lieu ainsi :

```
$ ./exemple-atexit
Allez... on quitte en revenant de main()
Sortie_1 : appelle exit()
Sortie_2
Sortie_2
Sortie_3
$
```

En revanche, si on remplace `exit()` par `_exit()` dans `sortie_1()`, on obtient :

```
$ ./exemple-atexit
Allez... on quitte en revenant de main()
Sortie_1 : appelle _exit()
$
```

Il existe une seconde fonction permettant d'enregistrer des routines de terminaison : `on_exit()`. Il s'agit d'une extension *Gnu*. La routine de terminaison recevra lors de son invocation deux arguments : le premier est un entier correspondant au code transmis à `exit()` ou `return` de `main()`. Le second argument est un pointeur `void *`, et la valeur de cet argument est programmée lors de l'appel de `on_exit()`.

Le prototype de `on_exit()` est déclaré dans `<stdlib.h>`, ainsi :

```
| int on_exit (void (* fonction) (int, void *), void * argument);
```

Le second argument est souvent utilisé pour passer un pointeur de fichier `FILE *` à terminer de traiter avant de finir. Voici un exemple où la routine de terminaison ne

fait que fermer le fichier transmis s'il est non `NULL`. Bien sûr, elle pourrait effectuer une tâche bien plus compliquée, comme mettre à jour un en-tête ou une table des matières en début de fichier.

```
exemple-on-exit.c :
#include <stdio.h>
#include <stdlib.h>

void gestion_sortie (int code, void * pointeur);

int main (void)
{
    FILE * fp;

    fp = fopen("exemple-atexit.c", "r");
    if (on_exit(gestion_sortie, (void *) fp) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");
    fp = fopen("exemple-on-exit.c", "r");
    if (on_exit(gestion_sortie, (void *) fp) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");

    if (on_exit(gestion_sortie, NULL) != 0)
        fprintf(stderr, "Erreur dans on_exit \n");
    fprintf(stdout, "Allez... on quitte en revenant de main()\n");
    return 4;
}

void gestion_sortie (int code, void * pointeur)
{
    fprintf(stdout, "Gestion Sortie appelee... code %d\n", code);
    if (pointeur == NULL) {
        fprintf(stdout, "Pas de fichier a fermer \n");
    } else {
        fprintf(stdout, "Fermeture d'un fichier \n");
        fclose((FILE *) pointeur);
    }
}
```

L'exécution suivante nous montre que les fonctions sont appelées, comme pour `atexit()`, dans l'ordre inverse de leur programmation (le pointeur `NULL` programmé en dernier apparaît en premier). Nous voyons également que le code de retour de `main()`, 4, est bien transmis à la routine de terminaison.

```
$ ./exemple-on-exit
Allez... on quitte en revenant de main()
Gestion Sortie appelée... code 4
Pas de fichier à fermer
Gestion Sortie appelée... code 4
```

```
Fermeture d'un fichier
Gestion Sortie appelée... code 4
Fermeture d'un fichier
$
```

Attendre la fin d'un processus fils

L'une des notions fondamentales dans la conception des systèmes Unix est la mise à disposition de l'utilisateur d'un grand nombre de petits utilitaires très spécialisés et très configurables grâce à des options en ligne de commande. Ces petits utilitaires peuvent être associés, par des redirections d'entrée-sortie, en commandes plus complexes, et regroupés dans des fichiers scripts simples à écrire et à déboguer.

Il est primordial dans ces scripts de pouvoir déterminer si une commande a réussi à effectuer son travail correctement ou non. On imagine donc l'importance qui peut être portée à la lecture du code de retour d'un processus. Cette importance est telle qu'un processus qui se termine passe automatiquement par un état spécial, *zombie*, en attendant que le processus père ait lu son code de retour. Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l'état *zombie*. Voici un exemple, dans lequel le processus fils attend deux secondes avant de se terminer, tandis que le processus père affiche régulièrement l'état de son fils en invoquant la commande `ps`.

```
exemple-zombie-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    pid_t pid;
    char commande[128];

    if ((pid = fork()) < 0) {
        fprintf(stderr, "echec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        /* Processus fils */
        sleep(2);
        fprintf(stdout, "Le processus fils %ld se termine \n",
                (long) getpid());
        exit(EXIT_SUCCESS);
    } else {
```

```

    /* Processus père */
    snprintf (commande, 128, "ps %ld", (long) pid);
    system(commande);
    sleep(1);
    system(commande);
    sleep(1);
    system(commande);
    sleep(1);
    system(commande);
    sleep(1);
    system(commande);
    sleep(1);
    system(commande);
}
return EXIT_SUCCESS;
}

```

En Python, l'écriture est proche ;

```

exemple-zombie-1.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys
import time

pid = os.fork()

if (pid == 0):
    time.sleep(2)
    print("Child process will terminate now!")

else:
    for i in range(1,5):
        os.system("ps {0}".format(pid))
        time.sleep(1)

sys.exit(0)

```

Le S en deuxième colonne indique que le processus fils est endormi au début, puis il se termine et passe à l'état zombie Z :

```

$ ./exemple_zombie_1
PID TTY      STAT   TIME COMMAND
949 pts/0    S       0:00 ./exemple_zombie_1
PID TTY      STAT   TIME COMMAND
949 pts/0    S       0:00 ./exemple_zombie_1

```

```

Le processus fils 949 se termine
  PID TTY          STAT       TIME COMMAND
  949 pts/0        Z          0:00 [exemple_zombie_ <defunct>]
  PID TTY          STAT       TIME COMMAND
  949 pts/0        Z          0:00 [exemple_zombie_ <defunct>]
  PID TTY          STAT       TIME COMMAND
  949 pts/0        Z          0:00 [exemple_zombie_ <defunct>]
  PID TTY          STAT       TIME COMMAND
  949 pts/0        Z          0:00 [exemple_zombie_ <defunct>]
$ ps 949
  PID TTY          STAT       TIME COMMAND
$

```

Lorsque le processus père se finit, on invoque manuellement la commande `ps`, et on s'aperçoit que le fils zombie a disparu. Dans ce cas, le processus numéro 1, `init`, adopte le processus fils orphelin et lit son code de retour, ce qui provoque sa disparition. Sur les systèmes n'utilisant pas la version Système V de `init`, (par exemple, `Systemd` ou `Upstart`), il se peut qu'un autre processus spécifique soit chargé de ce travail. Il est généralement visible dans la commande `ps` sous le nom « `init --user` ».

Dans ce second exemple, le processus père va se terminer au bout de 2 secondes, alors que le fils va continuer à afficher régulièrement le PID de son père.

```

exemple-zombie-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "echec fork()\n");
        exit(EXIT_FAILURE);
    }

    if (pid != 0) {
        /* Processus père */
        fprintf(stdout, "Père : mon PID est %ld\n", (long)getpid());
        sleep(2);
        fprintf(stdout, "Père : je me termine \n");
        exit(EXIT_SUCCESS);
    } else {
        /* Processus fils */

```

```

        fprintf(stdout, "Fils : mon pere est %ld\n",
                    (long)getppid ());
        sleep(1);
        fprintf(stdout, "Fils : mon pere est %ld\n",
                    (long)getppid ());
        sleep(1);
        fprintf(stdout, "Fils : mon pere est %ld\n",
                    (long)getppid ());
        sleep(1);
        fprintf(stdout, "Fils : mon pere est %ld\n",
                    (long)getppid ());
        sleep(1);
        fprintf(stdout, "Fils : mon pere est %ld\n",
                    (long)getppid ());
    }
    return EXIT_SUCCESS;
}

```

Voici un équivalent en Python.

```

exemple-zombie-2.py :
#!/usr/bin/python
from __future__ import print_function
import os
import time

if (os.fork() != 0):
    print("[PARENT] My PID is", os.getpid())
    time.sleep(2)
    print("[PARENT] Terminating now!")
else:
    for i in range(1,5):
        print("[CHILD] My parent is", os.getppid())
        time.sleep(1)

```

L'exécution suivante montre bien que le processus 1 adopte le processus fils dès que le père se termine. Au passage, on remarquera que, aussitôt le processus père terminé, le shell reprend la main et affiche immédiatement son symbole d'accueil (\$) :

```

$ ./exemple_zombie_2
Père : mon PID est 1006
Fils : mon père est 1006
Fils : mon père est 1006
Père : je me termine
$ Fils : mon père est 1
Fils : mon père est 1
Fils : mon père est 1

```

Pour lire le code de retour d'un processus fils, il existe quatre fonctions : `wait()`, `waitpid()`, `wait3()` et `wait4()`. Toutefois, `wait3()` et `wait4()` ne sont pas utilisés dans les applications courantes, et nous n'étudierons que `wait()` et `waitpid()`.

La fonction `wait()` de la bibliothèque C est déclarée dans `<sys/wait.h>` ; en Python, l'implémentation est `os.wait()`. Le prototype est le suivant :

```
| pid_t wait (int * status);
```

Lorsqu'on l'invoque, elle bloque le processus appelant jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID du fils terminé. Si le pointeur `status` est non `NULL`, il est renseigné avec une valeur informant sur les circonstances de la mort du fils. Si un processus fils était déjà en attente à l'état zombie, `wait()` revient immédiatement. Si on n'est pas intéressé par les circonstances de la fin du processus fils, il est tout à fait possible de fournir un argument `NULL`.

La manière dont sont organisées les informations au sein de l'entier `status` est opaque, et il faut utiliser les macros suivantes pour analyser les circonstances de la fin du fils.

- `WIFEXITED(status)` est vraie si le processus s'est terminé de son propre chef en invoquant `exit()` ou en revenant de `main()`. On peut obtenir le code de retour du processus fils, c'est-à-dire la valeur transmise à `exit()`, en appelant `WEXITSTATUS(status)`.
- `WIFSIGNALED(status)` indique que le fils s'est terminé à cause d'un signal, y compris le signal `SIGABRT` envoyé lorsqu'il appelle `abort()`. Le numéro du signal ayant tué le processus fils est disponible en utilisant la macro `WTERMSIG(status)`. À ce moment, la macro `WCOREDUMP(status)` signale si une image mémoire *core* a été créée.
- `WIFSTOPPED(status)` indique que le fils est stoppé temporairement. Le numéro du signal ayant stoppé le processus fils est accessible en utilisant `WSTOPSIG(status)`.

Attention, les macros `WEXITSTATUS`, `WTERMSIG` et `WSTOPSIG` n'ont de sens que si les macros `WIFxxx` correspondantes ont renvoyé une valeur vraie.

La fonction `wait()` peut échouer et renvoyer `-1`, en plaçant l'erreur `ECHILD` dans `errno` si le processus appelant n'a pas de fils. Dans notre premier exemple, le processus père va se dédoubler en une série de fils qui se termineront de manières variées. Le processus père restera en boucle sur `wait()` jusqu'à ce qu'il ne reste plus de fils.

```
| exemple-wait-1.c :  
| #include <stdio.h>  
| #include <stdlib.h>  
| #include <unistd.h>  
| #include <signal.h>  
| #include <sys/wait.h>  
  
| void affichage_type_de_termination (pid_t pid, int status);
```



```
int processus_fils                (int numero_fils);

int main (void)
{
    pid_t pid;
    int    status;
    int    numero_fils;

    for (numero_fils = 0; numero_fils < 4; numero_fils++) {
        switch (fork()) {
            case -1 :
                fprintf(stderr, "Erreur dans fork()\n");
                exit(EXIT_FAILURE);
            case 0 :
                printf("Fils %d : PID = %ld\n",
                       numero_fils, (long)getpid());
                return processus_fils(numero_fils);
            default :
                /* Processus père */
                break;
        }
    }
    /* Ici il n'y a plus que le processus père */
    while ((pid = wait(& status)) > 0)
        affichage_type_de_termination(pid, status);
    return EXIT_SUCCESS;
}

void affichage_type_de_termination (pid_t pid, int status)
{
    printf("Le processus %ld ", (long)pid);
    if (WIFEXITED(status)) {
        printf("s'est termine normalement avec le code %d\n",
               WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("s'est termine a cause du signal %d (%s)\n",
               WTERMSIG(status),
               sys_siglist[WTERMSIG(status)]);
    } if (WCOREDUMP(status)) {
        fprintf(stdout, "Fichier image core cree \n");
    }
    } else if (WIFSTOPPED(status)) {
        printf("s'est arrete a cause du signal %d (%s)\n",
               WSTOPSIG(status),
               sys_siglist[WSTOPSIG(status)]);
    }
}

int processus_fils (int numero_fils)
{

```

```
switch (numero_fils) {
    case 0 :
        return 1;
    case 1 :
        exit(2);
    case 2 :
        abort();
    case 3 :
        raise(SIGUSR1);
}
return numero_fils;
}
```

L'exécution suivante montre bien les différents types de fin des processus fils :

```
$ ./exemple_wait_1
Fils 0 : PID = 1353
Fils 1 : PID = 1354
Fils 2 : PID = 1355
Le processus 1355 s'est terminé à cause du signal 6 (Aborted)
Fichier image core créé
Le processus 1354 s'est terminé normalement avec le code 2
Le processus 1353 s'est terminé normalement avec le code 1
Fils 3 : PID = 1356
Le processus 1356 s'est terminé à cause du signal 10 (User defined 1)
$
```

Notons qu'il n'y a pas de différence entre un retour de la fonction `main()` (fils 0, *PID* 1353) et un appel `exit()` (fils 1, *PID* 1354). De même, on voit que l'appel `abort()` (fils 2, *PID* 1355) se traduit bien par un envoi du signal `SIGABRT` (6 sur notre machine), avec création d'un fichier `core`. Le signal `SIGUSR1` termine le processus mais ne crée pas d'image `core`.

Il y a deux inconvénients avec la fonction `wait()`, qui ont conduit à développer la fonction `waitpid()` que nous allons voir ci-après. Le premier problème, c'est que l'appel reste bloquant tant qu'aucun fils ne s'est terminé. Il n'est donc pas possible d'appeler systématiquement `wait()` dans une boucle principale du programme pour savoir où en est le fils. La solution est d'installer un gestionnaire pour le signal `SIGCHLD` qui est émis dès qu'un fils se termine ou est stoppé temporairement.

Le second problème vient du fait qu'il n'est pas possible d'attendre la fin d'un fils particulier. Dans ce cas, il faut alimenter dans le gestionnaire du signal `SIGCHLD` une liste des fils terminés, qu'on consultera dans le programme principal en attente d'un processus donné. Il ne faut pas oublier de bloquer temporairement `SIGCHLD` lors de la consultation de la liste, pour éviter qu'elle ne soit modifiée pendant ce temps par l'arrivée d'un signal.

Pour pallier ces deux problèmes, un appel système supplémentaire est disponible, `waitpid()`, dont le prototype est déclaré dans `<sys/wait.h>` ainsi :

```
pid_t waitpid (pid_t pid, int * status, int options);
```

Le premier argument, `pid`, permet de déterminer le processus fils dont on désire attendre la fin.

- Si `pid` est strictement positif, la fonction attend la fin du processus dont le *PID* correspond à cette valeur.
- Si `pid` vaut 0, on attend la fin de n'importe quel processus fils appartenant au même groupe que le processus appelant.
- Si `pid` vaut -1, on attend la fin de n'importe quel fils, comme avec la fonction `wait()`.
- Si `pid` est strictement inférieur à -1, on attend la fin de n'importe quel processus fils appartenant au groupe de processus dont le numéro est `-pid`.
- Le deuxième argument, `status`, a exactement le même rôle que `wait()`.

Le troisième argument permet de préciser le comportement de `waitpid()`, en associant par un éventuel OU binaire les constantes suivantes, voir tableau ci-après.

Nom	Signification
WNOHANG	Ne pas rester bloqué si aucun processus correspondant aux spécifications fournies par l'argument <code>pid</code> n'est terminé. Dans ce cas, <code>waitpid()</code> renverra 0.
WUNTRACED	Accéder également aux informations concernant les processus fils temporairement stoppés. C'est dans ce cas que les macros <code>WIFSTOPPED(status)</code> et <code>WSTOPSIG(status)</code> prennent leur signification.

Comme on le devine, il est aisé d'implémenter `wait()` à partir de `waitpid()` :

```
pid_t mon_wait (int * status)
{
    return waitpid(-1, status, 0);
}
```

Nous allons utiliser un programme de démonstration dans lequel un processus fils, qui reste en boucle, sera surveillé par le processus père, alors qu'un second fils, qui se termine au bout de quelques secondes, n'est pas pris en considération. Nous agirons sur une seconde console pour examiner les statuts des fils avec la commande `ps`, et pour stopper et relancer le premier fils.

```
exemple-wait-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
```

```

#include <sys/wait.h>

int main (void)
{
    pid_t    pid;
    int      status;

    switch (pid = fork()) {
        case -1 :
            fprintf(stderr, "Erreur dans fork()\n");
            exit(EXIT_FAILURE);
        case 0 : /* Fils 1 */
            fprintf(stdout, "Fils 1 : PID = %ld\n", (long)getpid());
            while (1)
                pause();
        default : /* Père */
            break;
    }

    /* Créons un fils qu'on n'attend pas */
    switch (fork ()) {
        case -1 :
            fprintf(stderr, "Erreur dans fork()\n");
            exit(EXIT_FAILURE);
        case 0 : /* Fils 2 */
            fprintf(stdout, "Fils 2 : PID = %ld\n", (long)getpid());
            sleep(2);
            exit(EXIT_SUCCESS);
        default : /* Père */
            break;
    }

    while (1) {
        sleep(1);
        if (waitpid(pid, & status, WUNTRACED | WNOHANG) > 0) {
            if (WIFEXITED(status)) {
                fprintf(stdout, "%ld termine par exit (%d)\n",
                        (long)pid, WEXITSTATUS(status));
                exit(EXIT_SUCCESS);
            } else if (WIFSIGNALED (status)) {
                fprintf(stdout, "%ld termine par signal %d\n",
                        (long)pid, WTERMSIG(status));
                exit(EXIT_SUCCESS);
            } else if (WIFSTOPPED(status)) {
                fprintf(stdout, "%ld stoppe par signal %d\n",
                        (long)pid, WSTOPSIG(status));
            }
        }
    }

    return EXIT_SUCCESS;
}

```

L'exécution suivante montre en seconde colonne les actions depuis l'autre *Xterm* :

```
$ ./exemple-wait-2
Fils 1 : PID = 1525
Fils 2 : PID = 1526

$ ps 1525 1526
  PID TTY          STAT COMMAND
 1525 pts/0        S      ./exemple_wait_2
 1526 pts/0        Z      [exemple_wait_2 <defunct>]
$ kill -STOP 1525
1525 stoppé par signal 19

$ ps 1525 1526
  PID TTY          STAT COMMAND
 1525 pts/0        T      ./exemple_wait_2
 1526 pts/0        Z      [exemple_wait_2 <defunct>]
$ kill -CONT 1525
$ kill -TSTP 1525
1525 stoppé par signal 20

$ kill -CONT 1525
$ kill -TERM 1525
1525 terminé par signal 15
$
```

Nous voyons que le fils 2, de PID 1526, reste à l'état zombie dès qu'il se finit car le père ne demande pas son code de retour. Le fait d'avoir appelé l'option `WUNTRACED` nous permet d'être informé lorsque le processus fils 1 est temporairement stoppé par un signal. Les fonctions `wait()` et `waitpid()` sont définies par SUSv4, contrairement aux deux fonctions `wait3()` et `wait4()`, qui sont d'inspiration BSD. Elles permettent, par rapport à `wait()` ou `waitpid()`, d'obtenir des informations supplémentaires sur le processus qui s'est terminé. Ces renseignements sont transmis par l'intermédiaire d'une structure `rusage`, définie dans `<sys/resource.h>`.

Depuis la version 2.6.9 du noyau Linux, un nouvel appel système a fait son apparition : `waitid()` qui permet de préciser les changements d'état des processus fils attendus, et de recevoir des informations supplémentaires :

```
int waitid (idtype_t idtype, id_t id, siginfo_t * info,
            int options);
```

Les deux premiers champs permettent de sélectionner les fils qui nous intéressent :

- si `idtype` vaut `P_PID`, on s'intéresse au fils dont le *PID* est précisé en second argument ;
- si `idtype` vaut `P_GID`, on s'intéresse à tous les fils dont le *GID* est indiqué en seconde position ;

- si `idtype` vaut `P_ALL`, on s'intéresse à tous les fils, et le second argument est ignoré.

Dans l'argument `options`, on peut préciser, outre `WNOHANG` et `WNOWAIT` comme `waitpid()` :

- `WEXITED` : on attend la fin d'un fils sélectionné ;
- `WSTOPPED` : on attend l'arrêt temporaire d'un fils ;
- `WCONTINUED` : on attend le redémarrage après l'arrêt temporaire d'un fils.

Lorsqu'une de ces conditions se présente, `waitid()` se termine en renvoyant zéro et en remplissant les champs suivant de la structure `siginfo_t` (que l'on retrouvera ultérieurement dans le traitement des signaux temps réel).

```
exemple-waitid.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>

int main (void)
{
    pid_t    pid;
    siginfo_t info;
    pid = fork();
    if (pid == -1) {
        fprintf(stderr, "Erreur dans fork()\n");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        while (1) {
            printf("[Fils %ld]: en fonctionnement\n",
                (long int) getpid());
            sleep(3);
            printf("Fils %ld va se stopper\n", (long int) getpid());
            raise(SIGSTOP);
        }
    }
    while (1) {
        fprintf(stderr, "    [Pere] : en attente\n");
        if (waitid(P_ALL, 0, & info, WSTOPPED) == 0) {
            if (info.si_code == CLD_STOPPED) {
                printf("    [Pere] : mon fils %ld arrete par signal %d\n",
                    (long)pid, info.si_status);

                sleep(2);
                printf("    [Pere] Je le relance\n");
                kill(info.si_pid, SIGCONT);
            }
        }
    }
}
```

```
}  
    return EXIT_SUCCESS;  
}
```

Champs	Signification
<code>si_pid</code>	<i>PID</i> du fils terminé.
<code>si_uid</code>	<i>UID</i> réel du fils terminé.
<code>si_code</code>	CLD_EXITED si le fils s'est terminé avec <code>exit()</code> ou <code>return</code> ; CLD_KILLED s'il a été tué par un signal ; CLD_STOPPED en cas d'arrêt temporaire et CLD_CONTINUED s'il a été relancé.
<code>si_status</code>	Si <code>si_code</code> contient CLD_EXITED : valeur de retour fournie à <code>exit()</code> ou <code>return</code> ; Si <code>si_code</code> contient CLD_KILLED, CLD_STOPPED, CLD_CONTINUED : numéro du signal reçu.

Signaler une erreur

Il y a des cas où la gestion d'erreur doit être moins drastique qu'un arrêt anormal du programme. Le langage Python propose pour cela le mécanisme des exceptions qui permettent en cas de problème une reprise automatique du programme en un point bien connu. En C, le traitement est plus compliqué : les appels système remplissent la variable globale externe `errno` avec une valeur numérique entière représentant le type d'erreur qui s'est produite. Toutes les valeurs représentant une erreur sont non nulles.

Attention : le fait que `errno` soit remplie avec une valeur non nulle n'est pas suffisant pour en déduire qu'une erreur s'est produite. Il faut pour cela que l'appel système échoue explicitement (la plupart du temps en renvoyant `-1` et non pas `0`). Ceci est encore plus vrai avec des routines de bibliothèque qui peuvent invoquer plusieurs appels système et remédier aux conditions d'erreur. `errno` sera alors modifiée à plusieurs reprises avant le retour de la fonction. SUSv4 autorise les fonctions de bibliothèque à modifier `errno` même lorsqu'elles réussissent à faire leur travail.

Il existe des constantes symboliques représentant chaque erreur possible. Elles sont définies dans le fichier `<errno.h>`. Toutefois, celui-ci inclut automatiquement `<bits/errno.h>`, `<linux/errno.h>` et `<asm/errno.h>`, qui définissent l'essentiel des constantes d'erreur. Il est bon de connaître l'existence de ces fichiers car un coup d'œil rapide permet parfois d'identifier une erreur qu'on n'avait pas prévue, à partir de son numéro. Les principales erreurs qu'on rencontre fréquemment dans les appels système sont décrites dans le tableau suivant. Nous en avons limité la liste aux plus courantes. Il en existe de nombreuses autres, par exemple dans le domaine de la programmation réseau, que nous détaillerons le moment venu.

Nom	Signification
E2BIG	La liste d'arguments fournie à l'une des fonctions de la famille <code>exec()</code> est trop longue.
EACCES	L'accès demandé est interdit, par exemple dans une tentative d'ouverture de fichier avec <code>open()</code> .
EAGAIN	L'opération est momentanément impossible, il faut réessayer. Par exemple, on demande une lecture non bloquante avec <code>read()</code> , mais aucune donnée n'est encore disponible.
EBADF	Le descripteur de fichier transmis à l'appel-système, par exemple <code>close()</code> , est invalide.
EBUSY	Le répertoire ou le fichier considéré est en cours d'utilisation. Ainsi, <code>umount()</code> ne peut démonter un périphérique si un processus l'utilise alors comme répertoire de travail.
ECHILD	Le processus attendu par <code>waitpid()</code> n'existe pas ou n'est pas un fils du processus appelant.
EDEADLK	Le verrouillage en écriture par <code>fcntl()</code> du fichier demandé conduirait à un blocage.
EDOM	La valeur transmise à la fonction mathématique est hors de son domaine de définition. Par exemple, on appelle <code>acos()</code> avec une valeur inférieure à -1 ou supérieure à 1.
EEXIST	Le fichier ou le répertoire indiqué pour une création existe déjà. Par exemple, avec <code>open()</code> , <code>mkdir()</code> , <code>mknod()</code> ...
EFAULT	Un pointeur transmis en argument pointe en dehors de l'espace d'adressage du processus. Cette erreur révèle un bogue grave dans le programme.
EFBIG	On a tenté de créer un fichier de taille plus grande que la limite autorisée pour le processus.
EINTR	L'appel système a été interrompu par l'arrivée d'un signal qui a été intercepté par un gestionnaire installé par le processus.
EINVAL	Un argument de type entier, ou représenté par une constante symbolique, a une valeur invalide ou incohérente.
EIO	Une erreur d'entrée-sortie de bas niveau s'est produite pendant un accès au fichier.
EISDIR	Le descripteur de fichier transmis à l'appel système, par exemple <code>read()</code> , correspond à un répertoire.
ELOOP	On a rencontré trop de liens symboliques successifs, il y a probablement une boucle entre eux.
EMFILE	Le processus a atteint le nombre maximal de fichiers ouverts simultanément.
EMLINK	On a déjà créé le nombre maximal de liens sur un fichier.
ENAMETOOLONG	Le chemin d'accès transmis en argument, par exemple pour <code>chdir()</code> , est trop long.
ENFILE	On a atteint le nombre maximal de fichiers ouverts simultanément sur l'ensemble du système.
ENODEV	Le fichier spécial de périphérique n'est pas valide, par exemple dans l'appel système <code>mount()</code> .
ENOENT	Un répertoire contenu dans le chemin d'accès fourni à l'appel système n'existe pas, ou est un lien symbolique pointant dans le vide.
ENOEXEC	Le fichier exécutable indiqué à l'un des appels de la famille <code>exec()</code> n'est pas dans un format reconnu par le noyau.

Nom	Signification
ENOLCK	Il n'y a plus de place dans la table système pour ajouter un verrou avec l'appel système <code>fcntl()</code> .
ENOMEM	Il n'y a plus assez de place mémoire pour allouer une structure supplémentaire dans une table système.
ENOSPC	Le périphérique sur lequel on veut créer un nouveau fichier ou écrire des données supplémentaires n'a plus de place disponible.
ENOSYS	La fonctionnalité demandée par l'appel-système n'est pas disponible dans le noyau. Il peut s'agir d'un problème de version ou d'options lors de la compilation du noyau.
ENOTBLK	Le fichier spécial qu'on tente de monter avec <code>mount()</code> ne représente pas un périphérique de type « bloc ». Cette erreur n'est pas décrite dans SUSv4.
ENOTDIR	Un élément du chemin d'accès fourni n'est pas un répertoire.
ENOTEMPTY	Le répertoire qu'on veut détruire n'est pas vide, ou le nouveau nom d'un répertoire à renommer existe déjà et n'est pas vide.
ENOTTY	Le fichier indiqué en argument à <code>ioctl()</code> n'est pas un terminal.
ENXIO	Le fichier spécial indiqué n'est pas reconnu par le noyau (par exemple un numéro de nœud majeur invalide).
EPERM	Le processus appelant n'a pas les autorisations nécessaires pour effectuer l'opération prévue. Souvent, il s'agit d'une fonctionnalité réservée à root.
EPIPE	Tentative d'écriture avec <code>write()</code> dans un tube dont l'autre extrémité a été fermée par le processus lecteur. Cette erreur n'est envoyée que si le signal <code>SIGPIPE</code> est bloqué ou ignoré.
ERANGE	Une valeur numérique attendue dans une fonction mathématique est invalide.
EROFS	On tente une modification sur un fichier appartenant à un système de fichiers monté en lecture seule.
ESPIPE	On essaye de déplacer le pointeur de lecture, avec <code>lseek()</code> , sur un descripteur de fichier ne le permettant pas, comme un tube ou une socket.
ESRCH	Le processus visé, par exemple avec <code>kill()</code> , n'existe pas.
ETXTBSY	On essaye d'exécuter un fichier déjà ouvert en écriture par un autre processus.
EWOULDBLOCK	Synonyme de <code>EAGAIN</code> qu'on rencontre dans la description de nombreuses fonctionnalités réseau.
EXDEV	On essaye de renommer un fichier ou de créer un lien matériel entre deux systèmes de fichier différents.

On voit, à l'énoncé d'une telle liste (qui ne représente qu'un tiers environ de toutes les erreurs pouvant se produire sous Linux), qu'il est difficile de gérer tous les cas à chaque appel système effectué par le programme.

Alors que faire si, par exemple, l'appel système `open()` échoue ? À lui seul, il peut déjà renvoyer une bonne quinzaine d'erreurs différentes. Tout dépend du degré de convivialité du

logiciel développé. Dans certains cas, on peut se contenter de mettre un message sur la sortie d'erreur « impossible d'ouvrir le fichier xxx », et arrêter le programme. À l'opposé, on peut aussi diagnostiquer qu'un des répertoires du chemin d'accès est invalide et afficher pour l'utilisateur la liste des répertoires du même niveau pour qu'il corrige son erreur.

Ainsi, certaines erreurs ne doivent jamais se produire dans un programme bien débogué. C'est le cas de `EFAULT`, qui signale un pointeur mal initialisé, de `EDOM` ou `ERANGE`, qui indiquent qu'une fonction mathématique a été appelée sans vérifier si les variables appartiennent à son domaine de définition. Ces cas-là peuvent être contrôlés dans des appels à `assert()`, car il s'agit de bogues à éliminer avant la distribution du logiciel.

Dans d'autres cas, le problème concerne le système, et le pauvre programme ne peut rien faire pour corriger l'erreur. C'est le cas par exemple de `ENOMEM`, qui indique que le noyau n'a plus assez de place mémoire, ou de `ENFILE`, qui se produit lorsque le nombre maximal de fichiers ouverts sur le système est atteint. Il n'y a guère d'autres possibilités alors que d'abandonner l'opération après avoir signalé le problème à l'utilisateur.

La règle absolue est de ne jamais passer sous silence les conditions d'erreur qui paraissent improbables. Si une application doit être distribuée largement et utilisée pendant de longues heures par ses utilisateurs, il est pratiquement certain qu'elle sera un jour confrontée au problème d'une partition disque saturée. Ignorer le code de retour de `write()` reviendra à ne pas sauvegarder le travail de l'utilisateur, alors qu'un simple message d'avertissement lui aurait permis d'effacer des fichiers inutiles et de refaire une sauvegarde.

Si on ne désire pas traiter au cas par cas toutes les erreurs possibles, on peut employer la fonction `strerror()`, déclarée dans `<string.h>` ainsi :

```
| char * strerror (int numero_erreur);
```

Cette fonction renvoie un pointeur sur une chaîne de caractères statique décrivant l'erreur produite. Cette chaîne de caractères peut être écrasée à chaque nouvel appel de `strerror()`. Pour éviter ce problème dans le cas d'une programmation multithread, on peut utiliser l'extension *Gnu* `strerror_r()`, déclarée ainsi :

```
| char * strerror_r (int numero_erreur, char * chaine,  
                   size_t longueur);
```

Cette fonction n'écrit jamais dans la chaîne plus d'octets que la longueur indiquée, y compris le caractère nul final.

Dans tous les cas, il convient de consulter la page de manuel des appels système et des fonctions de bibliothèque employés (en espérant que les informations soient à jour), et de prévoir une gestion adéquate pour les erreurs les plus fréquentes. Une gestion générique peut être mise en place pour les cas les plus rares. Prenons l'exemple de `open()`. Une manière assez simple mais correcte d'opérer serait :

```

while (1) {
    if ((fd = open(fichier, mode)) == -1) {
        assert(errno != EFAULT);
        switch (errno) {
            case EMFILE :
            case ENFILE :
            case ENOMEM :
                fprintf(stderr, "Erreur critique : %s\n",
                    strerror(errno));
                return -1;
            default :
                fprintf(stderr, "Erreur d'ouverture de %s : %s\n",
                    fichier, strerror(errno));
                break;
        }
        if (corriger_le_nom_pour_reessayer() != 0)
            /* L'utilisateur préfère abandonner */
            return -1;
        else
            continue; /* Recommencer */
    } else {
        /* Pas d'erreur */
        break;
    }
    return 0;
}

```

Cela permet à la fois de différencier les erreurs irrécupérables de celles qu'on peut corriger, et donne à l'utilisateur la possibilité de modifier sa demande ou d'abandonner l'opération.

Nous allons voir un petit exemple d'utilisation de `strerror()`, en l'invoquant pour une dizaine d'erreurs courantes :

```

exemple-strerror.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main (void)
{
    printf("strerror(EACCES) = %s\n", strerror(EACCES));
    printf("strerror(EAGAIN) = %s\n", strerror(EAGAIN));
    printf("strerror(EBUSY) = %s\n", strerror(EBUSY) );
    printf("strerror(ECHILD) = %s\n", strerror(ECHILD));
    printf("strerror(EEXIST) = %s\n", strerror(EEXIST));
    printf("strerror(EFAULT) = %s\n", strerror(EFAULT));
    printf("strerror(EINTR) = %s\n", strerror(EINTR) );
    printf("strerror(EINVAL) = %s\n", strerror(EINVAL));
}

```

```

printf("strerror(EISDIR) = %s\n", strerror(EISDIR));
printf("strerror(EMFILE) = %s\n", strerror(EMFILE));
printf("strerror(ENODEV) = %s\n", strerror(ENODEV));
printf("strerror(ENOMEM) = %s\n", strerror(ENOMEM));
printf("strerror(ENOSPC) = %s\n", strerror(ENOSPC));
printf("strerror(EPERM) = %s\n", strerror(EPERM) );
printf("strerror(EPIPE) = %s\n", strerror(EPIPE) );
printf("strerror(ESRCH) = %s\n", strerror(ESRCH) );

return EXIT_SUCCESS;
}

```

L'exécution montre que la fonction `strerror()` de la Glibc est sensible à la localisation :

```

$ echo $LC_ALL
fr_FR
$ ./exemple-strerror
strerror(EACCES) = Permission non accordée
strerror(EAGAIN) = Ressource temporairement non disponible
strerror(EBUSY) = Périphérique ou ressource occupé
strerror(ECHILD) = Aucun processus enfant
strerror(EEXIST) = Le fichier existe
strerror(EFAULT) = Mauvaise adresse
strerror(EINTR) = appel système interrompu
strerror(EINVAL) = Paramètre invalide
strerror(EISDIR) = Est un répertoire
strerror(EMFILE) = Trop de fichiers ouverts
strerror(ENODEV) = Aucun périphérique de ce type
strerror(ENOMEM) = Ne peut allouer de la mémoire
strerror(ENOSPC) = Aucun espace disponible sur le périphérique
strerror(EPERM) = Opération non permise
strerror(EPIPE) = Relais brisé (pipe)
strerror(ESRCH) = Aucun processus de ce type
$

```

Il existe également une fonction permettant d'afficher directement sur la sortie standard, précédée éventuellement d'une chaîne de caractères permettant de situer l'erreur. Cette fonction, nommée `perror()`, est déclarée dans `<stdio.h>` ainsi :

```

void perror (const char * s);

```

On l'utilise généralement de la manière suivante :

```

if ((fd = open(nom_fichier, mode)) == -1) {
    perror("open");
}

```

```
    exit(EXIT_FAILURE);  
}
```

Le message s'affiche ainsi :

open: Aucun fichier ou répertoire de ce type

Il s'agit dans ce cas de l'erreur ENOENT.

Notre exemple va utiliser `perror()` en cas d'échec de `fork()`. Pour faire échouer celui-ci, nous allons diminuer la limite `RLIMIT_NPROC` (nous verrons les limites ultérieurement), puis nous bouclerons sur un appel `fork()`.

```
exemple-perror.c :  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <sys/resource.h>  
#include <sys/wait.h>  
  
int main (void)  
{  
    struct rlimit limite;  
    pid_t  pid;  
  
    if (getrlimit(RLIMIT_NPROC, & limite) != 0) {  
        perror("getrlimit");  
        exit(EXIT_FAILURE);  
    }  
    limite.rlim_cur = 16;  
    if (setrlimit(RLIMIT_NPROC, & limite) != 0) {  
        perror("setrlimit");  
        exit(EXIT_FAILURE);  
    }  
    while (1) {  
        pid = fork();  
        if (pid == (pid_t) -1) {  
            perror("fork");  
            exit(EXIT_FAILURE);  
        }  
        if (pid != 0) {  
            fprintf(stdout, "%ld\n", (long)pid);  
            if (waitpid(pid, NULL, 0) != pid)  
                perror("waitpid");  
            break;  
        }  
    }  
}
```

```
}  
    return EXIT_SUCCESS;  
}
```

Comme nous avons déjà plusieurs processus qui tournent avant même de lancer le programme, nous n'atteindrons pas les seize `fork()`. Pour vérifier le nombre de processus en cours, nous allons d'abord lancer une série de commandes shell, avant d'exécuter le programme.

```
$ ps aux | grep `whoami` | wc -l  
10  
$ ./exemple_perror  
6922  
6923  
6924  
6925  
6926  
6927  
6928  
fork: Ressource temporairement non disponible  
$
```

Dans les dix processus qui tournaient avant le lancement, il faut compter la commande `ps` elle-même. Il est donc normal que `fork()` n'échoue qu'une fois arrivé au septième processus fils. Le message correspond à celui de l'erreur `EAGAIN`. En effet, le système considère que l'un des processus va se finir tôt ou tard et que `fork()` pourra réussir alors.

Un autre moyen d'accéder directement aux messages d'erreur est d'employer la table `sys_errlist[]`, définie dans `<errno.h>` ainsi :

```
const char * sys_errlist[];
```

Chaque élément de la table est un pointeur sur une chaîne de caractères décrivant l'erreur correspondant à l'index de la chaîne dans la table. Il existe une variable globale décrivant le nombre d'entrées dans la table :

```
int sys_nerr;
```

Il faut être très prudent avec les accès dans cette table car il y a des valeurs ne correspondant à aucune erreur. C'est le cas, par exemple, de 41 et 58 sous Linux avec la Glibc 2. Pareillement, dans la même configuration, `sys_nerr` vaut 135, ce qui signifie que les erreurs s'étendent de 0 à 134.

L'accès à la table doit donc être précautionneux, du genre :

```
if ((erreur < 0) || (erreur >= sys_nerr)) {
    fprintf(stderr, "Erreur invalide %d\n", erreur);
} else if (sys_errlist[erreur] == NULL) {
    fprintf(stderr, "Erreur non documentée %d\n", erreur);
} else {
    fprintf(stderr, "%s\n", sys_errlist[erreur]);
}
```

Nous allons utiliser la table `sys_errlist[]` pour afficher tous les libellés des erreurs connues.

```
exemple-sys-errlist.c :
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main (void)
{
    int i;
    for (i = 0; i < sys_nerr; i++)
        if (sys_errlist[i] != NULL)
            fprintf(stdout, "%d : %s\n", i, sys_errlist[i]);
        else
            fprintf(stdout, "*** Pas de message pour %d **\n", i);
    return EXIT_SUCCESS;
}
```

Dans l'exemple d'exécution suivant, nous avons éliminé quelques passages pour éviter d'afficher inutilement toute la liste :

```
$ ./exemple-sys-errlist
0 : Success
1 : Operation not permitted
2 : No such file or directory
3 : No such process
4 : Interrupted system call
5 : Input/output error
6 : Device not configured
7 : Argument list too long
8 : Exec format error
9 : Bad file descriptor
10 : No child processes
[...]
39 : Directory not empty
40 : Too many levels of symbolic links
```

```
** Pas de message pour 41 **
42 : No message of desired type
[...]
56 : Invalid request code
57 : Invalid slot
** Pas de message pour 58 **
59 : Bad font file format
60 : Device not a stream
61 : No data available
62 : Timer expired
[...]
121 : Remote I/O error
122 : Disk quota exceeded
123 : No medium found
124 : Wrong medium type
125 : Operation canceled
126 : Required key not available
127 : Key has expired
128 : Key has been revoked
129 : Key was rejected by service
130 : Owner died
131 : State not recoverable
132 : Operation not possible due to RF-kill
** Pas de message pour 133 **
** Pas de message pour 134 **
$
```

On remarque plusieurs choses : d'abord les messages de `sys_errlist[]` ne sont pas traduits automatiquement par la localisation, contrairement à `strerror()`, ensuite l'erreur 0, bien que documentée pour simplifier l'accès à cette table, n'est par définition pas une erreur, enfin les erreurs 41 et 58 n'existent effectivement pas.

Conclusion

Nous avons étudié dans ce chapitre les principaux points importants concernant la fin d'un processus, due à des causes normales ou à un problème irrémédiable.

Nous avons également essayé de définir un comportement raisonnable en cas de détection d'erreur, et nous avons analysé les méthodes pour obtenir des informations sur les raisons ayant conduit un processus fils à se terminer. En ce qui concerne le code de débogage, et les macros `assert()`, on trouvera des réflexions intéressantes dans [McConnell 1994] *Programmation professionnelle*.

Déroulement des Pthreads

Ainsi que nous l'avions évoqué en début de chapitre 2, il existe deux méthodes pour réaliser des applications multitâches sous Linux. La première – que nous avons examiné précédemment – reposait sur les processus. La seconde méthode, que nous allons aborder, repose sur les threads.

Les threads représentent un concept relativement nouveau dans le domaine de la programmation. Il s'agit d'une manière différente d'aborder la conception multitâche. Linux implémente les mécanismes qui, mis en œuvre par le noyau et des fonctions de bibliothèque, permettent d'accéder à la puissance des threads avec la portabilité de la norme SUSv4. Ces fonctionnalités étant standardisées, elles ont donné naissance au terme *Pthread* pour représenter les threads compatibles avec la norme Posix.1c qui les décrivait en premier lieu dans les années 1990.

Ce chapitre présentera les notions et les routines essentielles de la programmation multithread, ainsi que les mécanismes simples de synchronisation entre threads. Le prochain chapitre abordera des points plus avancés comme les annulations de threads ou les synchronisations complexes par des variables conditions.

Les fonctions présentées dans ces deux chapitres sont spécifiques au langage C. La programmation multithread en Python s'appuie sur des routines assez différentes, qu'on trouvera dans le module `threading` de ce langage. Nous ne les traiterons pas dans ce livre.

Présentation

Le mot *thread* peut se traduire par « fil d'exécution », c'est-à-dire un déroulement particulier du code du programme qui se produit parallèlement à d'autres entités en cours de progression. Les threads sont généralement présentés en premier lieu comme des processus allégés ne réclamant que peu de ressources pour les changements de contexte. Il faut ajouter à ceci un point important : les différents threads d'une application partagent un même espace d'adressage en ce qui concerne leurs données. La vision du programmeur est d'ailleurs plus orientée sur ce dernier point que sur la simplicité de commutation des tâches.

Les threads sont donc des tâches indépendantes partageant les mêmes données statiques et dynamiques. Chaque thread dispose personnellement d'une pile et d'un contexte d'exécution contenant les registres du processeur et un compteur d'instruction. Les méthodes de communication entre les threads sont alors naturellement plus simples que les communications entre processus. En contrepartie, l'accès concurrentiel aux mêmes données nécessite une synchronisation pour éviter les interférences, ce qui complique certaines portions de code.

Les threads définis par la norme SUSv4 sont indépendants de l'implémentation sous-jacente dans le système d'exploitation. Les applications sont donc portables sur des systèmes n'implémentant pas la notion de processus ni leurs méthodes de communication.

Les threads ne sont intéressants que dans les applications assurant plusieurs tâches en parallèle. Si chacune des opérations effectuées par un logiciel doit attendre la fin d'une opération précédente avant de pouvoir démarrer, il est totalement inutile d'essayer une approche multithread.

Implémentation

Pour implémenter les fonctionnalités de la norme SUSv4, il existe essentiellement deux possibilités : l'implémentation dans l'espace du noyau, et celle dans l'espace de l'utilisateur. En implémentation noyau, chaque thread est vu individuellement par le noyau ; en implémentation utilisateur, l'application n'est constituée que d'un seul processus, et la répartition en différents threads est assurée par une bibliothèque indépendante du noyau¹. Chaque implémentation a ses avantages et ses défauts.

1. Attention à ne pas confondre les threads applicatifs implémentés grâce à un support du noyau (ce dont nous traitons ici) et les threads du noyau (kernel threads) qui représentent des portions de code exécutées dans l'espace mémoire du noyau, avec les privilèges de ce dernier, tout en étant ordonnancés par le scheduler. Dans ce dernier cas, on rencontre par exemple les tâches events, ksoftirq ou migration que l'on voit sur la plupart des systèmes Linux actuels en tête des résultats de ps aux.

Point de vue	Implémentation dans l'espace du noyau	Implémentation dans l'espace de l'utilisateur
Implémentation des fonctionnalités SUSv4.	Nécessite la présence d'appels système spécifiques, qui n'existent pas nécessairement sur toutes les versions du noyau.	Portable de système Unix en système Unix sans modification du noyau.
Création d'un thread.	Nécessite un appel système.	Ne nécessite pas d'appel système, est donc moins coûteuse en ressource que l'implémentation dans le noyau.
Commutation entre deux threads.	Commutation par le noyau avec changement de contexte.	Commutation assurée dans la bibliothèque sans changement de contexte, est donc plus légère.
Ordonnancement des threads.	Chaque thread dispose des mêmes ressources CPU que les autres processus du système.	Utilisation globale des ressources CPU limitée à celle du processus d'accueil.
Priorités des tâches.	Chaque thread peut s'exécuter avec une priorité indépendante des autres, éventuellement en ordonnancement temps réel.	Les threads ne peuvent s'exécuter qu'avec des priorités inférieures à celle du processus principal.
Parallélisme.	Sur une machine multiprocesseur ou multicoeur, le noyau peut répartir les threads sur différents processeurs et profiter ainsi du véritable parallélisme.	Les threads sont condamnés à s'exécuter sur un seul processeur puisqu'ils sont contenus dans un unique processus.

Sous Linux, l'implémentation usuelle est effectuée dans l'espace noyau à l'aide de l'appel système `clone()` spécifique. Cet appel système est un genre de `fork()` grâce auquel l'espace d'adressage du processus (père) accueillera une nouvelle tâche. Toutefois, cette fonction ne concerne en principe jamais le programmeur applicatif, car la bibliothèque C fournit les fonctionnalités SUSv4 que nous allons décrire dans le reste de ce chapitre.

Jusqu'au noyau 2.4, la principale bibliothèque implémentant les threads s'appelait *LinuxThreads*. Puissante, elle souffrait toutefois de quelques limitations, notamment en ce qui concerne la gestion des signaux. Depuis le noyau 2.6 une nouvelle bibliothèque a été intégrée à la bibliothèque C : la *NPTL* (*Native Posix Thread Library*). En outre, le noyau offre un support nettement amélioré en ce qui concerne la création et l'ordonnancement des threads.

Il existait historiquement d'autres implémentations des Pthreads sous Linux, notamment *PCthreads* qui fonctionnait dans l'espace utilisateur, mais elles sont à présent considérées comme obsolètes.

Pour employer les fonctionnalités de la bibliothèque *NPTL*, il faut inclure l'en-tête `<pthread.h>` dans les fichiers source, ajouter l'option `-pthread` sur la ligne de com-

mande du compilateur gcc. Nous ajoutons ces options dans le fichier `Makefile` pour l'utiliser systématiquement avec tous les programmes de ce chapitre.

L'option `-pthread` a deux effets principaux : lors de la compilation elle permet de choisir une implémentation fonctionnant en contexte multithread pour toutes les routines de la bibliothèque C que l'on invoque (c'était le rôle de la constante symbolique `_REENTRANT` avant la *NPTL*) Pendant l'édition des liens cette option s'assure que l'on utilise la bibliothèque implémentant les fonctions de gestion des threads (ce que réalisait l'option `-lpthread` auparavant).

Création de threads

Il existe grosso modo des équivalents aux appels système `fork()`, `wait()` et `exit()`, qui permettent dans un contexte multithread de créer un nouveau thread, d'attendre la fin de son exécution, et de permettre à un thread de se terminer volontairement.

Un type opaque `pthread_t` est utilisé pour distinguer les différents threads d'une application, à la manière des *PID* qui permettent d'identifier les processus. Dans la bibliothèque *NPTL*, le type `pthread_t` est représenté « pour l'extérieur » sous forme d'un `unsigned long` mais en interne, une structure est utilisée. Il ne faut donc jamais faire de comparaison directe de variables de type `pthread_t` et éviter toute affectation. Une fonction de comparaison spécifique est définie par la norme SUSv4 :

```
| int pthread_equal (pthread_t thread_1, pthread_t thread_2);
```

Cette fonction renvoie une valeur non nulle s'ils sont égaux.

Notons que le comportement des fonctions du standard Pthread est un peu particulier en ce qui concerne les codes de retour. Toutes les fonctions sauf `pthread_equal()` renvoient zéro si elles réussissent leur travail, et renvoient directement le code d'erreur (défini dans `<errno.h>`) en cas d'erreur, sans toucher à la variable globale `errno`.

Pour créer un nouveau thread, on emploie la fonction `pthread_create()`. Celle-ci donne naissance à un nouveau fil d'exécution, qui va démarrer en invoquant la routine dont on passe le nom en argument. Lorsque cette routine se termine, le thread est éliminé. Cette routine fonctionne donc un peu comme la fonction `main()` des programmes C. Pour cette raison, le fil d'exécution original du processus est nommé thread principal (*main thread*). Le prototype de `pthread_create()` est le suivant :

```
| int pthread_create (pthread_t * thread,  
                    pthread_attr_t * attributs,  
                    void * (* fonction) (void * argument),  
                    void * argument);
```

Le premier argument est un pointeur qui sera initialisé par la routine avec l'identifiant du nouveau thread. Le second argument correspond aux attributs dont on désire doter le nouveau thread. Ces attributs seront détaillés plus loin. En général, on transmettra en second argument un pointeur NULL, car le thread reçoit alors les attributs standard par défaut.

Le troisième argument est un pointeur représentant la fonction centrale du nouveau thread. Celle-ci est invoquée dès la création du thread et reçoit en argument le pointeur passé en dernière position dans `pthread_create()`. Le type de l'argument étant `void *`, on pourra le transformer en n'importe quel type de pointeur pour passer un argument au thread. On pourra même employer une conversion explicite en `long` pour transmettre une valeur. Cet argument est généralement un numéro permettant au thread de déterminer le travail qu'il doit accomplir – en employant la même fonction centrale pour plusieurs threads –, mais cela peut aussi être un pointeur sur une structure que le nouveau thread doit manipuler avant de se terminer.

La routine `pthread_create()` renvoie zéro si elle réussit et une valeur non nulle sinon, correspondant à l'erreur survenue.

Voyons un exemple de création de thread :

```
exemple-pthread-create-1.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (void)
{
    pthread_t thr;
    if (pthread_create(& thr, NULL, fct_thread, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}

void * fct_thread(void * arg)
{
    while (1) {
        fprintf(stderr, "Nouveau Thread\n");
        sleep(1);
    }
}
```

Le thread principal, exécutant la fonction `main()`, va créer un nouveau thread, puis afficher indéfiniment le message « Thread Main » toutes les secondes. Le nouveau thread va afficher le message « Nouveau Thread » indéfiniment. Aussi verra-t-on à l'écran les messages entremêlés, montrant bien les deux exécutions parallèles. On pressera « *Contrôle-C* » pour arrêter le programme.

```
$ ./exemple-pthread-create-1
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
(Ctrl-C)
$
```

Si l'on regarde, sur une autre console, le résultat de la commande « `ps aux` » durant l'exécution du programme on ne voit pas les threads séparément :

```
$ ps aux
USER PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root    1  0.0  0.0  2080  644 ?        Ss   08:28   0:00 init [5]
[...]
cpb 3951  0.0  0.0  4680 1528 pts/1    Ss   08:32   0:00 bash
cpb 4302  0.0  0.0 11992  420 pts/1    Sl+  08:47   0:00 ./exemple-pthread-create-1
cpb 4306  0.0  0.0  4680 1504 pts/2    Ss   08:48   0:00 bash
cpb 4332  0.0  0.0  4268  964 pts/2    R+   08:48   0:00 ps aux
$
```

Pour voir les différents threads des processus, il faut ajouter l'option `m` de `ps` :

```
$ ps maux
USER PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root    1  0.0  0.0  2080  644 ?        -    08:28   0:00 init [5]
root    -  0.0  -      -     - -      Ss   08:28   0:00 -
[...]
```

```

cpb 3951 0.0 0.0 4680 1528 pts/1 - 08:32 0:00 bash
cpb - 0.0 - - - - Ss 08:32 0:00 -
cpb 4302 0.0 0.0 11992 420 pts/1 - 08:47 0:00 ./exemple-pthread-create-1
cpb - 0.0 - - - - Sl+ 08:47 0:00 -
cpb - 0.0 - - - - Sl+ 08:47 0:00 -
cpb 4306 0.0 0.0 4680 1504 pts/2 - 08:48 0:00 bash
cpb - 0.0 - - - - Ss 08:48 0:00 -
cpb 4392 0.0 0.0 4268 964 pts/2 - 08:52 0:00 ps maux
cpb - 0.0 - - - - R+ 08:52 0:00 -
$

```

Dans ce cas, chaque ligne correspondant à un processus est suivie d'une ligne pour chaque thread dans ce processus. Par exemple la ligne où le champ PID vaut 4306 (bash) est suivie d'une ligne dont le champ PID vaut « - ». Il s'agit de l'unique thread de ce processus, celui qui exécute la fonction `main()`.

Ainsi, depuis le noyau 2.6, la vision des tâches du système a légèrement évolué : on considère que tout processus contient au moins un thread (`main`) et l'on ne fait plus de distinction entre processus unithread et processus multithread.

La ligne du processus de PID 4302 (`exemple-pthread-create-1`) est suivie de deux lignes : l'une pour le thread `main` et l'autre pour le nouveau thread que nous avons créé.

Ces lignes contiennent des tirets « - » pour les champs qui dépendent du processus lui-même :

- PID : le *PID* est global au processus, la fonction `getpid()` renvoie la même valeur dans chaque thread ;
- %MEM : le taux d'occupation de la mémoire disponible est une valeur qui dépend de l'espace mémoire, donc du processus ;
- VSZ : la taille de l'espace virtuel de mémoire (*virtual memory size*) du processus ;
- rss : la taille de la mémoire physique utilisée (*resident set size*) dépend du processus ;
- tty : le terminal de contrôle est celui du processus, commun à tous les threads ;
- command : enfin, la ligne de commande est identique pour tous, c'est celle qui a initialisé le processus.

Mais toutes les autres informations peuvent varier d'un thread à l'autre :

- USER : l'*UID* réel du thread pourrait être modifié avec `setreuid()` ;
- %CPU : un thread effectuant beaucoup de calcul peut consommer une part beaucoup plus grande des ressources *CPU* qu'un autre thread qui dort la plupart du temps ;
- STAT : l'état d'exécution (*Sleeping*, *Running*, etc., voir chapitre 10) d'un thread est totalement indépendant de ceux des autres ;

- **START** : l'heure de démarrage d'un thread peut être largement postérieure à celle d'un autre thread.
- **TIME** : enfin la quantité de temps *CPU* consommé par un thread dépend uniquement de son activité personnelle et n'a pas de rapport avec celles des autres threads.

Nous avons remarqué que le *PID* est identique dans tous les threads d'un processus. Ceci va plus loin encore : il n'est pas possible d'identifier les threads depuis l'extérieur du processus. Leurs seuls identifiants sont les valeurs `pthread_t` fournies par `pthread_create()`, mais ces valeurs n'ont aucune signification en dehors de l'espace d'adressage du processus.

Passage d'argument à la création d'un thread

Lors de l'appel `pthread_create()`, on peut fournir en dernier paramètre une valeur (un pointeur générique de type `void *` ou toute donnée compatible), que le thread trouvera en argument de sa fonction principale. Voyons un exemple où nous forçons un transtypage (*cast*) d'un entier long en pointeur générique. Cette conversion est possible sur toutes les architectures supportant Linux :

```
exemple-pthread-create-2.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

#define NB_THREADS 5

int main (void)
{
    pthread_t thr[NB_THREADS];
    long n;
    for (n = 0; n < NB_THREADS; n++) {
        if (pthread_create(& thr[n], NULL,
                          fct_thread, (void *) n) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}
```



```
void * fct_thread(void * arg)
{
    long num = (long) arg;
    while (1) {
        fprintf(stderr, "Thread numero %ld\n", num+1);
        sleep(1);
    }
}
```

Lors de l'exécution nous avons six messages distincts chaque seconde :

```
$ ./exemple-pthread-create-2
Thread Main
Thread numero 4
Thread numero 5
Thread numero 3
Thread numero 2
Thread numero 1
Thread Main
Thread numero 4
Thread numero 5
Thread numero 3
Thread numero 2
Thread numero 1
Thread Main
Thread numero 4
Thread numero 5
Thread numero 3
Thread numero 2
Thread numero 1
      (Control-C)
$
```

Si nous avons plusieurs valeurs à transmettre à la création du thread, il suffirait de les regrouper dans une structure, ainsi :

```
exemple-pthread-create-3.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

typedef struct {
```

```

    int X;
    int Y;
} coordonnee_t;

int main (void)
{
    pthread_t thr;
    coordonnee_t * coord;

    coord = malloc(sizeof(coordonnee_t));
    if (coord == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    coord->X=10;
    coord->Y=20;
    if (pthread_create(& thr, NULL, fct_thread, coord) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}

void * fct_thread(void * arg)
{
    coordonnee_t * coord = (coordonnee_t *) arg;
    int X = coord->X;
    int Y = coord->Y;
    free(coord);
    while (1) {
        fprintf(stderr, "Thread X=%d, Y=%d\n", X, Y);
        sleep(1);
    }
}

```

À l'exécution, on voit bien que le thread a reçu les deux valeurs transmises :

```

$ ./exemple-pthread-create-3
Thread Main
Thread X=10, Y=20
Thread Main
Thread X=10, Y=20
Thread Main
Thread X=10, Y=20
          (Control-C)
$

```

Notez bien que la structure a été allouée, avec `malloc()`, dans le thread `main()`, et libérée avec `free()` dans le nouveau thread. Ceci fonctionne parfaitement car nous sommes dans le même espace de mémoire.

Partage d'espace mémoire

Nous avons affirmé que les threads se trouvent dans le même espace d'adressage de mémoire, mais nous n'en avons eu aucune confirmation pour le moment. Pour le vérifier, il nous suffit d'utiliser une variable globale comme un compteur que chacun incrémentera. L'exemple suivant est construit en rajoutant le compteur dans l'exemple-`pthread-create-2.c`.

exemple-pthread-create-4.c :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

#define NB_THREADS 5

int compteur = 0;

int main (void)
{
    pthread_t thr[NB_THREADS];
    long n;
    for (n = 0; n < NB_THREADS; n++) {
        if (pthread_create(&thr[n], NULL, fct_thread, (void *) n) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    while (1) {
        printf("Thread Main, compteur = %d\n", compteur);
        sleep(1);
    }
}

void * fct_thread(void * arg)
{
    long num = (long) arg;
    while (1) {
        printf("Thread %ld, compteur = %d \n", num+1, compteur);
        compteur ++;
    }
}
```

```
    sleep(1);  
  }  
}
```

On notera que le thread main ne fait qu'afficher le compteur, il ne le modifie pas, en revanche tous les autres l'incrémentent après affichage :

```
$ ./exemple-pthread-create-4  
Thread Main, compteur = 0  
Thread 4, compteur = 0  
Thread 5, compteur = 1  
Thread 2, compteur = 2  
Thread 1, compteur = 3  
Thread 3, compteur = 4  
Thread Main, compteur = 5  
Thread 4, compteur = 5  
Thread 5, compteur = 6  
Thread 2, compteur = 7  
Thread 1, compteur = 8  
Thread 3, compteur = 8  
Thread 4, compteur = 10  
Thread Main, compteur = 11  
Thread 5, compteur = 11  
Thread 1, compteur = 12  
Thread 2, compteur = 13  
Thread 3, compteur = 14  
Thread 4, compteur = 15  
Thread Main, compteur = 16  
Thread 5, compteur = 16  
Thread 1, compteur = 17  
Thread 2, compteur = 18  
Thread 3, compteur = 19  
    (Control-C)  
$
```

La variable globale compteur est donc bien partagée entre les threads. Notez que nous avons accédé en parallèle à cette variable sans aucune protection. C'est un comportement très dangereux dans un contexte multithread. Il faut toujours utiliser des mécanismes de verrouillage comme nous en verrons plus loin.

Fin d'un thread

Un thread peut se terminer volontairement, ainsi que nous allons le voir ici, ou involontairement (par une *annulation*) comme nous l'étudierons dans le prochain chapitre.

Lorsque la fonction principale d'un thread se termine, celui-ci est éliminé. Cette fonction doit renvoyer une valeur de type `void *` qui pourra être récupérée dans un autre thread. Il est aussi possible d'invoquer la fonction `pthread_exit()`, qui met fin au thread appelant tout en renvoyant le pointeur `void *` passé en argument.

```
void pthread_exit (void * retour);
```

Si l'on n'a aucune valeur à renvoyer, un pointeur `NULL` peut être utilisé.

Voici un exemple où trois threads sont créés, au bout de deux secondes l'un se termine avec `pthread_exit()`, et un deuxième thread se termine avec `return` au bout de quatre secondes.

```
exemple-pthread-exit.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main(void)
{
    pthread_t thr[3];
    long n;
    for (n = 0; n < 3; n++) {
        if (pthread_create(&thr[n], NULL, fct_thread,
                          (void *) n) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    while (1) {
        fprintf(stderr, "Thread Main\n");
        sleep(1);
    }
}

void * fct_thread(void * arg)
{
    long num = (long) arg;
    int i = 0;
```

```
while (1) {  
    fprintf(stderr, "Thread %ld, iteration %d\n", num, i);  
    if ((num == 0) && (i == 1))  
        pthread_exit(NULL);  
    if ((num == 1) && (i == 3))  
        return NULL;  
    i ++;  
    sleep(1);  
}  
}
```

À l'exécution, nos threads se terminent bien comme prévu :

```
$ ./exemple-pthread-exit-1  
Thread Main  
Thread 2, iteration 0  
Thread 1, iteration 0  
Thread 0, iteration 0  
Thread Main  
Thread 2, iteration 1  
Thread 1, iteration 1  
Thread 0, iteration 1  
Thread Main  
Thread 2, iteration 2  
Thread 1, iteration 2  
Thread Main  
Thread 2, iteration 3  
Thread 1, iteration 3  
Thread Main  
Thread 2, iteration 4  
Thread Main  
Thread 2, iteration 5  
Thread Main  
Thread 2, iteration 6  
      (Control-C)  
$
```

Notons qu'un thread ne doit jamais invoquer `exit()` sinon c'est tout le processus qui se termine. De même si le thread principal se termine, en faisant un `return` dans la fonction `main()`, tout le processus se finira.

En revanche, le thread principal peut se terminer avec `pthread_exit()` et laisser les autres threads s'exécuter comme ils veulent.

```
exemple-pthread-exit-2.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (void)
{
    pthread_t thr[3];
    long n;

    for (n = 0; n < 3; n++) {
        if (pthread_create(&thr[n], NULL, fct_thread,
                          (void *) n) != 0) {
            fprintf(stderr, "Erreur dans pthread_create\n");
            exit(EXIT_FAILURE);
        }
    }
    pthread_exit(NULL);
}

void * fct_thread(void * arg)
{
    longnum = (long) arg;
    while (1) {
        fprintf(stderr, "Thread %ld\n", num);
        sleep(1);
    }
}
```

Dans cet exemple, le thread principal se termine avec `pthread_exit()`, tandis que les autres threads qu'il a créé continuent de s'exécuter indéfiniment.

```
$ ./exemple-pthread-exit-2
Thread 1
Thread 2
Thread 0
Thread 1
Thread 2
Thread 0
Thread 1
Thread 2
Thread 0
    (Control-C)
$
```

Élimination d'un thread

Il est important de comprendre que lorsqu'un thread commet une erreur fatale, c'est tout le processus qui est tué. Dans l'exemple suivant un thread va tenter au bout de quelques secondes d'écrire dans un pointeur non-initialisé (NULL). Nous verrons que tout le processus est tué.

```
exemple-pthread-exit-3.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (void)
{
    pthread_t thr;
    if (pthread_create(& thr, NULL, fct_thread, NULL) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    while (1) {
        fprintf(stderr, "Thread MAIN en fonctionnement\n");
        sleep(1);
    }
}

void * fct_thread(void * arg)
{
    char * ptr = NULL;
    sleep(4);
    ptr[0] = 'A';
    return NULL;
}
```

Nous voyons que tout le processus est tué lorsqu'un thread se plante :

```
$ ./exemple-pthread-exit-3
Thread MAIN en fonctionnement
Thread MAIN en fonctionnement
Thread MAIN en fonctionnement
Thread MAIN en fonctionnement
Erreur de segmentation
$
```


Récupération de la valeur de retour

Pour récupérer la valeur de retour d'un thread terminé, on utilise la fonction `pthread_join()`. Celle-ci suspend l'exécution du thread appelant jusqu'à la terminaison du thread indiqué en argument. Elle remplit alors le pointeur passé en seconde position avec la valeur de retour du thread fini.

```
| int pthread_join (pthread_t thread, void ** retour);
```

La fonction `pthread_join()` peut échouer si le thread attendu n'existe pas, s'il est détaché, comme nous le verrons plus bas, ou si un risque de blocage se présente – par exemple si un thread demande à attendre sa propre fin.

Notons deux défauts importants de la fonction `pthread_join()` telle qu'elle est normalisée par SUSv4 : il n'est pas possible d'appeler cette fonction en demandant de récupérer la fin de n'importe quel thread qui se termine, il faut nécessairement préciser le thread attendu. En outre, il n'est pas possible de faire un appel non bloquant à cette fonction ; si le thread attendu n'est pas terminé, on restera bloqué dans `pthread_join()` jusqu'à ce qu'il se finisse.

Lorsqu'on désire employer une valeur de type entier comme code de retour, il est important de procéder en deux étapes pour la récupérer depuis le type `void *`. Ceci permet de conserver la portabilité du programme, même si la taille d'un `void *` est plus grande que celle d'un `int`. Nous rencontrerons cette situation dans le programme ci-dessous.

```
exemple-pthread-join-1.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (int argc, char * argv[])
{
    pthread_t * thr = NULL;
    void * ptr;
    int i;
    long n;
    thr = calloc(argc - 1, sizeof(pthread_t));
    if (thr == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
}
```

```
for (i = 1; i < argc; i++) {
    n = atoi(argv[i]);
    if (pthread_create(& thr[i-1], NULL, fct_thread,
                      (void *) n) != 0) {
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
}
for (i = 1; i < argc; i++) {
    pthread_join(thr[i - 1], & ptr);
    fprintf(stderr, "%d -> %ld\n", atoi(argv[i]), (long) ptr);
}
return EXIT_SUCCESS;
}

void * fct_thread(void * arg)
{
    long num = (long) arg;
    return (void *) (num * num);
}
```

Ce programme lance autant de thread qu'il a reçu d'arguments sur sa ligne de commande, en leur transmettant ces arguments. La fonction des threads calcule le carré de son argument et la renvoie en fin de thread. Enfin le thread *main* récupère les résultats et les affiche.

```
$ ./exemple-pthread-join-1 5 6 7
5 -> 25
6 -> 36
7 -> 49
$
```

Détachement des threads

Imaginons un processus qui lance successivement de très nombreux threads. Par exemple un serveur de base de données qui déclenche un thread à chaque connexion d'un utilisateur, pour gérer les requêtes qui lui seront transmises. Lorsqu'un thread se termine, il n'a pas de code de retour significatif pour l'application, aussi appellera-t-on probablement `pthread_exit(NULL)`.

Le thread principal ne s'intéresse pas à ce code de retour. En outre, il ne peut pas rester bloqué en attente sur un `pthread_join()` car un autre utilisateur peut demander une connexion avant la fin du thread précédent.

Aussi ignorerons-nous cette valeur nulle. Écrivons une première version naïve, où le thread *main* lancera un nouveau thread toutes les 10 ms (le serveur est plutôt chargé !). Ces threads ne feront rien, et se termineront immédiatement :

```
exemple-pthread-detach-1.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (int argc, char * argv[])
{
    pthread_t thr; /* Même variable écrasée à chaque fois */
    int nb_lances = 0;
    while (pthread_create(&thr, NULL, fct_thread, NULL)==0) {
        nb_lances ++;
        usleep(10000); /* 10 ms */
    }
    fprintf(stderr, "Echec de creation apres %d threads\n",
            nb_lances);
    return EXIT_SUCCESS;
}

void * fct_thread(void * arg)
{
    return NULL;
}
```

Le programme va indiquer combien de threads il a pu créer avant d'échouer. Au bout de 3 secondes d'exécution environ, on a le résultat suivant :

```
$ ./exemple-pthread-detach-1
Echec de creation apres 304 threads
$
```

Pourquoi cet échec au bout de ce nombre si faible de threads ?

En fait, lorsqu'un thread se termine, son code de retour est conservé pour pouvoir être éventuellement lu ultérieurement avec un `pthread_join()`. Et pour garder ce code de retour, le noyau Linux conserve la pile du thread en question. La mémoire occupée par cette pile sera libérée et l'espace réutilisé lors du `pthread_join()`.

Mais les piles sont positionnées dans l'espace d'adressage du processus (voir chapitre 14) avec un espacement assez important, visible avec la commande `ulimit -s` du shell.

```
$ ulimit -s
10240
$
```

Les piles sont espacées de 10 240 ko soient 10 Mo. Sur un processeur 32 bits, dès que 300 piles environ sont utilisées simultanément, la totalité des 3 Go de l'espace d'adressage en mode utilisateur est saturée.

Sur ce type de processeur, pour créer plus de 300 threads simultanément, il faudra diminuer l'espacement entre leurs piles comme nous le verrons dans le chapitre suivant.

Mais notre problème est de pouvoir créer autant de threads successifs que nous le voulons, sans être obligés de lire leurs codes de retour avec `pthread_join()`. Pour cela il faut *détacher* le thread, c'est-à-dire indiquer au noyau que dès la fin du thread sa pile et son code de retour doivent être libérés.

```
| int pthread_detach (pthread_t thread);
```

Un thread peut très bien invoquer `pthread_detach()` à propos d'un autre thread de l'application (en général juste après l'avoir créé). Contrairement aux processus, il n'y a pas de notion de hiérarchie chez les threads ni d'autorisations particulières pour modifier les paramètres d'un autre fil d'exécution. Cette fonction échoue si le thread n'existe pas ou s'il est déjà détaché.

Pour pouvoir se détacher lui-même, un thread doit connaître son identifiant de type `pthread_t`. La fonction `pthread_self()` est là pour cela :

```
| pthread_t pthread_self (void);
```

Voyons le comportement de notre programme, légèrement modifié pour afficher les centaines de threads lancés.

```
| exemple-pthread-detach-2.c :
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int nb_lances = 0;
while (pthread_create(& thr, NULL, fct_thread, NULL) == 0) {
    pthread_detach(thr);
    nb_lances ++;
    if ((nb_lances % 100) == 0)
        fprintf(stderr, "%d thread lances\n", nb_lances);
    usleep(10000); /* 10 ms */
}
fprintf(stderr, "Echec de creation apres %d threads\n",
        nb_lances);
return EXIT_SUCCESS;
}
```

Cette fois, il n'y a plus de limites au nombre de threads que l'on peut exécuter successivement :

```
$ ./exemple-pthread-detach-2
```

```
100 thread lances
200 thread lances
300 thread lances
400 thread lances
500 thread lances
600 thread lances
700 thread lances
800 thread lances
900 thread lances
1000 thread lances
1100 thread lances
1200 thread lances
1300 thread lances
1400 thread lances
1500 thread lances
1600 thread lances
1700 thread lances
1800 thread lances
1900 thread lances
2000 thread lances
2100 thread lances
2200 thread lances
```

```
(CONTROL-C)
```

```
$
```

Attributs des threads

Jusqu'à présent nous avons toujours passé un pointeur NULL en second argument de la fonction `pthread_create()`. Mais nous pouvons passer un pointeur sur une variable de type `pthread_attr_t` représentant les attributs du thread à sa création. Le pointeur NULL indique simplement que le thread dispose pour tous les attributs des valeurs par défaut telles qu'elles sont définies dans SUSv4.

Le type `pthread_attr_t` est opaque, il ne faut faire ni affectation ni comparaison directe, nous disposerons de fonctions d'accès pour lire ou écrire les valeurs pour les attributs qui nous intéressent.

Dans un premier temps, il faut invoquer la fonction `pthread_attr_init()` en lui transmettant un pointeur sur une variable de type `pthread_attr_t`.

```
| int pthread_attr_init (pthread_attr_t * attributs);
```

Cette fonction initialise la variable-attributs avec toutes les valeurs par défaut. Nous allons utiliser les accesseurs `pthread_attr_getXXX()` et `pthread_attr_setXXX()` pour lire et écrire les attributs qui nous intéressent.

Ici nous allons nous contenter de fixer l'état « détaché » du thread avant sa création. Pour cela nous utiliserons les accesseurs :

```
| int pthread_attr_getdetachstate (const pthread_attr_t * attributs,  
                                int * detachement);  
| int pthread_attr_setdetachstate (pthread_attr_t * attributs,  
                                int detachement);
```

Le paramètre `detachement` peut prendre les valeurs suivantes, voir tableau ci-après.

Nom	Signification
PTHREAD_CREATE_JOINABLE	Configuration par défaut, la valeur de retour du thread sera conservée jusqu'à ce qu'elle soit consultée par un autre thread.
PTHREAD_CREATE_DETACHED	Lors de la terminaison du thread, toutes ses ressources sont libérées immédiatement, il n'y a pas de valeur de retour valide.

Si la valeur indiquée pour l'attribut n'est pas acceptable, la fonction échoue avec l'erreur `EINVAL`.

L'objet de type `pthread_attr_t` est alors prêt à être utilisé dans `pthread_create()`. Le thread créé est configuré en fonction des attributs indiqués, et l'objet `pthread_attr_t` n'est plus utilisé pour le thread. Il peut être à nouveau modifié pour préparer la création d'un autre thread. Une fois qu'on n'en a plus besoin, cet objet

peut être détruit en employant la fonction `pthread_attr_destroy()`, qui peut libérer des données dynamiques internes.

```
| int pthread_attr_destroy (pthread_attr_t * attr);
```

L'invocation de cette fonction est donc indispensable pour assurer la portabilité du programme, même si elle n'a pas d'utilité dans l'implémentation *NPTL* actuelle. Les threads créés précédemment ne sont pas concernés par la destruction de l'objet attribut, ce dernier a été utilisé une fois pour toutes au moment de la création du thread, mais n'a plus de liaison avec lui.

Voici les modifications apportées à l'exemple précédent pour utiliser l'attribut de détachement plutôt que `pthread_detach()`.

```
| exemple-pthread-attr-1.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (int argc, char * argv[])
{
    pthread_t thr;
    pthread_attr_t attr;
    int nb_lances = 0;
    pthread_attr_init(& attr);
    pthread_attr_setdetachstate(& attr,
                               PTHREAD_CREATE_DETACHED);
    while (pthread_create(& thr, & attr, fct_thread, NULL)
           == 0) {
        nb_lances++;
        if ((nb_lances % 100) == 0)
            fprintf(stderr, "%d thread lances\n", nb_lances);
        usleep(10000);
    }
    pthread_attr_destroy(& attr); /* Pas indispensable ici */
    fprintf(stderr, "Echec de creation apres %d threads\n",
            nb_lances);
    return EXIT_SUCCESS;
}

void * fct_thread(void * arg)
{
    return NULL;
}
```

L'exécution confirme bien le détachement des threads dès leur création puisqu'on arrive à dépasser les 300 threads :

```
$ ./exemple-pthread-attr-1
100 thread lances
200 thread lances
300 thread lances
400 thread lances
500 thread lances
600 thread lances
700 thread lances
      (CONTROL-C)
$
```

Nous verrons d'autres attributs de threads dans les prochains chapitres.

Synchronisation entre threads

L'un des enjeux essentiels lors du développement d'applications multithreads est la synchronisation entre les différents fils d'exécution concurrents. Ce qui représente, somme toute, un aspect annexe des logiciels reposant sur plusieurs processus devient ici un point crucial. Les différents threads d'une application disposant d'un accès partagé immédiat à toutes les variables globales, descripteurs de fichiers, etc., leur synchronisation est indispensable pour éviter la corruption de données et les situations de blocage.

Il existe essentiellement deux cas où des données risquent d'être corrompues si l'accès aux ressources communes n'est pas synchronisé :

- Deux threads concurrents veulent modifier une variable globale, par exemple décrémenter un compteur dans une gestion de stocks. Le premier thread lit la valeur initiale V0 dans un registre du processeur. Il décrémente la valeur d'une unité. L'ordonnanceur commute les tâches et donne la main au second thread. Celui-ci lit la valeur initiale V0, la décrémente et écrit la nouvelle valeur V0-1 dans le compteur. L'ordonnanceur réactive le premier thread qui inscrit à son tour la valeur calculée V0-1 dans le compteur. Au final, le stock indique V0-1 unités alors qu'il aurait dû être décrémenté deux fois. Ceci présage de sérieux problèmes le jour de l'inventaire...
- Un thread modifie une structure de données globale tandis qu'un autre essaye de la lire. Le thread lecteur charge les premiers membres de la structure. L'ordonnanceur bascule le contrôle au thread écrivain, qui modifie toute la structure. Lorsque le second thread est réactivé, il lit la fin de la structure. Les premiers

membres qu'il a reçus ne sont pas cohérents avec les suivants. Le problème pourrait être le même avec une chaîne de caractères, ou même une simple variable de type réel ou entier long.

Les mutex

Pour accéder à des données globales, il est donc indispensable de mettre en œuvre un mécanisme de synchronisation. Le plus courant est un système d'exclusion mutuelle nommé *mutex* (*mutal exclusion*) reposant sur des variables de type `pthread_mutex_t`. Chaque variable sert de verrou pour l'accès à une partie des données globales.

Il existe deux états pour un mutex : libre ou verrouillé. Lorsqu'un mutex est verrouillé par un thread, on dit que ce dernier *tient* le mutex. Un mutex ne peut être tenu que par un seul thread à la fois. En conséquence, il existe essentiellement deux fonctions de manipulation des mutex : une fonction de verrouillage et une fonction de libération. Lorsqu'un thread demande à verrouiller un mutex déjà tenu par un autre thread, le premier est bloqué jusqu'à ce que le mutex soit libéré.

On peut initialiser un mutex de manière statique ou dynamique, en précisant certains attributs à l'aide d'un objet de type `pthread_mutexattr_t`. L'initialisation statique se fait à l'aide de la constante `PTHREAD_MUTEX_INITIALIZER` :

```
| pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Pour l'initialisation dynamique, on emploie `pthread_mutex_init()` avec une variable regroupant les attributs du mutex.

```
| int pthread_mutex_init (pthread_mutex_t * mutex,  
|                          const pthread_mutexattr_t * attributs);
```

Nous verrons des exemples d'attributs des mutex dans les prochains chapitres. On peut utiliser un pointeur `NULL` en second argument de `pthread_mutex_init()` si le mutex doit prendre les attributs par défaut. Étant donné que les mutex servent à synchroniser différents threads, on les déclare naturellement dans des variables globales ou dans des variables locales statiques.

Une fois qu'un mutex n'est plus utilisé, on libère la variable en appelant `pthread_mutex_destroy()`. Le mutex ne doit plus être verrouillé, sinon cette fonction échoue avec l'erreur `EBUSY`.

```
| int pthread_mutex_destroy (pthread_mutex_t * mutex);
```

La fonction de verrouillage s'appelle `pthread_mutex_lock()`. Si le mutex est libre, il est immédiatement verrouillé et attribué au thread appelant. Si le mutex est déjà maintenu par un autre thread, la fonction reste bloquée jusqu'à la libération du mutex, puis elle le verrouille à la disposition du thread appelant.

Le prototype de `pthread_mutex_lock()` est le suivant :

```
| int pthread_mutex_lock (pthread_mutex_t * mutex);
```

La libération d'un mutex se fait avec la fonction `pthread_mutex_unlock()`.

```
| int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

Enfin, il existe une fonction nommée `pthread_mutex_trylock()` fonctionnant comme `pthread_mutex_lock()`, à la différence qu'elle échoue avec l'erreur `EBUSY`, plutôt que de rester bloquée, si le mutex est déjà verrouillé.

```
| int pthread_mutex_trylock (pthread_mutex_t * mutex);
```

Il est généralement déconseillé d'employer `pthread_mutex_trylock()`. Notamment, si on désire surveiller plusieurs mutex à la fois, on évitera les constructions du genre :

```
| while (1) {  
    if (pthread_mutex_trylock(& mutex_1) == 0)  
        break;  
    if (pthread_mutex_trylock(& mutex_2) == 0)  
        break;  
    if (pthread_mutex_trylock(& mutex_3) == 0)  
        break;  
}
```

Ce code est très mauvais car il gâche inutilement des ressources *CPU*, alors qu'il est possible de le remplacer par une attente de conditions, comme nous le verrons dans le prochain chapitre.

Le programme suivant utilise un mutex comme verrou pour restreindre l'accès au flux `stdout`. Nous lançons en parallèle une dizaine de threads, qui vont attendre une durée aléatoire avant de demander un blocage du mutex. L'attente aléatoire sert à perturber un peu le déterminisme de l'ordonnanceur et à éviter de voir les threads se dérouler dans l'ordre croissant.

exemple-pthread-mutex-1.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static void * fct_threads (void * argument);
static int aleatoire (int maximum);

pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

int main (void)
{
    int i;
    pthread_t thread;

    for (i = 0; i < 5; i++)
        pthread_create(& thread, NULL, fct_threads, (void *) i);
    pthread_exit(NULL);
}

static void * fct_threads (void * argument)
{
    int numero = (int) argument;
    int nombre_iterations;
    int i;
    nombre_iterations = 1 + aleatoire(3);
    for (i = 0; i < nombre_iterations; i++) {
        sleep(aleatoire(3));
        pthread_mutex_lock(& mutex_stdout);
        printf("Le thread %d a obtenu le mutex\n", numero);
        sleep(aleatoire(3));
        printf("Le thread %d relache le mutex\n", numero);
        pthread_mutex_unlock(& mutex_stdout);
    }
    return NULL;
}

static int aleatoire (int maximum)
{
    double d;
    d = (double) maximum * rand();
    d = d / (RAND_MAX + 1.0);
    return ((int) d);
}
```

On remarque l'emploi de `pthread_exit()` en fin de fonction `main()` pour terminer le fil d'exécution principal, sans finir les autres threads. Ceci est parfaitement défini dans la norme SUSv4.

Attention toutefois, les premières implémentations de la *NPTL* contiennent un bogue qui termine toute l'application si le thread `main` invoque `pthread_exit()` avant que les autres threads n'aient réellement démarré. Une solution simple pour éviter ce problème (corrigé dans les implémentations ultérieures) est d'ajouter un petit sommeil d'une seconde avant le `pthread_exit()`.

Le déroulement du processus montre bien que l'accès est correct, malgré les demandes concurrentes de verrouillage du mutex.

```
$ ./exemple-pthread-mutex-1
Le thread 2 a obtenu le mutex
Le thread 2 relache le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relache le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relache le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relache le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relache le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relache le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relache le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relache le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
Le thread 0 a obtenu le mutex
Le thread 0 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
$
```


auquel cas on le supprimera finalement avec :

```
| int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

Le verrouillage dans le cas d'une demande d'accès en lecture seulement se fera ainsi :

```
| int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);  
| int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
```

La variante `pthread_rwlock_tryrdlock()` n'étant pas bloquante mais échouant immédiatement si l'accès est impossible.

Pour demander un accès en lecture et écriture sur la donnée, on appellera :

```
| int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);  
| int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);
```

Enfin, pour libérer le verrou on appelle :

```
| int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Le noyau garantit qu'un thread écrivain sera toujours seul à obtenir l'accès à un moment donné, tandis que les threads lecteurs pourront y accéder à plusieurs simultanément.

Dans la *NPTL*, on considère que la lecture est fréquente et l'écriture rare. Aussi c'est cette dernière qui est privilégiée : dès qu'un thread demande un accès avec `pthread_rwlock_wrlock()`, le noyau bloque toute autre demande d'accès ; il laisse les threads lecteurs déjà en place terminer leur travail, mais il n'en laisse pas entrer de nouveaux.

Conclusion

Nous avons vu dans ce chapitre l'essentiel des éléments de programmation multithread courants. On construira la plupart des applications en n'utilisant que les fonctions que nous avons vues ici. Toutefois, d'autres éléments du standard Pthread sont intéressants. Dans le prochain chapitre, nous traiterons des éléments avancés tels que les annulations, les variables-conditions, d'autres attributs des threads, etc. On retrouvera encore des éléments de programmation multithread dans les chapitres consacrés à l'ordonnancement et à la gestion mémoire.

Aspects avancés des Pthreads

Nous avons vu dans le chapitre précédent les éléments essentiels de programmation multithread. Il nous est d'ores et déjà possible de créer des applications multitâches complètes et efficaces. Toutefois, certaines contraintes peuvent encore nous limiter. Nous allons examiner dans ce chapitre quelques éléments avancés de la programmation multithread.

Annulation d'un thread

Un thread peut se terminer volontairement, comme nous l'avons vu précédemment en appelant `pthread_exit()` ou en invoquant `return` depuis sa fonction principale. Il peut également, nous allons le voir ici, se terminer prématurément sur une demande d'un autre thread, par le mécanisme des annulations.

Rappelons qu'un thread qui commet une erreur fatale (accès à un pointeur invalide, division par zéro, etc.) déclenche l'arrêt global de tout le processus.

L'annulation d'un thread est provoquée par un autre thread du même processus. Il est important de bien comprendre qu'une annulation ne peut jamais provenir de l'extérieur du processus. Il est impossible, depuis l'extérieur, d'arrêter un thread spécifiquement.

Un thread demande l'annulation d'un autre thread en appelant :

```
| int pthread_cancel (pthread_t thread);
```

Cette fonction renvoie 0 si elle réussit, ou l'erreur ESRCH si le thread visé n'existe pas ou plus. Le thread récepteur peut être configuré pour accepter la requête d'annulation, la refuser ou la repousser jusqu'à atteindre un point particulier de son exécution.

Lorsqu'un thread est annulé, il se comporte exactement comme s'il invoquait la fonction `pthread_exit()` avec l'argument spécial `PTHREAD_CANCELED`.

La suppression d'un thread est un phénomène assez subtil. On y a recours généralement lorsqu'un thread n'a plus d'intérêt parce qu'on a obtenu par un autre moyen le résultat qu'il devait fournir, ou après un abandon demandé par l'utilisateur. Prenons l'exemple d'une application recherchant un nombre qui vérifie certaines propriétés¹. Elle peut, pour tirer profit du parallélisme d'une machine multiprocesseur, scinder son espace de recherche en plusieurs portions et déclencher une série de threads, chacun explorant sa portion personnelle. Aussitôt qu'un thread aura trouvé la valeur recherchée, on pourra annuler ses confrères.

Toutefois, un thread manipule souvent des variables globales en employant des techniques de verrouillage que nous étudierons dans les prochains paragraphes. S'il se trouve annulé brutalement au moment de la mise à jour d'une variable globale, il peut la laisser dans un état instable ou abandonner un verrou en position bloquée. Il faut donc pouvoir interdire temporairement les demandes d'annulation dans certaines portions du code.

La fonction `pthread_setcancelstate()` permet de configurer le comportement du thread appelant vis-à-vis d'une requête d'annulation.

```
| int pthread_setcancelstate (int etat_annulation, int * ancien_etat);
```

Les valeurs possibles sont les suivantes.

Nom	Signification
PTHREAD_CANCEL_ENABLE	Le thread acceptera les requêtes d'annulation (comportement par défaut).
PTHREAD_CANCEL_DISABLE	Le thread ne tiendra pas compte des demandes d'annulation.

Le second argument de la fonction est un pointeur sur un entier dans lequel on pourra retrouver l'état précédent du thread. Si cet état ne nous intéresse pas, il est toujours possible de passer un pointeur NULL.

Dans l'exemple suivant, le thread principal crée deux nouveaux threads. Le premier est protégé contre les annulations, mais pas le second. Après trois secondes, le thread principal envoie des demandes d'annulation :

1. On peut imaginer, par exemple, la recherche par la force brute d'une clé de décryptage d'un message.


```
exemple-pthread-cancel.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fct_thread(void * arg);

int main (void)
{
    pthread_t thr1, thr2;
    if ((pthread_create(& thr1, NULL, fct_thread, (void *)1) !=0 )
        ||(pthread_create(& thr2, NULL, fct_thread, (void *)2) !=0 )){
        fprintf(stderr, "Erreur dans pthread_create\n");
        exit(EXIT_FAILURE);
    }
    sleep(3);
    fprintf(stderr, "[main] envoi des annulations\n");
    pthread_cancel(thr1);
    pthread_cancel(thr2);
    pthread_exit(NULL);
}

void * fct_thread(void * arg)
{
    long num = (long) arg;
    if (num == 1)
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while (1) {
        fprintf(stderr, "[Thread %ld] Je suis là\n", num);
        sleep(1);
    }
}
```

À l'exécution, on constate bien que le premier thread continue son travail alors que le second disparaît au moment de l'annulation :

```
$ ./exemple-pthread-cancel-1
[Thread 1] Je suis là
[Thread 2] Je suis là
[Thread 1] Je suis là
[Thread 2] Je suis là
[Thread 1] Je suis là
[Thread 2] Je suis là
[main] envoi des annulations
[Thread 1] Je suis là
[Thread 1] Je suis là
```

```
[Thread 1] Je suis la
           (Control-C)
$
```

Les requêtes d'annulation ne sont pas mémorisées, contrairement aux signaux par exemple, ce qui fait qu'un thread désactivant temporairement les requêtes pendant une zone de code critique ne se terminera pas lorsqu'il autorisera de nouveau les annulations, même si plusieurs demandes sont arrivées pendant ce laps de temps.

Il faut donc trouver un moyen plus souple pour empêcher l'annulation de se produire intempestivement, tout en acceptant les requêtes. Pour cela, on utilise tout simplement un mécanisme de synchronisation fondé sur un retardement des annulations jusqu'à atteindre des emplacements bien définis du code.

Le thread ainsi configuré ne se terminera pas dès la réception d'une annulation, mais continuera de s'exécuter jusqu'à atteindre un point d'annulation, que nous allons définir ci-après. Pour configurer ce comportement, on emploie la fonction `pthread_setcanceltype()` dont le prototype est :

```
| int pthread_setcanceltype (int type_annulation, int * ancien_type);
```

Le type d'annulation peut correspondre à l'une des constantes suivantes.

Nom	Signification
PTHREAD_CANCEL_DEFERRED	Le thread ne se terminera qu'en atteignant un point d'annulation (comportement par défaut).
PTHREAD_CANCEL_ASYNCHRONOUS	L'annulation prendra effet dès réception de la requête.

La norme SUSv4 décrit quatre fonctions qui constituent des points d'annulation, c'est-à-dire des fonctions dans lesquelles un thread est susceptible de se terminer brutalement :

- `pthread_cond_wait()` et `pthread_cond_timedwait()` que nous verrons dans un prochain paragraphe ;
- `pthread_join()` que nous avons déjà étudiée dans le précédent chapitre ;
- `pthread_testcancel()`.

Cette dernière fonction est déclarée ainsi :

```
| void pthread_testcancel (void);
```

Dès qu'on l'invoque, le thread peut se terminer si une demande d'annulation est en attente. On voit donc que dans le cas `PTHREAD_CANCEL_DEFERRED`, un thread ne sera jamais interrompu au milieu d'un calcul ou dans une boucle de manipulation des données. On pourra donc répartir des appels `pthread_testcancel()` dans ce type de code, aux endroits où on est sûr qu'une annulation ne présente pas de danger.

Il existe aussi dans SUSv4 un ensemble minimal d'appels système qui représentent des points d'annulation sur toutes les implémentations des *Pthreads* (certains de ces appels sont optionnels dans SUSv4 et ne sont pas implémentés sous Linux) :

```
accept(), aio_suspend(), clock_nanosleep(), close(), connect(),
creat(), fcntl(), fsync(), getmsg(), getpmsg(), lockf(),
mqreceive(), mqsend(), mq_timed_receive(), mq_timed_send(),
msgrcv(), msqsnd(), msync(), nanosleep(), open(), pause(), poll(),
pread(), pselect(), putmsg(), putpmsg(), pwrite(), read(), readv(),
recv(), recvfrom(), recvmmsg(), select(), sem_timedwait(),
sem_wait(), send(), sendmsg(), sendto(), sigpause(), sigsuspend(),
sigtimedwait(), sigwait(), sigwaitinfo(), sleep(), system(),
tcdrain(), usleep(), wait(), waitid(), waitpid(), write(), writev().
```

Ces fonctions ont en commun de pouvoir bloquer indéfiniment, ce qui représente un gâchis de ressource si le thread doit être annulé. Il existe aussi un nombre important de routines qui peuvent être des points d'annulation, si leurs concepteurs le désirent. On consultera à cet effet la documentation *Gnu* pour connaître les fonctions des bibliothèques concernées.

Un grand nombre de fonctions et d'appels système modifient temporairement des données statiques et ne peuvent pas se permettre d'être interrompus n'importe quand. Cela signifie qu'un appel système bloquant, comme `read()`, ne doit jamais être invoqué si le thread est configuré avec une annulation asynchrone. Les fonctions qui supportent l'annulation asynchrone (*async-cancel safe*) sont explicitement documentées comme telles. Étant donné qu'elles sont extrêmement rares, on se fixera comme règle de ne configurer un thread en mode d'annulation asynchrone que pour des portions de code où il réalise des boucles de calculs intenses, sans aucun appel système.

En résumé, on utilisera principalement les configurations suivantes :

Configuration	Utilisation
PTHREAD_CANCEL_DISABLE	Zone critique où l'on ne peut supporter aucune annulation, même si on invoque un appel système.
PTHREAD_CANCEL_ENABLE PTHREAD_CANCEL_DEFERRED	Comportement habituel des threads.
PTHREAD_CANCEL_ENABLE PTHREAD_CANCEL_ASYNCHRONOUS	Boucle de calculs gourmande en CPU, sans appel système.

Fonctions de nettoyage

Comme un thread peut être légitimement annulé pratiquement à n'importe quel moment, il convient de trouver un moyen de libérer les ressources qu'il peut main-

tenir, avant qu'il se termine vraiment. En effet, quand un thread disparaît, la mémoire dynamique qu'il s'était allouée n'est pas libérée automatiquement par le noyau, contrairement à la fin d'un processus. Le même problème se pose avec les fichiers ouverts, les tubes de communication, les sockets réseau... De plus, un thread peut avoir verrouillé une ou plusieurs ressources partagées, afin d'en avoir temporairement l'usage exclusif, et il convient de relâcher les verrous posés.

Pour cela, la norme SUSv4 propose un mécanisme assez élégant, bien qu'un peu surprenant à première vue. Lorsqu'un thread s'attribue une ressource – mémoire, fichier, verrous, etc. – qui nécessitera une libération ultérieure, il enregistre le nom d'une routine de libération dans une pile spéciale, avec la fonction `pthread_cleanup_push()`. Lorsque le thread se termine, les routines sont dépilées – dans l'ordre inverse de leur enregistrement – et exécutées.

Quand un thread désire libérer explicitement une ressource, à la fin d'une fonction par exemple, il appelle `pthread_cleanup_pop()`, qui extrait la dernière routine enregistrée et l'invoque. Les prototypes de ces deux fonctions sont les suivants :

```
void pthread_cleanup_push (void (* fonction) (void * argument),  
                           void * argument);  
void pthread_cleanup_pop (int execution_routine);
```

La routine `pthread_cleanup_push()` prend donc en premier argument un pointeur de fonction, et en second un pointeur générique pouvant représenter n'importe quel objet. Lorsque la routine de nettoyage est appelée, elle reçoit en argument le second pointeur. En général, on utilisera :

```
FILE * fp;  
fp = fopen(nom_fichier, "r");  
pthread_cleanup_push fclose, fp);
```

ou

```
char * buffer;  
buffer = malloc(BUFSIZE);  
pthread_cleanup_push(free, buffer);
```

ou encore

```
int fd;  
fd = open(nom_fichier, O_RDONLY);  
pthread_cleanup_push(close, (void *) fd);
```

Naturellement, il est possible de transmettre un pointeur sur une routine personnelle qui libère toutes les ressources tenues éventuellement par le thread.

Lorsqu'on désire invoquer explicitement la routine de libération, on emploie `pthread_cleanup_pop()` en lui passant un argument entier. Si cet argument est nul, la routine est retirée de la pile de nettoyage, mais elle n'est pas exécutée. Sinon, la routine est extraite et invoquée.

Ce mécanisme nécessite donc de soigner la conception du programme pour disposer les allocations et libérations autour des zones où le thread peut être annulé. On se disciplinera pour adopter ce comportement dans toutes les fonctions de l'application, en employant par exemple :

```
void routine_dialogue (char * nom_serveur, char
*nom_fichier_enregistrement)
{
    char * buffer;
    FILE * fichier;
    int     socket_serveur;
    int     nb_recus;

    buffer = malloc(BUFSIZE);
    if (buffer != NULL) {
        pthread_cleanup_push(free, buffer);

        socket_serveur = ouverture_socket(nom_serveur);
        if (socket_serveur >= 0) {
            pthread_cleanup_push(close, (void *) socket_serveur);

            fichier = fopen(nom_fichier_enregistrement, "w");
            if (fichier != NULL) {
                pthread_cleanup_push(fclose, fichier);

                while (1) {
                    nb_recus = lecture_socket(socket_serveur, buffer);
                    if (nb_recus < 0)
                        break;
                    if (fwrite(buffer, 1, nb_recus, fichier) != nb_recus)
                        break;
                }
                pthread_cleanup_pop(1); /* fclose(fichier) */
            }
            pthread_cleanup_pop(1); /* close(socket_serveur) */
        }
        pthread_cleanup_pop(1); /* free(buffer) */
    }
}
```

Pour obliger le programmeur à adopter un comportement cohérent dans toute la fonction, la norme SUSv4 impose une contrainte assez restrictive à l'utilisation de ces rou-

tines. En effet, les appels `pthread_cleanup_push()` et `pthread_cleanup_pop()` doivent se trouver dans la même fonction et dans le même bloc lexical. Cela signifie qu'ils doivent être au même niveau d'imbrication entre accolades. On peut le vérifier d'un coup d'œil en s'assurant que le `pthread_cleanup_pop()` se trouve bien au même niveau d'indentation que le `pthread_cleanup_push()` correspondant.

Pour comprendre la raison de cette restriction, il suffit de savoir que l'implémentation de ces routines dans la plupart des bibliothèques, dont *NPTL*, est réalisée par deux macros : la première comprend une accolade ouvrante alors que la seconde contient l'accolade fermante associée. Il importe donc de considérer ces deux fonctions comme une paire d'accolades et de ne pas essayer de les séparer de plus d'un bloc lexical.

On prendra comme habitude de faire systématiquement suivre un appel de la fonction `pthread_cleanup_pop()` d'un commentaire indiquant son effet, comme on peut le voir précédemment. On remarquera également que le fait de n'avoir plus qu'un seul point de sortie d'une routine oblige parfois à une indentation excessive. Pour éviter ce problème, on peut scinder la routine en plusieurs sous-fonctions ou utiliser des sauts `goto`, comme on les tolère parfois pour les gestions d'erreur.

Le principe des fonctions de nettoyage est puissant, mais complique sensiblement la compréhension du code source. Notez bien que ce mécanisme n'a d'intérêt que dans le cas où un thread est susceptible d'être annulé. Rappelons que l'annulation ne peut provenir que de l'intérieur du processus, jamais depuis un autre processus.

Cela signifie que si vous n'utilisez jamais `pthread_cancel()` dans votre application – ce qui est probablement le cas dans la majorité des processus multithreads – il est inutile d'avoir recours aux `pthread_cleanup_push()` et `pthread_cleanup_pop()`.

Variables conditions

Supposons qu'un thread ait besoin d'envoyer vers un client réseau toutes les données qu'il reçoit dans différents buffers (en provenance d'autres threads, par exemple). La première approche pourrait être :

```
while (1) {
    if (buffer_non_vide(buffer_1))
        envoyer_donnees(buffer_1);
    if (buffer_non_vide(buffer_2))
        envoyer_donnees(buffer_2);
    if (buffer_non_vide(buffer_3))
        envoyer_donnees(buffer_3);
    if (buffer_non_vide(buffer_4))
        envoyer_donnees(buffer_5);
}
```

Toutefois, ce code est très pénalisant pour les autres threads, car il consomme inutilement des ressources CPU en bouclant sur lui-même en permanence, tant qu'aucune donnée n'arrive. L'idéal serait de pouvoir s'arrêter en attente passive au début de la boucle, et être averti quand des données sont effectivement disponibles dans un des .

Pour cela, il existe un mécanisme de synchronisation spécifique : les variables « conditions » représentées par le type `pthread_cond_t`. Un thread peut se mettre en attente sur une variable condition, et un autre thread pourra le réveiller en agissant sur la variable condition quand l'événement attendu par le premier thread se sera produit.

Le principe est simple, reposant sur deux fonctions de manipulation des conditions : l'une est l'attente de la condition, le thread appelant restant bloqué jusqu'à ce qu'elle soit réalisée, et l'autre sert à signaler que la condition est remplie. C'est l'application qui affecte une signification à la variable condition.

Les variables conditions ont des attributs, de type `pthread_condattr_t`, qui n'ont pas d'utilité dans la bibliothèque NPTL. En conséquence, on initialisera généralement les conditions de manière statique :

```
| pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

ou en employant la fonction `pthread_cond_init()`, en passant un second argument `NULL`

```
| int pthread_cond_init (pthread_cond_t * cond,  
                        pthread_condattr_t * attr);
```

Une condition inutilisée est libérée avec `pthread_cond_destroy()`. Aucun autre thread ne doit être en attente sur la condition, sinon la libération échoue avec l'erreur `EBUSY`.

```
| int pthread_cond_destroy (pthread_cond_t * cond);
```

Voyons à présent l'utilisation effective d'une condition. Tout d'abord, il faut signaler qu'une condition est toujours associée à un mutex, cela pour éviter des problèmes de concurrence d'accès sur la variable.

Examinons d'abord le thread qui doit attendre une condition.

- On initialise la variable condition et le mutex qui lui est associé.
- Le thread bloque le mutex. Ensuite, il invoque la routine `pthread_cond_wait()` qui attend que la condition soit réalisée.
- Le thread libère le mutex.

Maintenant, voyons le thread qui réalise la condition.

- Le thread travaille jusqu'à avoir réalisé la condition attendue.
- Il bloque le mutex associé à la condition.
- Le thread appelle la fonction `pthread_cond_signal()` pour montrer que la condition est remplie.

- Le thread débloque le mutex.

Ce schéma est a priori surprenant puisqu'il semble que, lorsque le second thread désire signaler la réalisation de la condition, l'accès lui soit interdit, le premier thread ayant bloqué le mutex. En fait, la fonction `pthread_cond_wait()` fonctionne en trois temps.

- 1 D'abord, elle débloque le mutex associé à la condition, et elle se met en attente. Cette opération est réalisée de manière atomique vis-à-vis de la bibliothèque NPTL.
- 2 L'attente se poursuit jusqu'à ce que la réalisation de la condition soit indiquée.
- 3 La condition étant remplie, la fonction termine son attente et bloque à nouveau le mutex, avant de revenir dans le programme appelant.

Le scénario se déroule donc de la manière suivante.

Thread attendant la condition	Thread signalant la condition
Appel de <code>pthread_mutex_lock()</code> : blocage du mutex associé à la condition.	
Appel de <code>pthread_cond_wait()</code> : déblocage du mutex.	
... attente...	
	Appel de <code>pthread_mutex_lock()</code> sur le mutex.
	Appel de <code>pthread_cond_signal()</code> , qui réveille l'autre thread.
Dans <code>pthread_cond_wait()</code> , tentative de récupérer le mutex. Blocage.	
	Appel de <code>pthread_mutex_unlock()</code> . Le mutex étant libéré, l'autre thread se débloque.
Fin de <code>pthread_cond_wait()</code> .	
Appel de <code>pthread_mutex_unlock()</code> pour revenir à l'état initial.	

On peut vérifier qu'il n'y a pas de risque d'interblocage des deux threads ni de risque de perdre la signalisation d'une condition dès que le premier `pthread_mutex_lock()` a été invoqué. En général, le thread qui attend les conditions verrouille le mutex dès son initialisation, puis boucle indéfiniment autour de `pthread_cond_wait()`, ce qui garantit ainsi que les `pthread_cond_signal()` ne pourront se produire que lorsque le thread sera en attente. Dans ce cas, il n'y a aucun risque de perte de réalisation de la condition.

Les prototypes de ces deux fonctions sont les suivants :

```
int pthread_cond_signal (pthread_cond_t * condition);
int pthread_cond_wait  (pthread_cond_t * condition,
                       pthread_mutex_t * mutex);
```


Le terme « signal » présent dans `pthread_cond_signal()` ne doit pas être confondu avec les signaux Unix que nous traiterons plus loin. Les interactions entre threads et signaux seront d'ailleurs traitées à cette occasion.

L'exemple précédent deviendra donc :

```
pthread_cond_t cnd_donnees_buffer = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mtx_donnees_buffer = PTHREAD_MUTEX_INITIALIZER;

[...]

pthread_mutex_lock(& mtx_donnees_buffer);
while (1) {
    pthread_cond_wait(& cnd_donnees_buffer, & mtx_donnees_buffer);
    if (buffer_non_vide(buffer_1))
        envoyer_donnees(buffer_1);
    if (buffer_non_vide(buffer_2))
        envoyer_donnees(buffer_2);
    if (buffer_non_vide(buffer_3))
        envoyer_donnees(buffer_3);
    if (buffer_non_vide(buffer_4))
        envoyer_donnees(buffer_4);
    if (buffer_non_vide(buffer_5))
        envoyer_donnees(buffer_5);
}
```

Le thread qui inscrit des données dans le buffer le signalera ainsi :

```
pthread_mutex_lock(&mtx_donnees_buffer);
/* Écriture des données dans un buffer */
[...]
pthread_cond_signal (& cnd_donnees_buffer);
pthread_mutex_unlock(& mtx_donnees_buffer);
```

Ici, le fait de verrouiller le mutex avant d'écrire les données dans le buffer permet de garantir un accès exclusif à ces données. Toutefois, dans d'autres cas, le mutex ne servira qu'à encadrer l'appel à `pthread_cond_signal()`.

En fait, la norme SUSv4 autorise l'appel `pthread_cond_wait()` à se terminer de manière impromptue, même si la condition n'est pas réalisée. Cela, je suppose, permet de simplifier l'implémentation d'une bibliothèque Pthreads vis-à-vis des appels système interrompus par un signal.

Il faut donc accompagner l'invocation de `pthread_cond_wait()` d'une vérification de l'état de la condition. Dans l'exemple précédent, c'est ce qui est réalisé en testant chaque buffer pour vérifier s'il y en a un non vide. Plus généralement, il ne faut pas faire une confiance aveugle à la sortie d'un `pthread_cond_wait()` ; il faut retester la condition attendue.

Il existe une fonction d'attente temporisée, nommée `pthread_cond_timedwait()`, permettant de limiter le délai imparti pour la réalisation de la condition.

```
int pthread_cond_timedwait (pthread_cond_t * condition,  
                             pthread_mutex_t * mutex,  
                             const struct timespec * date);
```

Attention, on ne précise pas la durée d'attente, mais l'heure maximale jusqu'à laquelle la fonction peut attendre. La structure `timespec` contient un champ contenant le nombre de secondes écoulées depuis le 1^{er} janvier 1970, et un champ indiquant le complément en nanosecondes. Pour obtenir la date actuelle, on peut employer les appels système `time()` ou `gettimeofday()`. Tout cela sera étudié plus en détail dans le prochain chapitre.

Si le délai est dépassé, cette fonction échoue avec l'erreur `ETIMEDOUT`. Même dans ce cas, il est important de s'assurer que la condition n'est quand même pas vérifiée, notamment si plusieurs threads attendent la réalisation de la même condition.

En effet, la fonction `pthread_cond_signal()` garantit qu'un seul thread en attente sera réveillé. Lorsqu'on désire réveiller tous les threads qui surveillent cette condition, il faut employer `pthread_cond_broadcast()`. Dans un cas comme dans l'autre, aucune erreur ne se produit si aucun thread n'est en attente.

```
int pthread_cond_broadcast (pthread_cond_t * condition);
```

Les personnes découvrant la programmation multithread sont souvent surprises par le comportement de `pthread_cond_wait()` comme point d'annulation. En effet, lorsqu'un thread reçoit une demande d'annulation durant cette fonction d'attente, elle se termine, mais doit récupérer d'abord le mutex associé à la condition. Cela signifie qu'elle peut bloquer indéfiniment avant de se terminer.

Cette attitude peut surprendre si on considère l'annulation comme une demande de terminaison urgente. Mais ce n'est pas la bonne façon de voir cette fonctionnalité. Il est préférable d'imaginer la demande d'annulation à la manière des applications graphiques dans lesquelles le clic sur un bouton « Fermeture » ne termine pas nécessairement l'application, mais peut passer par une phase de sauvegarde éventuelle des données modifiées si l'utilisateur le désire.

L'annulation d'un thread doit laisser les données manipulées dans un état prévisible, et le seul état prévisible du mutex associé à un appel `pthread_cond_wait()` est le verrouillage. Bien entendu, le thread ne doit pas se terminer en laissant le mutex bloqué. Il faut donc utiliser une fonction de nettoyage :

```
pthread_mutex_lock(& mutex);  
pthread_cleanup_push(pthread_mutex_unlock, (void *) & mutex);  
while (! condition_realisee)  
    pthread_cond_wait(& condition, & mutex);  
pthread_cleanup_pop(1); /* pthread_mutex_unlock (& mutex) */
```

Pour terminer cette section sur les variables conditions, mentionnons qu'il existe deux fonctions `pthread_condattr_init()` et `pthread_condattr_destroy()` permettant de manipuler les attributs des conditions de manière dynamique.

```
int pthread_condattr_init (pthread_condattr_t * attributs);  
int pthread_condattr_destroy (pthread_condattr_t * attributs);
```

Cela n'a pas grand intérêt sous Linux, car l'implémentation de la bibliothèque NPTL ne gère pas d'attribut vraiment utile pour les variables conditions.

Types de mutex

Revenons quelques instants sur les mutex que nous avons étudiés dans le chapitre précédent. Nous l'avons vu, les mutex servent à protéger une variable, une portion de code, une ressource quelconque, de façon à garantir l'unicité de l'accès pendant une période de temps. Posons-nous toutefois quelques questions à leur propos. Par exemple, il est légitime de se demander ce qui peut se passer :

- si un thread essaye de déverrouiller un mutex qui a été verrouillé par un autre thread ;
- si un même thread essaye de verrouiller plusieurs fois de suite le même mutex ;
- si un thread essaye de déverrouiller un mutex qui n'est pas verrouillé ;

Contrairement à ce que l'on pourrait croire, les mutex de la norme SUSv4 n'ont, par défaut, pas de comportements bien définis dans ces situations. Pire, certains comportements par défaut sont plutôt surprenants. Prenons, par exemple, le cas d'un thread qui déverrouille un mutex tenu par un autre thread.

```
exemple-mutex-type-1.c :  
#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  
  
void * fonction_thread(void * arg);  
  
int main (void)  
{  
    pthread_t thr;  
    pthread_mutex_lock(& mtx);  
    pthread_create(& thr, NULL, fonction_thread, NULL);  
    pause();  
    pthread_exit(NULL);  
}
```

```
void * fonction_thread(void * arg)
{
    int ret;
    fprintf(stderr, "Thread deverrouille le mutex : ");
    ret = pthread_mutex_unlock(& mtx);
    fprintf(stderr, "resultat %d\n", ret);
    sleep(1);
    fprintf(stderr, "Thread re-verrouille le mutex : ");
    ret = pthread_mutex_lock(& mtx);
    fprintf(stderr, "resultat %d\n", ret);
    pthread_exit(NULL);
}
```

Ici, le thread `main()` a verrouillé le mutex dès le début du programme. Son exécution est suspendue par l'instruction `pause()` – que nous verrons ultérieurement avec le traitement des signaux Unix –, mais il est clair que ce thread s'attend à retrouver son mutex verrouillé lorsqu'il reviendra de sa pause.

Subrepticement, un autre thread demande à déverrouiller le même mutex, puis à le reverrouiller en son nom. Intuitivement, il semble que ce comportement soit impossible. Pourtant, il n'en est rien. Pour le vérifier, nous affichons le code de retour des opérations `pthread_mutex_unlock()` et `pthread_mutex_lock()`.

```
$ ./exemple-mutex-type-1
Thread deverrouille le mutex : resultat 0
Thread re-verrouille le mutex : resultat 0
      (Control-C)
$
```

Le code de retour nul indique une réussite des opérations. Dans cette première situation, les mutex semblent peu fiables. Du moins dans leur configuration par défaut.

Essayons à présent les autres situations à risque présentées plus haut. Dans ce second exemple, le même thread va essayer de verrouiller deux fois de suite le même mutex.

```
exemple-mutex-type-2.c :
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

int main (void)
{
```

```
fprintf(stderr, "Je verrouille le mutex...");
if (pthread_mutex_lock(& mtx) == 0)
    fprintf(stderr, "Ok\n");
else
    fprintf(stderr, "Erreur\n");
sleep(1);
fprintf(stderr, "Je verrouille a nouveau le mutex...");
if (pthread_mutex_lock(& mtx) == 0)
    fprintf(stderr, "Ok\n");
else
    fprintf(stderr, "Erreur\n");
pthread_exit(NULL);
}
```

À l'exécution, nous voyons que lors du second verrouillage, le thread reste bloqué indéfiniment.

```
$ ./exemple-mutex-type-2
Je verrouille le mutex...Ok
Je verrouille a nouveau le mutex...      (... Control-C)
$
```

Ce cas de *deadlock* pourrait pourtant être détecté facilement puisque c'est le même thread qui demande deux fois le même mutex. Cela sera possible en modifiant l'attribut *type* du mutex.

Enfin, dans la dernière situation, nous allons déverrouiller un mutex qui n'est pas verrouillé, et observer le code de retour de `pthread_mutex_unlock()`.

```
exemple-mutex-type-3.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

int main (void)
{
    int ret;
    fprintf(stderr, "Je deverrouille le mutex...");
    ret = pthread_mutex_unlock(& mtx);
    fprintf(stderr, "retour %d\n", ret);
    pthread_exit(NULL);
}
```

À l'exécution, le code de retour nul n'indique aucune erreur :

```
$ ./exemple-mutex-type-3
Je deverrouille le mutex...retour 0
$
```

Dans ces trois situations, les mutex adoptent un comportement par défaut plutôt surprenant et manquant probablement de robustesse.

La justification de ce choix dans SUSv4 consiste à conseiller une implémentation par défaut la plus rapide et efficace possible (en évitant donc tous les tests d'erreur). Pour la mise au point et le débogage, la norme SUSv4 introduit trois types de mutex incluant les vérifications d'erreurs attendues, que l'on configure à travers les attributs des mutex avec les fonctions suivantes :

```
int pthread_mutexattr_init (pthread_mutexattr_t * attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t * attr);
```

puis

```
int pthread_mutexattr_gettype (pthread_mutexattr_t * attr,
                               int * type);
int pthread_mutexattr_settype (pthread_mutexattr_t * attr,
                               int type);
```

Les types définis sont PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK et PTHREAD_MUTEX_RECURSIVE. En outre, un type PTHREAD_MUTEX_DEFAULT représente le comportement par défaut choisi par l'implémentation. Dans la NPTL, le type PTHREAD_MUTEX_DEFAULT est identique à PTHREAD_MUTEX_NORMAL.

Les comportements des différents types de mutex sont les suivants.

- PTHREAD_MUTEX_NORMAL : nous avons vu le comportement avec la NPTL ; pour la norme SUSv4, le déverrouillage d'un mutex tenu par un autre thread ou non verrouillé mène à un comportement non spécifié par le standard, et le double verrouillage du même mutex dans le même thread conduit à un blocage définitif (*deadlock*).
- PTHREAD_MUTEX_ERRORCHECK : les tentatives de déverrouiller un mutex non verrouillé ou appartenant à un autre thread échouent en renvoyant une erreur ; le deuxième appel `pthread_mutex_lock()` sur le même mutex dans le même thread échoue sans bloquer, en renvoyant une erreur.
- PTHREAD_MUTEX_RECURSIVE : les déverrouillages d'un mutex non verrouillé ou tenu par un autre thread échouent comme avec un mutex de type PTHREAD_MUTEX_ERRORCHECK ; le fait de verrouiller plusieurs fois le même mutex dans le même thread est accepté, mais il faudra autant de déverrouillages pour que le mutex devienne disponible pour les autres threads.

Pour pouvoir utiliser les mutex de type `ERRORCHECK` et `RECURSIVE`, il est nécessaire d'initialiser la constante symbolique `_XOPEN_SOURCE` à la valeur 500 avant d'inclure les fichiers d'en-tête de la bibliothèque C.

L'exemple suivant est une simple adaptation de `exemple-mutex-type-1.c` dans lequel on a initialisé dynamiquement le mutex ainsi :

exemple-mutex-type-4.c :

```
#define _XOPEN_SOURCE 500
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mtx;

void * fonction_thread(void * arg);

int main (void)
{
    pthread_t thr;
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(& attr);
    pthread_mutexattr_settype(&attr,
                              PTHREAD_MUTEX_ERRORCHECK);
    pthread_mutex_init(& mtx, & attr);
    pthread_mutex_lock(& mtx);
    pthread_create(& thr, NULL, fonction_thread, NULL);
    pause();
    pthread_exit(NULL);
}

void * fonction_thread(void * arg)
{
    int ret;
    fprintf(stderr, "Thread deverrouille le mutex : ");
    ret = pthread_mutex_unlock(& mtx);
    fprintf(stderr, "resultat %d\n", ret);
    sleep(1);
    fprintf(stderr, "Thread re-verrouille le mutex : ");
    ret = pthread_mutex_lock(& mtx);
    fprintf(stderr, "resultat %d\n", ret);
    pthread_exit(NULL);
}
```

Le résultat d'exécution montre que le déverrouillage d'un mutex n'appartenant pas au thread renvoie une erreur, en l'occurrence l'erreur 1 (`EPERM` – *Permission non accordée*) et que le mutex n'est pas libre par la suite :

```
$ ./exemple-mutex-type-4
Thread deverrouille le mutex : resultat 1
Thread re-verrouille le mutex : (Control-C)
$
```

Le résultat serait identique pour le type RECURSIVE. Nous encourageons le lecteur à appliquer la même modification de type de mutex dans les exemples 2 et 3 et à vérifier ainsi les comportements des différents types de mutex.

Taille de la pile

Les attributs `stackaddr` et `stacksize` permettent de configurer la pile utilisée par un thread. Comme nous l'avons déjà indiqué, chaque thread dispose d'une pile personnelle, dans laquelle sont allouées toutes ses variables automatiques. Il peut être parfois nécessaire de réclamer une pile de dimension plus grande que celle qui est fournie par défaut si le thread à créer fait un large usage de fonctions récursives, par exemple.

À l'inverse, il peut être nécessaire de réduire la taille de la pile accordée à un thread. Par défaut, la NPTL emploie des piles de 10 Mo, ce qui est bien souvent très supérieur aux besoins des threads, et ce qui limite le nombre de threads utilisables (de l'ordre de 300 threads). En réduisant la taille de la pile à 16 ko, on peut employer plusieurs milliers, voire dizaines de milliers de threads.

Comme nous le verrons ultérieurement, la pile d'un processus habituel n'est contrainte que par la limite de la zone nommée *tas*, dans laquelle les variables dynamiques sont allouées. En principe, la pile d'un tel processus pourrait croître jusqu'à remplir l'essentiel de l'espace d'adressage du programme, soit environ 3 Go. Dans le cas d'un programme multithread, les différentes piles doivent être positionnées à des emplacements figés dès la création des threads, ce qui impose *ipso facto* des limites de taille puisqu'elles ne doivent pas se rejoindre.

Les routines citées ci-après permettent respectivement de lire ou d'indiquer l'adresse de départ et la taille maximale de la pile du thread. La dimension minimale de la pile est disponible dans la constante symbolique `PTHREAD_STACK_MIN` (16 ko sur PC 32 bits sous Linux).

Naturellement, l'emploi de ces attributs est assez pointu, et on le réservera aux applications pour lesquelles ils sont réellement indispensables.

```
int pthread_attr_getstackaddr (const pthread_attr_t * attributs,
                               void ** valeur);
int pthread_attr_setstackaddr (pthread_attr_t * attributs,
```



```
void * valeur);  
int pthread_attr_getstacksize (const pthread_attr_t * attributs,  
                               size_t * valeur);  
int pthread_attr_setstacksize (pthread_attr_t * attributs,  
                               size_t valeur);
```

La réduction de la taille de la pile peut servir néanmoins dans les applications nécessitant plusieurs centaines de threads simultanés, comme nous l'avons indiqué dans le chapitre précédent.

Appel de fork()

On peut se demander ce qui se passe lorsqu'un thread appelle `fork()`. Le comportement est tout à fait logique. Le processus entier est dupliqué, y compris les zones de mémoire partagées avec les autres threads. En revanche, il n'y a dans le processus fils qu'un seul fil d'exécution, celui du thread qui a invoqué `fork()`, cela quel que soit le nombre de threads concurrents avant la séparation.

Un premier problème se pose, car les piles et les zones de mémoire dynamiquement allouées par les autres threads continuent d'être présentes dans l'espace mémoire du nouveau processus, même s'il n'a aucun moyen d'y accéder. Aussi, en théorie, ce mécanisme doit être restreint uniquement à l'utilisation de `exec()` après le `fork()`.

Un second problème peut se poser si un autre thread a verrouillé – dans le processus père – une ressource. Si le thread restant dans le processus fils a besoin de cette ressource, celle-ci persiste à être verrouillée, et on risque un blocage définitif.

Pour résoudre ce problème, il existe une fonction nommée `pthread_atfork()`, qui permet d'enregistrer des routines qui seront automatiquement invoquées si un thread appelle `fork()`. Les fonctions sont exécutées dans l'ordre inverse de leur enregistrement, comme avec une pile. La liste des fonctions mémorisées est commune à tous les threads.

On peut enregistrer trois fonctions avec `pthread_atfork()`. La première routine est appelée avant le `fork()` dans le thread qui l'invoque. Les deux autres routines sont appelées après la séparation, l'une dans le processus fils, et l'autre dans le processus père – toujours au sein du thread ayant invoqué `fork()`.

```
int pthread_atfork (void (* avant) (void),  
                   void (* dans_pere) (void),  
                   void (* dans_fils) (void));
```

Si un pointeur est nul, la routine est ignorée. Dans l'encadrement de `fork()`, on essaye d'éviter la situation suivante :

- Le thread 1 bloque un mutex.
- Le thread 2 appelle `fork()`, dupliquant l'ensemble de l'espace mémoire du processus, y compris le mutex bloqué.
- Le processus père continue de se dérouler normalement, le thread 1 libérant le mutex après ses modifications, et le thread 2 pouvant poursuivre son exécution.
- Dans le processus fils, le thread 2 veut accéder à la zone de données commune. Celle-ci étant verrouillée, il attend que le thread 1 libère le mutex, mais il n'y a pas de thread 1 dans le fils ! Le processus est définitivement bloqué.

La bonne manière de procéder est la suivante, un peu complexe, mais correcte :

- Avant que le thread 1 bloque le mutex – par exemple pendant son initialisation –, on installe la routine `avant()`, qui correspond à un blocage du mutex, ainsi que `dans_pere()` et `dans_fils()`, deux routines qui représentent une libération du mutex.
- Le thread 1 bloque le mutex.
- Le thread 2 appelle `fork()`. La routine `avant()` est invoquée. Correspondant à une demande de blocage du mutex, elle reste bloquée jusqu'à ce que le thread 1 ait terminé son travail.
- Le thread 1 libère le mutex.
- La routine `avant()` bloque le mutex et se termine.
- L'appel système `fork()` a lieu, les processus se séparent. Les routines `dans_pere()` et `dans_fils()` sont invoquées, libérant le mutex dans les deux contextes.
- Les threads 1 et 2 du processus père continuent normalement.
- Le thread 2 du processus fils peut accéder à la zone de données s'il le désire, le mutex est libre.

Nous voyons qu'il faut donc enregistrer une série de routines pour chaque mutex susceptible d'être employé dans le processus fils, ce qui complique – parfois excessivement – l'écriture des programmes.

En fait, l'appel `pthread_atfork()` est principalement employé dans des programmes expérimentaux, pour étudier justement les blocages dus aux partages de verrous. Dans des applications courantes, on évite généralement de se trouver dans cette situation. Pour cela, on essaye de ne pas utiliser `fork()`, ou de le faire suivre immédiatement d'un `exec()`. On peut aussi appeler `fork()` avant la création des threads et installer un mécanisme de communication entre processus, comme nous en verrons dans des chapitres ultérieurs.

Données globales privées

Les threads doivent souvent manipuler des données privées, non partagées avec les autres threads. Dans la plupart des cas, on peut simplement utiliser des variables

locales qui sont stockées dans la pile privée du thread au moment de l'entrée dans la fonction. Il y a pourtant des cas où un thread a besoin d'utiliser des variables privées disponibles de manière globale. On peut par exemple imaginer un module implémentant une bibliothèque de fonctions qui stocke certaines informations dans des variables statiques entre deux invocations de fonction.

Pour permettre ce comportement, la norme SUSv4 introduit la notion de clés associées à des données privées. La clé est une variable de type `pthread_key_t`, qui peut résider en variable statique. La bibliothèque associe la clé avec un pointeur `void *` différent pour chaque thread.

L'initialisation d'une clé privée se fait à l'aide de la fonction `pthread_key_create()`, à laquelle on peut éventuellement passer un pointeur sur une fonction de destruction qui libère le pointeur associé si le thread se termine avant que la clé ne soit détruite avec `pthread_key_delete()`.

```
int pthread_key_create (pthread_key_t * cle_privee,  
                        void (* fonction) (void *));  
int pthread_key_delete (pthread_key_t cle_privee);
```

Une fois qu'une clé a été initialisée, on utilise la fonction `pthread_setspecific()` pour l'associer à un pointeur représentant des données personnelles du thread.

```
int pthread_setspecific (pthread_key_t cle_privee, const void * data);
```

Pour lire les données associées à une clé, on emploie `pthread_getspecific()` :

```
void * pthread_getspecific (pthread_key_t cle_privee);
```

En imaginant un module permettant de charger un fichier de données, puis d'accéder ensuite à son contenu à travers des fonctions d'interrogation, on peut construire le schéma suivant :

```
pthread_key_t    cle;  
  
int ouverture_fichier (const char * nom_fichier);  
{  
    FILE * fp;  
    struct donnees * donnees;  
    int nb_donnees;  
    int i;  
    pthread_key_create(& cle, free);  
    fp = fopen(nom_fichier, "r");  
    /* Lecture nb_données */  
    [...]
```

```
donnees = (struct donnees *) calloc(nb_donnees,
                                     sizeof(struct donnees));
/* Lecture des données */
for (i = 0; i < nb_donnees; i++)
    [...]
pthread_setspecific(& cle, donnees);
return nb_donnees;
}

int resultat_donnee (int num)
{
    struct donnees * donnees;
    donnees = (struct donnees *) pthread_getspecific(& cle);
    return donnees[num].resultats;
}
```

En fait, la robustesse du programme serait sensiblement renforcée si on pouvait garantir que l'initialisation de la clé avec `pthread_key_create()` n'ait lieu qu'une seule fois, quel que soit le nombre de threads passant par `ouverture_fichier()`. Bien sûr, nous pouvons construire un mécanisme basé sur une variable d'état protégée par un mutex, mais la NPTL met directement à notre disposition une structure permettant d'assurer l'unicité d'exécution d'une portion de code.

Pour utiliser cette fonction, il faut préalablement définir une variable statique de type `pthread_once_t`, initialisée avec la constante `PTHREAD_ONCE_INIT`, qu'on passera par adresse à la routine `pthread_once()`. Le second argument est un pointeur sur une fonction d'initialisation qui ne sera ainsi appelée qu'une seule fois dans l'application.

```
int pthread_once (pthread_once_t * once,
                 void (* fonction) (void));
```

Naturellement, seule la première invocation de `pthread_once()` a un effet, les appels ultérieurs n'ayant plus aucune influence.

Conclusion

Nous avons observé, au gré de ces deux chapitres, l'essentiel des fonctions permettant d'employer des threads. Nous allons les rencontrer à nouveau à plusieurs reprises, notamment dans le chapitre consacré à l'ordonnancement sous Linux.

Fonctions horaires

Nous allons, dans les chapitres à venir, examiner comment le noyau Linux exécute les différentes tâches, comment il les choisit, les place sur le processeur (ou les processeurs) ou les en retire.

Pour commencer, nous devons nous munir des outils adéquats, c'est-à-dire savoir mesurer l'heure de manière assez précise pour nos expériences, ce qui fera, entre autres, l'objet de ce chapitre.

Il est fréquent qu'une application fasse un usage plus ou moins large de la date ou de l'heure. On peut désirer horodater des enregistrements ou des messages, mémoriser des dates de naissance, vider les données trop vieilles se trouvant en mémoire, ou simplement attendre une dizaine de secondes pour laisser à l'utilisateur la possibilité de réagir et de modifier la configuration par défaut.

Malgré tout, la manipulation des dates est souvent source de problèmes. Il m'a fallu écrire, pour une application de supervision d'un système de radiolocalisation, un module enregistrant diverses statistiques (nombre de trames reçues, états de certains bits d'alarme, etc.). Ces valeurs devaient être mémorisées et cumulées seconde par seconde sur la dernière minute, minute par minute sur la dernière heure, heure par heure sur les dernières vingt-quatre heures, et jour par jour pendant un an. Les complications commencent lorsqu'on sait que les événements à enregistrer n'arrivaient pas nécessairement toutes les secondes mais pouvaient se produire une ou deux fois par mois seulement. Bien entendu, il fallait conserver quand même les statistiques à jour en permanence et pouvoir les afficher à tout moment (en gérant notamment les problèmes dus aux années bissextiles). Ce genre de fonctionnalité devient vite assez acrobatique à élaborer, alors qu'il ne s'agit en réalité que d'une partie accessoire d'un logiciel servant par ailleurs à tout autre chose.

C'est peut-être en cela que la manipulation des données horaires pose des difficultés. Il s'agit souvent de fonctions annexes ou de simples routines d'affichage à l'écran, auxquelles on n'accorde pas toujours l'attention nécessaire. De plus, des cas particuliers peuvent se produire sortant largement du cadre des tests du logiciel. Le problème de l'année bissextile vient bien sûr immédiatement à l'esprit, mais on peut aussi citer l'horloge interne que l'administrateur ramène brutalement en arrière (ce qu'il ne devrait jamais faire en principe), ou le processus qui s'est endormi pendant une durée très longue (plusieurs jours) car on a débranché par mégarde un périphérique de communication, etc.

Nous nous intéresserons en premier lieu à la lecture de l'heure et à la configuration de l'horloge interne. Nous examinerons ensuite les fonctions de conversion à utiliser pour afficher des résultats, puis nous aborderons le problème des fuseaux horaires.

Horodatage et type `time_t`

L'horodatage sous Unix est réalisé à l'aide d'un type de donnée particulier, le type `time_t`. On y stocke le nombre de secondes écoulées depuis le 1^{er} janvier 1970, à 0 heure TU, qu'on considère comme le début de « l'Ère Unix » (*Epoch* en anglais). L'essentiel des datations est accompli à l'aide de ce repère, ce qui rend bien entendu le noyau insensible aux problèmes d'années bissextiles ou de changement de siècle. La norme Iso C9X indique uniquement que le type `time_t` permet des opérations arithmétiques, mais elle ne précise pas qu'il s'agit d'un nombre de secondes. En pratique c'est le cas sur tous les systèmes Unix, mais si on désire vraiment assurer la portabilité d'une manipulation arithmétique horaire, on passera par une conversion intermédiaire en structure `tm` que nous verrons plus loin.

Le type de donnée `time_t` étant exprimé en secondes, il est facile à manipuler car on peut aisément ajouter un délai pour programmer une alarme, sans se soucier du débordement sur la minute, l'heure ou le jour suivant. Traditionnellement, les données `time_t` sont implémentées sous forme d'entiers sur 4 octets. C'est le cas pour l'essentiel des implémentations de Linux sur processeurs 32 bits (sur les architectures 64 bits, `time_t` bénéficie d'un format étendu à 64 bits, ce qui évite les problèmes mentionnés ci-après).

Toutefois, l'appel système `time()` que nous verrons plus loin et qui renvoie une valeur de type `time_t` doit pouvoir échouer s'il détecte une erreur (par exemple si on lui passe un pointeur invalide) et renvoyer, comme c'est l'usage pour les appels système, la valeur -1. Pour cela, le type `time_t` doit donc être signé.

On peut donc gérer des dates jusqu'à un maximum de 0x7FFFFFFF, soit 2 147 483 647 secondes depuis le 1^{er} janvier 1970. Malheureusement, ce nombre

n'est pas aussi énorme qu'il en a l'air. Le mardi 19 janvier 2038 à 3 heures 14 minutes et 7 secondes TU, les compteurs `time_t` 32 bits signés, s'il en reste, basculeront à 0x80000000, soit -2 147 483 648 secondes, et reviendront donc au vendredi 13 décembre 1901, à 20 heures 45 minutes et 52 secondes T.U. !

Bien sûr, cela n'arrivera pas réellement, car d'ici là les noyaux Unix seront mis à jour pour traiter les données `time_t` avec un autre stockage sur 64 bits signés. Le problème qui se pose toutefois est l'interface des applications fonctionnant sur ces systèmes. Car si le noyau modifie la longueur du type `time_t`, cela vaudra également pour la bibliothèque C et les applications qui utilisent les fonctions de lecture d'heure que nous verrons ci-après.

Dans l'immense majorité des cas, une simple recompilation permettra de mettre à niveau le logiciel. Toutefois, le cas des applications disponibles uniquement sous forme binaire posera un problème essentiel, ainsi que pour les systèmes gérant des bases de données dans lesquelles les dates sont stockées, avec `fwrite()` par exemple, de manière binaire dans des fichiers. Il sera nécessaire d'écrire des outils de conversion des bases de données. En outre, il faut prendre en compte les systèmes de fichiers, où les horodatages sont stockés avec un type `time_t` et dont les datations des fichiers sont importantes.

L'an 2038 peut paraître bien éloigné aujourd'hui. Une partie des informaticiens actuels ne seront plus en activité à ce moment-là, donc le problème ne leur semble pas aussi crucial que cela. Pourtant, ce raisonnement est faux pour plusieurs raisons.

- Le vent de panique créé lors du passage à l'an 2000 devrait nous servir de leçon pour savoir qu'on ne peut pas prédire la durée de vie d'une application. Elle peut non seulement être utilisée bien plus longtemps que ce qu'on estimait lors de son écriture, mais cela semble encore plus vrai pour les logiciels dont les sources ne sont pas disponibles.
- Il existera de plus en plus de systèmes embarqués, qu'on trouvera dans les appareils électroménagers, hi-fi, voitures, appareils photographiques numériques... Le logiciel embarqué sera de plus en plus évolué, et une bonne partie sera constituée d'un noyau Linux sur lequel tournera l'application faisant fonctionner le matériel, mais également des outils de communication, pour la programmation, le paramétrage ou l'évolution du logiciel. La durée de vie de ces appareils pourra être très longue, et des applications conçues dans un avenir proche pourront fort bien continuer d'exister dans des équipements fonctionnant toujours en 2038.
- De nombreux programmes n'ont pas besoin d'attendre 2038 pour être confrontés à ce problème. Un logiciel de calcul astronomique peut par exemple être employé pour prévoir des événements 10, 15, 20 ans à l'avance. Il en est de même pour un programme faisant des calculs d'amortissements pour un emprunt sur 20 ou 25 ans. Il devra alors faire face au bogue de 2038 dès l'an 2013. Le délai restant est alors largement diminué.

Pour toutes ces raisons, il est important pour un programmeur applicatif de commencer à se préoccuper de l'utilisation qu'il fait des données `time_t`. Les manipulations internes dans le programme ne posent en fait pas vraiment de problème. Une recompilation du logiciel permettra de prendre en compte la nouvelle longueur lorsqu'il le faudra.

Les difficultés s'annoncent lorsqu'on doit stocker des dates dans un fichier ou les communiquer à un autre système. Dans un cas comme dans l'autre, si on est maître des deux extrémités de la transmission (lecture et écriture du fichier, ou émission et réception des données), on peut employer un subterfuge consistant à transférer les données de type `time_t` dans un entier long `long int`, qui dispose au moins de 64 bits sur tous les systèmes actuels. Ce sera alors cette variable qui sera utilisée pour la transmission ou le stockage. La conversion inverse supprimera les bits supplémentaires, inutilisés à ce moment-là, tant que le type `time_t` n'aura pas évolué.

Si cette solution n'est pas possible, il faut se contenter de bien documenter par des commentaires précis les emplacements où la taille des données `time_t` est prise en compte.

Lecture de l'heure

L'appel système le plus simple pour lire l'heure actuelle est `time()` qui est déclaré dans `<time.h>` ainsi :

```
| time_t time (time_t * heure);
```

Cet appel système renvoie la date et l'heure actuelles, et remplit la variable transmise en argument avec cette même valeur si le pointeur n'est pas `NULL`. Si jamais le pointeur est invalide, `time()` retourne la valeur d'erreur `((time_t) -1)`. Cet appel système est simple, portable, défini par SUSv4 et Ansi C, et nous avons vu que le type de donnée `time_t` est facile à manipuler.

En Python, l'appel de `time.time()` renvoie un nombre réel ajoutant, en partie décimale, l'intervalle de temps écoulé depuis la dernière seconde.

```
| exemple-time.py :  
| #! /usr/bin/python  
  
| from __future__ import print_function  
| import time  
  
| print("Current time is", time.time())
```


Voici un exemple d'exécution :

```
$ ./exemple-time.py
Current time is 1454202899.29
```

Il peut arriver cependant qu'on ait besoin de dater des événements avec une précision plus grande que la seconde. Pour cela, il existe plusieurs appels système fournissant une meilleure résolution. Le premier, hérité des systèmes Unix classiques, est `gettimeofday()`, déclaré dans `<sys/time.h>` avec les types des données qu'il emploie :

```
int gettimeofday (struct timeval * tv, struct timezone * tz);
```

Cette fonction remplit les deux structures sur lesquelles on passe des pointeurs – s'ils ne sont pas `NULL` –, et renvoie 0 si elle réussit et -1 en cas d'erreur. La structure `timeval` contient les deux membres suivants.

Nom	Type	Signification
<code>tv_sec</code>	<code>time_t</code>	Nombre de secondes écoulées depuis le 1 ^{er} janvier 1970.
<code>tv_usec</code>	<code>time_t</code>	Nombre de microsecondes depuis le dernier changement de <code>tv_sec</code> .

Bien entendu, on pourrait construire la fonction `time()` à partir de `gettimeofday()` ainsi :

```
time_t time (time_t * tm)
{
    struct timeval tv;
    gettimeofday(& tv, NULL);
    if (tm != NULL)
        * tm = tv.tv_sec;
    return tv.tv_sec;
}
```

Toutefois, sous Linux, l'implémentation est encore sous forme d'appel système indépendant, ce qui présente par ailleurs l'avantage d'une meilleure vérification de la validité du pointeur transmis.

La structure `timezone` contient deux membres (tableau ci-après).

Nom	Type	Signification
<code>tz_minuteswest</code>	<code>int</code>	Nombre de secondes de décalage vers l'ouest depuis Greenwich.
<code>tz_dsttime</code>	<code>int</code>	Type d'heure hiver/été appliqué localement.

La structure `timezone` est quasi obsolète et ne doit pas être utilisée. Nous verrons à la fin de ce chapitre comment accéder aux informations sur les fuseaux horaires. Le premier membre de `timezone` peut indiquer correctement la bonne valeur, mais le second n'est jamais mis à jour. Dans la plupart des cas, on n'utilisera jamais le second argument de `gettimeofday()`, et on passera donc un pointeur `NULL`.

Voyons donc les comportements de `time()` et de `gettimeofday()` :

```
exemple-gettimeofday-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

int main (void)
{
    struct timeval tv;

    if (gettimeofday(& tv, NULL) != 0) {
        perror("gettimeofday");
        exit(EXIT_FAILURE);
    }
    fprintf (stdout, "time() : %ld \n", time (NULL));
    fprintf (stdout, "gettimeofday() : %ld.%06ld\n",
             tv.tv_sec, tv.tv_usec);
    return 0;
}
```

L'exécution montre bien le même résultat au niveau seconde et une meilleure précision de `gettimeofday()`.

```
$ ./exemple-gettimeofday-1
time() : 1277749148
gettimeofday() : 1277749148.568455
$
```

Réalisons à présent une petite expérience : dans l'exemple suivant nous allons appeler `gettimeofday()` en boucle 20 fois, puis nous afficherons les résultats.

```
exemple-gettimeofday-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
```

```
int main (void)
{
    struct timeval tv[20];
    int i;
    for (i = 0; i < 20; i++)
        gettimeofday(& (tv[i]), NULL);
    for (i = 0; i < 20; i++)
        fprintf (stdout, "%ld.%06ld\n", tv[i].tv_sec, tv[i].tv_usec);
    return 0;
}
```

À l'exécution nous voyons qu'il y a plusieurs invocations de `gettimeofday()` par microseconde :

```
$ ./exemple-gettimeofday-2
1277749174.587307
1277749174.587308
1277749174.587308
1277749174.587309
1277749174.587309
1277749174.587309
1277749174.587310
1277749174.587310
1277749174.587311
1277749174.587311
[...]
```

Ceci varie grandement suivant le type de processeur, la fréquence, la charge du système (préférez l'exécution sur un système au repos), etc. Quoi qu'il en soit, pour la plupart des ordinateurs actuels, l'appel système `gettimeofday()` dure moins d'une microseconde, et sa précision est donc bien de cet ordre.

Ceci varie grandement suivant le type de processeur, la fréquence, la charge du système (préférez l'exécution sur un système au repos), etc. Quoi qu'il en soit, pour la plupart des ordinateurs actuels, l'appel système `gettimeofday()` dure moins d'une microseconde, et sa précision est donc bien de cet ordre. Cet appel système n'a pas d'équivalent direct en Python.

Le second appel système permettant de lire précisément l'heure est déclaré ainsi dans `<time.h>` :

```
int clock_gettime (clockid_t id, struct timespec * ts);
```

Le premier argument est un identifiant d'horloge ; nous préciserons son rôle plus loin. Il existe également deux autres appels système concomitants :

```
int clock_gettime (clockid_t id, const struct timespec * ts);
int clock_getres (clockid_t id, struct timespec * ts);
```

Le premier permet de fixer la valeur de l'horloge mais, en pratique, très peu sont modifiables. Le second permet d'obtenir la résolution de l'horloge concernée.

Il existe huit types d'horloges :

Nom	Signification
CLOCK_REALTIME	L'heure du système. Susceptible de varier de manière discontinue en cas de modification (par l'administrateur par exemple).
CLOCK_REALTIME_COARSE	Une version de CLOCK_REALTIME éventuellement moins précise mais plus rapide à obtenir. On évitera, par exemple, d'interroger un composant externe. (Spécifique Linux)
CLOCK_MONOTONIC	Une horloge synchronisée initialement avec CLOCK_REALTIME mais qui ne subira pas de variations discontinues en cas de modification de l'heure système.
CLOCK_MONOTONIC_COARSE	Une version de CLOCK_MONOTONIC plus rapide à consulter mais peut-être moins précise. (Spécifique Linux)
CLOCK_MONOTONIC_RAW	Une version de CLOCK_MONOTONIC s'appuyant sur un dispositif matériel sans mise à l'heure. (Spécifique Linux)
CLOCK_PROCESS_CPUTIME_ID	Temps précis consommé par le processus appelant.
CLOCK_THREAD_CPUTIME_ID	Temps précis consommé par le thread appelant.
CLOCK_BOOTTIME	Une horloge à la progression monotone depuis le démarrage du système.

Voici un exemple d'utilisation.

```
exemple-clock-gettime.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main (void)
{
    struct timespec ts;

    if (clock_gettime(CLOCK_REALTIME, &ts) != 0)
        perror("CLOCK_REALTIME");
    else
```

```
    printf("CLOCK_REALTIME : %ld.%09ld\n", ts.tv_sec, ts.tv_nsec);

    if (clock_gettime(CLOCK_REALTIME_COARSE, &ts) != 0)
        perror("CLOCK_REALTIME_COARSE");
    else
        printf("CLOCK_REALTIME_COARSE : %ld.%09ld\n", ts.tv_sec,
ts.tv_nsec);

    if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0)
        perror("CLOCK_MONOTONIC");
    else
        printf("CLOCK_MONOTONIC : %ld.%09ld\n", ts.tv_sec, ts.tv_nsec);

    if (clock_gettime(CLOCK_MONOTONIC_COARSE, &ts) != 0)
        perror("CLOCK_MONOTONIC_COARSE");
    else
        printf("CLOCK_MONOTONIC_COARSE : %ld.%09ld\n", ts.tv_sec,
ts.tv_nsec);

    if (clock_gettime(CLOCK_MONOTONIC_RAW, &ts) != 0)
        perror("CLOCK_MONOTONIC_RAW");
    else
        printf("CLOCK_MONOTONIC_RAW : %ld.%09ld\n", ts.tv_sec, ts.tv_nsec);

    if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts) != 0)
        perror("CLOCK_PROCESS_CPUTIME_ID");
    else
        printf("CLOCK_PROCESS_CPUTIME_ID : %ld.%09ld\n", ts.tv_sec,
ts.tv_nsec);

    if (clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts) != 0)
        perror("CLOCK_THREAD_CPUTIME_ID");
    else
        printf("CLOCK_THREAD_CPUTIME_ID : %ld.%09ld\n", ts.tv_sec,
ts.tv_nsec);

    if (clock_gettime(CLOCK_BOOTTIME, &ts) != 0)
        perror("CLOCK_BOOTTIME");
    else
        printf("CLOCK_BOOTTIME : %ld.%09ld\n", ts.tv_sec, ts.tv_nsec);

    return 0;
}
```

L'exécution montre les différentes mesures disponibles :

```
$ ./exemple-clock-gettime
CLOCK_REALTIME : 1454229908.677149988
CLOCK_REALTIME_COARSE : 1454229908.671209165
```

```
CLOCK_MONOTONIC : 256262.286990158
CLOCK_MONOTONIC_COARSE : 256262.281009576
CLOCK_MONOTONIC_RAW : 256262.278301934
CLOCK_PROCESS_CPUTIME_ID : 0.001019002
CLOCK_THREAD_CPUTIME_ID : 0.001024223
CLOCK_BOOTTIME : 256262.287004632
```

Cet appel système est également disponible dans le module `time` de Python à partir de la version 3.3 :

```
exemple-clock-gettime.py :
#!/usr/bin/python3

import time

print("CLOCK_REALTIME : ",
      time.clock_gettime(time.CLOCK_REALTIME))
print("CLOCK_MONOTONIC : ",
      time.clock_gettime(time.CLOCK_MONOTONIC))
print("CLOCK_MONOTONIC_RAW : ",
      time.clock_gettime(time.CLOCK_MONOTONIC_RAW))
print("CLOCK_PROCESS_CPUTIME_ID : ",
      time.clock_gettime(time.CLOCK_PROCESS_CPUTIME_ID))
print("CLOCK_THREAD_CPUTIME_ID : ",
      time.clock_gettime(time.CLOCK_THREAD_CPUTIME_ID))
```

Configuration de l'heure système

Le réglage de l'heure du système est une opération évidemment privilégiée, nécessitant un UID effectif nul ou la capacité `CAP_SYS_TIME`. Il existe trois appels système permettant de modifier l'heure de la machine : `settimeofday()`, qui est un héritage de BSD, `stime()`, qui provient de Système V, et `adjtimex()`, qui est spécifique à Linux. Leurs prototypes sont déclarés respectivement dans `<sys/time.h>`, `<time.h>` et `<sys/timex.h>` ainsi :

```
int settimeofday (const struct timeval * timeval,
                  const struct timezone * timezone);
int stime (time_t * heure);
int adjtimex (struct timex * timex);
```

L'appel système `settimeofday()` fonctionne à l'inverse de `gettimeofday()`, en configurant l'heure et éventuellement le fuseau horaire de la machine.

L'appel `stime()` peut très bien être implémenté à partir de `settimeofday()`, comme nous l'avons observé pour son antagoniste `time()`.

Enfin, `adjtimex()` sert non seulement à régler l'heure de l'horloge interne, mais permet aussi d'organiser des paramètres complexes pour ajuster la régularité de l'horloge et éviter des dérives périodiques. Ce sujet sort largement du cadre de notre étude, et nous laisserons le lecteur que cela intéresse se reporter directement aux sources du noyau, en étudiant les fichiers `kernel/time.c` et `arch/xxx/kernel/time.c`, ou à la *RFC 956*, qui décrit le principe de cet algorithme.

Il est fortement déconseillé d'utiliser directement ces appels système. En effet, l'horloge du noyau doit fonctionner de la manière la plus monotone possible. Modifier brutalement l'heure du système ou, pire encore, la faire revenir en arrière peuvent perturber gravement certains processus qui traitent des données horodatées. Pour configurer l'horloge de la machine, on préférera employer la fonction de bibliothèque `adjtime()`, extension Gnu spécifique déclarée dans :

```
int adjtime (const struct timeval * modification,
             struct timeval * ancienne);
```

Cette fonction prend en premier argument un pointeur sur une structure `timeval` contenant la différence entre l'heure désirée et l'heure actuelle. Cette différence peut notamment être négative, si on désire retarder l'horloge. La bibliothèque C va alors ralentir l'horloge système de manière à rattraper progressivement la valeur voulue. De même, lorsque la différence est positive, l'horloge sera accélérée pour combler peu à peu l'écart. Si le second argument est un pointeur non `NULL`, on y stocke la modification précédemment demandée et qui n'a pas fini d'être appliquée.

Cette fonction est très précieuse par exemple pour synchroniser plusieurs machines d'un réseau local en employant le protocole *NTP* (*Network Time Protocol*, défini dans la *RFC 1305*), ou *PTP* (*Precision Time Protocol*, normalisé sous la référence *IEC 61588*).

Conversions, affichages de dates et d'heures

Pour le moment nous n'avons manipulé la date et l'heure que sous forme de données de type `time_t` (ou de structures `timeval` qui l'encadrent en ajoutant les microsecondes). Nous avons observé que ce type est pratique (l'unité étant la seconde, il est très intuitif), robuste (pas de problème d'années bissextiles ou de changement de siècle), et portable (défini par *SUSv4* et *Ansi C*). Toutefois, malgré tous ces avantages, on arrive difficilement à faire comprendre à l'utilisateur que 1277749050 est plus commode que 28 juin 2010 à 20 heures 17 minutes et 30 secondes. Il faut donc trouver le moyen de

convertir les secondes du type `time_t` en éléments plus lisibles par un utilisateur moyen. La bibliothèque C nous fournit plusieurs routines de traduction.

Tout d'abord, il existe une structure de données permettant de représenter la date et l'heure sous forme intelligible. La structure `tm` est définie par le standard Ansi C et contient les membres suivants, qui sont tous de type `int`.

Nom	Signification
<code>tm_sec</code>	Nombre de secondes écoulées depuis le dernier changement de minute, dans l'intervalle 0 à 60.
<code>tm_min</code>	Nombre de minutes écoulées depuis le dernier changement d'heure, entre 0 et 59.
<code>tm_hour</code>	Nombre d'heures écoulées depuis minuit, dans l'intervalle 0 à 23.
<code>tm_mday</code>	Jour du mois, allant de 1 à 31.
<code>tm_mon</code>	Nombre de mois écoulés depuis le début de l'année, dans l'intervalle 0 à 11.
<code>tm_year</code>	Nombre d'années écoulées depuis 1900.
<code>tm_wday</code>	Nombre de jours écoulés depuis dimanche dans l'intervalle 0 à 6.
<code>tm_yday</code>	Nombre de jours écoulés depuis le 1er janvier, dans l'intervalle 0 à 365.
<code>tm_isdst</code>	Indicateur d'horaire d'été ou d'hiver.

Plusieurs points appellent des commentaires dans ce tableau.

- Les secondes peuvent aller de 0 à 60, car il existe parfois des secondes de rattrapage périodique, définies par les instances astronomiques internationales. Une minute officielle peut donc durer 59, 60 ou 61 secondes. En réalité, les fonctions de la bibliothèque C ne renvoient jamais de valeur supérieure à 59, comme cela est demandé par SUSv4 (elles arrondissent au besoin à la minute supérieure). Par contre, on peut légitimement remplir le champ `tm_sec` avec une valeur allant jusqu'à 60 en entrée des fonctions de la Glibc. Un programme portable devra donc être prêt à traiter ce cas, par exemple s'il utilise les secondes comme index dans un tableau de statistiques. Il faudra alors soit prévoir 61 emplacements, soit utiliser une astuce comme `index=(tm.tm_sec % 60)` ou `index=(tm.tm_sec<60? tm.tm_sec: 59)`.
- Le jour du mois commence à 1 et n'est donc pas directement utilisable comme index dans une table, mais il peut être affiché. Par contre, le numéro du mois débute à zéro. Il faut lui ajouter 1 pour l'affichage.
- Le membre `tm_year` indique le nombre d'années écoulées depuis 1900. L'an 2000 est donc représenté par un 100. Pour afficher l'année sur deux chiffres, on emploiera donc `(tm_year % 100)`. Ceci ne pose plus de problème pour les nouvelles applications puisque en cas d'erreur le problème apparaîtra dès les premiers tests avec par exemple un affichage 25/12/101. Par contre, de nombreux logiciels conçus jusqu'en 1999 peuvent souffrir d'un défaut d'attention du programmeur face à cette caractéristique.

- La semaine commence, à l'anglaise, le dimanche et pas le lundi. Le champ `tm_wday` va de 0 à 6, pouvant servir d'index dans un tableau initialisé ainsi :
`char * jours[7]={"D", "L", "Ma", "Me", "J", "V", "S"};`
- Le membre `tm_isdst` a une valeur positive si l'horaire d'été est en vigueur, nulle si l'horaire normal (hiver) fonctionne, et négative si cette information n'est pas disponible.

La bibliothèque Glibc ajoute également deux autres membres `tm_gmtoff` et `tm_zone`, qui correspondent respectivement au nombre de secondes qu'il faut ajouter à la date indiquée pour obtenir le temps TU, et au nom (sous forme de chaîne de caractères statique) du fuseau horaire employé. Ces deux champs ne sont pas standard et nous ne les traiterons pas ici.

Les routines de conversion de format de date renvoient traditionnellement des pointeurs sur des zones de mémoire allouées statiquement. Ces données sont donc écrasées à chaque nouvel appel de la même fonction. Ceci rend impossible leur utilisation dans un contexte multithread. Aussi la bibliothèque Glibc inclut-elle des extensions *Unix 98* avec le suffixe `_r` pour définir une version réentrante de chacune de ces routines.

Pour convertir une valeur de type `time_t` en structure `tm`, il existe deux fonctions, `localtime()` et `gmtime()`, et leurs homologues réentrantes :

```
struct tm * localtime (const time_t * date);
struct tm * localtime_r (const time_t * date, struct tm * tm);
struct tm * gmtime (const time_t * date);
struct tm * gmtime_r (const time_t * date, struct tm * tm);
```

Bien entendu, les deux premières routines renvoient l'heure locale, en se fondant sur la configuration des fuseaux horaires que nous verrons plus bas, alors que les deux dernières retournent l'heure TU.

En Python, les fonctions `time.localtime()` et `time.gmtime()` renvoient un objet `struct_time` dont les attributs sont identiques aux champs de la structure décrite précédemment. Une différence toutefois : le champ `tm_mon` prend ses valeurs entre 1 et 12 et non plus entre 0 et 11.

Voici un exemple d'emploi de ces routines :

```
exemple-localtime-1.c :
#include <stdio.h>
#include <time.h>

int main (void)
{
    time_t      temps;
    struct tm * tm;
```

```

time(& temps);
printf("time() = %ld \n", temps);
tm = localtime(& temps);
printf("localtime() = %02d/%02d/%02d - %02d:%02d:%02d %s\n",
       tm->tm_mday, tm->tm_mon + 1, tm->tm_year % 100,
       tm->tm_hour, tm->tm_min, tm->tm_sec,
       tm->tm_isdst>0 ? "Été":tm->tm_isdst==0 ? "Normal":"?");
tm = gmtime(& temps);
printf("gmtime()      = %02d/%02d/%02d - %02d:%02d:%02d %s\n",
       tm->tm_mday, tm->tm_mon + 1, tm->tm_year % 100,
       tm->tm_hour, tm->tm_min, tm->tm_sec,
       tm->tm_isdst>0 ? "Été":tm->tm_isdst==0 ? "Normal":"?");
return EXIT_SUCCESS;
}

```

En Python :

```

exemple-localtime.py :
#!/usr/bin/python

from __future__ import print_function
import time

t = time.time()
print("time.time()      =", t)

tm = time.localtime(t)
if (tm.tm_isdst > 0):
    dst = "Ete"
elif (tm.tm_isdst == 0):
    dst = "Normal"
else:
    dst = "?"
print("time.localtime() = {:02}/{:02}/{:02} {:02}:{:02}:{:02}
{}".format(
    tm.tm_mday, tm.tm_mon, tm.tm_year % 100,
    tm.tm_hour, tm.tm_min, tm.tm_sec, dst))

tm = time.gmtime(t)
if (tm.tm_isdst > 0):
    dst = "Ete"
elif (tm.tm_isdst == 0):
    dst = "Normal"
else:
    dst = "?"
print("time.gmtime()    = {:02}/{:02}/{:02} {:02}:{:02}:{:02}
{}".format(
    tm.tm_mday, tm.tm_mon, tm.tm_year % 100,
    tm.tm_hour, tm.tm_min, tm.tm_sec, dst))

```

Les exécutions suivantes du programme ont lieu dans le fuseau horaire de Paris :

```
$ ./exemple-localtime
time() = 1277749270
localtime() = 28/06/10 - 20:21:10 Été
gmtime()    = 28/06/10 - 18:21:10 Normal
$
```

L'horaire d'été est bien détecté, voyons l'horaire d'hiver, après ajout de 150 jours au résultat de `time()` :

```
$ ./exemple-localtime
time() = 1290709356
localtime() = 25/11/10 - 19:22:36 Normal
gmtime()    = 25/11/10 - 18:22:36 Normal
$
```

La traduction inverse est possible, grâce à la fonction `mktime()` (`time.mktime()` en Python) :

```
| time_t mktime (struct tm * tm);
```

Cette routine peut renvoyer `(time_t)-1` en cas d'erreur, mais elle essaye toutefois d'être la plus robuste possible. Elle ignore les membres `tm_yday` et `tm_wday` de la structure `tm` transmise, elle les recalcule grâce aux autres données et les remet à jour. Si un membre a une valeur invalide, la fonction `mktime()` calcule son débordement. Par exemple, 23h70 est corrigé pour correspondre à 0h10 du jour suivant.

On peut bien entendu utiliser une fonction de la famille `printf()` pour présenter le contenu d'une structure `tm`, comme nous l'avons fait précédemment, mais lorsqu'on désire afficher la date uniquement à titre informatif pour l'utilisateur, il est souvent plus simple d'utiliser l'une des fonctions `asctime()` et `ctime()`, qui renvoient des chaînes de caractères statiques, ou leurs homologues `asctime_r()` et `ctime_r()`, qui utilisent un buffer passé en argument, pouvant contenir au minimum 16 caractères.

```
| char * asctime (const struct tm * tm);
| char * asctime_r (const struct tm * tm, char * buffer);
| char * ctime (const time_t * date);
| char * ctime_r (const time_t * date, char * buffer);
```

La fonction `ctime()` est l'équivalent de `asctime(localtime (date))`. Le résultat de ces fonctions est une chaîne de caractères contenant :

- le jour de la semaine, parmi les abréviations Mon, Tue, Wed, Thu, Fri, Sat, Sun ;
- le nom du mois parmi Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec ;
- le numéro du jour dans le mois ;
- l'heure, les minutes et les secondes ;
- l'année sur quatre chiffres ;
- un caractère `'\n'` de retour à la ligne.

En voici une illustration très simple.

```
exemple-ctime.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
    time_t t;
    t = time(NULL);
    fprintf(stdout, "%s", ctime(& t));
    return EXIT_SUCCESS;
}
```

Dans la version en Python, on notera que `time.ctime()` n'ajoute pas de retour chariot final ; c'est la fonction `print()` qui le fait.

```
exemple-ctime.py :
#!/usr/bin/python

from __future__ import print_function
import time

t = time.time()
print(time.ctime(t))
```

```
$ ./exemple-ctime
Mon Jun 28 20:23:33 2010
$
```

Nous voyons qu'avec `ctime()` ou `asctime()` le format d'affichage est figé. De plus, le nom des jours et des mois est en anglais. Ces routines ne sont pas sensibles à la localisation du processus. Pour pallier ces problèmes, la bibliothèque C propose une rou-

tine définie par SUSv4, `strftime()`, très puissante mais légèrement plus compliquée puisqu'elle fonctionne un peu sur le principe de la famille `printf()`.

```
size_t strftime (char * buffer, size_t longueur,
                 const char * format, const struct tm * tm);
```

Cette fonction remplit le buffer passé en premier argument, dont la taille est indiquée en second argument. Si ce buffer est trop court, `strftime()` renvoie 0. Sinon, elle transmet le nombre de caractères écrits, sans compter le ‘\0’ final. Le contenu du buffer est constitué des champs de la structure `tm` indiqués dans la chaîne de format passée en troisième argument. La syntaxe de la commande équivalente en Python est plus simple :

```
time.strftime(format, tm)
```

Si `tm` n’est pas fourni, c’est l’heure locale actuelle qui est prise en considération.

Code	Signification
%%	Le caractère %.
%A	Le nom complet du jour de la semaine.
%a	Le nom abrégé du jour de la semaine.
%B	Le nom complet du mois.
%b	Le nom abrégé du mois.
%C	Le siècle (19 pour 1977, 20 pour 2015).
%c	La date et l’heure dans la représentation locale usuelle.
%D	La date, dans le format %m/%d/%y.
%d	Le jour du mois dans l’intervalle 1 à 31.
%e	Le jour du mois dans l’intervalle 1 à 31, précédé par un blanc pour les valeurs inférieures à 10, afin de permettre un alignement à droite.
%F	La date, dans le format %Y-%m-%d. Ce format est défini par la norme ISO 8601, il deviendra probablement de plus en plus répandu dans l’avenir.
%g	Le numéro de l’année sur deux chiffres, correspondant à la semaine en cours. Il peut y avoir une différence avec %y pour les premiers ou derniers jours de l’année.
%G	Comme %g, mais sur quatre chiffres.
%h	Comme %b.
%H	L’heure sur 24 heures et sur deux chiffres, de 00 à 23.
%I	L’heure sur 12 heures et sur deux chiffres, de 00 à 11.
%j	Le numéro du jour de l’année sur trois chiffres, de 001 à 366.

Code	Signification
%k	L'heure sur 24 heures, mais avec une espace devant les valeurs inférieures à 10, allant donc de 0 à 23. Extension Gnu.
%l	Comme %k, mais sur 12 heures. Extension Gnu.
%M	La minute sur deux chiffres, de 00 à 59.
%m	Le numéro du mois, sur deux chiffres, de 01 à 12.
%n	Un caractère '\n' de retour à la ligne.
%P	Comme %p, mais en majuscules. Extension Gnu.
%p	L'équivalent local des chaînes «am» ou «pm» de l'heure sur 12 heures. Minuit est considéré comme 0h am et midi comme 0h pm.
%R	L'heure et la minute au format %H:%M. Extension Gnu.
%r	L'heure complète, sur 12 heures, y compris les équivalents locaux de am et pm.
%S	Les secondes sur deux chiffres, de 00 à 60.
%s	Le nombre de secondes écoulées depuis le 1 ^{er} janvier 1970 à 0 heure TU. Extension Gnu.
%T	L'heure au format %H:%M:%S.
%t	Le caractère '\t' de tabulation.
%U	Le numéro de la semaine dans l'année, allant de 00 à 53. La semaine numéro 1 de l'année commence au premier dimanche. Les jours précédant ce dimanche sont dans la semaine 0.
%u	Le numéro du jour dans la semaine, de 1 à 7, avec 1 correspondant au lundi.
%V	Le numéro de la semaine dans l'année de 1 à 53. La référence prise ici commence au premier lundi de l'année, comme le précise la norme <i>ISO 8601</i> .
%W	Comme %V, mais de 0 à 53, les jours précédant le premier lundi étant dans la semaine 0.
%w	Le jour de la semaine de 0 à 6, en commençant le dimanche.
%X	La représentation locale usuelle de l'heure.
%x	La représentation locale usuelle de la date. Le compilateur nous avertit lorsqu'on utilise %x dans une chaîne constante que cette représentation se fait avec des années sur deux chiffres dans certaines localisations. Cet avertissement peut être ignoré si on est conscient de ce fait.
%Y	L'année sous forme de nombre décimal complet.
%y	L'année sur deux chiffres, sans le siècle.
%Z	Le nom du fuseau horaire abrégé, éventuellement vide.
%z	Le fuseau horaire indiqué sous forme numérique conforme à la <i>RFC 822</i> .

Lorsqu'on transmet un buffer NULL, `strftime()` nous indique le nombre de caractères qu'elle aurait dû écrire dedans. Ceci est très utile car, dans certaines situations, cette fonction peut renvoyer légitimement 0 alors que le buffer est bien assez grand. C'est le cas par exemple si on demande uniquement le code %p alors que la localisation permet seulement l'emploi du temps sur 24 heures. Nous allons employer cette méthode dans le programme suivant.

```
exemple-strftime.c :
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (int argc, char * argv[])
{
    int      i;
    int      lg;
    char *    buffer;
    struct tm * tm;
    time_t    heure;

    setlocale(LC_ALL, "");

    time(& heure);
    tm = localtime(& heure);

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : ", argv[i]);
        lg = strftime(NULL, SSIZE_MAX, argv[i], tm);
        if (lg > 0) {
            buffer = malloc(lg + 1);
            if (buffer == NULL) {
                perror("malloc");
                exit(EXIT_FAILURE);
            }
            strftime(buffer, lg + 1, argv[i], tm);
            fprintf(stdout, "%s", buffer);
            free(buffer);
        }
        fprintf(stdout, "\n");
    }
    return EXIT_SUCCESS;
}
```

L'équivalent en Python est plus concis :

```
exemple-strftime.py :
#!/usr/bin/python
from __future__ import print_function
import locale
import sys
import time

locale.setlocale(locale.LC_ALL, '')
sys.argv.pop(0)
```

```
for arg in sys.argv:  
    print(time.strftime(arg))
```

Ce programme permet d'afficher la date et l'heure courantes avec le format transmis en argument de la ligne de commande. En voici quelques exemples.

```
$ ./exemple-strftime "Le %d %B %Y, à %H heures %M"  
Le %d %B %Y, à %H heures %M : Le 28 juin 2010, à 20 heures 24  
$ ./exemple-strftime "%p"  
%p :  
$ ./exemple-strftime "%Z (%z)"  
%Z (%z) : CEST (+0200)  
$
```

Nous remarquons que dans la localisation française, le code %p (AM/PM) n'a pas de signification.

Bien entendu, la bibliothèque Glibc propose des fonctions permettant le cheminement inverse, c'est-à-dire la création d'une structure `tm` à partir d'une chaîne de caractères qui peut avoir été saisie par l'utilisateur. Deux fonctions existent, `strptime()` et `getdate()`, toutes deux déclarées dans `<time.h>` et définies par SUSv4.

```
char * strptime (const char * chaine_lue, const char * format,  
                 struct tm * tm);
```

Cette routine fonctionne un peu comme `sscanf()`. Elle examine le contenu de la chaîne transmise en premier argument, à la lumière du format précisé en second argument. Le résultat est alors stocké dans la structure `tm`, puis renvoie un pointeur sur le premier caractère de la chaîne initiale qui n'a pas été converti.

La mise en correspondance entre la chaîne lue et le format est faite octet par octet, chaque caractère du format devant avoir un équivalent dans la chaîne, sinon la lecture s'arrête. Bien entendu, des codes spéciaux identiques à ceux de `strftime()` peuvent être insérés pour lire les champs de la structure `tm`. Les codes étant les mêmes, la fonction `strptime()` est donc symétrique à `strftime()`, et peut aussi bien être employée pour relire des données écrites par un programme que pour lire une saisie humaine.

Si toute la chaîne a pu être analysée, le pointeur transmis correspond au caractère nul final, `'\0'`. Par contre, si aucune conversion n'a pu avoir lieu, le pointeur est `NULL`. Il faut donc systématiquement vérifier cette condition avant d'essayer de consulter le contenu du pointeur, sous peine de déclencher une erreur `SIGSEGV`.

La bibliothèque Glibc met à jour uniquement les champs de la structure `tm` qui ont été lus, ainsi que les champs `tm_wday` et `tm_yday`. Les autres membres ne sont pas modifiés. Pour vérifier le résultat de la fonction, il est donc conseillé d'initialiser tous les membres avec une valeur impossible, comme `-1` ou `INT_MAX`. Cela permettra de s'assurer de la réussite de la conversion. Si on veut éviter cette étape, on peut éventuellement initialiser tous les membres avec des zéros, ainsi la structure aura toujours un contenu cohérent.

La version de `time.strptime()` proposée par Python est légèrement différente : elle renvoie toujours une structure complète dont les champs non renseignés sont initialisés avec une date par défaut, à savoir le 1^{er} janvier 1900 à minuit.

Pour que `strptime()` soit déclarée dans `<time.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` avant l'inclusion de cet en-tête.

Le programme suivant va lire la ou les chaînes de formats successifs sur sa ligne de commande, en afficher un exemple sur `stdout`, et demander à l'utilisateur une saisie sur `stdin`. La même structure `tm` sera utilisée tout au long des saisies. Ensuite, le résultat sera affiché au complet.

```
exemple-strptime.c :
#define _XOPEN_SOURCE
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main (int argc, char * argv[])
{
    int          i;
    int          lg;
    time_t       heure;
    struct tm    tm;
    struct tm *  exemple;
    char *       buffer;
    char *       retour;

    setlocale(LC_ALL, "");
    time(& heure);
    exemple = localtime(& heure);
    memset(& tm, 0, sizeof(struct tm));

    for (i = 1; i < argc ; i ++) {
        lg = strptime(NULL, SSIZE_MAX, argv[i], exemple);
        if (lg > 0) {
            /* On alloue 2 octets de plus pour \n et \0 */
            buffer = malloc(lg + 2); /* retour à vérifier...*/
```

```

    strftime(buffer, lg + 2, argv[i], exemple);
    fprintf(stdout, "Format %s (exemple %s) : ",
            argv[i], buffer);
    while (1) {
        fgets(buffer, lg + 2, stdin);
        retour= strftime(buffer, argv[i], & tm);
        if (retour == NULL)
            fprintf(stdout, "Erreur > ");
        else
            break;
    }
    free(buffer);
}
}
puts(asctime(& tm));
return EXIT_SUCCESS;
}

```

Voici un exemple d'exécution simple, mais intéressant à plusieurs égards :

```

$ ./exemple-strptime "Le %x" "à %X"
Format Le %x (exemple Le 28/06/2010) : Le 4/07/1967
Format à %X (exemple à 20:26:13) : à 4:20:0
Tue Jul 4 04:20:00 1967

$ ./exemple_strptime %F %r
Format %F (exemple 2010-06-28) : 2222-2-2
Format %r (exemple 08:27:11 ) : 20:12:10
Erreur > 20:10
Erreur > 08:10:10
Erreur > 08:10:10 PM
Erreur > (Contrôle-C)
$

```

Nous remarquons plusieurs choses.

- Les versions précédentes de la bibliothèque C ne savaient pas calculer les jours pour les dates antérieures au 1^{er} janvier 1970. Ce défaut est de nos jours corrigé.
- le format %r pose des problèmes car, dans le cas d'une localisation française, la chaîne AM/PM est indéfinie. Lors d'un affichage avec strftime(), tout est masqué car une chaîne vide est affichée, mais lors d'une écriture, la mise en correspondance n'est pas possible. Ceci peut engendrer de sérieux problèmes, qui n'apparaîtront que lors de l'exportation d'une application.
- on peut également regretter l'absence de code d'erreur indiquant le type de problème qui s'est présenté.

La bibliothèque C met donc à notre disposition la fonction `getdate()` et sa correspondante réentrante `getdate_r()`, qui peuvent simplifier la lecture des chaînes contenant des données d'horodatage. Ces deux routines n'existent pas en Python.

```
struct tm * getdate (const char * chaine_lue);  
int getdate_err;  
int getdate_r (const char * chaine_lue, struct tm * tm);
```

La fonction `getdate()` analyse la chaîne transmise et renvoie un pointeur vers une structure `tm` statique représentant la date obtenue. En cas d'erreur, elle retourne un pointeur `NULL` et positionne la variable globale `getdate_err` avec un code d'erreur détaillé ci-après. La fonction `getdate_r()` n'emploie pas de structure statique, mais utilise le pointeur passé en second argument. En conséquence, elle renvoie un code de retour signalant les conditions d'erreur, mais n'utilise pas la variable `getdate_err`.

Pour réaliser l'analyse de la chaîne, ces routines utilisent la variable d'environnement `DATMSK`. Celle-ci doit contenir le chemin d'accès et le nom d'un fichier comprenant des motifs de conversion identiques à ceux qui sont employés par `strptime()`. Chaque motif possible est présenté sur une ligne du fichier, et ils sont essayés successivement jusqu'à ce que l'un d'eux soit correct. L'utilisateur a donc la possibilité de configurer le format de la conversion en fonction de ses habitudes (ou du logiciel employé pour fournir les données d'entrée alimentant la routine `getdate()` concernée).

En contrepartie, la possibilité d'indiquer soi-même le fichier contenant les motifs à utiliser peut constituer une faille de sécurité dans un programme *Set-UID* (car on peut alors consulter n'importe quel fichier du système, y compris ceux réservés à l'administrateur). Dans un tel cas, on évitera d'employer `getdate()` ou on figera lors de la compilation le contenu de la variable d'environnement `DATMSK`, en utilisant la routine `setenv()` étudiée au chapitre 4.

Les codes d'erreur transmis par `getdate()` dans `getdate_err` ou renvoyés par `getdate_r()` sont les suivants.

Valeur	Signification
0	Pas d'erreur.
1	Variable <code>DATMSK</code> non configurée ou contenant une chaîne vide.
2	Le fichier de motifs indiqué dans <code>DATMSK</code> ne peut pas être ouvert.
3	L'état du fichier de motifs n'est pas accessible.
4	Le fichier de motifs n'est pas un fichier régulier.
5	Impossible de lire le contenu du fichier de motifs.
6	Pas assez de mémoire disponible.
7	Impossible de trouver un motif permettant de réaliser une conversion correcte.
8	La chaîne contient des données invalides après conversion (par exemple 31 avril).

Pour que les prototypes de ces routines soient présents dans `<time.h>`, il faut remplir la constante symbolique `_XOPEN_SOURCE` avec la constante 500 avant d'inclure ce fichier d'en-tête.

Les membres de la structure `tm` qui ne sont pas renseignés par la chaîne fournie sont initialisés avec la date et l'heure de l'appel de la routine. Voici un programme qui emploie `getdate()` pour analyser les chaînes transmises en ligne de commande.

```
exemple-getdate.c :
#define _XOPEN_SOURCE 500
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

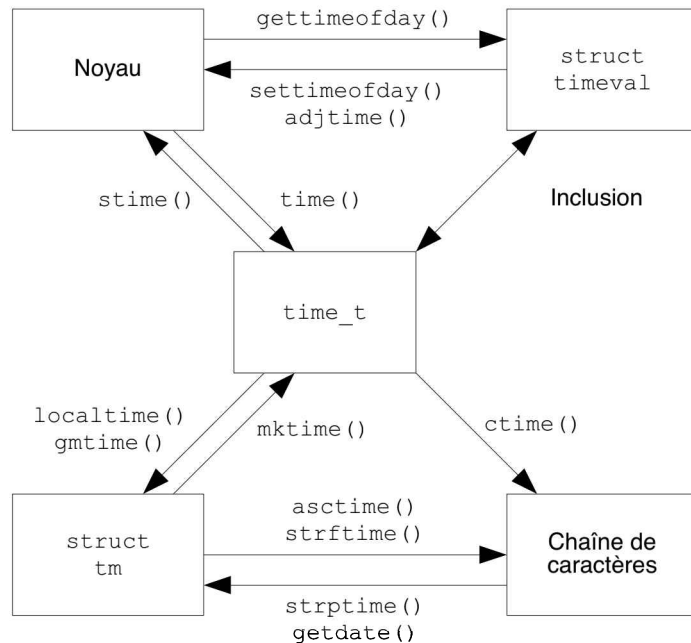
int main (int argc, char * argv[])
{
    struct tm * tm;
    int      i;

    for (i = 1; i < argc; i ++) {
        fprintf(stdout, "%s : ", argv[i]);
        tm = getdate(argv[i]);
        if (tm == NULL)
            switch (getdate_err) {
                case 1 :
                    fprintf(stdout, "DATEMSK indéfinie \n");
                    break;
                case 2 :
                case 3 :
                case 4 :
                case 5 :
                    fprintf(stdout, "Fichier de motifs invalide \n");
                    break;
                case 6 :
                    fprintf(stdout, "Pas assez de mémoire \n");
                    break;
                case 7 :
                    fprintf(stdout, "Conversion impossible \n");
                    break;
                case 8 :
                    fprintf(stdout, "Valeur invalide \n");
                    break;
            }
        else
            fprintf(stdout, "%s", asctime(tm));
    }
    return EXIT_SUCCESS;
}
```

Nous créons un fichier de motifs nommé `datemsk.txt`, qui contient plusieurs conversions possibles sur le thème de la date et de l'heure. Voici un exemple de quelques exécutions.

```
$ cat datemsk.txt
%F %H:%M:%S
%F %H:%M
%F
$ ./exemple-getdate 2010-06-28
2010-06-28 : DATEMSK indéfinie
$ export DATEMSK=datemsk.txt
$ ./exemple-getdate 2010-06-28
2010-06-28 : Mon Jun 28 20:30:41 2010
$ ./exemple-getdate "2010-06-28 05"
2010-06-28 05 : Conversion impossible
$ ./exemple-getdate "2010-06-28 05:06"
2010-06-28 05:06 : Mon Jun 28 05:06:00 2010
$ ./exemple-getdate "2010-06-28 05:06:07"
2010-06-28 05:06:07 : Mon Jun 28 05:06:07 2010
$ ./exemple-getdate "2010-04-31"
2010-04-31 : Valeur invalide
$
```

Figure 8–1
Conversions de données
d'horodatage



La possibilité de modifier soi-même le format des conversions est une fonctionnalité très puissante. Indiquons toutefois que `getdate()`, ou plutôt la fonction `strptime()` sous-jacente, n'est pas très robuste vis-à-vis de la localisation, et que l'application risque parfois de planter sur une faute de segmentation, notamment en employant les codes `%r`, `%X` ou `%x`.

La figure 8-1 récapitule toutes les routines de conversion que nous avons vues afin de passer d'une représentation d'une date à une autre.

Calcul d'intervalles

Les fonctions que nous avons étudiées permettent de travailler sur un instant précis. Pourtant, il est parfois indispensable de travailler sur des durées, sur des intervalles. Nous pouvons par exemple avoir besoin d'ajouter à l'heure actuelle un délai maximal de réaction afin de programmer une alarme.

La structure `timeval` que nous avons déjà rencontrée est utilisée pour enregistrer les durées, avec les membres suivants (tableau ci-après).

Type	Nom	Signification
<code>time_t</code>	<code>tv_sec</code>	Nombre de secondes.
<code>time_t</code>	<code>tv_usec</code>	Nombre de microsecondes.

La bibliothèque Glibc offre quelques fonctions d'assistance pour manipuler les structures `timeval`. Comme le champ `tv_usec` d'une telle structure doit toujours être compris entre 0 et 999.999, il n'est pas facile d'ajouter ou de soustraire ces données. Les fonctions d'aide sont les suivantes :

```
void timerclear (struct timeval * temporisation);
```

qui met à zéro les deux champs de la structure transmise.

```
void timeradd (const struct timeval * duree_1,  
               const struct timeval * duree_2,  
               struct timeval * duree_resultat);
```

additionne les deux structures (en s'assurant que les membres `tv_usec` ne dépassent pas 999.999) et remplit les champs de la structure résultat, sur laquelle on passe un pointeur en dernier argument. Une structure utilisée en premier ou second argument peut aussi servir pour récupérer le résultat, la bibliothèque C réalisant correctement la copie des données.

```
void timersub (const struct timeval * duree_1,  
               const struct timeval * duree_2,  
               struct timeval * duree_resultat);
```

soustrait la deuxième structure de la première (en s'assurant que les membres `tv_usec` ne deviennent pas négatifs) et remplit les champs de la structure résultat.

```
int timerisset (const struct timeval * temporisation);
```

est vraie si au moins l'un des deux membres de la structure est non nul.

Nous avons présenté ici des prototypes de fonctions, mais en réalité elles sont toutes les quatre implémentées sous forme de macros, qui évaluent plusieurs fois leurs arguments. Il faut donc prendre les précautions adéquates pour éviter les effets de bord.

Le type `time_t` étant signé et contenant des secondes, il est possible d'ajouter ou de soustraire des durées sans problème. Toutefois, si ce type de donnée devait évoluer, comme nous l'avons évoqué au début de ce chapitre, certaines soustractions seraient peut-être invalidées. Pour éviter ce genre de problème, on peut utiliser la fonction `difftime()`, qui permet de manière portable de calculer l'intervalle entre deux instants donnés :

```
double difftime (time_t instant_final, time_t instant_initial);
```

Le type de retour de cette fonction étant un double, nous sommes assurés qu'elle pourra gérer sans difficulté d'éventuelles extensions futures du type `time_t`.

Pour réaliser ce type d'opération en Python, on se reportera plutôt à son module `datetime`, en particulier à la fonction `timedelta()`.

Fuseau horaire

Un système Unix en général et Linux en particulier est fondé sur la notion de réseau, de fonctionnement multi-utilisateur et de connexion à Internet. Une machine donnée doit pouvoir accepter simultanément des utilisateurs provenant de plusieurs continents, résidant dans des fuseaux horaires totalement différents.

Traditionnellement, les stations Unix utilisent une horloge interne fonctionnant sur la référence TU. Il est donc préférable autant que possible de laisser l'horloge *CMOS* du PC travailler avec l'heure GMT. L'administrateur de la machine la configure alors pour indiquer dans quel fuseau horaire elle est installée physiquement, et le noyau peut ainsi horodater ses messages, par exemple avec l'heure locale. Malheureuse-

ment, pour cause de cohabitation avec d'autres systèmes d'exploitation moins performants, il est souvent nécessaire de laisser l'horloge interne tourner sur la référence locale. Les distributions Linux permettent de gérer ce type de désagrément.

De son côté, un utilisateur peut se connecter sur la machine depuis n'importe quel endroit du monde, et il est normal que le système lui fournisse des informations temporelles adaptées à son environnement. Ceci concerne bien entendu le résultat de la commande `date`, mais également les horodatages des fichiers affichés par `ls`, ou les informations contenues dans l'en-tête des messages électroniques.

Pour simplifier la tâche de l'utilisateur, une seule variable d'environnement sert pour toutes ces opérations : `TZ`. Celle-ci doit contenir le nom du fuseau horaire où se trouve l'utilisateur, et toutes les informations de dates et d'heures seront mises à jour automatiquement au moment de l'affichage.

En fait, les fonctions `localtime()`, `mktime()`, `ctime()` ou `strftime()` appellent automatiquement la routine `tzset()`, qui sert à initialiser les données correspondant à l'emplacement de l'utilisateur :

```
| void tzset (void);
```

Il n'y a normalement pas de raison de faire appel directement à cette routine puisqu'elle est invoquée par toute fonction prenant en compte la position horaire. `tzset()` configure également deux chaînes de caractères globales :

```
| char * tzname [2];
```

La chaîne `tzname[0]` contient le nom du fuseau horaire, déterminé depuis la variable d'environnement `TZ`. La chaîne `tzname[1]` comprend le nom de ce fuseau lorsqu'on bascule en heure d'été. En Python, la variable `time.tzname` est un tuple de ces deux chaînes, initialisé par `time.tzset()`.

La variable d'environnement `TZ` peut être remplie avec plusieurs champs successifs, seul le premier étant obligatoire.

- 1 Un nom de fuseau horaire, sur trois caractères au minimum.
- 2 Un décalage qui indique la valeur à ajouter à l'heure TU pour obtenir l'heure locale. La valeur est positive à l'ouest de Greenwich.
- 3 Le nom du fuseau à utiliser pour l'heure d'été.
- 4 Le décalage pour l'heure d'été.
- 5 La date de début de l'heure d'été, indiquée sous l'une des formes suivantes : un J suivi du numéro du jour dans l'année, sans compter le 29 février, même pour les années bissextiles, un simple numéro de jour, comptant éventuellement le

29 février, ou un M suivi du numéro du mois, puis d'un point, du numéro de la semaine, d'un point, et du numéro du jour de la semaine (le 0 étant le dimanche).

6 La date de fin de l'heure d'été.

Les informations concernant les fuseaux horaires préprogrammés sont stockées dans les répertoires `/usr/lib/zoneinfo` ou `/usr/share/zoneinfo` suivant les distributions Linux.

Le programme suivant va afficher quelques exemples de configuration :

```
exemple-tzname.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (void)
{
    tzset();
    fprintf(stdout, "tzname[0] = %s\n", tzname[0]);
    fprintf(stdout, "tzname[1] = %s\n", tzname[1]);
    return EXIT_SUCCESS;
}
```

Le script Python équivalent est le suivant.

```
exemple-tzname.py :
#!/usr/bin/python
from __future__ import print_function
import time

time.tzset()
(nodst, dst) = time.tzname
print("no DST =", nodst)
print("    DST =", dst)
```

Nous exécutons le programme qui suit en utilisant d'abord le fuseau horaire de Paris (CET), puis celui de Montréal.

```
$ date
lun. juin 28 20:37:08 CEST 2010
$ ./exemple-tzname
tzname[0] = CET
tzname[1] = CEST
$ export TZ=EST
```

```
$ date
lun. juin 28 13:37:15 EST 2010
$ ./exemple-tzname
tzname[0] = EST
tzname[1] = EST
$
```

Nous pouvons faire un essai en inventant notre propre fuseau horaire.

```
$ export TZ=""
$ date
lun. juin 28 18:37:33 UTC 2010
$ ./exemple-tzname
tzname[0] = UTC
tzname[1] = UTC
$ export TZ="RIEN -1:12"
$ ./exemple-tzname
tzname[0] = RIEN
tzname[1] =
$
```

Les possibilités de configuration du fuseau horaire utilisateur par utilisateur (et même session par session) offrent des perspectives très intéressantes en ce qui concerne les fonctionnalités de communications internationales, aussi bien pour le courrier électronique que pour les groupes de discussion, en permettant à chacun d'obtenir des informations temporelles intelligibles dans son propre environnement, sans avoir à s'interroger sur la position précise de ses interlocuteurs.

Conclusion

Nous avons examiné en détail l'essentiel des fonctionnalités d'horodatage offertes par Linux et la Glibc ainsi que leurs implémentations en Python.

Insistons encore sur la nécessité de mettre l'accent dès à présent sur la portabilité des programmes qui manipulent des données de type `time_t` et sur le risque d'évolution de celui-ci dans l'avenir.

À présent que nous disposons des outils de mesure du temps – et plus particulièrement de `gettimeofday()` – nous allons examiner les mises en sommeil, les attentes et les comportements périodiques des tâches.

9

Sommeil des processus et contrôle des ressources

Nous allons étudier dans la première partie de ce chapitre les méthodes permettant d'endormir un processus pendant une durée plus ou moins précise, et d'exécuter une tâche différée ou périodique. Nous aborderons ensuite les moyens de suivre l'exécution d'un programme et d'obtenir des informations statistiques le concernant. Enfin, nous examinerons les fonctions de limitation des ressources, permettant de restreindre l'utilisation du système par un processus.

Endormir un processus

La fonction la plus simple pour endormir temporairement un processus est `sleep()`, qui est déclarée ainsi dans `<unistd.h>` :

```
| unsigned int sleep (unsigned int nb_secondes);
```

Cette fonction endort le processus pendant la durée demandée et revient ensuite. À cause de la charge du système, il peut arriver que `sleep()` dure un peu plus longtemps que prévu. De même, si un signal interrompt le sommeil du processus, la fonction `sleep()` revient plus tôt que prévu, en renvoyant le nombre de secondes restantes sur la durée initiale. En Python, `time.sleep()` autorise un argument de type réel, le sommeil pouvant ainsi durer un nombre de secondes non entier, mais ne renvoie pas la durée de sommeil restante.

Voici un exemple dans lequel deux processus exécutent un appel à `sleep()`. Le processus père dort deux secondes avant d'envoyer un signal à son fils. Ce dernier essaye de dormir 10 secondes, mais sera réveillé plus tôt par le signal. On affiche l'heure avant et après l'appel `sleep()`. On présente également la durée restante :

```
exemple-sleep.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>

void gestionnaire_sigusr1(int numero)
{ /* Ne rien faire */
}

int main (void)
{
    pid_t          pid;
    time_t          heure;
    unsigned int    duree_sommeil;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    /* Installation d'un handler vide pour SIGUSR1 */
    signal(SIGUSR1, gestionnaire_sigusr1);

    if (pid == 0) {
        /* Fils */
        time(& heure);
        fprintf(stdout, "Avant : %s", ctime(& heure));
        duree_sommeil = sleep(10);
        time(& heure);
        fprintf(stdout, "Après : %s", ctime(& heure));
        fprintf(stdout, "Duree restante %u\n", duree_sommeil);
    } else {
        /* Père */
        sleep(2);
        /* Envoi d'un signal vers le fils */
        kill(pid, SIGUSR1);
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Le même programme peut s'écrire en Python :

```
exemple-sleep.py :
#!/usr/bin/python
from __future__ import print_function
import os
import signal
import time

def handler_usr1(signum, stack):
    pass

signal.signal(signal.SIGUSR1, handler_usr1)
pid = os.fork()
if (pid == 0):
    print("Before :", time.ctime())
    time.sleep(10)
    print("After :", time.ctime())
else:
    time.sleep(2)
    os.kill(pid, signal.SIGUSR1)
```

Voici un exemple d'exécution :

```
$ ./exemple-sleep
Avant : Wed Jun 30 08:02:10 2010
Après : Wed Jun 30 08:02:12 2010
Duree restante 8
$
```

La fonction `sleep()` est implémentée à partir de l'appel système `alarm()`, que nous étudierons ultérieurement avec les signaux. Il est vraiment déconseillé de les utiliser ensemble dans la même portion de programme. La bibliothèque Glibc implémente `sleep()` en prenant garde aux éventuelles interactions avec une alarme déjà programmée, mais ce n'est pas forcément le cas sur d'autres systèmes sur lesquels on peut être amené à porter le programme.

Lorsqu'on désire assurer une durée de sommeil assez précise malgré le risque d'interruption par un signal, on pourrait être tenté de programmer une boucle du type :

```
void sommeil (unsigned int duree_initiale)
{
    unsigned int duree_restante = duree_initiale;
    while (duree_restante > 0)
        duree_restante = sleep(duree_restante);
}
```

Malheureusement, ceci ne fonctionne pas, car lors d'une invocation de la fonction `sleep()` si un signal se produit au bout d'un dixième de seconde par exemple, la durée renvoyée sera quand même décrétementée d'une seconde complète. Si ce phénomène se produit à plusieurs reprises, un décalage certain peut se produire en fin de compte. Pour l'éviter, il faut recadrer la durée de sommeil régulièrement. On peut par exemple utiliser l'appel système `time()`, que nous avons vu dans le chapitre précédent :

```
void sommeil (unsigned int duree_initiale)
{
    time_t    heure_fin;
    time_t    heure_actuelle;

    heure_fin = time(NULL) + duree_initiale;

    while ((heure_actuelle = time(NULL)) < heure_fin)
        sleep(heure_fin-heure_actuelle);
}
```

Cette routine peut quand même durer un peu plus longtemps que prévu si le système est très chargé, mais elle restera précise sur des longues durées, même si de nombreux signaux sont reçus par le processus.

Si on désire avoir une résolution plus précise que la seconde, la fonction `usleep()` est disponible. Il faut imaginer que le `u` représente en réalité le μ de microseconde. Le prototype de cette fonction est déclaré dans `<unistd.h>` ainsi :

```
void usleep (unsigned long nb_micro_secondes);
```

Cette fonction n'existe pas en Python, car `time.sleep()` indique des fractions de secondes. L'appel endort le processus pendant le nombre indiqué de microsecondes, à moins qu'un signal ne soit reçu entre-temps. La fonction `usleep()` ne renvoyant pas de valeur, on ne sait pas si la durée voulue s'est écoulée ou non. Cette fonction est implémentée dans la bibliothèque *Glibc* en utilisant l'appel système `select()`, que nous verrons dans le chapitre consacré aux traitements asynchrones.

Voici une variante du programme précédent, utilisant `usleep()`. Nous allons montrer que cette fonction peut aussi être interrompue par l'arrivée d'un signal.

```
exemple-usleep-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
void gestionnaire_sigusr1(int numero)
{ /* Ne rien faire */
}

int main (void)
{
    pid_t          pid;
    time_t         heure;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Erreur dans fork \n");
        exit(EXIT_FAILURE);
    }
    signal(SIGUSR1, gestionnaire_sigusr1);

    if (pid == 0) {
        time(& heure);
        fprintf(stdout, "Avant : %s", ctime(& heure));
        usleep(10000000); /* 10 millions de us = 10 secondes */
        time(& heure);
        fprintf(stdout, "Après : %s", ctime(& heure));
    } else {
        usleep(2000000); /* 2 millions de us = 2 secondes */
        kill(pid, SIGUSR1);
        waitpid(pid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Le sommeil du processus fils, censé durer 10 secondes, est interrompu au bout de 2 secondes par le signal provenant du père.

```
$ ./exemple-usleep-1
Avant : Wed Jun 30 08:03:45 2010
Après : Wed Jun 30 08:03:47 2010
$
```

Il existe encore une autre fonction de sommeil, offrant une précision encore plus grande : `nanosleep()`. Cette fonction est définie par SUSv4. Elle est déclarée ainsi dans `<time.h>` :

```
| int nanosleep (const struct timespec * voulu, struct timespec * restant);
```

Le premier argument représente la durée de sommeil désirée, et le second argument, s'il est non NULL, permet de stocker la durée de sommeil restante lorsque la fonction a

été interrompue par l'arrivée d'un signal. Si `nanosleep()` dort pendant la durée désirée, elle renvoie 0. En cas d'erreur, ou si un signal la réveille prématurément, elle renvoie -1 (et place `EINTR` dans `errno` dans ce dernier cas).

La structure `timespec` servant à indiquer la durée de sommeil contient deux membres : l'un précisant le nombre de secondes, l'autre contenant la partie fractionnaire exprimée en nanosecondes (tableau suivant).

Nom	Type	Signification
<code>tv_sec</code>	<code>time_t</code>	Nombre de secondes écoulées depuis le 1 ^{er} janvier 1970.
<code>tv_nsec</code>	<code>long int</code>	Nombre de nanosecondes depuis le dernier changement de <code>tv_sec</code> .

Il est illusoire d'imaginer avoir une précision de l'ordre de la nanoseconde, ou même de la microseconde, sur un système multitâche. Même sur un système monotâche dédié à une application en temps réel, il est difficile d'obtenir une telle précision sans avoir recours à des boucles d'attente vides, ne serait-ce qu'en raison de l'allongement de durée dû à l'ordonnancement proprement dit.

Longtemps, les sommeils (ainsi que les *timers* que nous verrons plus loin) étaient cadencés par le *tick* du système. Celui-ci, configurable lors de la compilation du noyau Linux, prenait généralement pour valeur 10 ms pour les noyaux jusqu'au 2.4 et sur les systèmes embarqués, et 1 ms sur les noyaux 2.6. Ceci signifiait qu'il était impossible d'obtenir des sommeils ou des *timers* dont la durée soit inférieure à la milliseconde (voire à la dizaine de millisecondes).

Plus récemment, les *timers* à haute résolution (*hrtimers*) sont apparus, s'appuyant sur les améliorations du matériel. Grâce à ceux-ci, la granularité des sommeils peut descendre à quelques dizaines de microsecondes.

Voici un exemple où nous allons mesurer la durée effective d'un sommeil demandé avec `usleep()`. Le programme ci-dessous mesure la durée d'un seul sommeil, mais il serait intéressant de faire cette opération en boucle, pour vérifier la durée moyenne (et la durée maximale).

```
exemple-usleep-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

int main (int argc, char * argv[])
{
    struct timeval tv_avant;
    struct timeval tv_apres;
```



```
struct timeval tv_duree;
long int duree_sommeil;
long int sommeil_voulu;
int i;

if ((argc != 2)
    || (sscanf(argv[1], "%ld", & sommeil_voulu) != 1)) {
    fprintf(stderr, "usage: %s duree_sommeil_en_us\n",
            argv[0]);
    exit(EXIT_FAILURE);
}

for (i = 0; i < 10; i++) {
    gettimeofday(& tv_avant, NULL);
    usleep(sommeil_voulu);
    gettimeofday(& tv_apres, NULL);
    timersub(& tv_apres, & tv_avant, & tv_duree);
    duree_sommeil = tv_duree.tv_sec * 1000000
        + tv_duree.tv_usec;
}
fprintf(stdout, "%ld\n", duree_sommeil);
return EXIT_SUCCESS;
}
```

La durée de sommeil voulu est passée en argument sur la ligne de commande du processus.

```
$ ./exemple-usleep-2 100000
100059
$ ./exemple-usleep-2 10000
10050
$ ./exemple-usleep-2 1000
1035
$ ./exemple-usleep-2 100
144
$ ./exemple-usleep-2 50
110
$ ./exemple-usleep-2 10
80
$
```

Nous voyons que sur cette machine (les performances peuvent varier sensiblement d'un matériel à l'autre) la précision du sommeil est de l'ordre de la cinquantaine de microsecondes.

Utilisation des temporisations Unix

Pour chaque processus, il existe en fait trois temporisations Unix (*timers*) qui lui permettent de recevoir régulièrement un signal. Lors de la réception de ce signal, le processus peut exécuter une fonction préprogrammée (*handler*). Le principe des signaux sera approfondi dans le second volume.

Les temporisations peuvent être configurées pour s'exécuter une seule fois ou se relancer automatiquement en fonctionnement périodique. Les trois temporisations sont programmées par l'appel système `setitimer()`. On peut également consulter la programmation en cours grâce à l'appel `getitimer()`.

Le prototype de `setitimer()` est déclaré dans `<sys/time.h>`, en Python il s'agit d'une fonction du module `signal`.

```
int setitimer (int laquelle, const struct itimerval * valeur,  
              struct itimerval * ancienne);
```

Le premier argument permet de choisir quelle temporisation est utilisée parmi les trois constantes symboliques suivantes (tableau ci-après).

Nom	Signification
ITIMER_REAL	Le décompte de la temporisation a lieu en temps « réel », et lorsque le compteur arrive à zéro, le signal <code>SIGALRM</code> est envoyé au processus.
ITIMER_VIRTUAL	La temporisation ne décroît que lorsque le processus s'exécute en mode utilisateur. Un signal <code>SIGVTALRM</code> lui est envoyé à la fin du décompte.
ITIMER_PROF	Le décompte a lieu quand le processus s'exécute en mode utilisateur, mais également pendant qu'il s'exécute en mode noyau, durant les appels système. Au bout du délai programmé, le signal <code>SIGPROF</code> est émis.

L'utilisation de la temporisation `ITIMER_REAL` est la plus courante. Elle correspond généralement à ce que le programmeur entend par *timer*.

`ITIMER_VIRTUAL` s'utilise surtout conjointement à `ITIMER_PROF`, car ces temporisations permettent, par une simple soustraction, d'obtenir des statistiques sur le temps d'exécution passé par le processus en mode utilisateur et en mode noyau.

La temporisation `ITIMER_PROF` permet de rendre compte du déroulement du processus indépendamment des mécanismes d'ordonnancement, et donc d'avoir une indication quantitative de la durée d'une tâche quelle que soit la charge système. On peut utiliser cette technique pour comparer par exemple les durées de plusieurs algorithmes de calcul.

Pour lire l'état de la programmation en cours, on utilise `getitimer()` en C et `signal.getitimer()` en Python.

```
int getitimer (int laquelle, struct itimerval * valeur);
```

La structure `itimerval` servant à stocker les données concernant un timer est définie dans `<sys/time.h>` avec les deux membres suivants.

Type	Nom	Signification
struct timeval	it_interval	Valeur à reprogrammer lors de l'expiration du <i>timer</i> .
struct timeval	it_value	Valeur décroissante actuelle.

Les temporisations n'expirent jamais avant la fin du délai programmé, mais plutôt légèrement après, avec un retard constant dépendant de l'horloge interne du système. Si on désire faire des mesures critiques, il est possible de calibrer ce léger retard.

Avec la temporisation `ITIMER_REAL`, lorsque le signal `SIGALRM` est émis, le processus n'est pas nécessairement actif (contrairement aux deux autres temporisations). Il peut donc s'écouler un retard avant l'activation du processus et la délivrance du signal. Avec la temporisation `ITIMER_PROF`, le processus peut se trouver au sein d'un appel système, et un retard sera également possible avant l'appel du gestionnaire de signaux.

L'exemple suivant va utiliser conjointement les deux timers `ITIMER_VIRTUAL` et `ITIMER_PROF` pour mesurer les durées passées dans les modes utilisateur et noyau d'une routine qui fait une série de boucles consommant du temps processeur, suivie d'une série de copies d'un fichier vers le périphérique `/dev/null` pour exécuter de nombreux appels système.

Le gestionnaire de signaux commun aux deux temporisations départage les signaux, puis incrémente le compteur correspondant. Les temporisations sont réglées pour envoyer un signal tous les centièmes de seconde. Une routine d'affichage des données est installée par `atexit()` afin d'être invoquée en sortie du programme.

```
exemple-setitimer-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>

unsigned long int mode_utilisateur;
unsigned long int mode_utilisateur_et_noyau;

void gestionnaire_signaux (int numero);
void action_a_mesurer (void);

int main (void)
```

```

{
    struct itimerval timer;

    /* Préparation du timer */
    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = 10000; /* 1/100 s. */
    timer.it_interval.tv_sec = 0;
    timer.it_interval.tv_usec = 10000; /* 1/100 s. */

    /* Installation du gestionnaire de signaux */
    signal(SIGVTALRM, gestionnaire_signaux);
    signal(SIGPROF, gestionnaire_signaux);
    /* Programmation des timers */
    if ((setitimer(ITIMER_VIRTUAL, & timer, NULL) != 0)
        || (setitimer(ITIMER_PROF, & timer, NULL) != 0)) {
        fprintf(stderr, "Erreur dans setitimer \n");
        return EXIT_FAILURE;
    }
    /* Appel de la routine de travail effectif du processus */
    action_a_mesurer();

    /* Désinstallation des timers */
    setitimer(ITIMER_VIRTUAL, NULL, NULL);
    setitimer(ITIMER_PROF, NULL, NULL);
    fprintf(stdout, "Temps en mode utilisateur : %ld/100 s \n",
            mode_utilisateur);
    fprintf(stdout, "Temps en mode noyau : %ld/100 s \n",
            mode_utilisateur_et_noyau - mode_utilisateur);
    return EXIT_SUCCESS;
}

void gestionnaire_signaux (int numero)
{
    if (numero == SIGVTALRM)
        mode_utilisateur ++;
    else
        mode_utilisateur_et_noyau ++;
}

void action_a_mesurer (void)
{
    int i, j;
    FILE * fp1, * fp2;
    double x = 0.0;

    for (i = 0; i < 30000; i ++)
        for (j = 0; j < 30000; j ++)
            x += i * j;

```

```
for (i = 0; i < 50000; i ++) {
    if ((fp1 = fopen ("exemple_setitimer_2", "r")) != NULL) {
        if ((fp2 = fopen("/dev/null", "w")) != NULL) {
            while (fread(& j, sizeof (int), 1, fp1) == 1)
                fwrite(& j, sizeof (int), 1, fp2);
            fclose(fp2);
        }
        fclose(fp1);
    }
}
```

L'exécution affiche les résultats suivants.

```
$ ./exemple-setitimer-1
Temps en mode utilisateur : 470/100 s
Temps en mode noyau : 9/100 s
$ ./exemple-setitimer-1
Temps en mode utilisateur : 471/100 s
Temps en mode noyau : 10/100 s
$ ./exemple-setitimer-1
Temps en mode utilisateur : 470/100 s
Temps en mode noyau : 10/100 s
```

Nous voyons les quelques fluctuations dues à l'ordonnancement sur un système multitâche, même si les ordres de grandeur restent bien constants. Nous copions à présent ce programme dans `exemple-setitimer-2.c` en ne conservant plus que la routine de travail effectif, ce qui nous donne cette fonction `main()` :

```
int main (void)
{
    action_a_mesurer();
    return EXIT_SUCCESS;
}
```

Nous pouvons alors utiliser la fonction `times` de Bash, qui permet de mesurer les temps cumulés d'exécution en mode noyau et en mode utilisateur du shell et des processus qu'il a lancés.

```
$ sh -c "./exemple-setitimer-2 ; times"
0m0.003s 0m0.001s
0m4.701s 0m0.096s
```

```
$ sh -c "./exemple-setitimer-2 ; times"
0m0.002s 0m0.002s
0m4.696s 0m0.096s
$ sh -c "./exemple-setitimer-2 ; times"
0m0.001s 0m0.001s
0m4.696s 0m0.089s
$
```

Nous voyons que les résultats sont comparables, même s'ils présentent également une variabilité due à l'ordonnancement multitâche.

Nous allons voir un exemple facile d'utilisation de `signal.setitimer()` en Python. La configuration s'effectue simplement en indiquant la durée avant le premier signal et la période de répétition. Les durées sont des nombres réels de secondes, la partie fractionnaire étant prise en considération.

Dans notre exemple, un processus installe un handler pour le signal, programme le timer, puis se met en boucle à afficher un point chaque seconde. Lorsque le timer se déclenche, on affiche la chaîne "Alarme !".

```
exemple-setitimer.py :
#!/usr/bin/python
from __future__ import print_function
import signal
import sys
import time

def handler_sigalrm(signum, stack):
    print("Alarme !", file=sys.stderr)

signal.signal(signal.SIGALRM, handler_sigalrm)
signal.setitimer(signal.ITIMER_REAL, 5, 5)
while True:
    print(".", file=sys.stderr, end='')
    time.sleep(1)
```

On notera que les écritures s'effectuent sur la sortie interactive `stderr` afin de s'assurer que les messages seront visibles immédiatement (nous détaillerons ceci dans le chapitre 21).

```
$ ./exemple-setitimer.py
....Alarme !
....Alarme !
....Alarme !
..^C
```

Timers temps réel

Depuis le noyau 2.6, Linux intègre un ensemble d'appels système permettant de réaliser des tâches périodiques plus élaborées. Le principe est simple : on programme un timer avec un délai initial et une période de répétition donnés. Au déclenchement du timer, le noyau délivrera un signal au processus et réarmera la temporisation. La création et la destruction d'un timer se font avec les routines suivantes.

```
int timer_create (clockid_t clock, struct sigevent * event,  
                 timer_t * timer);  
int timer_delete (timer_t timer);
```

Le premier argument de `timer_create()`, de type `clockid_t`, décrit une horloge fournie par le système sur laquelle le timer doit s'appuyer. Chaque implémentation peut proposer ses propres horloges, et éventuellement proposer la définition d'horloges par l'application. Toutefois, certaines ont une signification standard.

- `CLOCK_REALTIME` : cette horloge représente l'écoulement effectif du temps tel qu'il est perçu par le système. La valeur de l'horloge peut être modifiée (par exemple avec la commande `date` depuis le shell).
- `CLOCK_MONOTONIC` : contrairement à la précédente, la valeur de l'horloge monotone est strictement croissante. La modification de l'heure système n'influe pas sur elle.
- `CLOCK_PROCESS_CPUTIME_ID` : une horloge qui démarre en même temps que le processus. Elle représente la durée complète d'exécution du processus, même lorsqu'il est endormi.
- `CLOCK_THREAD_CPUTIME_ID` : cette horloge est semblable à la précédente mais démarre lorsque le thread courant débute. Elle peut donc être décalée par rapport à la précédente.

Le second argument est une structure décrivant la manière dont le noyau doit notifier le processus de l'expiration du timer. Cette structure est également utilisée dans les notifications d'entrées-sorties asynchrones telles que nous les retrouverons dans le troisième volume de ce livre.

Elle contient un premier champ `sigev_notify` qui indique, quand il contient la valeur `SIGEV_SIGNAL` que le noyau doit notifier le processus en lui envoyant un signal (dont le numéro est indiqué dans le champ `sigev_signo`)

Lorsque ce champ contient `SIGEV_THREAD`, la notification se fait en démarrant un thread qui exécutera la fonction indiquée dans le membre `sigev_notify_function` avec les attributs du champ `sigev_notify_attributes`.

Le troisième argument enfin est un pointeur sur un objet de type `timer_t` qui contiendra au retour l'identifiant du timer créé.

Pour amorcer le timer, il faut utiliser la fonction suivante :

```
int timer_settime (timer_t timer, int options,
                  const struct itimerspec * spec,
                  struct itimerspec * precedent);
```

Le premier argument est l'identifiant du timer obtenu précédemment, si le second argument est nul, les valeurs indiquées dans la structure `itimerspec` du troisième argument sont relatives à l'heure actuelle. Si ce second argument contient la valeur `TIMER_ABSTIME`, les valeurs sont des heures absolues.

La structure `itimerspec` est très proche de la structure `itimerval` que nous avons vue plus haut. Elle contient deux champs `it_value` et `it_interval` de type `struct timespec` (deux champs `tv_sec` et `tv_nsec`) décrivant (en secondes et nanosecondes) le délai avant expiration du *timer* et la période de répétition.

Ces fonctions de temporisation étant issues de la norme Posix.1b (temps réel), il faut ajouter l'option `-lrt` (*real time library*) lors de l'édition des liens. Ceci est présent dans le fichier `Makefile` fourni avec les sources des programmes du livre.

Dans l'exemple suivant nous allons programmer deux timers : le premier s'exécute avec une fréquence fournie par l'utilisateur sur la ligne de commande. À chaque expiration nous incrémentons un compteur global. Le second timer fonctionne avec une période d'une seconde. Il affiche la valeur du compteur (qui correspond donc à la fréquence réelle de l'autre timer) et le réinitialise.

```
exemple-timer-create.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

volatile int compteur = 0;

void gestionnaire_usr1(int numero)
{
    compteur ++;
}

void gestionnaire_usr2(int numero)
{
    fprintf(stderr, "%d\n", compteur);
    compteur = 0;
}

int main(int argc, char * argv[])
{
```



```
long int      frequence;
timer_t timer1, timer2;
struct sigevent event;
struct itimerspec itimer;

if ((argc != 2)
    || (sscanf(argv[1], "%ld", & frequence) != 1)
    || (frequence < 2) || (frequence > 1000000000)) {
    fprintf(stderr, "usage: %s frequence\n", argv[0]);
    exit(EXIT_FAILURE);
}
signal(SIGUSR1, gestionnaire_usr1);
signal(SIGUSR2, gestionnaire_usr2);
event.sigev_notify = SIGEV_SIGNAL;
event.sigev_signo = SIGUSR1;
if (timer_create(CLOCK_REALTIME, &event, &timer1)!=0) {
    perror("timer_create");
    exit(EXIT_FAILURE);
}
event.sigev_notify = SIGEV_SIGNAL;
event.sigev_signo = SIGUSR2;
if (timer_create(CLOCK_REALTIME, &event, &timer2)!=0) {
    perror("timer_create");
    exit(EXIT_FAILURE);
}
itimer.it_value.tv_sec = 0;
itimer.it_value.tv_nsec = 1000000000/frequence;
itimer.it_interval.tv_sec = 0;
itimer.it_interval.tv_nsec = 1000000000/frequence;
if (timer_settime(timer1, 0, & itimer, NULL) != 0) {
    perror("timer_settime");
    exit(EXIT_FAILURE);
}
itimer.it_value.tv_sec = 1;
itimer.it_value.tv_nsec = 0;
itimer.it_interval.tv_sec = 1;
itimer.it_interval.tv_nsec = 0;
if (timer_settime(timer2, 0, & itimer, NULL) != 0) {
    perror("timer_settime");
    exit(EXIT_FAILURE);
}
while (1)
    pause();
return EXIT_SUCCESS;
}
```

Lors de son exécution, le programme montre les limites d'un timer géré par l'ordonnanceur. Pour de faibles valeurs, le comptage est assez bon, mais très vite les fluctuations des timers apparaissent.

```
$ ./exemple-timer-create 10
9
10
10
10
      (Control-C)
$ ./exemple-timer-create 50
49
50
50
50
      (Control-C)
$ ./exemple-timer-create 100
99
100
100
100
      (Control-C)
$ ./exemple-timer-create 500
497
499
500
499
      (Control-C)
$ ./exemple-timer-create 1000
998
1000
999
998
      (Control-C)
$ ./exemple-timer-create 5000
4917
4937
4960
4942
      (Control-C)
$ ./exemple-timer-create 10000
9790
9853
9768
9760
      (Control-C)
$
```

Le noyau Linux standard n'est pas fait pour réaliser des tâches temps réel de haute précision. Si l'on a besoin d'obtenir des temporisations avec une précision inférieure

à la milliseconde, il faudra utiliser une autre solution (patch PREEMPT_RT, Xenomai, etc. – voir [BLAESS 2015] *Solutions temps réel sous Linux*).

Notifications par descripteur

Depuis le noyau 2.6.25, Linux nous propose des timers dont la notification est obtenue par l'intermédiaire d'un descripteur de fichier. Les fonctions suivantes sont déclarées dans `<sys/timerfd.h>` :

```
int timerfd_create (int clock, int flags);
int timerfd_settime (int fd, int options,
                    const struct itimerspec *spec,
                    struct itimerspec *oldspec);
int timerfd_gettime (int fd, struct itimerspec *spec);
```

La fonction `timerfd_create()` permet d'obtenir un descripteur de timer : le champ `clock` est identique à celui de `timer_create()` et le champ `flags` peut contenir (avec un OU binaire) : `TFD_NONBLOCK` pour que le descripteur soit lisible sans bloquer et `TFD_CLOEXEC` pour que ce descripteur soit automatiquement fermé lors d'un appel système `execve()`.

La fonction `timerfd_settime()` permet de configurer la durée initiale et la période du timer comme `timer_settime()` vu précédemment.

Il devient alors possible de lire le contenu du descripteur avec un appel système `read()` : si aucune notification n'est arrivée, l'appel est bloquant jusqu'au prochain déclenchement ; si, au contraire, (au moins) une notification est déjà en attente, l'appel revient immédiatement. Le contenu lisible par `read()` est un entier long de huit octets contenant le nombre de notifications qui était en attente. En fonctionnement normal, la valeur lue est donc 1.

Un intérêt de ce type d'implémentation est la compatibilité avec des appels système comme `select()` ou `poll()` que nous verrons dans le chapitre 25. Le timer peut servir à réveiller une tâche en attente sur de multiples descripteurs.

L'exemple ci-après est très simple : tous les dixièmes de seconde, le processus se réveille pour afficher le nombre de notifications reçues.

```
exemple-timerfd.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdarg.h>
#include <sys/timerfd.h>
```


Le type `clock_t` représente un temps processeur écoulé sous forme d'impulsions d'horloge théoriques. Nous précisons qu'il s'agit d'impulsions théoriques car il y a une différence d'ordre de grandeur importante entre ces quantités et la véritable horloge système utilisée par l'ordonnanceur. À cause d'une différence entre les standards *Ansi C* et *Posix*, cette valeur n'a plus aucune signification effective. Pour obtenir une durée en secondes, il faut diviser la valeur `clock_t` par la constante `CLOCKS_PER_SEC`. Cette constante vaut un million sur l'essentiel des systèmes Unix dérivant de *Système V*, ainsi que sous Linux. On imagine assez bien que le séquençement des tâches est loin d'avoir effectivement lieu toutes les microsecondes... En Python, la fonction `time.clock()` fournit une valeur équivalente. On notera que depuis Python 3.3, on lui préfère la fonction `time.perf_counter()` dont la mesure est plus précise.

On ne sait pas avec quelle valeur la fonction `clock()` démarre. Il s'agit parfois de zéro, mais ce n'est pas obligatoire. Aussi est-il nécessaire de mémoriser la valeur initiale et de la soustraire pour connaître la durée écoulée.

Sous Linux, `clock_t` est un entier long, mais ce n'est pas toujours le cas sur d'autres systèmes. Il importe donc de forcer le passage en virgule flottante pour pouvoir effectuer l'affichage. Notre programme d'exemple va mesurer le temps processeur écoulé tant en mode utilisateur qu'en mode noyau, dans la routine que nous avons déjà utilisée dans les exemples précédents.

Le programme `exemple_clock.c` contient donc la fonction `main()` suivante, en plus de la routine `action_a_mesurer()`.

```
exemple-clock.c :
int main (void)
{
    clock_t debut_programme;
    double duree_ecoulee;

    debut_programme = clock();

    action_a_mesurer();

    duree_ecoulee = clock() - debut_programme;
    duree_ecoulee = duree_ecoulee / CLOCKS_PER_SEC;
    fprintf(stdout, "Duree = %f \n", duree_ecoulee);
    return EXIT_SUCCESS;
}
```

Les résultats sont les suivants.

```
$ ./exemple-clock
Duree = 4.830000
$ ./exemple-clock
Duree = 4.840000
$
```

Comme il fallait s'y attendre, il s'agit de la somme des temps obtenus avec nos programmes précédents, avec – comme toujours – une légère variation en fonction de la charge système.

Sous Linux, `clock_t` est équivalent à un `long int`. Il y a donc un risque de débordement de la valeur maximale au bout de `LONG_MAX` impulsions théoriques. `LONG_MAX` est une constante symbolique définie dans `<limits.h>`. Sur un PC, `LONG_MAX` vaut 2.147.483.647, et `CLOCKS_PER_SEC` vaut 1.000.000. Cela donne donc une durée avant le dépassement de 2 147 secondes, soit 35 minutes.

Rappelons qu'il s'agit là de temps processeur effectif, et qu'il est assez rare qu'un programme cumule autant de temps d'exécution. Mais cela peut arriver, notamment avec des programmes de calcul ou de traitement d'image. Si on désire suivre les durées d'exécution de tels programmes (particulièrement pour comparer des algorithmes), il faut disposer d'un mécanisme permettant d'obtenir des mesures plus longues.

Depuis Python 3.3, la fonction `process_time()` renvoie le temps CPU consommé (en secondes et décimales) depuis le démarrage du processus appelant.

En C, l'appel système `times()` fournit des informations plus précises que `clock()`. Il est déclaré ainsi dans `<sys/times.h>` :

```
clock_t times (struct tms * measure);
```

La valeur renvoyée par cet appel système est le nombre de « jiffies », c'est-à-dire le nombre de cycles d'horloge exécutés depuis le démarrage du système, ou `(clock_t)-1` en cas d'échec. Il s'agit cette fois de la véritable horloge de l'ordonnanceur, qui a une période de 1/1 000 s sur la plupart des PC. On peut utiliser cette valeur renvoyée, un peu à la manière de celle qui est fournie par la fonction `clock()`, mais en utilisant une autre constante de conversion, celle renvoyée par la commande `sysconf(_SC_CLK_TCK)`.

Notre premier exemple ne s'occupera pas de l'argument de la fonction `times()`, en passant un pointeur `NULL`, afin de se soucier uniquement de la valeur de retour. Nous calquons notre programme sur le précédent, avec la fonction `main()` suivante :

```
exemple-times-1.c :
int main (void)
```

exemple-times-1.c :

```
int main (void)
{
    clock_t debut_programme;
    double duree_ecoulee;
    debut_programme = times(NULL);
    fprintf(stdout, "Jiffies au debut %ld \n", debut_programme);
    action_a_mesurer();
    fprintf(stdout, "Jiffies en fin %ld \n", times (NULL));
    duree_ecoulee = (double) (times(NULL) - debut_programme);
    duree_ecoulee = duree_ecoulee / sysconf(_SC_CLK_TCK);
    fprintf(stdout, "Duree = %f \n", duree_ecoulee);
    return EXIT_SUCCESS;
}
```

L'exécution est la suivante.

```
$ ./exemple-times-1
Jiffies au debut 429808629
Jiffies en fin 429809113
Duree = 4.840000
$
```

Cette fois-ci, la durée est celle de l'exécution totale du programme et non le temps processeur consommé (il y a peu d'écart car notre machine est actuellement faiblement chargée).

Voici à présent le détail de la structure `tms`, qu'on passe en argument de `times()` afin qu'elle soit remplie. La définition se trouve dans `<sys/times.h>`.

Type	Nom	Signification
clock_t	tms_utime	Temps processeur passé en mode utilisateur.
clock_t	tms_stime	Temps processeur passé en mode noyau.
clock_t	tms_cutime	Temps processeur passé en mode utilisateur par les processus fils terminés du programme appelant.
clock_t	tms_cstime	Temps processeur passé en mode noyau par les processus fils terminés du programme appelant.

Les deux derniers membres contiennent les temps utilisateur et noyau cumulés de tous les processus fils terminés au moment de l'appel. Nous allons utiliser ces membres après avoir invoqué une commande passée en argument au programme. Cela permet d'avoir une fonctionnalité du même style que la commande `times` intégrée au shell `bash`.

```
exemple-times-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/times.h>

int main (int argc, char * argv[])
{
    struct tms mesure;
    double duree;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s <commande> \n", argv [0]);
        exit(EXIT_FAILURE);
    }
    system(argv[1]);
    times(& mesure);
    duree = (double) mesure.tms_cutime / sysconf(_SC_CLK_TCK);
    fprintf(stdout, "Temps CPU mode utilisateur = %f \n", duree);
    duree = (double) mesure.tms_cstime / sysconf(_SC_CLK_TCK);
    fprintf(stdout, "Temps CPU en mode noyau      = %f \n", duree);
    return EXIT_SUCCESS;
}
```

L'exécution, en reprenant toujours notre même routine de test, donne les résultats suivants.

```
$ ./exemple-times-2 ./exemple-setitimer-2
Temps CPU mode utilisateur = 4.690000
Temps CPU en mode noyau    = 0.090000
$ ./exemple-times-2 ./exemple-setitimer-2
Temps CPU mode utilisateur = 4.700000
Temps CPU en mode noyau    = 0.090000
$ ./exemple-times-2 ./exemple-setitimer-2
Temps CPU mode utilisateur = 4.700000
Temps CPU en mode noyau    = 0.090000
$
```

Les statistiques obtenues sont compatibles avec les données que nous avons déjà observées.

Obtenir des statistiques sur un processus

Il est parfois utile d'obtenir des informations sur les performances d'un programme donné. On peut ainsi, dans certains cas, optimiser les algorithmes ou réétudier cer-

tains goulets d'étranglement où le programme est ralenti. La fonction `getrusage()` permet d'obtenir les statistiques concernant le processus appelant ou l'ensemble de ses fils qui se sont terminés. Dans ce dernier cas, les valeurs sont cumulées sur la totalité des processus concernés.

Le prototype de `getrusage()` est déclaré dans `<sys/resource.h>` ; en Python, on trouve cette fonction dans le module `resource` :

```
| int getrusage (int lesquelles, struct rusage * statistiques);
```

Le premier argument de cette fonction indique quelles statistiques nous intéressent. Il peut s'agir de l'une des constantes symboliques suivantes.

Nom	Signification
RUSAGE_SELF	Obtenir les informations concernant le processus appelant.
RUSAGE_CHILDREN	Obtenir les statistiques sur les processus fils terminés.

Le second argument est une structure `rusage`. Cette dernière est remplie lors de l'appel. La fonction renvoie 0 si elle réussit, et -1 si elle échoue. En fait, sous Linux, l'appel système `getrusage()` ne remplit qu'un petit nombre de champs de la structure `rusage`.

Type	Nom	Signification
struct timeval	ru_utime	Temps passé par le processus en mode utilisateur.
struct timeval	ru_stime	Temps passé par le processus en mode noyau.
long	ru_minflt	Nombre de fautes de pages mineures (n'ayant pas nécessité de rechargement depuis le disque).
long	ru_majflt	Nombre de fautes de pages majeures (ayant nécessité un rechargement des données depuis le disque).
long	ru_nswap	Nombre de fois où le processus a été entièrement swappé.
long	ru_maxrss	Volume maximal de mémoire physique employée (en kilo-octets).
long	ru_inblock	Nombre de lectures de données.
long	ru_outblock	Nombre d'écritures de données.
long	ru_nvcsw	Nombre de changements de contextes volontaires (attente d'événement externe).
long	ru_nicsw	Nombre de préemptions (tâche plus prioritaire devenue prête ou consommation de la tranche de temps accordée).

Les autres champs sont mis à zéro lors de l'appel. De cette façon, une application désirant tirer parti de leur contenu lorsqu'ils seront vraiment remplis par une nouvelle version du noyau peut vérifier si leur valeur est non nulle et les utiliser alors.

Voici un exemple d'utilisation où, comme dans le programme précédent, nous lançons la commande passée en argument grâce à la fonction `system()`. Si aucun argument n'est fourni, le processus affiche les statistiques le concernant.

```
exemple-rusage.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>

int main (int argc, char * argv[])
{
    int lesquelles;
    struct rusage statistiques;

    if (argc == 1) {
        lesquelles = RUSAGE_SELF;
    } else {
        system(argv[1]);
        lesquelles = RUSAGE_CHILDREN;
    }
    if (getrusage(lequelles, & statistiques) != 0) {
        fprintf(stderr, "Impossible d'obtenir les statistiques \n");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Temps en mode utilisateur %ld s, %ld ms \n",
        statistiques.ru_utime.tv_sec,
        statistiques.ru_utime.tv_usec / 1000);
    fprintf(stdout, "Temps en mode noyau %ld s, %ld ms \n",
        statistiques.ru_stime.tv_sec,
        statistiques.ru_stime.tv_usec / 1000);
    fprintf(stdout, "\n");
    fprintf(stdout, "Nombre de fautes de pages mineures : %ld \n",
        statistiques.ru_minflt);
    fprintf(stdout, "Nombre de fautes de pages majeures : %ld \n",
        statistiques.ru_majflt);
    fprintf(stdout, "Nombre de swaps du processus : %ld \n",
        statistiques.ru_nswap);
    return EXIT_SUCCESS;
}
```

On l'exécute toujours avec la même routine de test.

```
$ ./exemple-rusage ./exemple-setitimer-2
Temps en mode utilisateur 4 s, 697 ms
Temps en mode noyau      0 s, 93 ms
```

```
Nombre de fautes de pages mineures : 421
Nombre de fautes de pages majeures : 0
Nombre de swaps du processus      : 0
$
```

Les durées sont cohérentes avec les statistiques déjà obtenues. Le nombre de fautes de pages est beaucoup plus difficile à interpréter. Voici un exemple en Python dans lequel on affiche aussi le nombre de changements de contextes :

```
exemple-rusage.py :
#!/usr/bin/python
from __future__ import print_function
import os
import resource
import sys
import time

if len(sys.argv) == 1:
    who = resource.RUSAGE_SELF
else:
    os.system(sys.argv[1])
    who = resource.RUSAGE_CHILDREN

ru = resource.getrusage(who)
print("User CPU time      =", ru.ru_utime)
print("System CPU time   =", ru.ru_stime)
print("Maximal RSS (kb)   =", ru.ru_maxrss)
print("Voluntary switches =", ru.ru_nvcsw)
print("Preemptions        =", ru.ru_nivcsw)
```

Exécutons le programme en lui demandant de lancer la commande `ls`.

```
$ ./exemple-rusage.py ls
exemple-clock exemple-setitimer-2.c exemple-times-1
[ ...] (affichage de ls)
User CPU time      = 0.003014
System CPU time    = 0.0
Maximal RSS (kb)   = 2700
Voluntary switches = 1
Preemptions        = 32
```

Nous voyons que la commande a été préemptée trente-deux fois. D'autres tâches plus prioritaires l'ont donc interrompue. Relançons la même commande en lui donnant la priorité temps réel la plus élevée (nous en reparlerons en détail dans le prochain chapitre).

```
$ sudo chrt -f 99 ./exemple-rusage.py ls
exemple-clock exemple-setitimer-2.c exemple-times-1
[ ...] (affichage de ls)
User CPU time      = 0.0
System CPU time    = 0.002467
Maximal RSS (kb)   = 2696
Voluntary switches = 1
Preemptions        = 0
```

Cette fois, notre commande n'a jamais été préemptée. C'est parfaitement cohérent avec le fonctionnement de l'ordonnanceur.

Limiter les ressources consommées par un processus

Les processus disposent d'un certain nombre de limites, supervisées par le noyau. Celles-ci se rapportent à des paramètres concernant des ressources système dont l'utilisation par le processus est surveillée étroitement. Certaines limites se justifient principalement dans le cas d'un système multi-utilisateur, pour éviter de léser les autres personnes connectées. C'est le cas par exemple du nombre maximal de processus pour un même utilisateur, ou de la taille maximale d'un fichier. D'autres limites peuvent intéresser le programmeur, pour surveiller le comportement de son application, comme la limite maximale de la pile ou du temps processeur consommé.

Comme la plupart des stations Linux sont réservées à un seul utilisateur, une partie importante des limites ne se justifie pas forcément, et les distributions ont un comportement très libéral dans leur configuration par défaut.

On accède aux ressources d'un processus grâce à la fonction `getrlimit()`, déclarée dans `<sys/resource.h>` ainsi :

```
| int getrlimit(int ressource, struct rlimit * limite);
```

Le premier argument permet de préciser la ressource concernée, et la structure `rlimit`, transmise en second argument, est remplie avec la limite demandée. La fonction renvoie 0 si elle réussit, et -1 en cas d'échec.

Chaque limite est composée de deux valeurs : une souple et une stricte. La limite souple peut être augmentée ou diminuée au gré de l'utilisateur, tout en ne dépassant jamais la limite stricte. Celle-ci peut être diminuée par l'utilisateur, mais ne peut être augmentée que par root ou un processus ayant la capacité `CAP_SYS_RESOURCE`.

Les limites sont transmises aux processus fils et aux programmes lancés par un `exec()`. Il est donc courant que l'administrateur système impose des valeurs aux limites strictes dans les fichiers d'initialisation du shell de connexion des utilisateurs.

Il existe une valeur spéciale, définie par la constante symbolique `RLIM_INFINITY`, pour indiquer que le processus n'a pas de limitation pour la ressource concernée.

Les différentes ressources sont les suivantes.

Nom	Signification
<code>RLIMIT_CPU</code>	Temps processeur consommable par le processus. S'il dépasse sa limite souple, il reçoit le signal <code>SIGXCPU</code> toutes les secondes. S'il l'ignore et atteint sa limite stricte, le noyau le tue par <code>SIGKILL</code> .
<code>RLIMIT_FSIZE</code>	Taille maximale en octets d'un fichier créé par le processus. Si on essaye de dépasser cette valeur, un signal <code>SIGXFSZ</code> est envoyé au processus si aucun octet n'a été écrit. Si ce signal est ignoré, les écritures après la limite se solderont par un échec avec l'erreur <code>EFBIG</code> .
<code>RLIMIT_DATA</code>	Taille maximale de la zone de données d'un processus. Cette limite est mise en place au chargement du programme.
<code>RLIMIT_STACK</code>	Taille maximale de la pile.
<code>RLIMIT_CORE</code>	Taille maximale d'un éventuel fichier d'image mémoire <code>core</code> . Si cette limite est mise à zéro, aucun fichier <code>core</code> ne sera créé en cas d'arrêt anormal du processus.
<code>RLIMIT_RSS</code>	Taille maximale de l'ensemble des données se trouvant simultanément en mémoire (cette limite n'est pas utilisée par les noyaux 2.6).
<code>RLIMIT_NPROC</code>	Nombre maximal de processus simultanés pour l'utilisateur.
<code>RLIMIT_NOFILE</code>	Nombre maximal de fichiers ouverts simultanément par un utilisateur.
<code>RLIMIT_MEMLOCK</code>	Taille maximale de la zone verrouillée en mémoire centrale en empêchant son transfert dans le <code>swap</code> . On étudiera le détail de ce mécanisme avec les fonctions comme <code>mlock()</code> .

Ces limites sont surtout utiles pour empêcher un utilisateur de s'approprier trop de ressources au détriment des autres. Elles n'ont pas un grand intérêt pour le programmeur, à quelques exceptions près.

- La limite de temps CPU : pour des applications effectuant de lourds calculs ou des logiciels fonctionnant sans interruption pendant plusieurs mois, il est bon de vérifier que la limite de temps CPU n'est pas trop restrictive. Sinon, il faudra demander à l'administrateur système une augmentation de la limite stricte.
- La limite de taille de fichier : en général, cette limite est suffisamment grande pour les applications courantes. Elle peut cependant être contraignante, par exemple pour un système d'enregistrement sur une longue durée de données provenant d'un réseau. Il peut alors être nécessaire de scinder un gros fichier en plusieurs petits. Notons que cette limitation ne fonctionne pas sur tous les systèmes de fichiers et que d'autres limites peuvent être imposées, comme le système de quotas de disques, ou des limites sur le serveur d'une partition montée par *NFS*. Lorsqu'un processus tente de dépasser cette taille, il reçoit un signal `SIGXFSZ` si aucun octet n'a pu être écrit.

- La limite de taille des fichiers core : lorsqu'un programme a terminé sa phase de débogage et qu'il est livré aux utilisateurs, les éventuels fichiers core qui peuvent être créés lorsqu'il s'arrête anormalement ne présentent aucun intérêt pour l'utilisateur final. Ils laissent même une impression de finition négligée s'ils ont tendance à se multiplier dans tous les répertoires où l'utilisateur se trouve lorsqu'il lance l'application. Il est donc conseillé, au début du programme, de mettre à zéro cette limite lorsqu'on décide que le code est suffisamment stable pour être distribué aux clients.
- Le nombre maximal de processus simultanés : il est conseillé de mettre une valeur suffisamment haute (par exemple, 256) dans les fichiers d'initialisation du shell de connexion, mais de ne pas laisser la limite infinie. En effet, cela permet de prévenir des erreurs de programmation où on boucle sur un `fork()`. Au bout d'un certain temps, celui-ci échouera et, de cette manière, root pourra stopper tous les processus du groupe fautif depuis une autre console.

Pour accéder aux limites, les shells offrent une commande intégrée, qui nous suffira dans la plupart des cas puisque les limites sont transmises aux processus fils, donc aux applications lancées par le shell.

Attention

Le shell n'utilise pas toujours les mêmes unités que celles manipulées par le noyau. Ainsi la limite `FSIZE` (taille maximale de fichier) est exprimée en kilo-octets par le shell et en octets par le noyau.

Avec `tcsh`, la commande est `limit`. Si on l'invoque seule, elle affiche l'état des limites souples actuelles. Avec l'option `-h`, elle s'occupe des limites strictes (*hard*).

```
% limit
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    10240 kbytes
coredumpsize 0 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      1024
% limit -h
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    unlimited
coredumpsize unlimited
```

```
memoryuse      unlimited
vmemoryuse     unlimited
descriptors    1024
memorylocked   32 kbytes
maxproc        32621
%
```

Nous voyons une différence sur les limitations de taille de la pile et des fichiers core, qui doivent probablement être fixées dans un script d'initialisation de `tcsh`.

Si on veut modifier une limite, on indique le nom de la ressource, tel qu'il apparaît dans la liste précédente, suivi de la valeur désirée (en secondes pour le temps CPU, en Ko pour les autres limites). Si on ajoute l'option `-h`, on modifie la limite stricte.

```
% limit cputime 240
% limit coredumpsize 100
% limit
cputime      4:00
filesize     unlimited
datasize     unlimited
stacksize    10240 kbytes
coredumpsize 100 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      1024
% limit -h coredumpsize 1000
% limit -h coredumpsize 2000
limit: coredumpsize: ne peut positionner la limite matérielle
% limit -h
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    unlimited
coredumpsize 1000 kbytes
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  1024
memorylocked 32 kbytes
maxproc      32621
% limit coredumpsize 4000
limit: coredumpsize: ne peut positionner la limite
%
```

Cet exemple nous montre bien qu'on ne peut que réduire les limites *hard* (la tentative de ramener `coredumpsize` stricte à 2000 échoue). De même, une limite souple ne peut pas être programmée au-dessus de la limite stricte (idem pour `coredumpsize` à 4 000).

Sous `bash`, la commande est « `ulimit` ». Elle peut être suivie de `-S` (par défaut) ou de `-H`, pour indiquer une limite souple ou stricte, puis d'une lettre s'appliquant au type de ressource recherchée, et éventuellement de la nouvelle valeur. Le tableau suivant reprend les lettres correspondant aux limites.

Option	Signification
a	Toutes les limites (affichage seulement).
c	Taille d'un fichier <code>core</code> .
d	Taille du segment de données d'un processus.
f	Taille maximale d'un fichier.
m	Taille maximale des données se trouvant simultanément en mémoire.
s	Taille de la pile.
t	Temps processeur maximal.
n	Nombre de fichiers ouverts simultanément.
u	Nombre maximal de processus par utilisateur.
v	Quantité de mémoire virtuelle disponible.

Voici les mêmes manipulations que sous `tcsh`, effectuées cette fois-ci sous `bash`.

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size                (blocks, -f) unlimited
pending signals          (-i) 32621
max locked memory        (kbytes, -l) 32
max memory size          (kbytes, -m) unlimited
open files               (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority       (-r) 0
stack size               (kbytes, -s) 10240
cpu time                 (seconds, -t) unlimited
max user processes       (-u) 1024
virtual memory           (kbytes, -v) unlimited
file locks               (-x) unlimited
```



```
$ ulimit -Ha
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 32621
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) unlimited
cpu time                (seconds, -t) unlimited
max user processes      (-u) 32621
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
$
```

Manifestement, sur notre machine, les fichiers d'initialisation de bash (/etc/profile) et de tcsh (/etc/csh.cshrc) sont configurés avec les mêmes limitations pour les fichiers core.

```
$ ulimit -t 240
$ ulimit -c 100
$ ulimit -t -c
cpu time                (seconds, -t) 240
core file size          (blocks, -c) 100
$ ulimit -Hc 1000
$ ulimit -Hc 2000
bash: ulimit: core file size: cannot modify limit: Opération non permise
$ ulimit -Hc
1000
$ ulimit -c 4000
bash: ulimit: core file size: cannot modify limit: Opération non permise
$
```

On retrouve évidemment les mêmes restrictions quand on tente de relever une limite forte ou d'augmenter une limite souple au-dessus de la valeur stricte correspondante.

Nous allons à présent étudier comment consulter et modifier une ressource à partir d'un programme C. La fonction `getrlimit()` dont nous avons fourni le prototype plus haut nous remplit une structure `rlimit`, contenant les membres suivants.

Type	Nom	Signification
rlim_t	rlim_cur	Limite souple (valeur actuelle).
rlim_t	rlim_max	Limite stricte.

Le type de donnée `rlim_t` est défini sous Linux, avec la Glibc, sur architecture PC comme un long int. Sur certaines machines, il peut toutefois s'agir d'un « long long int ». On peut donc utiliser un format d'affichage « long long » pour être tranquille. Voici un programme qui affiche les limites strictes, suivies des limites souples entre parenthèses.

exemple-getrlimit.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>

void affichage_limite (char * libelle, int numero);

int main (void)
{
    affichage_limite("temps CPU en secondes      ", RLIMIT_CPU);
    affichage_limite("taille maxi d'un fichier   ", RLIMIT_FSIZE);
    affichage_limite("taille maxi zone de données", RLIMIT_DATA);
    affichage_limite("taille maxi de la pile      ", RLIMIT_STACK);
    affichage_limite("taille maxi fichier core   ", RLIMIT_CORE);
    affichage_limite("taille maxi résidente      ", RLIMIT_RSS);
    affichage_limite("nombre maxi de processus   ", RLIMIT_NPROC);
    affichage_limite("nombre de fichiers ouverts ", RLIMIT_NOFILE);
    affichage_limite("taille mémoire verrouillée ", RLIMIT_NOFILE);
    return EXIT_SUCCESS;
}

void affichage_limite (char * libelle, int numero)
{
    struct rlimit limite;
    if (getrlimit(numero, & limite) != 0) {
        fprintf(stdout, "Impossible d'accéder à la limite de %s\n",
            libelle);
        return;
    }
    fprintf(stdout, "Limite de %s : ", libelle);
    if (limite.rlim_max == RLIM_INFINITY)
        fprintf(stdout, "illimitée ");
    else
        fprintf(stdout, "%lld ", (long long int) (limite.rlim_max));
    if (limite.rlim_cur == RLIM_INFINITY)
        fprintf(stdout, "(illimitée)\n");
}
```

```
    else
        fprintf(stdout, "(%lld)\n", (long long int)(limite.rlim_cur));
}
```

Voici un exemple d'exécution, qui nous fournit les mêmes résultats que les précédents exemples, directement depuis le shell.

```
$ ./exemple-getrlimit
Limite de temps CPU en secondes      : illimitée (illimitée)
Limite de taille maxi d'un fichier  : illimitée (illimitée)
Limite de taille maxi zone de données : illimitée (illimitée)
Limite de taille maxi de la pile     : illimitée (10485760)
Limite de taille maxi fichier core   : illimitée (0)
Limite de taille maxi résidente      : illimitée (illimitée)
Limite de nombre maxi de processus   : 32621 (1024)
Limite de nombre de fichiers ouverts : 1024 (1024)
Limite de taille mémoire verrouillée : 1024 (1024)
$
```

En Python, la fonction `resource.getrlimit()` renvoie un tuple contenant la limite souple et la limite stricte.

```
exemple-rlimit.py :
#!/usr/bin/python
from __future__ import print_function
import resource

(soft, hard) = resource.getrlimit(resource.RLIMIT_NPROC)
print("Nb processes:", soft, "/", hard)
(soft, hard) = resource.getrlimit(resource.RLIMIT_NOFILE)
print("Nb files:", soft, "/", hard)
(soft, hard) = resource.getrlimit(resource.RLIMIT_DATA)
print("Data size:", soft, "/", hard)
(soft, hard) = resource.getrlimit(resource.RLIMIT_STACK)
print("Stack :", soft, "/", hard)
```

Voici un affichage de quelques limites, la valeur -1 représentant une limite infinie.

```
$ ./exemple-rlimit.py
Nb processes: 29955 / 29955
Nb files: 1024 / 4096
Data size: -1 / -1
Stack : 8388608 / -1
```

La fonction `setrlimit()` permet de fixer une limite, avec les restrictions que nous avons vues avec les shells. Le prototype de la fonction est :

```
int setrlimit (int ressource, struct rlimit * limite);
```

Attention

Lorsqu'on désire modifier par exemple la limite souple, il est nécessaire de lire auparavant l'ensemble de la structure `rlimit` correspondante afin d'avoir la bonne valeur pour la limite stricte. En effet, les deux champs doivent être correctement renseignés.

On remarquera que le fait de diminuer une limite stricte ne réduit pas nécessairement la limite souple correspondante. On peut temporairement se retrouver avec une limite souple supérieure à la limite stricte. En revanche, à la tentative suivante de modification de la limite souple, elle sera soumise à la nouvelle restriction.

Il faut également savoir que même `root` ne peut pas augmenter le nombre de fichiers ouverts simultanément (`RLIMIT_NOFILE`) au-dessus de la limite imposée par le noyau (`NR_OPEN` défini dans `<linux/limits.h>`).

Conclusion

Lorsqu'un processus doit se mettre en attente pendant une période déterminée, il existe plusieurs méthodes de sommeil avec des durées plus ou moins précises, que nous avons étudiées en détail dans ce chapitre.

Nous avons vu que la précision des sommeils et des temporisations est limitée par l'ordonnancement multitâche. Certaines améliorations seront possibles en utilisant des priorités temps réel comme nous le verrons dans le prochain chapitre. Lorsqu'il est nécessaire d'avoir une précision sensiblement meilleure, il faut se tourner vers des versions de Linux modifiées pour le temps réel strict (*hard realtime*) comme `PREEMPT_RT`, `Xenomai` ou `RTAI`.

Nous avons également observé les fonctions permettant d'obtenir des informations sur l'utilisation des ressources système par un processus et de limiter cet accès aux ressources afin de ne pas léser les autres utilisateurs.

10

Ordonnancements sous Linux

Dans ce chapitre, nous allons approcher les mécanismes sous-jacents lors de l'exécution des programmes. Nous étudierons tout d'abord les différents états dans lesquels une tâche peut se trouver, ainsi que l'influence du noyau sur leurs transitions. Nous analyserons ensuite les méthodes simples permettant de modifier la priorité d'une tâche par rapport aux autres. Nous verrons également comment un processus peut choisir le CPU (ou le cœur de processeur) sur lequel il fonctionnera.

En toute rigueur, les concepts abordés dans ce chapitre s'appliquent uniquement aux tâches, c'est-à-dire aux threads. Toutefois, lorsque nous serons dans le cas d'un processus unithread, je prendrai la liberté, pour simplifier l'expression, de parler de « l'ordonnancement du processus » et non de « l'ordonnancement du thread principal du processus ».

États d'une tâche

Indépendamment de toute mécanique d'ordonnancement, une tâche peut se trouver dans un certain nombre d'états différents, en fonction de ses activités. Ces états peuvent être examinés à l'aide de la commande `ps` aux (`ps` pour examiner les différents threads des processus) ou en regardant le contenu du pseudo-fichier `/proc/<pid>/status`. Ce dernier contient en effet une ligne `State:...` indiquant l'état du processus. Nous utiliserons de préférence cette seconde méthode, car elle permet d'éviter de lancer un processus supplémentaire (`ps`), qui est le seul à être réellement actif au moment de l'invocation, sur une machine uniprocasseur du moins.

Le tableau ci-après montre les différents états d'une tâche.

État	Anglais	Signification
Exécution	<i>Running</i> (R)	La tâche est en cours de fonctionnement, elle effectue un travail actif.
Sommeil	<i>Sleeping</i> (S)	La tâche est en attente d'un événement extérieur. Elle se met en sommeil. Le sommeil est dit « interruptible » car la tâche peut-être réveillée prématurément si elle reçoit un signal.
Sommeil ininterruptible	<i>Down</i> (D)	La tâche est en attente absolue d'un élément extérieur. Rien ne pourra la réveiller avant que cet événement se produise.
Arrêt	<i>Stopped</i> (T)	La tâche a été temporairement arrêtée par un signal. Elle ne s'exécute plus et ne réagira qu'à un signal de redémarrage.
Zombie	<i>Zombie</i> (Z)	La tâche s'est terminée, mais on n'a pas encore lu son statut de terminaison.

Il n'y a pas grand-chose à ajouter sur l'état *Running*. La tâche s'exécute normalement, elle a accès au processeur de la machine et avance dans son code exécutable.

Les tâches en sommeil sont généralement en attente d'une ressource ou d'un événement extérieur. Dans la plupart des cas, elles attendent le résultat d'une opération d'entrée-sortie. Lorsqu'un programme veut écrire sur son terminal de sortie standard, le noyau prend le contrôle (à travers l'appel système `write()` sous-jacent) et assure l'écriture. Toutefois, la durée de réaction d'un terminal est relativement longue. Aussi le noyau bascule-t-il le processus en sommeil, en attente de la fin de l'écriture. Une fois que celle-ci est terminée, le processus se réveille et reprend son exécution.

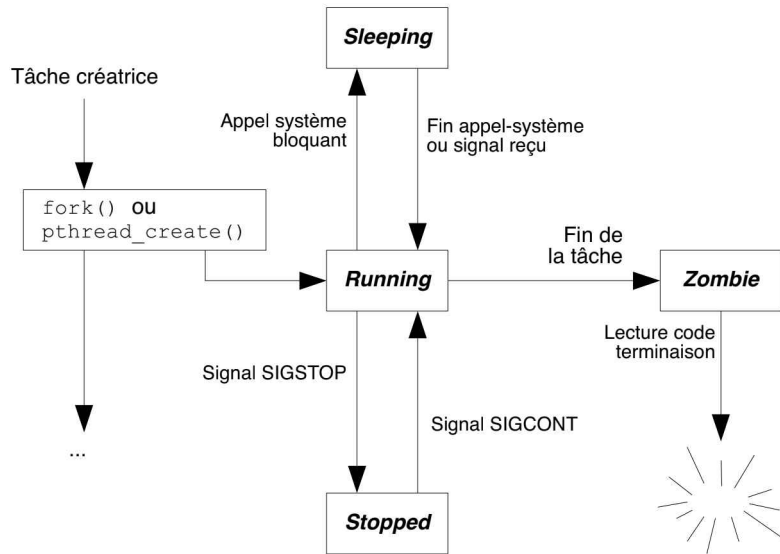
Notons enfin qu'il existe deux types de sommeil : *interruptible* et *ininterruptible*. Dans le premier état, la tâche peut être réveillée prématurément par l'arrivée d'un signal. Dans le second cas, la tâche ne peut être réveillée que lorsque l'événement attendu se produit. Le sommeil ininterruptible est rare mais peut s'observer parfois, notamment lorsque des problèmes de montage de systèmes de fichiers (par exemple par protocole *NFS*) entrent en considération. On peut ainsi trouver parfois un processus `mount` bloqué dans un sommeil ininterruptible suite à un problème matériel ; il ne faut pas s'étonner qu'il ne réponde à aucun signal – y compris `SIGKILL` (9) – et ne disparaîtra pas avant le prochain *reboot* du système.

Lorsqu'une tâche reçoit un signal `SIGSTOP`, elle est arrêtée temporairement mais pas terminée définitivement. Ce signal peut être engendré par l'utilisateur (avec `/bin/kill` par exemple), par le terminal (généralement avec la touche *Contrôle-Z*), ou encore par un débogueur comme `gdb` qui interrompt la tâche pour l'exécuter pas à pas. Le signal `SIGCONT` permet à la tâche de redémarrer ; il est déclenché soit par `/bin/kill`, soit par le shell (commandes internes `bg` ou `fg`), ou encore par le débogueur pour reprendre l'exécution.

Enfin, une tâche qui se termine dispose d'un statut de terminaison. Ce dernier indique les conditions de terminaison du thread ou du processus : s'il s'est terminé volontairement – avec `pthread_exit()`, `exit()` ou `return` depuis sa fonction principale – ou prématurément, par une annulation d'un thread ou un signal tuant un processus. Dans le premier cas, on peut également connaître la valeur fournie à `pthread_exit()`, `exit()` ou `return`. Dans le second on peut retrouver le numéro du signal ainsi que la présence éventuelle d'un fichier core image de la mémoire virtuelle du processus. Nous avons déjà abordé ces notions avec l'appel système `waitpid()` au chapitre 5 et la programmation multithread aux chapitres 6 et 7.

Tant que l'on n'a pas lu cette valeur de terminaison, la tâche reste dans un état intermédiaire entre la vie et la mort, toutes ses ressources ont été libérées, mais elle conserve une place dans la table des tâches du système. On dit qu'elle est à l'état *Zombie*. Un processus zombie peut conserver cet état jusqu'au *reboot* du système si son père ne lit jamais son statut de terminaison. Un thread zombie peut persister jusqu'à la terminaison de son processus.

Figure 10–1
États successifs d'une tâche



Dans l'exemple suivant, nous allons faire passer un processus – et son fils – par tous ces stades. Tout d'abord, notre processus va consulter son propre état dans le pseudo-système de fichiers `/proc`, puis il va se scinder avec `fork()` avant de s'endormir pendant quelques secondes (en attente donc d'un signal de réveil). Son fils profitera de ce laps de temps pour afficher l'état du père, puis se terminera immédiatement. À son réveil, le processus père examinera l'état de son fils avant et après avoir lu le code de retour. Ensuite, le processus se met en sommeil, en attente d'une saisie de caractères.

```
exemple-status.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

void affiche_status (pid_t pid)
{
    FILE * fp;
    char  chaine[80];
    sprintf(chaine, "/proc/%ld/status", (long) pid);
    if ((fp = fopen(chaine, "r")) == NULL) {
        fprintf(stdout, "Processus inexistant\n");
        return;
    }
    while (fgets(chaine, 80, fp) != NULL)
        if (strncmp(chaine, "State", 5) == 0) {
            fputs(chaine, stdout);
            break;
        }
    fclose(fp);
}

int main (void)
{
    pid_t pid;
    char chaine[5];

    fprintf(stdout, "PID = %ld\n", (long) getpid());
    fprintf(stdout, "Etat attendu = Running \n");
    affiche_status(getpid());
    if ((pid = fork()) == -1) {
        perror("fork ()");
        exit(EXIT_FAILURE);
    }
    if (pid != 0) {
        sleep(5);
        fprintf(stdout, "Consultation de l'etat de mon fils...\n");
        fprintf(stdout, "Etat attendu = Zombie \n");
        affiche_status(pid);
        fprintf(stdout, "Pere : Lecture code retour du fils...\n");
        wait(NULL);
        fprintf(stdout, "Consultation de l'etat de mon fils...\n");
        fprintf(stdout, "Etat attendu = inexistant\n");
        affiche_status(pid);
    } else {
        fprintf(stdout, "Fils : consultation de l'etat du pere...\n");
        fprintf(stdout, "Etat attendu = Sleeping \n");
    }
}
```



```
    affiche_status(getppid());  
    fprintf(stdout, "Fils : Je me termine \n");  
    exit(EXIT_SUCCESS);  
}  
fprintf(stdout, "Attente de saisie de chaine \n");  
fgets(chaine, 5, stdin);  
exit(EXIT_SUCCESS);  
}
```

Nous allons déjà exécuter cet exemple, jusqu'à la saisie de chaîne, en observant les différents états.

```
$ ./exemple-status  
PID = 3415  
Etat attendu = Running  
State: R (running)  
Fils : consultation de l'etat du pere...  
Etat attendu = Sleeping  
State: S (sleeping)  
Fils : Je me termine  
Consultation de l'etat de mon fils...  
Etat attendu = Zombie  
State: Z (zombie)  
Pere : Lecture code retour du fils...  
Consultation de l'etat de mon fils...  
Etat attendu = inexistant  
Processus inexistant  
Attente de saisie de chaine
```

Le processus est donc en attente de saisie. Nous allons à ce moment lancer gdb depuis une autre console et utiliser la commande attach de gdb pour déboguer le processus en cours de fonctionnement. Nous pourrions alors lancer la commande « cat /proc/3331/status » à l'aide de l'instruction shell de gdb :

```
$ gdb exemple_status  
GNU gdb (GDB)  
[...]  
(gdb) attach 3415  
Attaching to program: exemple-status, process 3415  
[...]  
(gdb) shell cat /proc/3415/status | grep State  
State:t (tracing stop)  
(gdb) detach
```

```
Detaching from program: exemple_status, process 3415
(gdb) quit
$
```

Cela nous permet de vérifier que le processus en cours de débogage est bien arrêté entre les exécutions pas à pas.

Fonctionnement multitâche, priorités

Jusqu'à présent, nous avons considéré que la tâche est seule active sur un processeur qui lui est attribué. Cela peut être le cas sur une machine multiprocesseur (*SMP*) ou multicœur peu chargée, mais c'est rare. Sur un système multitâche, l'accès au processeur est une ressource comme les autres (mémoire, disque, terminaux, imprimantes...) qui doit se partager entre les threads concurrents.

Dans le cadre de ce paragraphe, nous considérerons principalement le cas des machines uniprocresseurs. On pourra presque toujours généraliser notre propos aux ordinateurs multiprocresseurs dès que le nombre de processus concurrents sur le système augmentera.

Le noyau est chargé d'assurer la commutation entre les différents programmes pour leur offrir l'accès au processeur. C'est très facile lorsqu'une tâche s'endort. N'oublions pas que toutes les transitions entre états que nous avons vues plus haut se passent toujours par l'intermédiaire d'un appel système, d'une interruption matérielle ou de l'arrivée d'un signal. Aussi, le noyau a toujours le contrôle total de l'état des tâches. Lorsque l'une d'elles s'endort, en attente d'une saisie de l'opérateur par exemple, le noyau peut alors élire un autre thread pour lui attribuer l'accès au processeur. Le temps CPU disponible est partagé de manière assez équitable entre les différentes tâches en attente ; on parle d'*ordonnanceur temps-partagé*. Le but de ce type d'ordonancement est d'obtenir un comportement global du système le plus fluide possible.

Seulement ce n'est pas encore suffisant, on ne peut guère compter sur la bonne volonté de chaque tâche pour s'endormir régulièrement afin de laisser un peu de temps CPU disponible pour les autres. La moindre erreur de programmation du type :

```
while (1);
/* Exécuter un travail */
```

bloquerait totalement le système (notez le « ; » en trop à la suite du `while()`, qui engendre une boucle infinie).

Pour éviter ce problème, le noyau doit interrompre au bout d'un certain temps une tâche qui s'exécute sans avoir besoin de s'endormir (en effectuant des calculs par

exemple), afin qu'elles cèdent la place à une autre tâche. On dit que le noyau réalise une *préemption* de la tâche, ce qui a donné naissance au terme de *multitâche préemptif*.

Avec ce mécanisme, le noyau peut intervenir lorsqu'un thread dépasse le quantum de temps qui lui est imparti. Le thread passe alors à l'état *Prêt*. Un thread prêt est donc simplement un thread en cours d'exécution qui a été suspendu par le noyau et qui reprendra son travail dès que celui-ci lui réaffectera le processeur.

Au niveau de la commande `ps` ou du pseudo-fichier `/proc/<pid>/status`, il n'y a aucune différence entre une tâche effectivement en exécution et une tâche prête. Elles sont toutes deux indiquées par la lettre `R`. Cela explique pourquoi la commande `ps` `maux` présente parfois une liste contenant simultanément plusieurs threads à l'état `R`, même sur une machine uniprocasseur.

Un corollaire de ce mécanisme est qu'un thread qui se réveille, par exemple à cause d'une opération d'entrée-sortie terminée ou de l'arrivée d'un signal, ne passe pas directement de l'état *Sommeil* à l'état *Exécution*, mais passe par un état transitoire *Prêt*, et c'est l'ordonnanceur qui décidera du véritable passage en *Exécution*.

Une application n'a donc habituellement aucune raison de se soucier de l'ordonnement assuré par le noyau, tout se passe de manière totalement transparente. Un thread a toujours l'impression d'être en cours d'exécution. Malgré tout, il faut être conscient qu'une application qui effectue de larges plages d'opérations sans réclamer d'entrée-sortie emploie beaucoup la seule ressource qui soit vraiment indispensable pour tous les programmes : le CPU. Cette application pénalisera donc les autres logiciels qui font un usage plus raisonnable du processeur.

Le noyau utilise, pour pallier ce problème, le principe de priorité double. Une valeur *statique* de priorité est donnée à tout processus dès son démarrage et peut partiellement être corrigée par un appel système approprié. L'ordonnanceur se sert de cette valeur statique pour calculer une nouvelle valeur, nommée priorité *dynamique*, et qui est remise à jour à chaque fois qu'une tâche est traitée par l'ordonnanceur. Cette priorité dynamique est entièrement interne au noyau et ne peut pas être modifiée. Plus une tâche utilise le temps CPU qui lui est imparti, plus le noyau diminue sa priorité dynamique. Au contraire, plus le thread rend vite la main lorsqu'on l'exécute, plus le noyau augmente sa priorité. Avec cette politique, les tâches qui exploitent peu le processeur – déclenchant une opération d'entrée-sortie et s'endormant aussitôt en attente du résultat – passeront beaucoup plus rapidement de l'état *Prêt* à l'*Exécution* effective que les tâches qui grignotent tous les cycles CPU qu'on leur offre, sans jamais redonner la main intentionnellement.

Cette organisation de l'ordonnanceur permet d'améliorer en partie la fluidité du système. Malgré tout, lorsqu'on lance sur une machine uniprocasseur plusieurs processus qui dévorent les cycles CPU en boucle, une nette diminution des performances du

système est sensible. Pourtant, une grande partie des programmes qui font beaucoup de calculs et peu d'entrées-sorties ne présentent pas de caractère d'urgence. En voici quelques exemples.

- Une application reçoit des informations numériques, les affiche et les rediffuse. Elle exécute régulièrement des opérations de calcul statistique dont le résultat n'est imprimé qu'une fois par mois.
- Un système d'enregistrement collecte des blocs de données, les regroupe en petits fichiers correspondant chacun à une heure de trafic, puis les transfère sur un répertoire accessible en FTP anonyme, où d'autres machines viendront les récupérer. Pour diminuer le volume des transferts ultérieurs, on comprime les fichiers en invoquant `gzip` par exemple.
- Une application de création d'image numérique permet d'utiliser un modèleur affichant la scène en préparation sous forme de squelette adapté à la définition de l'écran. Avant le transfert vers le système de photocomposition définitif, l'image en très haute résolution est calculée par un processus complexe de calcul (*ray-tracing* par exemple).

Dans tous ces exemples, on remarque qu'une partie du travail est fortement consommatrice de cycles CPU (le calcul statistique, la compression de données, le traitement d'image) alors que le résultat n'est pas indispensable immédiatement. On ne peut toutefois pas se reposer entièrement sur le noyau pour « freiner » systématiquement ce genre d'opérations, car il existe aussi de nombreux cas où on attend impatiemment le résultat d'un travail intense du processeur. Le meilleur exemple pour le développeur est probablement la compilation d'un logiciel relativement conséquent, et la surdose de caféine ou de nicotine qui finit par meubler l'attente obligatoire...

Il est donc nécessaire de pouvoir donner au noyau une indication de la priorité qu'on affecte à tel ou tel travail. Lorsque plusieurs threads seront prêts, le noyau choisira d'abord celui dont la priorité dynamique est la plus importante. Lors du calcul de la priorité dynamique, l'ordonnanceur utilise la priorité statique conjointement à d'autres facteurs, comme le fait que le thread ait relâché le processeur avant l'expiration de son délai, l'emplacement réel du thread – sur les systèmes multiprocesseurs – ou la disponibilité immédiate de son espace d'adressage complet.

Plus un thread a une priorité dynamique élevée, plus la tranche de temps qu'on lui allouera sera longue. C'est un moyen de punir les programmes qui bouclent, en les laissant travailler quand même, mais sans trop perturber les autres applications. Lorsqu'un thread a consommé tous ses cycles, il ne sera réélu pour l'accès au processeur que dans le cas où aucun autre thread plus courtois n'est prêt.

Lorsqu'un processus désire influencer sur sa propre priorité statique, il peut utiliser l'appel système `nice()`. On indique à celui-ci la « gentillesse » dont le processus appe-

lant désire faire preuve. La déclaration de cette fonction se trouve dans le module `os` en Python, et dans `<unistd.h>` pour le C :

```
| int nice (int increment);
```

La valeur transmise est ajoutée à notre gentillesse vis-à-vis des autres processus. Cela signifie qu'un incrément positif diminue la priorité du processus, alors qu'un incrément négatif augmente sa priorité. Seul un processus ayant l'UID effectif de root ou la capacité `CAP_SYS_NICE` peut diminuer sa gentillesse. La plage de valeur utilisée en interne par l'ordonnanceur pour les priorités statiques s'étale de 0 à 40. Toutefois, par convention on présente la gentillesse d'un processus sur une échelle allant de -20 (processus très égoïste) à +20, la valeur 0 étant celle par défaut. Un utilisateur normal ne peut donc avoir accès qu'à la plage allant de 0 à 20.

Dans l'exemple suivant, le programme va lancer cinq processus fils, chacun d'eux prenant une valeur de gentillesse différente et se mettant à boucler sur un comptage. Le processus père les laisse travailler pendant 5 et les arrête. Pour synchroniser le début et la fin du comptage pour les différents fils, nous utilisons un signal émis par le père à destination du groupe de processus. Pour être sûr que les fils ont tous démarré avant d'envoyer le signal, le père respecte un petit sommeil d'une seconde ; c'est une méthode imparfaite, mais qui fonctionne sur un système peu chargé.

```
| exemple-nice.c :
| #define _GNU_SOURCE
| #include <signal.h>
| #include <stdio.h>
| #include <stdlib.h>
| #include <sys/wait.h>
| #include <unistd.h>
|
| volatile long compteur = 0;
| static int gentillesse;
|
| void gestionnaire (int numero)
| {
|     if (compteur != 0) {
|         fprintf(stdout, "Fils %ld (nice %2d) Compteur = %9ld\n",
|                 (long) getpid(), gentillesse, compteur);
|         exit(EXIT_SUCCESS);
|     }
| }
|
| #define NB_FILS 5
```

```
int main (void)
{
    pid_t    pid;
    int      fils;

    /* Création d'un nouveau groupe de processus */
    setpgid(0, 0);

    signal(SIGUSR1, gestionnaire);
    for (fils = 0; fils < NB_FILS; fils++) {
        if ((pid = fork()) < 0) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pid != 0)
            continue;
        gentillesse = fils * (20 / (NB_FILS - 1));
        if (nice(gentillesse) < 0) {
            perror("nice");
            exit(EXIT_FAILURE);
        }
        /* Attente signal de démarrage */
        pause();
        /* Comptage */
        while (1)
            compteur++;
    }
    /* Processus père */
    signal(SIGUSR1, SIG_IGN);
    sleep(1);
    kill(-getpgid(0), SIGUSR1);
    sleep(5);
    kill(-getpgid(0), SIGUSR1);
    while (wait(NULL) > 0)
        ;
    exit(EXIT_SUCCESS);
}
```

L'exécution montre bien que les processus ont eu un accès différent au processeur, inversement proportionnel à leur indice de gentillesse.

```
$ ./exemple-nice
Fils 10709 (nice 20) Compteur = 13934355
Fils 10706 (nice 5) Compteur = 313725172
Fils 10707 (nice 10) Compteur = 103045984
Fils 10705 (nice 0) Compteur = 961402650
Fils 10708 (nice 15) Compteur = 33756480
$
```

On peut obtenir la même chose avec le script Python suivant :

```
exemple-nice.py :
#!/usr/bin/python
from __future__ import print_function
import os
import signal
import sys
import time

count = 0
child = 0
nice = 0

def handler_usr1(signum, stack):
    print("Child {}, nice={:2}, count={}"
          .format(child, nice, count))

os.setpgid(0, 0)

for child in range(0, 5):
    pid = os.fork()
    if (pid != 0):
        continue
    # Child branch
    signal.signal(signal.SIGUSR1, handler_usr1)
    nice = child * 4
    os.nice(nice)
    signal.pause()
    while True:
        count = count + 1

# Parent branch
signal.signal(signal.SIGUSR1, signal.SIG_IGN)
time.sleep(1)
os.kill(-os.getpgid(0), signal.SIGUSR1)
time.sleep(5)
os.kill(-os.getpgid(0), signal.SIGUSR1)
time.sleep(1)
os.kill(-os.getpgid(0), signal.SIGKILL)
```

Le résultat est comparable :

```
$ ./exemple-nice.py
Child 4, nice=16, count=0
Child 3, nice=12, count=0
Child 2, nice= 8, count=0
Child 1, nice= 4, count=0
```

```
Child 0, nice= 0, count=0
Child 1, nice= 4, count=6960702
Child 0, nice= 0, count=16870305
Child 2, nice= 8, count=2841699
Child 3, nice=12, count=1157554
Child 4, nice=16, count=490630
```

Cela intéressera donc le programmeur dont l'application déclenche plusieurs processus, certains effectuant beaucoup de calculs peu urgents. De telles tâches auront intérêt à diminuer leur priorité (augmenter leur gentillesse) pour conserver un fonctionnement plus fluide au système.

Modification de la priorité d'un autre processus

Pouvoir modifier sa propre priorité est une bonne chose, mais il y a de nombreux cas où on aimerait changer la priorité d'un autre processus déjà en exécution. L'exemple est fourni par l'utilitaire `renice`, qui permet de diminuer la priorité d'un processus s'il charge trop lourdement le processeur ou au contraire de l'augmenter (à condition d'avoir les privilèges nécessaires). Ces opérations sont possibles grâce à deux appels système `getpriority()` et `setpriority()` qui sont déclarés dans `<sys/wait.h>` :

```
int getpriority (int classe, int identifiant);
int setpriority (int classe, int identifiant, int valeur);
```

Ces deux appels système ne travaillent pas obligatoirement sur un processus particulier, mais peuvent agir sur un groupe de processus ou sur tous les processus appartenant à un utilisateur donné.

Les fonctions `os.getpriority()` et `os.setpriority()` donnent accès aux mêmes fonctionnalités en Python. En fonction de la classe indiquée en premier argument, l'identifiant fourni en second est interprété différemment.

Valeur de classe	Type d'identifiant
<code>PRIO_PROCESS</code>	PID du processus visé.
<code>PRIO_PGRP</code>	PGID du groupe de processus concerné.
<code>PRIO_USER</code>	UID de l'utilisateur dont on vise tous les processus.

La valeur de retour de `getpriority()` correspond à la priorité statique du processus, qui s'étend dans l'intervalle `PRIO_MIN` à `PRIO_MAX`, qui valent typiquement -20 et 20. Aussi lorsque `getpriority()` renvoie -1, on ne peut être sûr qu'il s'agit d'une erreur qu'à condition d'avoir mis `errno` à 0 avant l'appel, et en vérifiant alors son état.

Lorsque plusieurs processus sont concernés par `getpriority()`, la valeur renvoyée se rapporte à la plus petite de toutes leurs priorités. Les valeurs de priorité considérées ici représentent en réalité des quantités de gentillesse. Plus la valeur est petite (proche de -20), plus le processus est prioritaire. L'exemple suivant va nous permettre de consulter les priorités.

```
exemple-getpriority.c :
#include <ctype.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

void syntaxe (char * nom)
{
    fprintf(stderr, "Syntaxe : %s <classe> <identifiant> \n", nom);
    fprintf(stderr, "      <classe> = P (PID)\n");
    fprintf(stderr, "              G (PGID)\n");
    fprintf(stderr, "              U (UID)\n");
    exit(EXIT_FAILURE);
}

int main (int argc, char * argv[])
{
    int classe;
    int identifiant;
    int priorite;

    if (argc != 3)
        syntaxe(argv[0]);
    if (toupper(argv[1][0]) == 'P')
        classe = PRIO_PROCESS;
    else if (toupper(argv[1][0]) == 'G')
        classe = PRIO_PGRP;
    else if (toupper(argv[1][0]) == 'U')
        classe = PRIO_USER;
    else
        syntaxe(argv[0]);
    if (sscanf(argv[2], "%d", & identifiant) != 1)
        syntaxe(argv[0]);
    errno = 0;
    priorite = getpriority(classe, identifiant);
    if ((priorite == -1) && (errno != 0))
        perror(argv[2]);
    else
        fprintf(stderr, "%d : %d\n", identifiant, priorite);
    return EXIT_SUCCESS;
}
```

L'exemple d'exécution ci-dessous présente un intérêt limité. Il est plus utile de comparer les valeurs obtenues par ce programme et celles qu'on visualise avec la commande `ps`, ou encore mieux avec le logiciel `top`.

```
$ ./exemple-getpriority P 2982
2982 : 15
$ ./exemple-getpriority U 500
500 : 0
$ ./exemple-getpriority P 6
6 : -20
$
```

`setpriority()` fonctionne de manière symétrique, en fixant la nouvelle priorité statique du ou des processus indiqués. Bien sûr, des restrictions s'appliquent en ce qui concerne les droits d'accès au processus, et seul `root` (ou un programme ayant le privilège `CAP_SYS_NICE`) peut rendre un processus plus prioritaire.

Les fonctions `nice()`, `getpriority()` ou `setpriority()` ne sont pas définies par `Ansi C` (qui n'inclut pas le concept de multitâche) mais elles sont indiquées dans `SUSv4`. Toutefois, si ces appels système peuvent suffire pour de petites opérations de configuration administrative (accélérer un calcul par rapport à un autre, ou diminuer la priorité des jobs ne présentant pas de caractère d'urgence), ils sont largement insuffisants dès qu'on a réellement besoin de configurer le comportement de l'ordonnanceur en détail.

Pour cela nous devons utiliser les mécanismes temps réel que nous allons étudier plus loin.

Systèmes multiprocesseurs, migrations

Les processeurs actuels évoluent de plus en plus vers un modèle multicœur. En outre, pour certains systèmes gourmands en temps CPU, peuvent se voir doter de plusieurs processeurs symétriques (*SMP*). Il est donc très intéressant pour une application de profiter du parallélisme pour accélérer son exécution.

Attention

Dans ce chapitre nous considérerons que les processeurs multicœurs offrent un comportement identique aux systèmes multiprocesseurs. Lorsque nous parlerons de l'exécution d'une tâche sur un processeur, cela s'appliquera également à l'exécution sur un cœur.

Le noyau Linux gère parfaitement les systèmes multiprocesseurs – surtout depuis le noyau 2.6. Il est capable de déplacer une tâche d'un processeur à l'autre pour équilib-

brer la charge du système. Ce rôle est assuré par les threads du noyau `migration/0`, `migration/1`, etc. (un thread par processeur). Toutefois certaines applications trouveront plus intéressant de gérer elles-mêmes le positionnement des tâches sur les processeurs. Pour cela quelques fonctions de bibliothèque existent.

Notons bien que toutes ces fonctions de bibliothèque sont des extensions GNU qui s'appuient sur des appels système spécifiques à Linux. Leur utilisation n'est donc pas portable sur d'autres systèmes d'exploitation. Pour pouvoir utiliser ces routines on définira la constante symbolique :

```
#define _GNU_SOURCE
```

avant l'inclusion du fichier d'en-tête `<sched.h>`.

Consultation du processeur utilisé

La première information intéressante est de savoir sur quel processeur une tâche s'exécute. Pour cela, nous disposons de la fonction :

```
int sched_getcpu (void);
```

Cette routine renvoie le numéro de processeur (commençant à 0) de la tâche appelante. En voici un exemple d'utilisation, dans lequel nous passons en argument sur la ligne de commande le nombre de threads à exécuter en parallèle.

```
exemple-sched-getcpu.c :
#define GNU_SOURCE
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fonction (void * arg);

int main(int argc, char * argv[])
{
    long n;
    pthread_t thr;
    int nb_threads;

    if ((argc != 2)
        || (sscanf(argv[1], "%d", & nb_threads) != 1)) {
```

```
    fprintf(stderr, "usage : %s nb_threads\n", argv[0]);
    exit(1);
}
for (n = 0; n < nb_threads; n++)
    pthread_create(& thr, NULL, fonction, (void *) n);
pthread_exit(NULL);
}

void * fonction (void * arg)
{
    long num = (long) arg;
    int precedent = -1;
    int cpu;
    while(1) {
        cpu = sched_getcpu();
        if (cpu != precedent) {
            fprintf(stdout, "[%ld] %d\n", num, cpu);
            precedent = cpu;
        }
    }
}
```

Voici une exécution sur un processeur *dual-core* où l'on voit les deux threads migrer entre les deux cœurs en fonction de la charge du système.

```
$ ./exemple-sched-getcpu 2
[0] 1
[1] 1
[1] 0
[0] 0
[1] 1
[1] 0
[0] 1
^C
$
```

À l'heure actuelle, la bibliothèque Python n'implémente pas la fonction `sched_getcpu()`.

Choix des processeurs autorisés pour une tâche

Nous pouvons choisir sur quel processeur une tâche s'exécute, ou plutôt sur quel ensemble de processeurs elle est autorisée à s'exécuter. C'est ce que l'on nomme l'affinité de la tâche pour un ensemble de processeur. Pour cela nous manipulerons des

variables de type `cpu_set_t` qui représentent des ensembles de processeurs. Pour renseigner ces variables, on emploie les macros suivantes :

```
CPU_ZERO(cpu_set_t * ensemble);
CPU_SET (int cpu, cpu_set_t * ensemble);
CPU_CLR (int cpu, cpu_set_t * ensemble);
CPU_ISSET (int cpu, cpu_set_t * ensemble);
```

Et l'on peut utiliser les fonctions suivantes :

```
int sched_setaffinity (pid_t pid, size_t taille_cpuset,
                      cpu_set_t * ensemble);
int sched_getaffinity (pid_t pid, size_t taille_cpuset,
                      cpu_set_t * ensemble);
```

pour fixer ou consulter l'affinité du thread principal d'un processus, ou encore

```
int pthread_setaffinity_np (pthread_t thr, size_t taille_cpuset,
                           const cpu_set_t * ensemble);
int pthread_getaffinity_np (pthread_t thr, size_t taille_cpuset,
                           cpu_set_t * ensemble);
```

L'argument `taille_cpuset` doit être renseigné avec la valeur `sizeof(cpu_set_t)`.

Dans cet exemple, nous avons repris la même fonction de thread que dans l'exemple précédent, et modifié la fonction `main()` pour tous les threads aient la même affinité, uniquement sur le processeur mentionné en second argument.

```
exemple-pthread-setaffinity.c :
[...]
int main(int argc, char * argv[])
{
    long n;
    int num_cpu;
    int nb_threads;
    pthread_t thr;
    cpu_set_t cpuset;

    if ((argc != 3)
        || (sscanf(argv[1], "%d", & nb_threads) != 1)
        || (sscanf(argv[2], "%d", & num_cpu) != 1)) {
        fprintf(stderr, "usage : %s nb_threads num_cpu\n", argv[0]);
        exit(1);
    }
    CPU_ZERO(& cpuset);
    CPU_SET(num_cpu, & cpuset);
    for (n = 0; n < nb_threads; n++) {
```

```
    pthread_create(& thr, NULL, fonction, (void *) n);  
    pthread_setaffinity_np(thr, sizeof(cpu_set_t), & cpuset);  
}  
pthread_exit(NULL);  
}  
[...]
```

Cette fois, tous les threads s'exécutent bien sur le CPU demandé, sans migration.

```
$ ./exemple-pthread-setaffinity 5 0  
[0] 0  
[1] 0  
[2] 0  
[4] 0  
[3] 0  
^C  
$ ./exemple-pthread-setaffinity 5 1  
[0] 1  
[1] 1  
[3] 1  
[2] 1  
[4] 1  
^C  
$  
$ ./exemple-pthread-setaffinity 5 2  
[0] 0  
[1] 1  
[2] 0  
[3] 1  
[4] 0  
[2] 1  
^C  
$
```

L'appel `pthread_setaffinity_np()` avec un numéro de CPU trop grand – 2, sur notre ordinateur *dual-core* – échoue et n'a pas d'effet sur l'affinité des threads.

Les fonctions `os.sched_getaffinity()` et `os.sched_setaffinity()` ne sont disponibles qu'à partir de la version 3 de la bibliothèque de Python. Dans l'exemple suivant, un processus parcourt inlassablement les CPU disponibles en effectuant à chaque fois une boucle active de quelques secondes.

```
exemple-setaffinity.py :  
#!/usr/bin/python3  
import os  
  
nb_cpu = os.sysconf(os.sysconf_names['SC_NPROCESSORS_ONLN'])  
while True:  
    for i in range(0, nb_cpu):  
        cpuset = [i]  
        os.sched_setaffinity(0, cpuset)  
        for j in range(0,100000000):  
            pass
```

On remarquera comment on peut obtenir le nombre de CPU disponibles avec `sysconf()`.

Lorsque le programme s'exécute, il est intéressant de lancer un moniteur système comme l'outil `gkrellm`, pour voir la boucle active charger les différents CPU l'un après l'autre.

On notera qu'il existe un utilitaire en ligne de commande permettant de placer un processus sur un ensemble de CPU : `taskset`. Par exemple la commande suivante lance notre application `exemple-sched-getcpu` en garantissant qu'elle s'exécutera entièrement sur le processeur 0.

```
$ taskset -c 0 ./exemple-sched-getcpu 5  
[3] 0  
[4] 0  
[2] 0  
[1] 0  
[0] 0  
^C  
$
```

Il n'y a plus de migration dans ce cas. Il peut être intéressant pour une application complexe de définir précisément un ensemble de CPU sur lesquels des threads de calculs peuvent s'exécuter (en utilisant éventuellement les ordonnancements temps réel que nous allons voir ci-après) tout en conservant les threads interactifs (réseau, interface graphique, etc.) sur un processeur différent afin d'éviter de pénaliser les temps de réponse vis-à-vis de l'utilisateur.

Ordonnancements temps réel

Nous avons observé plus haut les limitations de l'ordonnancement temps-partagé. Ce type d'ordonnancement est présent sur les systèmes Unix depuis leur origine. L'article célèbre de présentation de ce système d'exploitation, en 1974, s'appelait ainsi « *The Unix Time-Sharing System* », écrit par Dennis M. Ritchie et Ken Thompson. Toutefois, le besoin d'ajouter des possibilités temps réel aux systèmes Unix s'est fait sentir dans les années 1980, et la norme Posix.1b établie dans les années 1990 a proposé une interface de programmation pour ce type d'ordonnancement.

Les fonctionnalités Posix.1b, sont disponibles à condition que la constante symbolique `_POSIX_PRIORITY_SCHEDULING` soit définie lorsqu'on inclut le fichier d'en-tête `<unistd.h>`. Elles sont alors déclarées dans le fichier `<sched.h>`. On pourra donc utiliser un code du genre :

```
#include <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
    #include <sched.h>
#else
    #warning "Fonctionnalités temps réel non disponibles"
#endif
```

et dans le corps du programme :

```
int main(void)
{
    #ifdef _POSIX_PRIORITY_SCHEDULING
        /* Basculement en ordonnancement temps réel */
    #else
        /* Utilisation uniquement de nice() */
    #endif
    ...
}
```

Il existe trois types d'ordonnancement. En fait, il n'y a qu'un seul ordonnanceur, mais il choisit ou il rejette les tâches selon trois politiques possibles. La configuration se fait thread par thread. Elle n'est pas nécessairement globale. Par contre, la modification de la politique d'ordonnancement associée à un thread est une opération privilégiée, car il existe un – gros – risque de bloquer complètement le système.

Les trois algorithmes d'ordonnancement sont nommés *FIFO*, *RR* et *OTHER*. Les deux premiers sont considérés comme temps réel, le dernier n'est pas normalisé et dépend donc du système. Sous Linux, l'ordonnancement *OTHER* correspond à l'ordonnancement temps-partagé que nous avons examiné précédemment.

Les tâches temps réel disposent d'une valeur de priorité temps réel différente des valeurs de priorités que l'on rencontre pour les tâches temps-partagé. Les priorités temps réel sont toujours supérieures à celles des priorités temps-partagé, mais les intervalles ne sont pas figés suivant les systèmes. Il est important, pour respecter la portabilité d'un programme, d'utiliser les appels systèmes `sched_get_priority_max()` et `sched_get_priority_min()`, qui donnent les valeurs minimales et maximales des priorités associées à une politique d'ordonnement donnée. En Python, toutes les fonctions `sched_xxx` sont apparues dans la version 3.3.

```
int sched_get_priority_min (int politique);
int sched_get_priority_max (int politique);
```

Leur emploi est évident.

```
exemple-get-priority-min-max.c :
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    fprintf(stdout, "Ordonnement FIFO :\n %d <= prio <= %d\n",
        sched_get_priority_min(SCHED_FIFO),
        sched_get_priority_max(SCHED_FIFO));
    fprintf(stdout, "Ordonnement RR :\n %d <= prio <= %d\n",
        sched_get_priority_min(SCHED_RR),
        sched_get_priority_max(SCHED_RR));
    fprintf(stdout, "Ordonnement OTHER :\n %d <= prio <= %d\n",
        sched_get_priority_min(SCHED_OTHER),
        sched_get_priority_max(SCHED_OTHER));
    return EXIT_SUCCESS;
}
```

Le script Python équivalent est :

```
exemple-get-priority-min-max.py :
#!/usr/bin/python3
import os

print ("FIFO : {} <= prio <= {}".format(
    os.sched_get_priority_min(os.SCHED_FIFO),
    os.sched_get_priority_max(os.SCHED_FIFO)))
print ("RR : {} <= prio <= {}".format(
    os.sched_get_priority_min(os.SCHED_RR),
    os.sched_get_priority_max(os.SCHED_RR)))
```

```
print ("OTHER: {} <= prio <= {}".format(
    os.sched_get_priority_min(os.SCHED_OTHER),
    os.sched_get_priority_max(os.SCHED_OTHER)))
```

Les intervalles, sous Linux, sont les suivants.

```
$ ./exemple-get-priority-min-max
Ordonnancement FIFO :
  1 <= prio <= 99
Ordonnancement RR :
  1 <= prio <= 99
Ordonnancement OTHER :
  0 <= prio <= 0
$
```

Nous voyons que la priorité temps réel d'une tâche ordonnancée en *OTHER* est toujours nulle, mais qu'elle est pondérée par la valeur de *nice* (étudiée plus haut) afin de permettre de calculer la priorité dynamique.

Lorsqu'une application doit attribuer des priorités différentes à ses threads, exécutés sur un modèle *RR* ou *FIFO*, elle doit d'abord consulter les valeurs fournies par ces deux appels système, et échelonner ses priorités dans l'intervalle disponible. C'est la seule manière d'être vraiment portable selon SUSv4.

Ordonnancement sous algorithme FIFO

L'algorithme FIFO est celui d'une file d'attente (*First In First Out* – premier arrivé, premier servi).

Dans cette optique, il existe une liste des tâches pour chaque priorité temps réel. Le premier thread de la priorité la plus haute s'exécute jusqu'à ce qu'il relâche le processeur. Si un autre thread de même priorité est prêt, il est élu. Sinon, l'ordonnanceur passe au niveau de priorité inférieur et en extrait le premier thread prêt. Ce mécanisme se répète jusqu'à la priorité temps réel 1. Dès qu'une tâche de priorité supérieure à celle qui est en cours d'exécution est de nouveau prête (parce qu'elle attendait une entrée-sortie qui vient de se terminer par exemple), l'ordonnancement lui attribue immédiatement le processeur.

Il existe un appel système particulier, `sched_yield()`, qui permet à un thread en cours d'exécution de relâcher volontairement le processeur.

```
int sched_yield (void);
```

L'ordonnanceur peut alors faire tourner un autre thread du même niveau de priorité, s'il y en a un prêt. En revanche, si aucun autre thread de même niveau n'est prêt à s'exécuter, le processeur est réattribué au thread qui vient d'invoquer `sched_yield()`. Le noyau n'élit jamais une tâche si une autre de priorité supérieure est prête.

Sur d'autres systèmes d'exploitation temps réel – non Posix – on a coutume d'appeler `sleep(0)` pour obtenir le même résultat que `sched_yield()`.

L'ordonnement FIFO est le plus violent et le plus égoïste qui soit. Le plus fort a toujours raison. Le thread le plus prioritaire a toujours le processeur dès qu'il est prêt à s'exécuter. Il existe bien entendu un très sérieux risque de blocage du système (du moins sur une machine uniprocasseur) si on exécute un simple

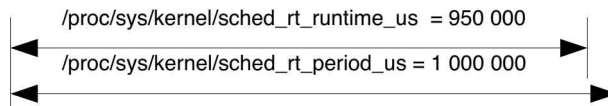
```
while (1)
;
```

avec une priorité élevée.

Pour éviter ce genre de désagrément, le noyau Linux (depuis sa version 2.6.21) offre un garde-fou plutôt discutable : lorsque l'ensemble des tâches temps réel ont consommé 950 ms de temps CPU de manière ininterrompue, l'ordonnanceur réserve les 50 ms suivantes pour les tâches temps partagé. Ceci permet de s'assurer qu'une tâche temps réel en boucle ne bloquera pas complètement le système, et pourra être interrompue par un signal (pression sur *Contrôle-C* dans son terminal par exemple).

Les durées sont configurables et ce mécanisme, plutôt contraire aux principes du temps réel, est – heureusement – débrayable.

Figure 10-2
Garde-fou temps-partagé



Les pseudo-fichiers de `sched_rt_runtime_us` et `sched_rt_period_us` contenus dans `/proc/sys/kernel` contiennent la durée (en microsecondes) réservée au temps réel par période (en microsecondes également). Pour s'assurer que les tâches temps réel disposent bien de l'intégralité du temps CPU si elles le demandent, on écrira :

```
echo -1 > /proc/sys/kernel/sched_rt_runtime
```

ou

```
sysctl kernel.sched_rt_runtime=-1
```

dans un script d'initialisation au boot.

Un programme se trouvant seul au niveau de priorité FIFO le plus élevé est sûr de s'exécuter de bout en bout sans être perturbé. Par contre, si deux tâches s'exécutent au même niveau FIFO, la progression parallèle des deux n'est pas très prévisible. La commutation s'effectue parfois volontairement, en invoquant `sched_yield()`, et parfois sur des appels système bloquants qui endorment une tâche.

Le comportement d'une tâche seule est presque totalement déterministe (aux retards induits par les interruptions matérielles près). Cela permet d'assurer un comportement temps réel quasi parfait. Par contre, deux tâches concurrentes à la même priorité ont des progressions imprévisibles. Pour améliorer tout cela, un second type d'ordonnancement temps réel a été défini.

Ordonnancement sous algorithme RR

L'ordonnancement RR (*Round Robin*, tourniquet) est une simple variante de celui qui a été décrit précédemment, incorporant un aspect préemptif au temps partagé. Chaque thread se voit attribuer une tranche de temps fixe. Lorsqu'il a atteint sa limite, le noyau l'interrompt et le met en état Prêt. Ensuite, il le repousse à la fin de la liste des tâches associée à sa priorité. Si un autre thread du même niveau est prêt, il sera choisi. Si aucun autre thread de même priorité n'est prêt, le noyau redonne la main à la tâche qu'il vient d'interrompre. On ne donne jamais le processeur à un thread de plus faible priorité.

La différence avec l'algorithme FIFO réside donc uniquement dans le cas où plusieurs tâches sont simultanément prêtes avec la même priorité (et si aucun thread de plus haute priorité n'est prêt). Dans le cas de l'algorithme FIFO, la première tâche qui arrive reçoit le processeur et le conserve jusqu'à ce qu'elle s'endorme ou qu'elle le relâche volontairement avec `sched_yield()`. Avec l'ordonnancement RR, chaque thread prêt de la plus haute priorité sera régulièrement choisi pour s'exécuter, quitte à interrompre l'un de ses confrères qui ne veut pas s'arrêter de lui-même.

Si deux threads ont la même priorité, chacun aura donc l'impression de s'exécuter deux fois moins vite que s'il était seul, mais aucun des deux ne sera bloqué pour une période a priori inconnue, comme c'était le cas avec l'ordonnancement FIFO.

Ordonnancement sous algorithme OTHER

Le troisième type d'ordonnancement est l'algorithme *OTHER* (autre), qui n'est pas réellement défini par Posix.1b. L'implémentation de cet algorithme est laissée à la discrétion des concepteurs du noyau. Sur certains systèmes, il peut s'agir d'ailleurs de l'algorithme RR, avec des plages de priorité plus faibles.

Sous Linux, il s'agit de l'ordonnancement temps-partagé dont nous avons déjà parlé, utilisant une priorité dynamique recalculée en fonction de la priorité statique et de l'usage que le thread fait du laps de temps qui lui est imparti.

Récapitulation

L'ordonnanceur fonctionne donc ainsi.

- S'il existe un ou plusieurs thread *FIFO* ou *RR* prêts, ils sont sélectionnés en premier. Celui dont la priorité est la plus grande est choisi. S'il s'agit d'un thread *RR*, on programme un délai au bout duquel le thread sera rejeté en fin de sa liste de priorité s'il n'a pas rendu le processeur auparavant.
- Si aucun thread *FIFO* ou *RR* n'est prêt, le noyau recalcule les priorités dynamiques des tâches *OTHER* prêtes, en fonction de leurs priorités statiques, de leur utilisation du CPU, et d'autres paramètres (emplacement sur une machine multiprocesseur, disponibilité de l'espace mémoire...). En fonction de la priorité dynamique, un thread est élu, et le noyau lui attribue le processeur pendant un délai maximal.
- Si aucun thread n'est prêt, le noyau peut arrêter le processeur sur certaines architectures, jusqu'à l'arrivée d'une interruption signalant un changement d'état.

Temps Réel ?

Nous avons évoqué à plusieurs reprises le terme d'ordonnancement temps réel. Certains sont sceptiques, et à juste titre, sur l'emploi de ce mot à propos de Linux ou de tout système Unix en général.

Il existe deux classes de problèmes relevant de la programmation temps réel.

- 1 Le temps réel strict (*hard realtime*), impose pour chaque opération des délais totalement infranchissables, sous peine de voir des événements catastrophiques se produire. Il s'agit par exemple du contrôle de la mise à feu d'un réacteur d'avion, du déclenchement d'un Airbag, ou de l'émission des impulsions laser en microchirurgie. La sensibilité par rapport à la limite temporelle est telle qu'il est non seulement impensable de soumettre la tâche aux retards dus à d'autres tâches, mais également impossible d'admettre la moindre tolérance par rapport au travail même du noyau. L'arrivée d'une interruption devant faire basculer une tâche à l'état Prêt, la vérification des tâches en cours, pour finalement laisser la main au même thread, peut induire un retard critique dans ces systèmes. Pour ce type d'application, Linux n'est pas approprié (pour l'instant du moins, même si des évolutions prometteuses s'annoncent en ce sens). Il est dans ce cas indispensable de se tourner vers d'autres systèmes d'exploitation spécialisés dans le temps réel strict, voire le projet Xenomai dont le principe est de faire tourner Linux comme une tâche d'un micro-noyau temps réel.

- 2 Les applications temps réel souples (*soft realtime*) n'ont pas de contraintes aussi strictes que les précédentes. Les limites temporelles existent toujours, mais les conséquences d'un dépassement faible ne sont pas aussi catastrophiques. Dans ce genre d'application, il est important de subdiviser le système en sous-unités réalisant des tâches bien précises, et dont les priorités peuvent être fixées avec précision. Une application pourra par exemple privilégier la réception et le décodage de données provenant de divers équipements. La transmission des alarmes sur défaut sera probablement traitée aussi avec une haute importance, tandis que l'affichage continu à destination d'un opérateur pourra être abordé avec une priorité légèrement plus faible (si un retard de présentation n'est pas critique). Enfin, on emploiera une priorité radicalement moindre pour des tâches administratives de statistiques, d'impression de copies d'écran ou de journalisation des changements d'état.

En utilisant un ordonnancement *RR*, voire *FIFO*, Linux peut être parfaitement adapté à des applications du domaine temps réel souple. Le comportement est déterministe entre les tâches. Un programme de plus faible priorité ne viendra jamais perturber un processus de haute priorité. Les seuls écarts temporels possibles sont dus à la gestion interne du noyau, qui est optimisée, et n'induit que des retards infimes.

Suivant la configuration choisie à sa compilation, le noyau peut être préemptible (*qui peut être préempté*), à ne pas confondre avec préemptif (*qui peut préempter une tâche*, ce qui était déjà le cas). Ceci signifie que l'exécution d'un appel système n'est pas nécessairement atomique, et qu'un thread peut se trouver interrompu même lorsqu'il exécute du code appartenant au noyau.

L'ordonnancement temps réel Linux est de bonne qualité, même s'il est limité à une classe *soft realtime* ; il existe plusieurs moyens de l'améliorer. Le lecteur intéressé pourra se reporter à [BLAESS 2015] *Solutions temps réel sous Linux*.

Modification de la politique d'ordonnancement

Les sources habituelles d'informations traitant des processus (`ps`, `top`, `/proc/<pid>/...`) ne nous indiquent pas la politique d'ordonnancement avec laquelle s'exécute un thread. Nous allons donc créer un petit programme qui va nous servir de frontal pour l'appel système `sched_getscheduler()`. Tous les appels système que nous allons étudier ici sont définis par la norme Posix.1b et sont déclarés dans `<sched.h>` :

```
| int sched_getscheduler (int pid);
```

Cet appel système renvoie `-1` en cas d'erreur, sinon il renvoie l'une des trois constantes `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`, en fonction de l'ordonnancement du processus dont on fournit le PID. Si on passe un PID nul, cette fonction renvoie la politique du processus appelant.

Le second paramètre de ces appels système est le type d'ordonnancement désiré : `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`. Pour passer un processus en temps réel (l'un des deux premiers ordonnancements) il faut que l'appelant dispose de la capacité `CAP_SYS_NICE` ou soit exécuté par root.

La politique d'ordonnancement est héritée au travers d'un `fork()` ou d'un `exec()`. Il est donc possible pour un processus de modifier sa propre politique, puis de lancer une commande indépendante. C'est ce que réalise l'utilitaire `chrt` disponible sur la majeure partie des distributions Linux.

La priorité temps réel d'une tâche est manipulée par l'intermédiaire d'une structure de type `sched_param`. Celle-ci peut contenir divers champs, mais le seul qui soit défini par Posix.1b est `sched_priority`, qui représente bien entendu cette priorité temps réel. Les deux appels système `sched_getparam()` et `sched_setparam()` permettent de lire ou modifier la priorité du thread principal d'un processus, tandis que `pthread_getschedparam()` consulte l'ordonnancement d'un thread quelconque.

```
int sched_getparam (pid_t pid, struct sched_param * param);
int sched_setparam (pid_t pid, const struct sched_param * param);

int pthread_getschedparam (pthread_t thr, int * ordo,
                           struct sched_param *param);
```

Toutes ces fonctions renvoient 0 si la consultation ou la modification a réussi, et -1 en cas d'échec.

En Python, une fonction `os.sched_param(priorite)` permet de créer un objet `sched_param` à passer en argument de `os.sched_setscheduler()`.

Nous pouvons réaliser une petite expérience pour vérifier le fonctionnement de l'ordonnancement temps réel. Le programme suivant exécute une boucle consommatrice de CPU, à une priorité élevée, mais se termine au bout de vingt secondes sur réception d'un signal d'alarme qu'il ne gère pas. Dans les fichiers sources du livre, il en existe une version C et une version Python ; leurs comportements sont identiques.

```
exemple-boucle-temps-reel.py :
#!/usr/bin/python3
import os
import signal

try:
    os.sched_setscheduler(0, os.SCHED_FIFO, os.sched_param(99))
    signal.alarm(20)
    while True:
        pass
except PermissionError:
    print("You need root permission to run as realtime process")
```


Pour exécuter ce programme, il faut avoir les droits root. Je vous conseille de lancer un exemplaire de l'application sur chaque processeur (ou cœur). Pour cela, il suffit d'exécuter les opérations suivantes (ici sur un *dual-core*).

```
$ su
Password:
# echo -l > /proc/sys/kernel/sched_rt_runtime_us
# sleep 5 ; taskset -c 0 ./exemple-boucle-temps-reel.py
```

Puis profiter du répit de 5 secondes pour lancer dans un autre terminal.

```
# sleep 5 ; taskset -c 1 ./exemple-boucle-temps-reel.py
```

Et ainsi de suite pour chaque cœur. On verra alors le système totalement gelé pendant vingt secondes, seules nos boucles sont exécutées. Une fois le délai écoulé, le fonctionnement reprend tout à fait normalement.

Pendant la boucle temps réel, les interruptions matérielles sont toutefois servies normalement. Sur les versions classiques de Linux, certaines opérations complexes (par exemple le traitement de la pile réseau) sont réalisées immédiatement au retour du gestionnaire d'interruption. Au détriment de notre tâche temps réel. Nous pouvons le vérifier en appelant la commande ping sur une autre machine de notre réseau afin de vérifier si notre système répond aux demandes d'écho réseau. Sur les systèmes Linux actuels, le ping vers notre machine continue à fonctionner, ce qui signifie que notre noyau continue à répondre aux messages externes, alors même qu'une tâche utilisateur extrêmement prioritaire est active. Les évolutions actuelles tendent à différer ces réponses dans des threads noyau ordonnancés avec une priorité parfois moindre que les tâches utilisateur. Ceci améliorera le déterminisme des processus ordonnancés en temps réel.

Il existe un autre paramètre, consultable mais non modifiable : il s'agit de la durée de la tranche de temps affectée à un processus lorsqu'il est ordonnancé sur le mode *RR*. L'appel `sys_sched_rr_get_interval()` permet de lire cette durée.

```
| int sched_rr_get_interval (pid_t pid, struct timespec * intervalle);
```

La structure `timespec` que nous avons déjà rencontrée propose le champ `tv_sec`, qui représente des secondes, et le champ `tv_nsec`, qui contient le complément en nano-secondes.

Nous pouvons à présent écrire un programme qui nous donne toutes les informations concernant l'ordonnancement des processus indiqués en argument sur sa ligne de commande :

```
exemple_ordonnancement.c :
#include <errno.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/resource.h>

int main (int argc, char * argv[])
{
    int                i;
    int                pid;
    int                ordonnancement;
    struct sched_param param;
    struct timespec    intervalle;

    for (i = 1; i < argc; i++) {
        fprintf(stdout, "%s: ", argv[i]);
        if ((sscanf(argv[i], "%d", & pid) != 1)
            || (ordonnancement = sched_getscheduler(pid)) < 0) {
            fprintf(stdout, "invalide\n");
            continue;
        }
        switch(ordonnancement) {
            case SCHED_OTHER :
                fprintf(stdout, "OTHER priorite (nice) %d\n",
                    getpriority(PRIO_PROCESS, pid));
                break;
            case SCHED_FIFO :
                sched_getparam(pid, & param);
                fprintf(stdout, "FIFO priorite %d\n",
                    param.sched_priority);
                break;
            case SCHED_RR :
                sched_getparam(pid, & param);
                sched_rr_get_interval(pid, & intervalle);
                fprintf(stdout, "RR priorite %d, intervalle %ld ns\n",
                    param.sched_priority,
                    intervalle.tv_sec * 1000000000 + intervalle.tv_nsec);
                break;
            default :
                fprintf(stdout, "??? \n");
                break;
        }
    }
}
```

```
}  
return EXIT_SUCCESS;  
}
```

Les processus courants sont, nous l'avons déjà dit, essentiellement ordonnancés en temps partagé.

```
$ ./exemple-ordonnancement 0  
0: OTHER priorite (nice) 0  
$ ./exemple-ordonnancement 1 2 3  
1: OTHER priorite (nice) 0  
2: OTHER priorite (nice) 0  
3: FIFO priorite 99
```

Vérifions la modification de la priorité temps partagé.

```
$ nice -n 12 ./exemple-ordonnancement 0  
0: OTHER priorite (nice) 12  
$
```

À présent nous allons basculer un shell en temps réel sur un autre terminal.

```
$ su -  
Password:  
# chrt -f 99 sh  
sh-4.1# echo $$  
5224
```

Et vérifions sa configuration.

```
$ ./exemple-ordonnancement 5224  
5224: FIFO priorite 99  
$
```

Essayons avec un ordonnancement *Round-Robin*.

```
sh-4.1# exit
exit
# chrt -r 50 sh
sh-4.1# echo $$
5232
#
```

Vérifions :

```
$ ./exemple-ordonnancement 5232
5232: RR priorite 50, intervalle 99984800 ns
$
```

L'intervalle de rotation entre threads ordonnancés en *RR* est de 100 ms environ.

Conclusion

L'ordonnancement des processus est un domaine très intéressant et plus facile à appréhender qu'on pourrait le croire au premier abord. La présentation générale des principes d'un ordonnanceur est décrite dans [Tanenbaum 1997] *Operating Systems, Design and Implementation*.

On trouvera une description plus complète des ordonnancements temps réel et de leurs extensions dans [BLAESS 2015] *Solutions temps réel sous Linux*.

La limitation des ordonnancements temps réel sous Linux est liée à la gestion des interruptions par le noyau, et par les actions – parfois conséquentes – qu'il réalise au détriment de l'espace applicatif (gestion mémoire virtuelle, systèmes de fichiers cryptés, protocoles réseau, etc.). Si on désire dépasser ces limites, on peut se tourner vers des extensions temps réel strict pour Linux, comme RTAI ou Xenomai qui retardent le traitement non urgent des interruptions pour laisser les tâches de hautes priorités s'exécuter en premier.

11

Gestion classique des signaux

La gestion des signaux entre processus est une partie passionnante de la programmation sous Unix. C'est aussi celle qui peut conduire aux dysfonctionnements les plus subtils, avec des bogues très difficiles à détecter de par leur nature fondamentalement intempestive.

On peut traiter les signaux de deux façons : une classique, en partie définie par la norme Ansi C, que nous étudierons dans ce chapitre, et une plus performante, définie à l'origine par les normes Posix.1 et Posix.1b puis de nos jours par SUSv4, ainsi que nous le verrons dans les prochains chapitres.

Généralités

Le principe est a priori simple : un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même)¹. Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement (aux délais dus à l'ordonnancement près) une mesure spécifique. Le destinataire peut soit ignorer le signal, soit le capturer – c'est-à-dire dérouter provisoirement son exécution vers une routine particulière qu'on nomme gestionnaire de signal –, soit laisser le système traiter le signal avec un comportement par défaut. En Python, le comportement est légèrement différent : c'est l'interpréteur qui reçoit le signal et le traite dans un gestionnaire bas niveau, puis il déroute l'exécution du script en cours pour appeler

1. Nous parlons bien dans ce chapitre de processus, l'envoi de signaux entre threads (d'un même processus uniquement) est anecdotique et déconseillé.

le gestionnaire installé par ce dernier. On ne gèrera donc pas des signaux de très bas niveau comme SIGFPE ou SIGSEGV qui indiquent des erreurs commises au niveau de l'exécutable interpréteur.

La plupart des signaux qui nous intéressent ne sont pas émis par des processus applicatifs, mais directement par le noyau en réponse à des conditions logicielles ou matérielles particulières. Certains signaux font partie d'une classe distincte, les signaux temps réel, définis par la norme Posix.1b. Nous les étudierons à part, dans le chapitre 13.

Il existe un nombre déterminé de signaux (64 sous Linux). Chaque signal dispose d'un nom défini sous forme de constante symbolique commençant par SIG et d'un numéro associé. Il n'y a pas de nom pour le signal numéro 0, car cette valeur a un rôle particulier que nous verrons plus tard. Toutes les définitions concernant les signaux se trouvent dans le fichier d'en-tête `<signal.h>` (ou dans d'autres fichiers qu'il inclut lui-même) ; pour le langage Python, il s'agit du module `signal`.

Il est important de toujours utiliser le nom symbolique du signal plutôt que son numéro, car celui-ci peut varier d'un système à l'autre, voire selon la machine employée, même avec une version identique du noyau. La constante symbolique `NSIG` définie par la bibliothèque C (et `signal.NSIG` en Python) correspond au nombre de signaux, y compris un pseudo-signal numéroté 0. Il arrive que, sur certains systèmes, cette constante soit nommée `_NSIG`. Pour assurer la portabilité d'un programme, on peut donc inclure en début de fichier des directives pour le préprocesseur, du type :

```
#include <signal.h>

#ifndef NSIG
    #ifndef _NSIG
        #error "NSIG et _NSIG indéfinis"
    #else
        #define NSIG _NSIG
    #endif
#endif
```

Sur les systèmes prenant en charge les signaux temps réel, comme Linux, il existe deux valeurs supplémentaires importantes : `SIGRTMIN` et `SIGRTMAX`. Il s'agit des numéros du plus petit et du plus grand signal temps réel. Ces derniers, en effet, s'étendent sur une plage continue en dessous de `NSIG-1`. Sous Linux, l'organisation des signaux est la suivante :

Valeur	Signal
0	Signaux classiques (non temps réel)
...	
31	

Valeur	Signal
32	SIGRTMIN
...	(signaux temps réel)
63	SIGRTMAX
64	NSIG

Attention : SIGRTMIN et SIGRTMAX ne sont pas nécessairement des constantes symboliques du préprocesseur. SUSv4 autorise leur implémentation sous forme de variables. Leur valeur n'est donc pas toujours disponible à la compilation. En revanche, la constante symbolique `_POSIX_REALTIME_SIGNALS` doit être définie dans `<unistd.h>`

Il nous arrivera de vouloir balayer uniquement la liste des signaux classiques. Pour cela, nous définirons une variable locale `NB_SIG_CLASSIQUES` permettant de travailler sur tous les systèmes :

```
#ifdef _POSIX_REALTIME_SIGNALS
#define NB_SIG_CLASSIQUES SIGRTMIN
#else
#define NB_SIG_CLASSIQUES NSIG
#endif
```

Chaque signal classique a une signification bien précise, et les processus ont par défaut un comportement adapté à la situation représentée par le signal. Par exemple, le signal indiquant la fin d'un processus fils est ignoré par défaut. En revanche, la déconnexion du terminal de contrôle envoie un signal qui arrête le processus, et une référence mémoire invalide le termine et crée un fichier d'image mémoire (core) permettant le débogage du programme.

Pour qu'un fichier core soit créé, un certain nombre de conditions sont nécessaires : permission d'écriture dans le répertoire de lancement du processus, pas d'exécution *Set-UID* ou *Set-GID* et pas de limitation dans la taille de ces fichiers. Cette limite est fixée avant le lancement du processus par la commande shell `ulimit`. Ainsi, il faudra penser à exécuter la commande :

```
$ ulimit -c unlimited
```

avant de démarrer une session de débogage, pour s'assurer que les fichiers core soient créés sans limites de taille. (Cette opération peut également être réalisée directement dans le programme, comme nous l'avons vu dans le chapitre 9).

Avant d'étudier les mécanismes d'émission et de réception des signaux, nous allons analyser précisément ceux qui sont disponibles sous Linux, en observant les conditions dans lesquelles les signaux sont émis, le comportement par défaut d'un processus qui les reçoit, et l'intérêt éventuel d'installer un gestionnaire pour les capturer.

Liste des signaux sous Linux

Certains signaux sont révélateurs de conditions d'erreur. Ils sont en général délivrés immédiatement après la détection du dysfonctionnement. On ne peut donc pas vraiment parler de comportement asynchrone. En revanche, tous les signaux indiquant une interaction avec l'environnement (action sur le terminal, connexion réseau...) ont une nature fortement asynchrone. Ils peuvent se produire à tout moment du programme.

Les noms indiqués ci-après existent sous forme de constantes symboliques pour la bibliothèque C et en tant que constantes Python (préfixées du nom du module `signal`).

Signaux SIGABRT et SIGIOT

Il s'agit de deux synonymes sous Linux. SIGIOT est le nom historique sous Système V, mais SIGABRT est préférable, car il est défini par les normes Ansi C et SUSv4. SIGABRT est déclenché lorsqu'on appelle la fonction `abort()` de la bibliothèque C. Le comportement par défaut est de terminer le programme en créant un fichier core. La routine `abort()` sert, comme nous l'avons vu dans le chapitre précédent, à indiquer la fin anormale d'un programme (détection d'un bogue interne, par exemple).

Il est possible d'ignorer le signal SIGABRT (si on le reçoit depuis un autre processus), mais il faut savoir que la fonction `abort()` (`os.abort()` en Python) restitue le comportement par défaut avant d'envoyer SIGABRT vers le processus appelant lui-même.

La routine `abort()` de la bibliothèque C est complexe, car la norme SUSv4 réclame plusieurs fonctionnalités assez difficiles à concilier.

- Si le processus ignore SIGABRT, `abort()` doit réinitialiser le comportement par défaut avant d'envoyer ce signal.
- Si le processus capture le signal SIGABRT et si son gestionnaire redonne ensuite la main à la routine `abort()`, celle-ci doit vider tous les flux d'entrée-sortie en utilisant `fflush()` et les fermer avant de terminer le programme.
- Si le processus capture le signal SIGABRT et s'il ne rend pas la main à `abort()`, celle-ci ne doit pas toucher aux flux d'entrée-sortie. Pour ne pas revenir, le gestionnaire peut soit sortir directement du programme en appelant `exit()` ou `_exit()`, soit effectuer un saut non local `siglongjmp()` vers une autre routine.

Rajoutons à cela que la routine `abort()` de la bibliothèque C doit traiter le cas des programmeurs distraits qui invoquent `abort()` depuis le gestionnaire de signal SIGABRT lui-même, ainsi que l'appel simultané depuis plusieurs threads. La complexité de cette tâche est toutefois résolue par la routine `abort()`, dont on peut examiner l'implémentation dans les sources de la bibliothèque Glibc.

Nous parlerons souvent des sauts non locaux au cours de ce chapitre. Nous les détaillerons lorsque nous étudierons la manière de terminer un gestionnaire de signal. Pour le moment, on peut considérer qu'il s'agit d'une sorte de « goto » permettant de sauter d'une fonction au sein d'une autre, en nettoyant également la pile.

Signaux SIGALRM, SIGVTALRM et SIGPROF

SIGALRM (attention à l'orthographe, il n'y a pas de second A...) est un signal déclenché par le noyau à l'expiration du délai programmé grâce à l'appel système `alarm()` (`signal.alarm()` en Python). Nous l'étudierons plus en détail ultérieurement, car il est souvent utilisé pour programmer une limite de temporisation pour des appels système bloquants (comme une lecture depuis une connexion réseau). On programme un délai maximal avant d'invoquer l'appel système susceptible de rester coincé, et l'arrivée du signal SIGALRM l'interrompt, avec un code d'erreur EINTR dans `errno`.

SIGALRM est également utilisé pour la programmation de temporisations avec `setitimer()` (ou `signal.setitimer()` en Python) Cet appel système sert à fournir un suivi temporel de l'activité d'un processus. Il y a trois types de temporisations : la première fonctionne en permanence et déclenche SIGALRM à son expiration, la deuxième ne fonctionne que lorsque le processus s'exécute et déclenche SIGVTALRM, et la troisième décompte le temps cumulé d'exécution du code du processus et celui du code du noyau exécuté lors des appels système, puis déclenche SIGPROF. L'utilisation conjointe des deux dernières temporisations permet un suivi de l'activité du processus.

Il est donc formellement déconseillé d'utiliser simultanément les fonctions de comptabilité de `setitimer()` et la programmation de `alarm()`.

Notons de surcroît que l'implémentation de la fonction `sleep()` de la bibliothèque Glibc utilise également le signal SIGALRM. Cette fonction prend bien garde de ne pas interférer avec les éventuelles autres routines d'alarme du processus, sauf si le gestionnaire de SIGALRM installé par l'utilisateur se termine par un saut non local ; dans ce cas, elle échoue. Il peut arriver que, sur d'autres systèmes, la routine `sleep()` soit moins prévenante et qu'elle interfère avec `alarm()`. Pour une bonne portabilité d'un programme, il vaut donc mieux éviter d'utiliser les deux conjointement.

SIGALRM est défini par SUSv4 ; SIGVTALRM et SIGPROF également, mais sous forme d'extensions X/Open (pas toujours disponibles dans tous les Unix). Par défaut, ces trois signaux terminent le processus en cours.

Signaux SIGBUS et SIGSEGV

Ces deux signaux indiquent respectivement une erreur d'alignement des adresses sur le bus et une violation de la segmentation. Ils n'ont pas la même signification, mais sont généralement dus au même type de bogue : l'emploi d'un pointeur mal initialisé.

Le signal SIGBUS est dépendant de l'architecture matérielle, car il représente en fait une référence à une adresse mémoire invalide (par exemple, un mauvais alignement de mots de deux ou quatre octets sur des adresses paires ou multiples de quatre).

Le signal SIGSEGV correspond à une adresse correcte, mais pointant en dehors de l'espace d'adressage affecté au processus. Ce signal est fréquemment rencontré lors du développement en C mais ne concerne généralement pas le développeur Python, les pointeurs étant absents de ce langage. Il est possible avec le module `ctypes` d'effectuer des échanges de données entre code Python et fonctions de bibliothèques écrites en C. Dans ce cas, des *segmentation faults* peuvent se produire, mais cela déborde du cadre de ce livre.

Ces deux signaux arrêtent par défaut le processus en créant un fichier `core`. Dans un cas comme dans l'autre, il est mal vu d'ignorer ces types de signaux, qui sont révélateurs de bogues. SUSv4 souligne d'ailleurs que le comportement d'un programme ignorant SIGSEGV est indéfini. À la rigueur, on peut capturer le signal, afficher un message à l'utilisateur et reprendre l'exécution dans un contexte propre par un saut non local `longjmp()`, comme nous le décrivons ci-après pour SIGILL.

Signaux SIGCHLD et SIGCLD

Le signal SIGCHLD est émis par le noyau vers un processus dont un fils vient de se terminer ou d'être stoppé. Ce processus peut alors soit ignorer le signal (ce qui est le comportement par défaut, mais qui est déconseillé par SUSv4), soit le capturer pour invoquer l'appel système `wait()` qui précisera le PID du fils concerné et le code de terminaison s'il s'est fini.

Le signal SIGCLD est un synonyme de SIGCHLD sous Linux. Il s'agit d'une variante historique sur Système V. Le fonctionnement de SIGCLD était différent de celui de SIGCHLD et était particulièrement discutable. Les nouvelles applications doivent donc uniquement considérer SIGCHLD, qui est défini par SUSv4.

Par défaut, SIGCHLD est ignoré, mais ce comportement est légèrement différent de celui qui est adopté si on demande explicitement d'ignorer ce signal. Lorsqu'un processus fils se termine, et tant que son père n'a pas exécuté un appel `wait()`, il devient zombie si SIGCHLD est capturé par un gestionnaire ou s'il est traité par défaut. En revanche, le processus fils disparaît sans rester à l'état zombie si SIGCHLD est volontairement ignoré. SUSv4 déconseille toutefois qu'un processus ignore SIGCHLD s'il est

destiné à avoir des descendants. Un gestionnaire de signal minimal permettra d'éliminer facilement les zombies.

Signaux SIGFPE et SIGSTKFLT

SIGFPE correspond théoriquement à une *Floating Point Exception*, mais il n'est en fait nullement limité aux erreurs de calcul en virgule flottante. Il inclut par exemple l'erreur de division entière par zéro. Il peut également se produire si le système ne possède pas de coprocesseur arithmétique, si le noyau a été compilé sans l'option pour l'émuler, et si le processus exécute des instructions mathématiques spécifiques.

SIGFPE est défini par Ansi C et SUSv4 ; par défaut, ce signal arrête le processus en créant un fichier core. Une application devra donc le laisser tel quel durant la phase de débogage afin de déterminer toutes les conditions dans lesquelles il se produit (en analysant *post-mortem* le fichier core). Théoriquement, un programme suffisamment défensif ne devrait laisser passer aucune condition susceptible de déclencher un signal SIGFPE, quelles que soient les données saisies par l'utilisateur...

On notera toutefois que le débogage peut parfois être malaisé en raison du retard du signal provenant du coprocesseur. Le signal n'est pas nécessairement délivré immédiatement après l'exécution de la condition d'erreur, mais peut survenir après plusieurs instructions.

SIGSTKFLT est un signal indiquant une erreur de pile, mais il n'est jamais déclenché par le noyau Linux. Ce signal, non conforme SUSv4, arrête par défaut le processus. On n'en trouve pas trace d'utilisation dans les sources du noyau, aussi est-il conseillé de ne pas s'en préoccuper.

Signal SIGHUP

Ce signal correspond habituellement à la déconnexion (*Hang Up*) du terminal de contrôle du processus. Le noyau envoie SIGHUP, suivi du signal SIGCONT que nous verrons plus bas, au processus *leader* de la session associée au terminal. Ce processus peut d'ailleurs se trouver en arrière-plan. La gestion de ce signal se trouve dans le fichier `drivers/char/tty/tty_io.c` des sources du noyau.

SIGHUP est aussi envoyé, suivi de SIGCONT, à tous les processus d'un groupe qui devient orphelin. Rappelons qu'un processus crée un groupe en utilisant `setpgid()`, l'identifiant du groupe étant égal au PID du processus *leader*. Ses descendants futurs appartiendront automatiquement à ce nouveau groupe, à moins qu'ils ne créent leur propre groupe. Lorsque le processus *leader* se termine, le groupe est dit orphelin. À ce moment, le noyau envoie SIGHUP, suivi de SIGCONT si le groupe contient des processus arrêtés.

Enfin, il est courant d'envoyer manuellement le signal `SIGHUP` (en utilisant la commande `/bin/kill`) à certains processus démons, pour leur demander de se réinitialiser bien que ce rôle revienne de plus en plus au signal `SIGUSR1`. Comme un démon n'a pas de terminal de contrôle, ce signal n'a pas de signification directe. Il est alors souvent utilisé pour demander au démon de relire ses fichiers de configuration (par exemple pour `xinetd`, `named`, `sendmail`...). En outre, ces démons ferment puis rouvrent leurs fichiers de journalisation (*log*) à la réception de ce signal, ce qui permet de réaliser une rotation de ces fichiers.

`SIGHUP` est décrit par `SUSv4` et, par défaut, il termine le processus cible. On notera que la commande `nohup` permet de lancer un programme en l'immunisant contre `SIGHUP`. En fait, `nohup` est parfois un script shell utilisant la fonctionnalité `TRAP` de l'interpréteur de commandes. Ce programme permet de lancer une application en arrière-plan, en redirigeant sa sortie vers le fichier `nohup.out`, et de se déconnecter en toute tranquillité.

Signal SIGILL

Ce signal est émis lorsque le processeur détecte une instruction assembleur illégale. Le noyau est prévenu par l'intermédiaire d'une interruption matérielle, et il envoie un signal `SIGILL` au processus fautif. Cela ne doit jamais se produire dans un programme normal, compilé pour la bonne architecture matérielle. Mais il se peut toutefois que le fichier exécutable soit corrompu, ou qu'une erreur d'entrée-sortie ait eu lieu lors du chargement du programme et que le segment de code contienne effectivement un code d'instruction invalide.

Un autre problème possible peut être causé par les systèmes ne disposant pas de coprocesseur arithmétique. Le noyau utilise alors un émulateur arithmétique qui peut déclencher le signal `SIGILL` dans certains cas rares d'instructions mathématiques inconnues.

En fait, l'occurrence la plus courante du signal `SIGILL` est révélatrice d'un bogue de débordement de pile. Par exemple, un tableau de caractères déclaré en variable locale (et donc alloué dans la pile) peut avoir été débordé par une instruction de copie de chaîne sans limites de longueur, comme `strcpy()`. La pile peut très bien être corrompue, et l'exécution du programme est totalement perturbée, avec une large confusion entre code et données.

Le comportement par défaut est d'arrêter le processus et de créer un fichier d'image mémoire `core`. Il s'agit d'un signal décrit dans la norme `Ansi C`. Il n'est d'aucune utilité d'ignorer ce signal. Un programme intelligemment conçu peut, en cas d'arrivée de `SIGILL`, adopter l'un des deux comportements suivants (dans le gestionnaire de signaux).

- S'arrêter en utilisant `exit()`, pour éviter de laisser traîner un fichier `core` qui ne sera d'aucune aide à l'utilisateur, après avoir éventuellement affiché un message signalant la présence d'un bogue à l'auteur.

- Utiliser une instruction de saut non local `longjmp()`, que nous verrons un peu plus loin. Cette instruction permet de reprendre le programme à un point bien défini, avec un contexte propre. Il est toutefois préférable d'indiquer à l'utilisateur que l'exécution a été reprise à partir d'une condition anormale.

Signal SIGINT

Ce signal est émis vers tous les processus du groupe en avant-plan lors de la frappe d'une touche particulière du terminal : la touche d'interruption. Sur les claviers de PC sous Linux, ainsi que dans les terminaux Xterm, il s'agit habituellement de la séquence de touches *Contrôle-C*. L'affectation de la commande d'interruption à la touche choisie peut être modifiée par l'intermédiaire de la commande `stty`, que nous détaillerons ultérieurement en étudiant les terminaux. L'affectation courante est indiquée au début de la deuxième ligne en invoquant `stty -a` :

```
$ stty -a
speed 9600 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
[...]
$
```

SIGINT est défini par la norme Ansi C et SUSv4, et arrête par défaut le processus en cours.

SIGINT est capturé directement par l'interpréteur Python qui notifie le programme par une exception `KeyboardInterrupt`.

Signaux SIGIO et SIGPOLL

Ces deux signaux sont synonymes sous Linux. SIGPOLL est le nom historique sous Système V, SIGIO celui de BSD.

SIGIO est envoyé lorsqu'un descripteur de fichier change d'état et permet la lecture ou l'écriture. Il s'agit généralement de descripteurs associés à des tubes, des sockets de connexion réseau, ou un terminal. La mise en place d'un gestionnaire de signaux pour SIGIO nécessite également la configuration du descripteur de fichiers pour accepter un fonctionnement asynchrone. Nous y reviendrons ultérieurement dans les pages consacrées aux entrées-sorties asynchrones.

L'utilisation de SIGIO permet à un programme d'accéder à un traitement d'entrée-sortie totalement asynchrone par rapport au déroulement du programme principal. Les données arrivant pourront servir par exemple à la mise à jour de variables globales con-

sultées régulièrement dans le cours du programme normal. Il est alors important, lors de la lecture ou de l'écriture de ces variables globales par le programme normal (en dehors du gestionnaire de signaux), de bien bloquer le signal SIGIO durant les modifications, comme nous apprendrons à le faire dans le reste de ce chapitre.

Signal SIGKILL

SIGKILL est l'un des deux seuls signaux (avec SIGSTOP) qui ne puisse ni être capturé ni être ignoré, ni même être temporairement bloqué par un processus. À sa réception, tout processus est immédiatement arrêté. C'est une garantie pour s'assurer qu'on pourra toujours reprendre la main sur un programme particulièrement rétif. Le seul processus qui ne puisse pas recevoir SIGKILL est `init`, le processus de PID 1.

SIGKILL est traditionnellement associé à la valeur 9, d'où la célèbre ligne de commande « `kill -9 xxx` », équivalente à « `kill -KILL xxxx` ». On notera que l'utilisation de SIGKILL doit être considérée comme un dernier recours, le processus ne pouvant se terminer proprement. On préférera essayer par exemple SIGTERM auparavant, puis SIGQUIT éventuellement.

Notons qu'un processus zombie n'est pas affecté par SIGKILL (ni par aucun autre signal d'ailleurs). Si un processus est stoppé, il sera relancé avant d'être terminé par SIGKILL.

Le noyau lui-même n'envoie que rarement ce signal. C'est le cas lorsqu'un processus a dépassé sa limite de temps d'exécution, ou lors d'un problème grave de manque de place mémoire. La bibliothèque Glibc n'utilise ce signal que pour tuer un processus qu'elle vient de créer lors d'un `popen()` et à qui elle n'arrive pas à allouer de flux de données.

Signal SIGPIPE

Ce signal est émis par le noyau lorsqu'un processus tente d'écrire dans un tube qui n'a pas de lecteur. Ce cas peut aussi se produire lorsqu'on tente d'envoyer des données dans une socket TCP/IP (traitée dans le chapitre sur la programmation réseau) dont le correspondant s'est déconnecté.

Le signal SIGPIPE (défini dans SUSv4) arrête par défaut le processus qui le reçoit. Toute application qui établit une connexion réseau doit donc soit intercepter le signal, soit (ce qui est préférable) l'ignorer. Dans ce dernier cas en effet, les appels système comme `write()` renverront une erreur EPIPE dans `errno`. On peut donc gérer les erreurs au coup par coup dès le retour de la fonction. Nous reviendrons sur ce problème dans les chapitres traitant des communications entre processus.

En Python, SIGPIPE est automatiquement ignoré par l'interpréteur ; il n'y a donc pas à craindre de mauvaise surprise lors d'une écriture dans une socket.

Signal SIGQUIT

Comme SIGINT, ce signal est émis par le pilote de terminal lors de la frappe d'une touche particulière : la touche QUIT. Celle-ci est affectée généralement à la séquence de touches *Contrôle-* (ce qui nécessite sur les claviers français la séquence triple *Contrôle-AltGr-*). SIGQUIT termine par défaut les processus qui ne l'ignorent pas et ne le capturent pas, mais en engendrant en plus un fichier d'image mémoire (core).

Signaux SIGSTOP, SIGCONT, et SIGTSTP

SIGSTOP est le deuxième signal ne pouvant être ni capturé ni ignoré, comme SIGKILL. Il a toutefois un effet nettement moins dramatique que ce dernier, puisqu'il ne s'agit que d'arrêter temporairement le processus visé. Celui-ci passe à l'état stoppé.

Le signal SIGCONT a l'effet inverse ; il permet de relancer un processus stoppé. Si le processus n'est pas stoppé, ce signal n'a pas d'effet. Le redémarrage a toujours lieu, même si SIGCONT est capturé par un gestionnaire de signaux de l'utilisateur ou s'il est ignoré.

SIGTSTP est un signal ayant le même effet que SIGSTOP, mais il peut être capturé ou ignoré, et il est émis par le terminal de contrôle vers le processus en avant-plan. Lors d'un « `stty --all` », la touche affectée à ce signal est indiquée par « `susp=` » et non par « `stop=` », qui correspond à un arrêt temporaire de l'affichage sur le terminal. Dans la plupart des cas, il s'agit de la séquence de touches *Contrôle-Z*.

Il est rare qu'une application ait besoin de capturer les signaux SIGTSTP et SIGCONT, mais cela peut arriver si elle doit gérer le terminal de manière particulière (par exemple en affichant des menus déroulants), et si elle désire effacer l'écran lorsqu'on la stoppe et le redessiner lorsqu'elle redémarre.

Dans la plupart des cas, on ne s'occupera pas du comportement de ces signaux.

Signal SIGTERM

Ce signal est une demande « gentille » de terminaison d'un processus. Il peut être ignoré ou capturé pour terminer proprement le programme en ayant effectué toutes les tâches de nettoyage nécessaires. Traditionnellement numéroté 15, ce signal est celui qui est envoyé par défaut par la commande `/bin/kill`.

SIGTERM est défini par Ansi C et SUSv4. Par défaut, il termine le processus concerné. En principe, une application bien conçue devrait installer systématiquement un gestionnaire pour SIGTERM et SIGINT afin d'assurer une fin « propre » au programme lorsque l'utilisateur a besoin de l'arrêter rapidement.

Signal SIGTRAP

Ce signal est émis par le noyau lorsque le processus a atteint un point d'arrêt. SIGTRAP est utilisé par les débogueurs comme gdb. Il n'a pas d'intérêt pour une application classique. Le comportement par défaut d'un processus recevant SIGTRAP est de s'arrêter avec un fichier core. On peut à la rigueur l'ignorer dans une application qui n'a plus besoin d'être déboguée et qui ne désire pas créer de fichier core intempestif.

Signaux SIGTTIN et SIGTTOU

Ces signaux sont émis par le terminal en direction d'un processus en arrière-plan, qui essaye respectivement de lire depuis le terminal ou d'écrire dessus. Lorsqu'un processus en arrière-plan tente de lire depuis son terminal de contrôle, tous les processus de son groupe reçoivent le signal SIGTTIN. Par défaut, cela stoppe les processus (sans les terminer). Toutefois, il est possible d'ignorer ce signal ou de le capturer. Dans ce cas, l'appel système de lecture `read()` échoue et renvoie un code d'erreur EIO dans `errno`.

Un phénomène assez semblable se produit dans le cas où un processus essaye d'écrire sur son terminal de contrôle alors qu'il se trouve en arrière-plan. Tous les processus du groupe reçoivent SIGTTOU, qui les stoppe par défaut. Si ce signal est ignoré ou capturé, l'écriture a quand même lieu. Notons que l'interdiction d'écrire sur le terminal par un processus en arrière-plan n'est valable que si l'attribut `TOSTOP` du terminal est actif (ce qui n'est pas le cas par défaut).

Nous voyons ci-après l'exemple avec le processus `/bin/ls` qu'on invoque en arrière-plan. Par défaut, il écrit quand même sur le terminal. Lorsqu'on active l'attribut `TOSTOP` du terminal, `/bin/ls` est automatiquement stoppé quand il tente d'écrire. On peut le relancer en le ramenant en avant-plan avec la commande `fg` du shell.

```
$ ls &
[1] 2839
$ bin      etc      lost+found root    usr
boot      home    mnt          sbins   var
dev       lib     proc         tmp
[1]+  Done                  ls
$ stty tostop
$ stty
speed 9600 baud; line = 0;
-brkint -imaxbel
tostop
$ ls &
[1] 2842
$
```



```
[1]+ Stopped (tty output)  ls
$ fg
ls
bin      etc      lost+found  root      usr
boot     home     mnt         sbin      var
dev      lib      proc        tmp
$
```

Signal SIGURG

Ce signal est émis par le noyau lorsque des données « hors bande » arrivent sur une connexion réseau. Ces données peuvent être transmises sur une socket TCP/IP avec une priorité plus grande que pour les données normales. Le processus récepteur est alors averti par ce signal de leur arrivée. Le comportement par défaut est d'ignorer ce signal. Seule une application utilisant un protocole réseau assez complexe pourra avoir besoin d'intercepter ce signal. Notons qu'une telle application peut détecter également l'arrivée de données hors bande différemment, par le biais de l'appel système `select()`.

Signaux SIGUSR1 et SIGUSR2

Ces deux signaux sont à la disposition du programmeur pour ses applications. Le fait que seuls deux signaux soient réservés à l'utilisateur peut sembler restreint, mais cela permet déjà d'établir un système – minimal – de communications interprocessus.

Ces deux signaux sont définis par SUSv4. Par défaut, ils terminent le processus visé, ce qui peut sembler discutable.

Signal SIGWINCH

`SIGWINCH` (*Window changed*) indique que la taille du terminal a été modifiée. Ce signal, ignoré par défaut, est principalement utile aux applications se déroulant en plein écran texte. Lorsque ces dernières sont exécutées dans un Xterm, il est plus sympathique qu'elles se reconfigurent automatiquement lorsqu'on modifie la taille de la fenêtre du Xterm.

C'est, par exemple, le cas de l'utilitaire `less`, entre autres, qui intercepte `SIGWINCH` pour recalculer ses limites d'écran et rafraîchir son affichage.

Une application interactive plein écran gérant ce signal bénéficie d'un « plus » non négligeable en termes d'ergonomie utilisateur. C'est un peu le premier degré du fonctionnement dans un environnement graphique multifenêtré.

Toutefois, la modification asynchrone des caractéristiques d'affichage peut parfois être difficile à implémenter au niveau programmation. La méthode la plus simple consiste à modifier un drapeau placé dans une variable globale, qui sera consultée régulièrement dans la boucle principale de fonctionnement du programme, là où des modifications de taille d'écran peuvent plus aisément être prises en compte.

Signaux SIGXCPU et SIGXFSZ

Ces signaux sont émis par le noyau lorsqu'un processus dépasse une de ses limites souples des ressources système. Nous avons vu dans le chapitre 9, avec les appels système `getrlimit()` et `setrlimit()`, que chaque processus est soumis à plusieurs types de limites système (par exemple le temps d'utilisation du CPU, la taille maximale d'un fichier ou de la zone de données, le nombre maximal de fichiers ouverts...). Pour chaque limite, il existe une valeur souple, modifiable par l'utilisateur, et une valeur stricte, modifiable uniquement par root ou par un processus ayant la capacité `CAP_SYS_RESOURCE`.

Lorsqu'un processus a dépassé sa limite souple d'utilisation du temps processeur, il reçoit chaque seconde le signal `SIGXCPU`. Celui-ci arrête le processus par défaut, mais il peut être ignoré ou capturé. Lorsque le processus tente de dépasser sa limite stricte, le noyau le tue avec un signal `SIGKILL`.

`SIGXFSZ` fonctionne de manière similaire et est émis par le noyau lorsqu'un processus tente de créer un fichier trop grand. Le signal n'est émis toutefois que si l'appel système `write()` tente de dépasser en une seule fois la taille maximale. Dans les autres cas, l'écriture au-delà du seuil programmé déclenche une erreur `EFBIG` de l'appel `write()`.

Signaux temps réel

Les signaux temps réel représentent une extension importante car ils donnent accès aux fonctionnalités définies à l'époque par la norme Posix.1b. De nos jours, ils sont décrits dans l'extension RTS (*real time signals*) de SUSv4. Pour vérifier l'existence de ces signaux en cas de portage d'une application, on peut tester à la compilation la présence de la constante symbolique `_POSIX_REALTIME_SIGNALS` dans le fichier `<unistd.h>`.

Les signaux temps réel sont réservés à l'utilisateur ; ils ne sont pas déclenchés par des événements détectés par le noyau. Ce sont donc des extensions de `SIGUSR1` et `SIGUSR2`. Ils ne sont pas représentés par des constantes, mais directement par leurs valeurs, qui s'étendent de `SIGRTMIN` à `SIGRTMAX`, bornes incluses. Ces signaux sont particuliers – nous y reviendrons –, car ils sont ordonnés en fonction de leur priorité, empilés à la réception (donc plusieurs signaux du même type peuvent être reçus très rapidement), et ils fournissent au processus des informations plus précises que les autres signaux, dont nous parlerons au chapitre 13.

Voici donc l'essentiel des signaux gérés par Linux. Il existe quelques signaux inutilisés dans la plupart des cas (SIGLOST, qui surveille les verrous sur les fichiers NFS), et d'autres dont le comportement diffère suivant l'architecture matérielle (SIGPWR / SIGINFO, qui préviennent d'une chute d'alimentation électrique).

Pour obtenir le libellé d'un signal dont le numéro est connu (comme nous le ferons par la suite dans un gestionnaire), il existe plusieurs méthodes.

- La fonction `strsignal()` disponible en tant qu'extension Gnu (il faut donc utiliser la constante symbolique `_GNU_SOURCE` à la compilation) dans `<string.h>`. Cette fonction renvoie un pointeur sur une chaîne de caractères allouée statiquement (donc susceptible d'être écrasée à chaque appel) et contenant un libellé descriptif du signal.
- La fonction `psignal()` affiche sur la sortie d'erreur standard la chaîne passée en second argument (si elle n'est pas NULL), suivie d'un deux-points et du descriptif du signal dont le numéro est fourni en premier argument. Cette fonction est plus portable que `strsignal()`, elle est définie dans `<signal.h>`.
- Il existe une table globale de chaînes de caractères contenant les libellés des signaux :

```
char * sys_siglist[numero_signal].
```

Les prototypes de `strsignal()` et `psignal()` sont les suivants :

```
char * strsignal (int numero_signal);  
int psignal (int numero_signal, const char * prefixe_affiche);
```

Voici un exemple permettant de consulter les libellés :

```
exemple-strsignal.c :  
#define _GNU_SOURCE  
#include <stdio.h>  
#include <stdlib.h>  
#include <signal.h>  
#include <string.h>  
#include <unistd.h>  
  
int main (void)  
{  
    int i;  
  
    fprintf(stdout, "strsignal() :\n");  
    for (i = 1; i < NSIG; i ++)  
        fprintf(stdout, "%s\n", strsignal(i));  
  
    fprintf (stdout, "\nsys_siglist[] : \n");  
    for (i = 1; i < NSIG; i ++)  
        fprintf(stdout, "%d : %s\n", i, sys_siglist[i]);  
}
```

```
    return EXIT_SUCCESS;  
}
```

Voici un exemple d'exécution (abrégé) :

```
$ ./exemple-strsignal  
strsignal() :  
Hangup  
Interrupt  
Quit  
Illegal instruction  
Trace/breakpoint trap  
Aborted  
Bus error  
Floating point exception  
Killed  
User defined signal 1  
Segmentation fault  
User defined signal 2  
Broken pipe  
Alarm clock  
Terminated  
Stack fault  
Child exited  
Continued  
Stopped (signal)  
Stopped  
Stopped (tty input)  
Stopped (tty output)  
Urgent I/O condition  
CPU time limit exceeded  
File size limit exceeded  
Virtual timer expired  
Profiling timer expired  
Window changed  
I/O possible  
Power failure  
Bad system call  
Unknown signal 32  
Unknown signal 33  
Real-time signal 0  
Real-time signal 1  
Real-time signal 2  
[...]
```

```
Real-time signal 28
Real-time signal 29
Real-time signal 30
sys_siglist[] :
1 : Hangup
2 : Interrupt
3 : Quit
4 : Illegal instruction
5 : Trace/breakpoint trap
6 : Aborted
7 : Bus error
8 : Floating point exception
9 : Killed
[...]
30 : Power failure
31 : Bad system call
32 : (null)
33 : (null)
[...]
63 : (null)
64 : (null)
$
```

Nous allons à présent étudier comment un signal est envoyé à un processus, et sous quelles conditions il lui parvient. Pour l'instant, nous parlerons des signaux classiques et nous réserverons les modifications concernant les signaux temps réel au chapitre 13.

Émission d'un signal sous Linux

Pour envoyer un signal à un processus, on utilise l'appel système `kill()`, qui est particulièrement mal nommé, car il ne tue pas toujours sa cible. Il est déclaré ainsi :

```
| int kill (pid_t pid, int numero_signal);
```

Le premier argument correspond au PID du processus visé, et le second au numéro du signal à envoyer. Rappelons qu'il est essentiel d'utiliser la constante symbolique correspondant au nom du signal et non la valeur numérique directe. En Python, la fonction `os.kill(pid, numero)` rend le même service.

Il existe une application `/bin/kill` qui sert de frontal en ligne de commande à l'appel système `kill()`. Cet utilitaire prend en option sur la ligne de commande un numéro de signal précédé d'un tiret « - » et suivi du ou des PID des processus à tuer. Le numéro de signal peut être remplacé par le nom symbolique du signal, avec ou sans le préfixe SIG.

Par exemple, sur Linux version x86, « `kill -13 1234` » est équivalent à « `kill -PIPE 1234` ». Si on ne précise pas de numéro de signal, `/bin/kill` utilise `SIGTERM` par défaut.

En fait, le premier argument de l'appel système `kill()` peut prendre diverses valeurs.

- S'il est strictement positif, le signal est envoyé au processus dont le PID correspond à cet argument.
- S'il est nul, le signal est envoyé à tous les processus du groupe auquel appartient le processus appelant.
- S'il est négatif, sauf pour `-1`, le signal est envoyé à tous les processus du groupe dont le PGID est égal à la valeur absolue de cet argument.
- S'il vaut `-1`, `SUSv4` indique que le comportement est indéfini. Sous Linux, le signal est envoyé à tous les processus sur le système, sauf au processus `init` (PID=1) et au processus appelant. Cette option est utilisée, par exemple dans les scripts de *shutdown*, pour tuer les processus avant de démonter les systèmes de fichiers et d'arrêter le système.

Bien entendu, l'émission du signal est soumise à des contraintes d'autorisation.

- Tout d'abord, un processus ayant son UID effectif nul ou la capacité `CAP_KILL` peut envoyer des signaux à tout processus.
- Un processus peut envoyer un signal à un autre processus si l'UID réel ou effectif de l'émetteur est égal à l'UID réel ou sauvé de la cible.
- Enfin, si le signal est `SIGCONT`, il suffit que le processus émetteur appartienne à la même session que le processus visé.

Dans tous les autres cas, l'appel système `kill()` échoue et remplit `errno` avec le code `EPERM`. Si toutefois aucun processus ne correspond au premier argument – quelles que soient les autorisations –, `errno` contiendra `ESRCH`. Le second argument de `kill()`, le numéro du signal à envoyer, doit être positif ou nul, et inférieur à `NSIG`, sinon `kill()` renvoie l'erreur `EINVAL`.

Nous voyons ici l'intérêt du signal numéro 0. Il ne s'agit pas vraiment d'un signal – rien n'est émis –, mais il permet de savoir si un processus est présent sur le système. Si c'est le cas, l'appel système `kill()` réussira si on peut atteindre le processus par un signal, ou il échouera avec `EPERM` si on n'en a pas le droit. Si le processus n'existe pas, il échouera avec `ESRCH`. Voici par exemple un moyen de l'employer :

```
int processus_present (pid_t pid)
{
    if (kill(pid, 0) == 0)
        return VRAI;
    if (errno == EPERM)
        return VRAI;
    return FAUX;
}
```

Bien entendu, il est toujours possible qu'un processus qui existait encore lors de l'appel `kill(pid, 0)` se soit terminé avant le retour de cette fonction, ou que l'effet inverse se produise.

Il existe une fonction de la bibliothèque C nommée `raise()`, et déclarée ainsi :

```
| void raise (int numero_signal);
```

Elle est équivalente à `kill(getpid(), numero_signal)` et permet de s'envoyer un signal. Elle n'a pas d'équivalent en Python, et il ne faut pas la confondre avec le mot-clé permettant de remonter une exception. Cette fonction est incluse dans la norme Ansi C, car elle ne nécessite pas de notion de PID, à la différence de `kill()`, qui elle est définie par SUSv4.

Un processus qui s'envoie un signal à lui-même, que ce soit par `raise(num)` ou par `kill(getpid(), num)`, est assuré que le signal sera délivré avant le retour de l'appel système `kill()`. Si le signal est capturé, le gestionnaire sera exécuté dans tous les cas avant le retour de `kill()`. Ce n'est bien entendu pas le cas lorsque le signal est destiné à un autre processus. Le signal est mis en attente (suspendu) jusqu'à ce qu'il soit effectivement délivré au processus visé.

Il existe également une fonction de bibliothèque nommée `killpg()` permettant d'envoyer un signal à un groupe de processus. Elle est déclarée ainsi :

```
| int killpg (pid_t pgid, int numero_signal)
```

Si le PGID indiqué est nul, elle envoie le signal au groupe du processus appelant. Comme on pouvait s'y attendre, elle fait directement appel à `kill(-pgid, numero_signal)` après avoir vérifié que PGID est positif ou nul. Un tel appel système est intéressant, car il peut détruire automatiquement tous ses descendants en se terminant. Pour cela, il suffit que le processus initial crée son propre groupe en invoquant `setpgrp()`, et il pourra tuer tous ses descendants en appelant `killpg(getpgrp(), SIGKILL)` avant de se finir.

L'émission des signaux est, nous le voyons, une chose assez simple et portable. Nous allons observer comment le noyau se comporte pour délivrer un signal en fonction de l'action programmée par le processus.

Délivrance des signaux

Un processus peut bloquer temporairement un signal. Si celui-ci arrive pendant ce temps, il reste en attente jusqu'à ce que le processus le débloque. Il est bien sûr

impossible de bloquer SIGKILL ou SIGSTOP. Le fait de bloquer un signal permet de protéger des parties critiques du code, par exemple l'accès à une variable globale modifiée également par le gestionnaire du signal en question.

Les concepteurs de Linux avaient choisi à l'origine de représenter la liste des signaux en attente pour une tâche par un masque de bits contenu dans un entier long. Chaque signal correspondait donc à un bit précis. Depuis Linux 2.2 a été introduit le concept des signaux temps réel, pour lesquels il existe une file d'attente des signaux. Un même signal temps réel peut être en attente en plusieurs occurrences. La représentation interne des signaux en attente a donc été modifiée pour devenir une table.

Un processus possède également un masque de bits indiquant quels sont les signaux bloqués. Lorsqu'un processus reçoit un signal bloqué, ce dernier reste en attente jusqu'à ce que le processus le débloque. Les signaux classiques ne sont pas empilés, ce qui signifie qu'une seule occurrence d'un signal donné est conservée lorsque plusieurs exemplaires arrivent consécutivement alors que le signal est bloqué. Les signaux temps réel sont empilés, ce qui veut dire que le noyau conserve l'ensemble des exemplaires d'un signal arrivé alors qu'il est bloqué.

À son réveil, le processus traite la liste des signaux en attente non bloqués. Un signal ignoré est simplement éliminé. Le système assure ensuite la gestion des signaux dont le comportement est celui par défaut (ignorer, stopper, arrêter, créer un fichier core). Sinon, il invoque le gestionnaire de signaux installé par l'utilisateur.

Il existe sous Unix des appels système rapides et des appels système lents. Un appel rapide est ininterrompible (hormis par une commutation de tâche de l'ordonnanceur). En revanche, un appel lent peut rester bloqué pendant une durée indéterminée. Savoir si un appel système est rapide ou lent n'est pas toujours simple. Classiquement, tous les appels concernant des descripteurs de fichiers (`open`, `read`, `write`, `fcntl`...) peuvent être lents dès lors qu'ils travaillent sur une socket réseau, un tube, voire des descripteurs de terminaux. Bien entendu, les appels système d'attente comme `wait()`, `select()`, `poll()` ou `pause()` peuvent attendre indéfiniment. Certains appels système servant aux communications interprocessus, comme `sem_get()` qui gère des sémaphores ou `mq_send()` et `mq_receive()` qui permettent de transmettre des messages, peuvent rester bloqués au gré du processus correspondant.

Prenons l'exemple d'une lecture depuis une connexion réseau. L'appel système `read()` est alors bloqué aussi longtemps que les données se font attendre. Si un signal non bloqué, pour lequel un gestionnaire a été installé, arrive pendant un appel système lent, ce dernier est interrompu. Le processus exécute alors le gestionnaire de signal. À la fin de l'exécution de celui-ci (dans le cas où il n'a pas mis fin au programme ni exécuté de saut non local), il y a plusieurs possibilités.

Le noyau peut faire échouer l'appel interrompu, qui transmet alors le code d'erreur `EINTR` dans `errno`. Le programme utilisateur devra alors réessayer son appel. Cela implique d'encadrer tous les appels système lents avec une gestion du type :

```
do {  
    nb_lus = read(socket_rcpt, buffer, nb_octets_a_lire);  
} (while ((nb_lus == -1) && (errno == EINTR)));
```

Cela est tout à fait utilisable si, dans les portions de code où des signaux sont susceptibles de se produire, on utilise peu d'appels système lents. Notons d'ailleurs que les fonctions de la bibliothèque C, par exemple `fread()`, gèrent elles-mêmes ce genre de cas.

En outre, le fait de faire volontairement échouer une lecture est un moyen d'éviter un blocage définitif, en utilisant un délai maximal par exemple. L'appel système `alarm()`, qui déclenche un signal `SIGALRM` lorsque le délai prévu est écoulé, est bien sûr le plus couramment utilisé.

```
alarm(delai_maximal_en_secondes);  
nb_lus = read(socket_rcpt, buffer, nb_octets_a_lire);  
  
alarm(0); /* Arrêter la temporisation si pas  
          écoulée entièrement */  
if (nb_lus != nb_octets_a_lire) {  
    if (errno == EINTR)  
        /* Délai dépassé... */
```

Ce code est très imparfait ; nous en verrons d'autres versions quand nous étudierons plus en détail l'alarme. De plus, nous ne savons pas quel signal a interrompu l'appel système, ce n'est pas nécessairement `SIGALRM`.

Il faut alors tester tous les retours de fonctions du type `read()` ou `write()`, et les relancer éventuellement si le signal reçu n'a pas d'influence sur le travail en cours. C'est d'autant plus contraignant avec le développement des applications fonctionnant en réseau, où une grande partie des appels système autrefois rapides – `read()` depuis un fichier sur disque – peuvent bloquer longuement, le temps d'interroger un serveur distant. La surcharge en termes de code nécessaire pour encadrer tous les appels système susceptibles de bloquer est parfois assez lourde.

Une autre possibilité, introduite initialement par les systèmes BSD, est de demander au noyau de relancer automatiquement les appels système interrompus. Ainsi, le code utilisant `read()` ne se rendra pas compte de l'arrivée du signal, le noyau ayant fait redémarrer l'appel système comme si de rien n'était. L'appel `read()` ne renverra jamais plus l'erreur `EINTR`.

Cela peut se configurer aisément, signal par signal. Il est donc possible de demander que tous les signaux pour lesquels un gestionnaire est fourni fassent redémarrer automatiquement les appels système interrompus, à l'exception par exemple de `SIGALRM` qui peut servir à programmer un délai maximal. Dans l'exemple précédent, la lecture reprendra automatiquement et ne se terminera que sur une réussite ou une réelle condition d'erreur, sauf bien entendu si on la temporise avec `alarm()`.

Nous allons à présent étudier le moyen de configurer le comportement d'un processus à la réception d'un signal précis.

Réception des signaux avec l'appel système `signal()`

Un processus peut demander au noyau d'installer un gestionnaire pour un signal particulier, c'est-à-dire une routine spécifique qui sera invoquée lors de l'arrivée de ce signal. Le processus peut aussi vouloir que le signal soit ignoré lorsqu'il arrive, ou laisser le noyau appliquer le comportement par défaut (souvent une terminaison du programme).

Pour indiquer son choix au noyau, il y a deux possibilités.

- L'appel système `signal()`, défini par Ansi C et SUSv4, présente l'avantage d'être très simple (on installe un gestionnaire en une seule ligne de code), mais il peut parfois poser des problèmes de fiabilité de délivrance des signaux et de compatibilité entre les divers systèmes Unix. Il est parfaitement implémenté en Python, de manière portable et déterministe.
- L'appel système `sigaction()` est légèrement plus complexe puisqu'il implique le remplissage d'une structure, mais il permet de définir précisément le comportement désiré pour le gestionnaire, sans ambiguïté suivant les systèmes puisqu'il est complètement défini par SUSv4. Cet appel n'est pas implémenté par Python, car dans ce langage `signal()` ne pose pas les problèmes de portabilité de son homologue en C.

Nous allons tout d'abord voir la syntaxe et l'utilisation de `signal()`, car il est souvent employé, puis nous étudierons dans le prochain chapitre `sigaction()`, qui est généralement plus adéquat pour contrôler finement le comportement d'un programme. Notons au passage l'existence d'une ancienne fonction, `sigvec()`, obsolète de nos jours et approximativement équivalente à `sigaction()`.

L'appel système `signal()` présente un prototype qui surprend toujours au premier coup d'œil, alors qu'il est extrêmement simple en réalité :

```
| void (*signal (int numero_sig, void (*gestionnaire) (int)))(int);
```

Il suffit en fait de le décomposer, en créant un type intermédiaire correspondant à un pointeur sur une routine de gestion de signaux :

```
| typedef void (* gestion_t)(int);
```

et le prototype de `signal()` devient :

```
| gestion_t signal (int numero_signal, gestion_t gestionnaire);
```

En d'autres termes, `signal()` prend en premier argument un numéro de signal. Bien entendu, il faut utiliser la constante symbolique correspondant au nom du signal plutôt que la valeur numérique directe. Le second argument est un pointeur sur la routine qu'on désire installer comme gestionnaire de signal. L'appel système nous renvoie un pointeur sur l'ancien gestionnaire, ce qui permet de le sauvegarder pour éventuellement le réinstaller plus tard.

La fonction Python `signal.signal(num, handler)` est équivalente.

Il existe deux constantes symboliques qui peuvent remplacer le pointeur sur un gestionnaire, `SIG_IGN` et `SIG_DFL`, qui sont définies dans `<signal.h>` (`signal.SIG_IGN` et `signal.SIG_DFL` en Python).

La constante `SIG_IGN` demande au noyau d'ignorer le signal indiqué. Par exemple l'appel système `signal(SIGINT, SIG_IGN)` protège le processus contre les tentatives d'arrêt par un *Contrôle-C*.

Avec la constante `SIG_DFL`, on demande au noyau de réinstaller le comportement par défaut pour le signal considéré. Nous avons vu l'essentiel des actions par défaut. Elles sont également documentées dans la page de manuel `signal(7)`.

Si l'appel système `signal()` échoue, il renvoie une valeur particulière, elle aussi définie dans `<signal.h>` : `SIG_ERR`.

L'erreur positionnée dans `errno` est alors généralement `EINVAL`, qui indique un numéro de signal inexistant. Si on essaie d'ignorer ou d'installer un gestionnaire pour les signaux `SIGKILL` ou `SIGSTOP`, l'opération n'a pas lieu. La documentation de la fonction `signal()` de Glibc indique que la modification est silencieusement ignorée, mais en réalité l'appel système `sigaction()` – interne au noyau –, sur lequel cette fonction est bâtie, renvoie `EINVAL` dans `errno` dans ce cas.

L'erreur `EFAULT` peut aussi être renvoyée dans `errno` si le pointeur de gestionnaire de signal n'est pas valide.

Un gestionnaire de signal est une routine comme les autres, qui prend un argument de type entier et qui ne renvoie rien. L'argument transmis correspond au numéro du signal ayant déclenché le gestionnaire. Il est donc possible d'écrire un unique ges-

tionnaire pour plusieurs signaux, en répartissant les actions à l'aide d'une construction switch-case.

Il arrive que le gestionnaire de signal puisse recevoir d'autres informations dans une structure transmise en argument supplémentaire (comme le PID du processus ayant envoyé le signal). Pour cela, il faut installer nécessairement le gestionnaire avec l'appel système `sigaction()` que nous verrons plus bas.

Le gestionnaire de signal étant une routine sans spécificité, il est possible de l'invoquer directement dans le corps du programme si le besoin s'en fait sentir.

Nous allons pouvoir installer notre premier gestionnaire de signal. Nous allons tenter de capturer tous les signaux. Bien entendu, `signal()` échouera pour `SIGKILL` et `SIGSTOP`. Pour tous les autres signaux, notre programme affichera son PID suivi du numéro de signal et de son nom. Il faudra disposer d'une seconde console (ou d'un autre Xterm) pour pouvoir le tuer finalement.

```
exemple-signal.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void gestionnaire (int numero)
{
    fprintf(stdout, "\n%d a reçu le signal %d (%s)\n",
            (long) getpid(), numero, sys_siglist[numero]);
}

int main (void)
{
    int i;

    for (i = 1; i < NSIG; i++)
        if (signal(i, gestionnaire) == SIG_ERR)
            fprintf(stderr, "Signal %d non capture\n", i);

    while (1)
        pause();
}
```

Voici un exemple d'exécution avec, en seconde colonne, l'action effectuée sur un autre terminal :

```
$ ./exemple-signal
Signal 9 non capture
```

```

Signal 19 non capture
Signal 32 non capture
Signal 33 non capture
  (Contrôle-C)
10899 a reçu le signal 2 (Interrupt)
  (Contrôle-AltGr-\)
10899 a reçu le signal 3 (Quit)
  (Contrôle-Z)
10899 a reçu le signal 20 (Stopped)
                                     $ kill 10899

10899 a reçu le signal 15 (Terminated)
                                     $ kill -KILL 10899

Processus arrêté
$

```

Le signal 9 non capturé correspond à SIGKILL, et le 19 à SIGSTOP, quant à 32 et 33, il s'agit de signaux temps réel que la Glibc réserve pour son propre usage. Elle empêche donc leur emploi par les applications.

Ce programme a également un comportement intéressant vis-à-vis du signal SIGSTOP, qui le stoppe temporairement. Le shell reprend alors la main. Nous pouvons toutefois ramener le processus en avant-plan, ce qui lui transmet le signal SIGCONT :

```

$ ./exemple-signal
Signal 9 non capture
Signal 19 non capture
Signal 32 non capture
Signal 33 non capture
  (Contrôle-C)
10975 a reçu le signal 2 (Interrupt)
                                     $ kill -STOP 10975

[1]+  Stopped      ./exemple-signal
$
                                     $ ps 10975
                                     PID TTY      STAT TIME COMMAND
                                     10975 pts/2  T  0:00 ./exemple-signal

$ fg
./exemple-signal
10975 a reçu le signal 18 (Continued)
                                     $ kill -KILL 10975

Processus arrêté
$

```

Le champ STAT de la commande ps contient T, ce qui correspond à un processus stoppé ou suivi (*traced*).

En Python, le gestionnaire de signal reçoit deux arguments : le numéro du signal reçu comme en C et un objet représentant la pile d'appel du processus. Le comportement du script suivant est exactement le même que celui du programme précédent.

```
exemple-signal.py :  
#!/usr/bin/python  
  
from __future__ import print_function  
import signal  
import os  
  
def handler(num, stack):  
    print("Signal", num, "recu")  
  
for sig in range(1, signal.NSIG):  
    try:  
        signal.signal(sig, handler)  
    except:  
        print("Unable to catch signal", sig)  
  
while True:  
    signal.pause()
```

Il faut savoir que sous Linux, la constante symbolique SIG_DFL est définie comme valant 0 (c'est souvent le cas, même sur d'autres systèmes Unix). Lors de la première installation d'un gestionnaire, l'appel système signal() renvoie donc, la plupart du temps, cette valeur (à moins que le shell n'ait modifié le comportement des signaux auparavant). Il y a là un risque d'erreur pour le programmeur distrait qui peut écrire machinalement :

```
if (signal(...) != 0)  
    /* Erreur */
```

comme on a l'habitude de le faire pour d'autres appels système. Ce code fonctionnera à la première invocation, mais échouera par la suite puisque signal() renvoie l'adresse de l'ancien gestionnaire. Ne pas oublier, donc, de détecter les erreurs ainsi :

```
if (signal(...) == SIG_ERR)  
    /* Erreur */
```

Nous avons pris soin dans l'exécution de l'exemple précédent de ne pas invoquer deux fois de suite le même signal. Pourtant, cela n'aurait pas posé de problème avec Linux et la Glibc, comme en témoigne l'essai suivant :

```
$ ./exemple-signal
Signal 9 non capture
Signal 19 non capture
Signal 32 non capture
Signal 33 non capture
  (Contrôle-C)
11628 a reçu le signal 2 (Interrupt)
  (Contrôle-C)
11628 a reçu le signal 2 (Interrupt)
  (Contrôle-C)
11628 a reçu le signal 2 (Interrupt)
```

Toutefois, il existe de nombreux systèmes Unix (de la famille Système V) sur lesquels un gestionnaire de signal ne reste pas en place après avoir été invoqué. Une fois que le signal est arrivé, le noyau repositionne le comportement par défaut. Ce dernier peut être observé sous Linux avec Glibc en définissant la constante symbolique `_XOPEN_SOURCE` avant d'inclure `<signal.h>`. En voici un exemple :

```
exemple-signal-2.c :
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void gestionnaire (int numero)
{
    fprintf(stdout, "\n%d a reçu le signal %d\n",
            (long) getpid(), numero);
}
[... suite identique à exemple-signal.c ]
```

Voici un exemple d'exécution dans lequel on remarque que la première frappe de *Contrôle-Z* est interceptée, mais pas la seconde, qui stoppe le processus et redonne la main au shell. On redémarre alors le programme avec la commande `fg`, et on presse *Contrôle-C*. Sa première occurrence sera bien interceptée, mais pas la seconde.

```
$ ./exemple-signal-2
Signal 9 non capture
Signal 19 non capture
Signal 32 non capture
Signal 33 non capture
```

```

    (Contrôle-Z)
11795 a reçu le signal 20
    (Contrôle-Z)
[1]+  Stopped                  ./exemple-signal-2
$ fg
./exemple-signal-2

11795 a reçu le signal 18
    (Contrôle-C)
11795 a reçu le signal 2
    (Contrôle-C)
$

```

Le signal 18 correspond à SIGCONT, que le shell a envoyé en remplaçant le processus en avant-plan. Sur ce type de système, il est nécessaire que le gestionnaire de signaux s'installe à nouveau à chaque interception d'un signal. On doit donc utiliser un code du type :

```

int gestionnaire (int numero_signal)
{
    signal(numero_signal, gestionnaire);
    /* Traitement effectif du signal reçu */
}

```

Il est toutefois possible que le signal arrive de nouveau avant que le gestionnaire ne soit réinstallé. Ce type de comportement à risque conduit à avoir des signaux non fiables.

Un deuxième problème se pose avec ces anciennes versions de `signal()` pour ce qui concerne le blocage des signaux. Lorsqu'un signal est capturé et que le processus exécute le gestionnaire installé, le noyau ne bloque pas une éventuelle occurrence du même signal. Le gestionnaire peut alors se trouver rappelé au cours de sa propre exécution. Nous allons le démontrer avec ce petit exemple, dans lequel un processus fils envoie deux signaux à court intervalle à son père, lequel utilise un gestionnaire lent, qui compte jusqu'à 3.

```

exemple-signal-3.c :
#define _XOPEN_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void gestionnaire (int numero)
{
    int i;

```



```
    signal(numero, gestionnaire);
    fprintf(stdout, "debut du gestionnaire de signal %d\n", numero);
    for (i = 1; i < 4; i++) {
        fprintf(stdout, "%d\n", i);
        sleep(1);
    }
    fprintf(stdout, "fin du gestionnaire de signal %d\n", numero);
}

int main (void)
{
    signal(SIGUSR1, gestionnaire);

    if (fork() == 0) {
        kill(getppid(), SIGUSR1);
        sleep(1);
        kill(getppid(), SIGUSR1);
    } else {
        while (1)
            pause();
    }
    return EXIT_SUCCESS;
}
```

Voici ce que donne l'exécution de ce programme :

```
$ ./exemple-signal-3
début du gestionnaire de signal 10
1
début du gestionnaire de signal 10
1
2
3
fin du gestionnaire de signal 10
2
3
fin du gestionnaire de signal 10
(Contrôle-C)
$
```

Les deux comptages sont enchevêtrés, ce qui n'est pas grave car la variable *i* est allouée de manière automatique dans la pile, et il y a donc deux compteurs différents pour les deux invocations du gestionnaire. Mais cela pourrait se passer autrement si la variable de comptage était statique ou globale. Il suffit de déplacer le « `int i` » pour le placer en variable globale avant le gestionnaire, et on obtient l'exécution suivante :

```
$ ./exemple-signal-3
début du gestionnaire de signal 10
1
début du gestionnaire de signal 10
1
2
3
fin du gestionnaire de signal 10
fin du gestionnaire de signal 10
(Contrôle-C)
$
```

Cette fois-ci, le compteur global était déjà arrivé à 4 lorsqu'on est revenu dans le premier gestionnaire, celui qui avait lui-même été interrompu par le signal. Pour éviter ce genre de désagrément, la version moderne de `signal()`, disponible sous Linux, bloque automatiquement un signal lorsqu'on exécute son gestionnaire, puis le débloque au retour. On peut le vérifier en supprimant la ligne `#define _XOPEN_SOURCE` et on obtient (même en laissant le compteur en variable globale) :

```
$ ./exemple-signal-3
début du gestionnaire de signal 10
1
2
3
fin du gestionnaire de signal 10
début du gestionnaire de signal 10
1
2
3
fin du gestionnaire de signal 10
(Contrôle-C)
$
```

Comme on pouvait s'y attendre, les deux exécutions du gestionnaire de signal sont séparées. On peut noter au passage que si on rajoute un troisième

```
sleep(1);
kill(getppid(), SIGUSR1);
```

dans le processus fils, il n'y a pas de différence d'exécution. Seules deux exécutions du gestionnaire ont lieu. C'est dû au fait que, sous Linux, les signaux classiques ne sont pas empilés, et l'arrivée du troisième `SIGUSR1` se fait alors que le premier gestionnaire

n'est pas terminé. Aussi, un seul signal est mis en attente. Remarquons également que lorsqu'on élimine la définition `_XOPEN_SOURCE`, on peut supprimer l'appel `signal()` à l'intérieur du gestionnaire : celui-ci est automatiquement réinstallé, comme on l'a déjà indiqué.

Bien sûr, toutes ces expérimentations tablent sur le fait que l'exécution des processus se fait de manière homogène, sur un système peu chargé. Si tel n'est pas le cas, les temps de commutation entre les processus père et fils, ainsi que les délais de délivrance des signaux, peuvent modifier les comportements de ces exemples.

Nous voyons que la version de `signal()` disponible sous Linux, héritée de celle de BSD, est assez performante et fiable puisqu'elle permet, d'une part, une réinstallation automatique du gestionnaire lorsqu'il est invoqué et, d'autre part, un blocage du signal concerné au sein de son propre gestionnaire. Une dernière question se pose, qui concerne le redémarrage automatique des appels système lents interrompus.

Pour cela, la bibliothèque Glibc nous offre une fonction de contrôle nommée `siginterrupt()`.

```
| int siginterrupt (int numero, int interrompre);
```

Elle prend en argument un numéro de signal, suivi d'un indicateur booléen. Elle doit être appelée après l'installation du gestionnaire et, si l'indicateur est nul, les appels système lents seront relancés automatiquement. Si l'indicateur est non nul, les appels système échouent, avec une erreur `EINTR` dans `errno`.

Voici un petit programme qui prend une valeur numérique en argument et la transmet à `siginterrupt()` après avoir installé un gestionnaire pour le signal `TSTP` (touche *Contrôle-Z*). Il exécute ensuite une lecture bloquante depuis le descripteur de fichier 0 (entrée standard). Le programme nous indique à chaque frappe sur *Contrôle-Z* si la lecture est interrompue ou non. On peut terminer le processus avec *Contrôle-C*.

```
| exemple-siginterrupt.c :
| #include <stdio.h>
| #include <stdlib.h>
| #include <unistd.h>
| #include <errno.h>
| #include <signal.h>
|
| void gestionnaire (int numero)
| {
|     fprintf(stdout, "\ngestionnaire de signal %d\n", numero);
| }
```

```
int main (int argc, char *argv[])
{
    int i;

    if ((argc != 2) || (sscanf(argv[1], "%d", & i) != 1)) {
        fprintf (stderr, "Syntaxe : %s {0|1}\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    signal(SIGTSTP, gestionnaire);
    siginterrupt(SIGTSTP, i);

    while (1) {
        fprintf(stdout, "appel read()\n");
        if (read(STDIN_FILENO, &i, sizeof (int)) < 0)
            if (errno == EINTR)
                fprintf(stdout, "EINTR \n");
    }
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution :

```
$ ./exemple-siginterrupt 0
appel read()
  (Contrôle-Z)
gestionnaire de signal 20
  (Contrôle-Z)
gestionnaire de signal 20
  (Contrôle-C)
$ ./exemple-siginterrupt 1
appel read()
  (Contrôle-Z)
gestionnaire de signal 20
EINTR
appel read()
  (Contrôle-Z)
gestionnaire de signal 20
EINTR
appel read()
  (Contrôle-C)
$
```

En supprimant la ligne `siginterrupt()`, on s'aperçoit que le comportement est identique à « `exemple_siginterrupt 0` ». Les appels système lents sont donc relancés automatiquement sous Linux par défaut. Si, en revanche, nous définissons la constante `_XOPEN_SOURCE` comme nous l'avons fait précédemment, en supprimant la ligne `siginterrupt()`, on observe que les appels lents ne sont plus relancés.

Conclusion

Nous voyons donc que l'appel système `signal()` donne accès, sous Linux, avec la bibliothèque Glibc, à des signaux fiables. Nous pouvons aussi compiler des sources datant d'anciens systèmes Unix et se fondant sur un comportement moins fiable de `signal()`, simplement en définissant des constantes symboliques à la compilation (consulter à ce sujet le fichier `/usr/include/features.h`). Malheureusement, ce n'est pas le cas sur tous les systèmes, aussi est-il préférable d'employer la fonction `sigaction()`, que nous allons étudier dans le prochain chapitre et qui permet un paramétrage plus souple du comportement du programme.

L'appel `signal()` doit surtout être réservé au cas où l'on désire ignorer un signal ou lui rendre son comportement par défaut. Pour installer un gestionnaire dans une application portable, on préférera `sigaction()`.

En revanche, l'implémentation en Python est parfaitement fiable (modèle BSD) quelle que soit la plate-forme Unix.

12

Gestion portable des signaux

La gestion des signaux de manière portable n'est pas beaucoup plus compliquée que ce que nous avons vu dans le chapitre précédent. L'appel `systeme_sigaction()` que nous allons étudier tout d'abord permet de réaliser toutes les opérations de configuration du gestionnaire et du comportement des signaux. On notera qu'il n'existe pas d'équivalent en Python ; ce n'est pas nécessaire car le comportement `signal.signal()` est parfaitement défini suivant le modèle BSD.

Nous examinerons ensuite le principe des ensembles de signaux, qui permettent d'assurer les blocages temporaires avec `sigprocmask()`. Enfin, nous observerons les « bonnes manières » d'écrire un gestionnaire de signal, ce que nous mettrons en pratique avec une étude de l'alarme `SIGALRM`.

Réception des signaux avec `sigaction()`

La routine `sigaction()` prend trois arguments et renvoie un entier valant 0 si elle réussit, et -1 si elle échoue. Le premier argument est le numéro du signal (comme toujours, il faut utiliser la constante symbolique définissant le nom du signal). Les deux autres arguments sont des pointeurs sur des structures `sigaction` (pas d'inquiétude, il n'y a pas d'ambiguïté avec le nom de la routine dans la table des symboles du compilateur). Ces structures définissent précisément le comportement à adopter en cas de réception du signal considéré. Le premier pointeur est le nouveau comportement à programmer, alors que le second pointeur sert à sauvegarder l'ancienne action. Le prototype est donc le suivant :

```
int sigaction (int numero,  
               const struct sigaction * nouvelle,  
               struct sigaction * ancienne);
```

Si le numéro indiqué est inférieur ou égal à 0, supérieur ou égal à NSIG, ou égal à SIGKILL ou SIGSTOP, `sigaction()` échoue en plaçant EINVAL dans `errno`.

Si le pointeur sur la nouvelle structure `sigaction` est NULL, aucune modification n'a lieu, seul l'ancien comportement est sauvegardé dans le second pointeur. Parallèlement, si le pointeur sur l'ancienne structure est NULL, aucune sauvegarde n'a lieu.

Voyons maintenant le détail de la structure `sigaction`. Celle-ci est définie dans `<sigaction.h>` qui est inclus par `<signal.h>`.

Nom	Type
<code>sa_handler</code>	<code>sighandler_t</code>
<code>sa_mask</code>	<code>sigset_t</code>
<code>sa_flags</code>	<code>int</code>
<code>sa_restorer</code>	<code>void (*) (void)</code>

Attention ! L'ordre des membres de cette structure n'est pas fixé suivant les systèmes, et il a changé entre différentes versions du noyau Linux. Il ne faut donc surtout pas le considérer comme immuable et éviter par exemple d'initialiser la structure de manière statique.

Le premier membre de cette structure correspond à un pointeur sur le gestionnaire du signal, comme nous en transmettions à l'appel système `signal()`. Le champ `sa_handler` peut également prendre comme valeur `SIG_IGN` pour ignorer le signal ou `SIG_DFL` pour appliquer l'action par défaut. Un gestionnaire doit donc être défini ainsi :

```
void gestionnaire_signal (int numero);
```

Le second membre est du type `sigset_t`, c'est-à-dire un ensemble de signaux. Nous verrons plus bas des fonctions permettant de configurer ce type de données. Cet élément correspond à la liste des signaux qui sont bloqués pendant l'exécution du gestionnaire. Le signal ayant déclenché l'exécution du gestionnaire est automatiquement bloqué, sauf si on demande explicitement le contraire (voir page ci-contre `SA_NODEFER`). Une tentative de blocage de SIGKILL ou SIGCONT est silencieusement ignorée.

Enfin, le troisième membre `sa_flags` contient un OU binaire entre différentes constantes permettant de configurer le comportement du gestionnaire de signal.

Nom	Signification
SA_NOCLDSTOP	Cette constante ne concerne que le gestionnaire pour le signal SIGCHLD. Lorsqu'elle est présente, ce gestionnaire n'est pas invoqué lorsqu'un processus fils a été stoppé temporairement (avec le signal SIGSTOP ou SIGTSTP). Par contre, il sera appelé pour les processus fils qui se terminent définitivement. Pour tout autre signal que SIGCHLD, cette constante est ignorée.
SA_RESTART	Lorsqu'elle est présente, cette constante indique que les appels système lents interrompus par le signal concerné sont automatiquement redémarrés. On l'utilise généralement pour tous les signaux, sauf pour SIGALRM s'il sert à installer un délai maximal pour une fonction pouvant rester bloquée.
SA_INTERRUPT	Cette constante n'est pas définie par SUSv4. Elle a le comportement exactement inverse de SA_RESTART, en empêchant le redémarrage automatique des appels système interrompus. Il suffit, pour porter sous Linux des programmes utilisant cette valeur, de la supprimer (éventuellement avec un <code>#define SA_INTERRUPT 0</code>) et de ne pas utiliser la constante SA_RESTART dans ce cas.
SA_NODEFER	Demander explicitement qu'un signal ne soit pas bloqué à l'intérieur de son propre gestionnaire. En réalité, sous Linux, elle est identique à SA_NOMASK, constante qui empêche de bloquer les signaux mentionnés dans <code>sa_mask</code> . Sa portée est donc plus grande. Si on désire obtenir le comportement « classique » de SA_NODEFER, il vaut mieux supprimer le signal considéré de l'ensemble de signaux de <code>sa_mask</code> .
SA_RESETHAND	Lorsqu'un gestionnaire de signal est invoqué et que cette constante a été fournie à <code>sigaction()</code> , le comportement par défaut est réinstallé pour le signal concerné. C'était le comportement normal des premières versions de <code>signal()</code> , ce que nous avons forcé avec la constante <code>_XOPEN_SOURCE</code> dans notre programme <code>exemple-signal-2.c</code> .
SA_ONESHOT	Cette constante n'est pas définie par SUSv4. Elle est équivalente à SA_RESETHAND.
SA_SIGINFO	Il s'agit d'une valeur décrite par SUSv4 pour les signaux temps réel, mais qui peut être utilisée aussi pour les signaux classiques. Un gestionnaire de signaux installé avec cette option recevra des informations supplémentaires, en plus du numéro du signal qui l'a déclenché. Le gestionnaire doit accepter trois arguments : le premier est toujours le numéro du signal, le second est un pointeur sur une structure de type <code>siginfo_t</code> , le troisième argument, de type <code>void *</code> , n'est pas documenté dans les sources du noyau. Nous détaillerons cette possibilité plus bas.
SA_ONSTACK	Dans ce cas, le gestionnaire du signal en question utilise une pile différente de celle du reste du programme. Nous fournirons plus loin un exemple d'utilisation de cette possibilité.

Lorsqu'on utilise l'attribut SA_SIGINFO, on considère que la structure `sigaction` contient un champ supplémentaire, nommé `sa_sigaction`, permettant de stocker le pointeur sur le gestionnaire. En réalité, tout est souvent implémenté sous forme d'union, une seule et même zone servant à stocker l'adresse des différents types de gestionnaires, mais les prototypes différents permettant une vérification à la compilation. Cela signifie aussi que les noms des membres présents dans la structure

sigaction peuvent être en réalité des macros permettant d'accéder à des champs dont le nom est plus complexe, et qu'il faut éviter, sous peine de voir le compilateur échouer, d'appeler une variable `sa_sigaction`, par exemple.

Le gestionnaire de signal devra dans ce cas avoir la forme suivante :

```
void gestionnaire_signal (int numero,
                        siginfo_t * info,
                        void * inutile);
```

La structure `siginfo_t` peut également être implémentée de manière assez complexe, avec des champs en union. De manière portable, on peut accéder aux membres suivants, définis par SUSv4.

Nom	Signification
<code>si_signo</code>	Indique le numéro de signal.
<code>si_sigval</code>	Est à son tour une union qui n'est utilisée qu'avec les signaux temps réel, et que nous détaillerons donc ultérieurement.
<code>si_code</code>	Indique la provenance du signal. Il s'agit d'une combinaison binaire par OU entre diverses constantes, variant en fonction du signal reçu. Si <code>si_code</code> est strictement positif, le signal provient du noyau. Sinon, il provient d'un utilisateur (même root). Nous verrons des exemples avec les signaux temps réel.
<code>si_errno</code>	Ce champ contient la valeur de <code>errno</code> lors de l'invocation. Permet par exemple de la rétablir en sortie de gestionnaire.
<code>si_pid</code> et <code>si_uid</code>	Ces membres ne sont valides que si le signal provient d'un utilisateur (<code>si_code</code> négatif ou nul), ou si le signal <code>SIGCHLD</code> a été émis par le noyau. Ils identifient l'émetteur du signal ou le processus fils qui s'est terminé.

Les informations fournies grâce à la structure `siginfo_t` peuvent être très importantes en termes de sécurité pour des programmes susceptibles d'avoir des privilèges particuliers (*Set-UID root*). Cela permet de vérifier que le signal est bien émis par le noyau et non par un utilisateur essayant d'exploiter une faille de sécurité. Nous reparlerons de ces données dans le prochain chapitre, car elles concernent également les signaux temps réel.

Lorsqu'on utilise l'attribut `SA_ONSTACK` lors de l'invocation de `sigaction()`, la pile est alors sauvegardée. Il faut déclarer une pile différente à l'aide de l'appel système `sigaltstack()`. Les variables locales automatiques étant utilisées dans les routines, y compris les gestionnaires de signaux alloués dans la pile, il peut être intéressant dans certains cas de réserver avec `malloc()` une place mémoire suffisamment importante pour accueillir des variables assez volumineuses. Les constantes `MINSIGSTKSZ` et `SIGSTKSZ`, définies dans `<signal.h>`, correspondent respectivement à la taille minimale et à la taille optimale pour la pile réservée à un gestionnaire de signal. Une structure de type `stack_t`, définie dans `<signal.h>`, contient trois champs.

Nom	Type	Signification
ss_sp	void *	Pointeur sur la pile.
ss_flags	int	Attributs.
ss_size	size_t	Taille de la pile.

On alloue donc la place voulue dans le champ `ss_sp` d'une variable `stack_t`, puis on invoque l'appel `sigaltstack()` en lui fournissant cette nouvelle pile. Elle sera alors utilisée pour tous les gestionnaires de signaux qui emploient l'attribut `SA_ONSTACK` dans leur installation par `sigaction()`. Le prototype de `sigaltstack()` est le suivant :

```
| int sigaltstack (stack_t * nouvelle, stack_t * ancienne);
```

L'appel permet éventuellement de sauvegarder l'ancienne pile en utilisant un second argument non nul. Dans le cas où le premier argument est `NULL`, aucune modification n'a lieu, on obtient simplement l'état actuel de la pile. Cela permet de vérifier si cette pile est en cours d'utilisation ou non. Les deux constantes symboliques `SS_DISABLE` et `SS_ONSTACK` indiquent respectivement dans le champ `ss_flags` que la pile est désactivée ou qu'elle est en cours d'utilisation.

Nous verrons un exemple d'utilisation de pile spécifique pour les gestionnaires de signaux à la fin du paragraphe sur les exemples d'utilisation de `sigaction()`.

Configuration des ensembles de signaux

Avant de voir des exemples d'utilisation de `sigaction()`, nous allons regarder les différentes primitives permettant de modifier les ensembles de signaux de type `sigset_t`. Ce type est opaque, et il faut absolument utiliser les routines décrites ci-dessous pour y accéder.

À titre d'exemple, nous rappelons que le noyau Linux a considéré successivement `sigset_t` comme un `unsigned long`, un tableau de 2 `unsigned long`, et une structure, et la bibliothèque Glibc comme un tableau de 32 `unsigned long` (se réservant de la place pour des extensions jusqu'à 1 024 signaux). La définition réelle du type `sigset_t` peut donc varier suivant les machines, mais également selon les versions du noyau et même le type de fichier d'en-tête utilisé (noyau ou bibliothèque C).

Les routines suivantes sont définies par SUSv4 :

```
| int sigemptyset(sigset_t * ensemble);  
| int sigfillset (sigset_t * ensemble);  
| int sigaddset (sigset_t * ensemble, int numero_signal);  
| int sigdelset (sigset_t * ensemble, int numero_signal);  
| int sigismember(const sigset_t * ensemble, int numero_signal);
```

La première routine, `sigemptyset()`, permet de vider un ensemble, c'est-à-dire de l'initialiser sans aucun signal. Il ne faut pas utiliser une initialisation du genre « `ensemble=0` », car elle n'est pas suffisante dans le cas où le type `sigset_t` est un tableau (dans la Glibc, par exemple). Parallèlement, `sigfillset()` permet de remplir un ensemble avec tous les signaux connus sur le système. Ces deux routines renvoient 0 si elles réussissent et -1 sinon, c'est-à-dire si `ensemble` vaut NULL ou pointe sur une zone mémoire invalide.

Les routines `sigaddset()` et `sigdelset()` permettent respectivement d'ajouter un signal à un ensemble ou d'en supprimer. Elles renvoient 0 si elles réussissent ou -1 si elles échouent (si le numéro de signal est invalide ou si `ensemble` est NULL). Le fait d'ajouter un signal à un ensemble qui le contient déjà ou de supprimer un signal d'un ensemble auquel il n'appartient pas ne constitue pas une erreur.

La dernière routine, `sigismember()`, permet de savoir si un signal appartient à un ensemble ou pas ; elle renvoie 1 si c'est le cas, ou 0 sinon. Elle peut également renvoyer -1 en cas d'erreur.

La bibliothèque Glibc ajoute trois fonctionnalités, qui sont des extensions Gnu (nécessitant donc la constante `_GNU_SOURCE` à la compilation), et qui permettent de manipuler les ensembles de signaux de manière globale :

```
int sigisemptyset (const sigset_t * ensemble)
int sigandset (sigset_t * ensemble_resultat,
               const sigset_t * ensemble_1,
               const sigset_t * ensemble_2)
int sigorset (sigset_t * ensemble_resultat,
              const sigset_t * ensemble_1,
              const sigset_t * ensemble_2)
```

`sigisemptyset()` indique si l'ensemble considéré est vide. `sigandset()` permet d'effectuer un ET binaire entre deux ensembles de signaux et d'obtenir dans l'ensemble résultat la liste des signaux qui leur sont communs. `sigorset()` permet symétriquement d'effectuer un OU binaire pour obtenir en résultat l'ensemble des signaux présents dans l'un ou l'autre des deux ensembles.

Exemples d'utilisation de `sigaction()`

Notre premier exemple consistera à installer deux gestionnaires : l'un pour SIGQUIT (que nous déclenchons au clavier avec *Contrôle-Alt-Gr-*), qui ne fera pas redémarrer les appels système lents interrompus, le second, celui de SIGINT (*Contrôle-C*), aura pour particularité de ne pas se réinstaller automatiquement. La seconde occurrence de SIGINT déclenchera donc le comportement par défaut et arrêtera le processus.

```
exemple-sigaction-1.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void gestionnaire (int numero)
{
    switch (numero) {
        case SIGQUIT :
            fprintf(stdout, "\nSIGQUIT reçu\n");
            break;
        case SIGINT :
            fprintf(stdout, "\nSIGINT reçu\n");
            break;
    }
}

int main (void)
{
    struct sigaction action;

    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = 0;

    if (sigaction(SIGQUIT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur %d\n", errno);
        exit(EXIT_FAILURE);
    }

    action.sa_flags = SA_RESTART | SA_RESETHAND;

    if (sigaction(SIGINT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur %d\n", errno);
        exit(EXIT_FAILURE);
    }

    /* Lecture continue, pour avoir un appel système lent bloqué */
    while (1) {
        int i;
        fprintf(stdout, "appel read()\n");
        if (read(0, & i, sizeof(int)) < 0)
            if (errno == EINTR)
                fprintf(stdout, "EINTR \n");
    }
    return EXIT_SUCCESS;
}
```

L'exécution de ce programme montre bien les différences de caractéristiques entre les signaux :

```
$ ./exemple-sigaction-1
appel read()
  (Contrôle-AltGr-\)
SIGQUIT reçu
EINTR
appel read()
  (Contrôle-C)
SIGINT reçu
  (Contrôle-AltGr-\)
SIGQUIT reçu
EINTR
appel read()
  (Contrôle-C)
$
```

SIGQUIT interrompt bien l'appel `read()`, mais pas SIGINT. De même, le gestionnaire de SIGQUIT reste installé et peut être appelé une seconde fois, alors que SIGINT reprend son comportement par défaut et termine le processus la seconde fois.

Nous n'avons pas sauvegardé l'ancien gestionnaire de signaux lors de l'appel de `sigaction()` (troisième argument). Il est pourtant nécessaire de le faire si nous installons temporairement une gestion de signaux propre à une seule partie du programme. De même, lorsqu'un processus est lancé en arrière-plan par un shell ne gérant pas le contrôle des jobs, celui-ci force certains signaux à être ignorés. Les signaux concernés sont SIGINT et SIGQUIT. Dans le cas d'un shell sans contrôle de jobs, ces signaux seraient transmis autant au processus en arrière-plan qu'à celui en avant-plan.

Voici un exemple de programme permettant d'afficher les signaux dont le comportement n'est pas celui par défaut.

```
exemple-sigaction-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

int main (void)
{
    int i;
    struct sigaction action;
    for (i = 1; i < NSIG; i++) {
```

```

    if (sigaction(i, NULL, & action) != 0)
        fprintf(stderr, "%d : Erreur\n", i);
    if (action.sa_handler != SIG_DFL) {
        fprintf(stdout, "%d (%s) : ",
            i, sys_siglist[i]);
        if (action.sa_handler == SIG_IGN)
            fprintf(stdout, "ignore\n");
        else
            fprintf(stdout, "personnalise\n");
    }
}
return EXIT_SUCCESS;
}

```

Pour l'exécuter de manière probante, il faut arrêter le contrôle des jobs. Sous Bash, cela s'effectue à l'aide de la commande `set +m`. Au début de notre exemple, Bash a un contrôle des jobs activé.

```

$ ./exemple-sigaction-2
32 : Erreur
33 : Erreur
$ ./exemple-sigaction-2 &
[1] 8936
$ 32 : Erreur
33 : Erreur
  (Entrée)
[1]+ Done                  ./exemple-sigaction-2
$ set +m
$ ./exemple-sigaction-2
32 : Erreur
33 : Erreur
$ ./exemple-sigaction-2 &
[1] 8950
$ 2 (Interrupt) : ignore
3 (Quit) : ignore
32 : Erreur
33 : Erreur
  (Entrée)
$

```

On voit qu'il n'y a effectivement de différence que pour les processus en arrière-plan si le contrôle des jobs est désactivé. Il vaut donc mieux vérifier, au moment de l'installation d'un gestionnaire pour ces signaux, si le shell ne les a pas volontairement ignorés. Dans ce cas, on les laisse inchangés :

```
struct sigaction action, ancienne;
action.sa_handler = gestionnaire;
/* ... Initialisation de action ... */
if (sigaction(SIGQUIT, & action, & ancienne) != 0)
    /* ... Gestion d'erreur ... */
if (ancienne.sa_handler != SIG_DFL) {
    /* Réinstallation du comportement original */
    sigaction(SIGQUIT, & ancienne, NULL);
}
```

Ceci n'est important que pour SIGQUIT et SIGINT.

Profitons de cet exemple pour préciser le comportement des signaux face aux appels système `fork()` et `exec()` utilisés notamment par le shell. Lors d'un `fork()`, le processus fils reçoit le même masque de blocage des signaux que son père. Les actions des signaux sont également les mêmes, y compris les gestionnaires installés par le programme. Par contre, les signaux en attente n'appartiennent qu'au processus père.

Lors d'un `exec()`, le masque des signaux bloqués est conservé. Les signaux ignorés le restent. C'est comme cela que le shell nous transmet le comportement décrit ci-dessus pour SIGINT et SIGQUIT. Mais les signaux qui étaient capturés par un gestionnaire reprennent leur comportement par défaut. C'est logique car l'ancienne adresse du gestionnaire de signal n'a plus aucune signification dans le nouvel espace de code du programme exécuté.

On peut s'interroger sur la pertinence de mélanger dans un même programme les appels à `signal()` et à `sigaction()`. Cela ne pose aucun problème majeur sous Linux avec Glibc. Le seul inconvénient vient du fait que `signal()` ne peut pas sauvegarder et rétablir ultérieurement autant d'informations sur le gestionnaire de signal que `sigaction()`. Ce dernier en effet peut sauver et réinstaller les attributs comme `NOCLDSTOP` ou `NODEFER`, au contraire de `signal()`.

Lorsqu'il faut sauvegarder un comportement pour le restituer plus tard, il faut donc impérativement utiliser `sigaction()`, sauf si tout le programme n'utilise que `signal()`.

Lorsqu'on installe un gestionnaire avec `signal()` sous Linux et qu'on examine le comportement du signal avec `sigaction()`, on retrouve dans le champ `sa_handler` la même adresse que celle de la routine installée avec `signal()`. Ceci n'est toutefois pas du tout généralisable à d'autres systèmes, et il ne faut pas s'appuyer sur ce comportement. Voici un exemple de test.

```
exemple-sigaction-3.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
```



```
void gestionnaire (int inutilise)
{
}

int main (void)
{
    struct sigaction action;

    signal(SIGUSR1, gestionnaire);
    sigaction(SIGUSR1, NULL, & action);

    if (action.sa_handler == gestionnaire)
        fprintf(stdout, "Meme adresse\n");
    else
        fprintf(stdout, "Adresse differente\n");

    return EXIT_SUCCESS;
}
```

Sous Linux, pas de surprise !

```
$ uname -r
2.6.33.6-147.fc13.i686
$ ./exemple-sigaction-3
Meme adresse
$
```

Nous terminerons cette section avec un exemple d'installation d'une pile spécifique pour les gestionnaires de signaux. Nous allons mettre en place un gestionnaire commun pour les signaux SIGQUIT et SIGTERM, qui vérifiera si la pile est en cours d'utilisation ou non en examinant le champ `ss_flags`. Nous n'installerons la pile spéciale que pour le signal SIGQUIT, ce qui nous permettra de vérifier la différence entre les deux signaux.

```
exemple-sigaltstack.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

void gestionnaire (int numero)
{
    stack_t pile;
```

```
fprintf(stdout, "\nSignal %d reçu\n", numero);

if (sigaltstack(NULL, & pile) != 0) {
    fprintf(stderr, "Erreur sigaltstack %d\n", errno);
    return;
}
if (pile.ss_flags & SS_DISABLE)
    fprintf(stdout, "La pile speciale est inactive\n");
else
    fprintf(stdout, "La pile speciale est active\n");

if (pile.ss_flags & SS_ONSTACK)
    fprintf(stdout, "Pile speciale en cours d'utilisation\n");
else
    fprintf(stdout, "Pile speciale pas utilisee actuellement\n");
}

int main (void)
{
    stack_t pile;
    struct sigaction action;

    if ((pile.ss_sp = malloc(SIGSTKSZ)) == NULL) {
        fprintf(stderr, "Pas assez de memoire\n");
        exit(EXIT_FAILURE);
    }
    pile.ss_size = SIGSTKSZ;
    pile.ss_flags = 0;

    if (sigaltstack(& pile, NULL) != 0) {
        fprintf(stderr, "Erreur sigaltstack() %d\n", errno);
        exit(EXIT_FAILURE);
    }

    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART | SA_ONSTACK;
    if (sigaction(SIGQUIT, & action, NULL) != 0) {
        fprintf(stderr, "Erreur sigaction() %d \n", errno);
        exit(EXIT_FAILURE);
    }

    action.sa_handler = gestionnaire;
    sigemptyset(& (action.sa_mask));
    action.sa_flags = SA_RESTART;
    if (sigaction(SIGTERM, & action, NULL) != 0) {
        fprintf(stderr, "Erreur sigaction() %d \n", errno);
        exit(EXIT_FAILURE);
    }
}
```

```
fprintf(stdout, "PID = %ld\n", (long) getpid());
fflush(stdout);

while (1)
    pause();

return EXIT_SUCCESS;
}
```

Voici un exemple d'utilisation.

```
$ ./exemple-sigaltstack
PID = 9151
  (Contrôle-AltGr-\)
Signal 3 reçu
La pile speciale est active
Pile speciale en cours d'utilisation

Signal 15 reçu
La pile speciale est active
Pile speciale pas utilisee actuellement
^C
$
```

Comme prévu, la pile spéciale des signaux reste active en permanence, mais elle n'est utilisée que lorsque le gestionnaire est invoqué par SIGQUIT.

Blocage des signaux

Nous avons mentionné à plusieurs reprises qu'un processus pouvait bloquer à volonté un ensemble de signaux, sauf SIGKILL et SIGSTOP. Cette opération se fait principalement grâce à l'appel système `sigprocmask()`. Cette routine est très complète puisqu'elle permet aussi bien de bloquer ou de débloquent des signaux, que de fixer un nouveau masque complet ou de consulter l'ancien masque de blocage. Le prototype de `sigprocmask()` est le suivant :

```
int sigprocmask (int methode,
                 const sigset_t * ensemble,
                 sigset_t * ancien);
```

Quelle que soit la méthode choisie, si le troisième argument `ancien` est un pointeur non NULL, il sera rempli avec le masque actuel de blocage des signaux. Le premier

argument permet d'indiquer l'action attendue, par l'intermédiaire de l'une des constantes symboliques suivantes.

Nom	Signification
SIG_BLOCK	On ajoute la liste des signaux contenus dans l'ensemble transmis en second argument au masque de blocage des signaux. Il s'agit d'une addition au masque en cours.
SIG_UNBLOCK	On retire les signaux contenus dans l'ensemble en second argument au masque de blocage des signaux. S'il existe un ou plusieurs signaux débloqués en attente, au moins un de ces signaux est immédiatement délivré au processus, avant le retour de l'appel système <code>sigprocmask()</code> .
SIG_SETMASK	Le second argument est utilisé directement comme masque de blocage pour les signaux. Comme pour SIG_UNBLOCK, la modification du masque peut entraîner le déblocage d'un ou de plusieurs signaux en attente. L'un au moins de ces signaux est alors délivré immédiatement avant le retour de <code>sigprocmask()</code> .

La fonction `sigprocmask()` renvoie 0 si elle réussit, et -1 en cas d'erreur, c'est-à-dire avec `errno` valant `EINVAL` en cas de méthode inexistante ou `EFAULT` si l'un des pointeurs est mal initialisé. Les signaux `SIGKILL` et `SIGSTOP` sont silencieusement éliminés du masque transmis en second argument, sans déclencher d'erreur. Il est donc possible de transmettre un ensemble de signaux remplis avec `sigfillset()` pour tout bloquer ou tout débloquer, sans se soucier de `SIGKILL` et `SIGSTOP`.

L'appel système `sigprocmask()` doit remplacer totalement les anciens appels `sigblock()`, `sigmask()`, `siggetmask()`, `sigsetmask()` qui sont désormais obsolètes.

On notera également qu'il n'est malheureusement pas possible de bloquer les signaux en Python.

L'utilité principale d'un blocage des signaux est la protection des portions critiques de code. Imaginons qu'un gestionnaire modifie une variable globale comme une structure. Lorsque le programme principal est en train de lire ou de modifier cette variable, il risque d'être interrompu par ce signal au milieu de la lecture et d'avoir des incohérences entre les premiers et les derniers champs lus. L'effet peut être encore pire avec une chaîne de caractères. Pour éviter cette situation, on bloque temporairement l'arrivée du signal concerné pendant la lecture ou la modification de la variable globale.

Imaginons qu'on ait une variable globale du type :

```
typedef struct {  
    double X;  
    double Y;  
} point_t;  
  
point_t centre, pointeur;
```

Que le gestionnaire de SIGUSR1 modifie la variable centre :

```
void gestionnaire_sigusr1 (int inutilise)
{
    centre.X = pointeur.X;
    centre.Y = pointeur.Y;
}
```

On protégera dans le corps du programme l'accès à cette variable :

```
sigset_t ensemble, ancien;

sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);

sigprocmask(SIG_BLOCK, & ensemble, & ancien);
X1 = centre.X * zoom;          /* Voici la portion critique */
Y1 = centre.Y * zoom;          /* protégée de SIGUSR1      */
sigprocmask(SIG_SETMASK, & ancien, NULL);

cercle(centre.X, centre.Y, rayon);
```

Ceci peut paraître un peu lourd, mais l'ensemble en question n'a besoin d'être initialisé qu'une seule fois, et on peut aisément définir des macros pour encadrer les portions de code critiques.

Il est important qu'un processus puisse consulter la liste des signaux bloqués en attente, sans pour autant en demander la délivrance immédiate. Cela s'effectue à l'aide de l'appel système `sigpending()`, dont le prototype est :

```
int sigpending (sigset_t * ensemble);
```

Comme on pouvait s'y attendre, cette routine remplit l'ensemble transmis en argument avec les signaux en attente. Voici un programme qui permet d'en voir le fonctionnement. Tout d'abord, nous installons un gestionnaire qui indique le numéro du signal reçu, et ce pour tous les signaux. Ce gestionnaire ne relance pas les appels système lents interrompus. Ensuite, nous bloquons tous les signaux, sauf SIGINT (*Contrôle-C*), et nous lançons une lecture bloquée, pendant laquelle nous pouvons appuyer sur des touches spéciales au clavier (*Contrôle-Z*, *Contrôle-AltGr-*) ou envoyer des signaux depuis une autre console. Lorsque nous appuyons sur *Contrôle-C*, SIGINT non bloqué fait échouer la lecture, et le programme continue, nous indiquant la liste des signaux bloqués en attente. Nous débloquons alors tous les signaux et nous les regardons arriver.

```
exemple-sigpending.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>

#ifdef _POSIX_REALTIME_SIGNALS
#define NB_SIG_CLASSIQUES SIGRTMIN
#else
#define NB_SIG_CLASSIQUES SIGRTMIN
#endif

void gestionnaire (int numero)
{
    fprintf(stdout, "%d (%s) recu\n", numero, sys_siglist[numero]);
}

int main (void)
{
    int i;
    struct sigaction action;
    sigset_t ensemble;

    action.sa_handler = gestionnaire;
    sigfillset (& (action . sa_mask));
    action.sa_flags = 0; /* Pas de SA_RESTART */
    for (i = 1; i < NSIG; i++)
        if (sigaction(i, & action, NULL) != 0)
            fprintf(stderr, "%ld : %d pas capture\n",
                    (long) getpid(), i);

    /* On bloque tout sauf SIGINT */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGINT);
    sigprocmask(SIG_BLOCK, & ensemble, NULL);

    /* un appel systeme lent bloque */
    read(0, & i, sizeof(int));

    /* Voyons maintenant qui est en attente */
    sigpending(& ensemble);
    for (i = 1; i < NB_SIG_CLASSIQUES; i++)
        if (sigismember(& ensemble, i))
            fprintf(stdout, "en attente %d (%s)\n",
                    i, sys_siglist[i]);

    /* On débloque tous les signaux pour les voir arriver */
    sigemptyset(& ensemble);
```

```
sigprocmask(SIG_SETMASK, & ensemble, NULL);  
return EXIT_SUCCESS;  
}
```

Voici un exemple d'exécution avec, en seconde colonne, les commandes saisies sur une autre console.

```
$ ./exemple-sigpending  
9422 : 9 pas capture  
9422 : 19 pas capture  
9422 : 32 pas capture  
9422 : 33 pas capture  
  (Contrôle-Z)  
  (Contrôle-AltGr-\  
                                     $ kill 9422  
                                     $ kill -USR1 9422  
                                     $ kill -PIPE 9422  
                                     $ kill -PIPE 9422  
  
  (Contrôle-C)  
(Interrupt) reçu  
en attente 3 (Quit)  
en attente 10 (User defined signal 1)  
en attente 13 (Broken pipe)  
en attente 15 (Terminated)  
en attente 20 (Stopped)  
3 (Quit) reçu  
10 (User defined signal 1) reçu  
13 (Broken pipe) reçu  
15 (Terminated) reçu  
20 (Stopped) reçu  
$
```

On remarque deux choses : d'abord le signal PIPE a été émis deux fois mais n'est reçu qu'en un seul exemplaire. C'est normal, Linux n'empile pas les signaux classiques. Ensuite, les signaux sont délivrés dans l'ordre numérique et pas dans leur ordre chronologique d'arrivée. C'est en fait une conséquence de la première remarque. Notez bien que la délivrance par ordre numérique croissant est simplement un détail d'implémentation sous Linux, et n'est absolument pas normalisée par SUSv4.

Il n'y a pas non plus de notion de priorité entre les signaux classiques. Si on désire introduire un ordre précis, on peut débloquent signal par signal en utilisant plusieurs fois de suite `sigprocmask()`.

Attente d'un signal

Il y a de nombreuses occasions dans un programme où on désire attendre passivement l'arrivée d'un signal qui se produira de manière totalement asynchrone (par exemple pour se synchroniser avec un processus fils).

Pour cela, l'appel système le plus évident est `pause()`. Celui-ci endort le processus jusqu'à ce qu'il soit interrompu par n'importe quel signal. Il est déclaré dans `<unistd.h>`, ainsi :

```
int pause (void);
```

Cet appel système renvoie toujours -1 en remplissant `errno` avec la valeur `EINTR`.

Le problème qui se pose souvent est d'arriver à encadrer correctement `pause()`, de façon à éviter de perdre des signaux. Imaginons que `SIGUSR1` dispose d'un gestionnaire faisant passer à 0 une variable globale nommée `attente`. On désire bloquer l'exécution du programme jusqu'à ce que cette variable ait changé. La première version – naïve – de ce programme serait :

```
attente = 1;
while (attente != 0)
    pause();
```

Nous utilisons une boucle `while()` car il se peut que l'appel système `pause()` soit interrompu par un autre signal qui ne modifie pas la variable.

Le gros problème de ce type de comportement est que le signal peut arriver entre le test (`attente!=0`) et l'appel `pause()`. Si le signal modifie la variable à ce moment-là, et si le programme ne reçoit plus d'autres signaux, le processus restera bloqué indéfiniment dans `pause()`.

Un autre problème se pose, car on peut avoir d'autres tâches à accomplir dans la boucle (en rapport, par exemple, avec les autres signaux reçus), et le signal peut éventuellement arriver dans ces périodes gênantes.

Pour éviter cette situation, on pourrait vouloir bloquer le signal temporairement ainsi :

```
sigset_t ensemble;
sigset_t ancien;

sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);
sigprocmask(SIG_BLOCK, & ensemble, & ancien);
attente = 1;
```



```
while (attente != 0) {
    sigprocmask(SIG_UNBLOCK, & ensemble, NULL);
    pause();
    sigprocmask(SIG_BLOCK, & ensemble, NULL);
    /* Traitement pour les autres signaux */
}
sigprocmask(SIG_SETMASK, & ancien, NULL);
```

Malheureusement, nous avons indiqué qu'un signal bloqué en attente était délivré avant le retour de `sigprocmask()`, qui le débloque. Nous avons ainsi encore augmenté le risque d'un blocage infini dans `pause()`. On pourrait vérifier de nouveau (`attente!=0`) entre `sigprocmask()` et `pause()`, mais le signal pourrait encore s'infiltrer entre ces deux étapes et bloquer indéfiniment.

Il existe un appel système, `sigsuspend()`, qui permet de manière atomique de modifier le masque des signaux et de se mettre en attente. Une fois qu'un signal non bloqué arrive, `sigsuspend()` restitue le masque original avant de se terminer. Son prototype est :

```
int sigsuspend (const sigset_t * ensemble);
```

Attention : l'ensemble transmis est celui des signaux que l'on bloque, pas celui des signaux que l'on attend.

Voici comment l'utiliser, pour attendre l'arrivée de `SIGUSR1`.

```
sigset_t    ensemble;
sigset_t    ancien;
int         sigusr1_dans_masque = 0;

sigemptyset(& ensemble);
sigaddset(& ensemble, SIGUSR1);
sigprocmask(SIG_BLOCK, & ensemble, & ancien);
if (sigismember(& ancien, SIGUSR1)) {
    sigdelset(& ancien, SIGUSR1);
    sigusr1_dans_masque = 1;
}
/* Initialisation, etc. */
attente = 1;
while (attente != 0) {
    sigsuspend(& ancien);
    /* Traitement pour les éventuels autres signaux */
}
if (sigusr1_dans_masque)
    sigaddset(& ancien, SIGUSR1);
sigprocmask(SIG_SETMASK, & ancien, NULL);
```

On remarquera que nous prenons soin de restituer l'ancien masque de blocage des signaux en sortie de routine, et qu'en transmettant cet ancien masque à `sigsuspend()`, nous permettons l'arrivée d'autres signaux que `SIGUSR1`.

Signalons qu'il existe un appel système `sigpause()` obsolète, qui fonctionnait approximativement comme `sigsuspend()`, mais en utilisant un masque de signaux contenu obligatoirement dans un entier de type `int`.

Écriture correcte d'un gestionnaire de signaux

En théorie, suivant le *C Ansi*, la seule chose qu'on puisse faire dans un gestionnaire de signaux est de modifier une ou plusieurs variables globales de type `sig_atomic_t` (défini dans `<signal.h>`). Il s'agit d'un type entier – souvent un `int` d'ailleurs – que le processeur peut traiter de manière atomique, c'est-à-dire sans risque d'être interrompu par un signal. Il faut déclarer la variable globale avec l'indicateur « volatile » pour signaler au compilateur qu'elle peut être modifiée à tout moment, et pour qu'il ne se livre pas à des optimisations (par exemple en gardant la valeur dans un registre du processeur). Dans ce cas extrême, le gestionnaire ne fait que positionner l'état d'une variable globale, qui est ensuite consultée dans le corps du programme.

Nous avons vu qu'avec une gestion correcte des blocages des signaux, il est en fait possible d'accéder à n'importe quel type de données globales. Le même problème peut toutefois se présenter si un signal non bloqué arrive alors qu'on est déjà dans l'exécution du gestionnaire d'un autre signal. C'est à ce moment que le champ `sa_mask` de la structure `sigaction` prend tout son sens.

Une autre difficulté est de savoir si on peut invoquer, depuis un gestionnaire de signal, un appel-système ou une fonction de bibliothèque. Une partie des fonctions de bibliothèque ne sont pas réentrantes. Cela signifie qu'elles utilisent en interne des variables statiques ou des structures de données complexes, comme `malloc()`, et qu'une fonction interrompue en cours de travail dans le corps principal du programme ne doit pas être rappelée depuis un gestionnaire de signal. Prenons l'exemple de la fonction `ctime()`. Celle-ci prend en argument un pointeur sur une date du type `time_t`, et renvoie un pointeur sur une chaîne de caractères décrivant la date et l'heure. Cette chaîne est allouée de manière statique et est écrasée à chaque appel. Si elle est invoquée dans le corps du programme, interrompue et rappelée dans le gestionnaire de signal, au retour de ce dernier, la valeur renvoyée dans le corps du programme principal ne sera pas celle qui est attendue. Les fonctions de bibliothèque qui utilisent des variables statiques le mentionnent dans leurs pages de manuel. Il est donc nécessaire de les consulter avant d'introduire la fonction dans un gestionnaire.

Il est important également d'éviter résolument les fonctions qui font appel indirectement à `malloc()` ou à `free()`, comme `tempnam()`.

Il existe une liste minimale, définie par SUSv4, des appels système pouvant être invoqués depuis un gestionnaire de signal (*async-signal-safe*). Ces appels sont soit réentrants, soit non interruptibles par les signaux. Remarquons qu'un appel *async-signal-safe* est également utilisable sans restriction en programmation multithread, mais que la réciproque n'est pas toujours vraie (par exemple `malloc()` est correct pour les programmes multithreads mais ne doit pas être invoqué dans un gestionnaire de signal). Les appels *async-signal-safe* de SUSv4 sont :

- `_Exit, _exit,`
- `abort, accept, access, aio_error, aio_return, aio_suspend, alarm,`
- `bind,`
- `cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, chdir, chmod, chown,`
`clock_gettime, close, connect, creat,`
- `dup, dup2,`
- `execle, execve,`
- `fchmod, fchown, fcntl, fdasyn, fork, fpathconf, fstat, fsync, ftruncate,`
- `getegid, geteuid, getgid, getgroups, getpeername, getpgrp, getpid, getppid,`
`getsockname, getsockopt, getuid,`
- `kill,`
- `link, listen, lseek, lstat,`
- `mkdir, mkfifo,`
- `open,`
- `pathconf, pause, pipe, poll, posix_trace_event, pselect,`
- `raise, read, readlink, recv, recvfrom, recvmsg, rename, rmdir,`
- `select, sem_post, send, sendmsg, sendto, setgid, setpgid, setsid, setsockopt,`
`setuid, shutdown, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset,`
`sigismember, sleep, signal, sigpause, sigpending, sigprocmask, sigqueue,`
`sigset, sigsuspend, socket, socketpair, stat, symlink, sysconf,`
- `tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsetattr, tcsetpgrp,`
`time, timer_getoverrun, timer_gettime, timer_settime, times,`
- `umask, uname, unlink, utime,`
- `wait, waitpid, write.`

Les fonctions d'entrée-sortie sur des flux, `fprintf()` par exemple, ne doivent pas être utilisées sur le même flux entre le programme principal et un gestionnaire, à cause du risque important de mélange imprévisible des données. Par contre, il est tout à fait possible de réserver un flux de données pour le gestionnaire (`stderr` par exemple), ou de l'employer si on est sûr que le programme principal ne l'utilise pas au même moment.

Il est très important qu'un gestionnaire de signal employant le moindre appel système sauvegarde le contenu de la variable globale `errno` en entrée du gestionnaire et qu'il la restitue en sortie. Cette variable est en effet modifiée par la plupart des fonctions système, et le signal peut très bien s'être déclenché au moment où le programme principal terminait un appel système et se préparait à consulter `errno`.

Notons, pour terminer, que dans les programmes s'appuyant sur l'environnement graphique *X11*, il ne faut en aucun cas utiliser les routines graphiques (*Xlib*, *Xt*, *Motif*...), qui ne sont pas réentrantes. Il faut alors utiliser des variables globales comme indicateurs des actions à exécuter dans le corps même du programme.

Il peut arriver que le travail du gestionnaire soit d'effectuer simplement un peu de nettoyage avant de terminer le processus. L'arrêt peut se faire avec l'appel système `_exit()` ou `exit()`. Néanmoins, il est souvent préférable que le processus père sache que son fils a été tué par un signal et qu'il ne s'est pas terminé normalement. Pour cela, il faut reprogrammer le comportement original du signal et se l'envoyer à nouveau. Bien sûr, cela ne fonctionne qu'avec des signaux qui terminent par défaut le processus (comme `SIGTERM`). De plus, dans certains cas (comme `SIGSEGV`), un fichier d'image mémoire core sera créé.

```
exemple-fatal.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void gestionnaire_signal_fatal (int numero)
{
    /* Effectuer le nettoyage :      */
    /* Couper proprement les connexions réseau */
    /* Supprimer les fichiers de verrouillage */
    /* Tuer éventuellement les processus fils */

    fprintf(stdout, "\nJe fais le menage !\n");
    fflush(stdout);
    signal(numero, SIG_DFL);
    raise(numero);
}

int main (void)
{
    struct sigaction action;
    action.sa_handler = gestionnaire_signal_fatal;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);
    fprintf(stdout, "mon pid est %ld\n", (long) getpid());
```

```
if ((sigaction(SIGTERM, & action, NULL) != 0)
    || (sigaction(SIGSEGV, & action, NULL) != 0)) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
while (1)
    pause();
return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution. On envoie les signaux depuis une autre console. Le shell (Bash en l'occurrence) nous indique que les processus ont été tués par des signaux.

```
$ ./exemple-fatal
mon pid est 9881

Je fais le menage !
Complété
$ ./exemple-fatal
mon pid est 9897

Je fais le menage !
Erreur de segmentation (core dumped)
$
```

Les messages « *Complété* » ou « *Erreur de segmentation* » sont affichés par le shell lorsqu'il se rend compte que son processus se termine anormalement.

Il y a ici un point de sécurité à noter : certains programmes, souvent *Set-UID root*, disposent temporairement en mémoire de données que même l'utilisateur qui les a lancés ne doit pas connaître. Cela peut concerner par exemple le fichier « shadow » des mots de passe ou les informations d'authentification servant à établir la liaison avec un hôte distant. Dans ce genre d'application, il est important que le programme écrase ces données sensibles avant de laisser le gestionnaire par défaut créer une éventuelle image mémoire core qu'on pourrait examiner par la suite.

Utilisation d'un saut non local

Une troisième manière de terminer un gestionnaire de signaux est d'utiliser un saut non local `siglongjmp()`. Dans ce cas, l'exécution reprend dans un contexte différent, qui a été sauvegardé auparavant. On évite ainsi certains risques de bogues dus à l'arrivée intempestive de signaux, tels que nous en utiliserons pour `SIGALRM` à la fin de ce chapitre. De même, cette méthode permet de reprendre le contrôle d'un pro-

gramme qui a, par exemple, reçu un signal indiquant une instruction illégale. SUSv4 précise que le comportement d'un programme qui ignore les signaux d'erreur du type SIGFPE, SIGILL, SIGSEGV est indéfini. Nous avons vu que certaines de ces erreurs peuvent se produire à la suite de débordements de pile ou de mauvaises saisies de l'utilisateur dans le cas des routines mathématiques. Certaines applications désirent rester insensibles à ces erreurs et reprendre leur exécution comme si de rien n'était. C'est possible grâce à l'emploi de `sigsetjmp()` et `siglongjmp()`. Ces deux appels système sont des extensions des anciens `setjmp()` et `longjmp()`, qui posaient des problèmes avec les gestionnaires de signaux.

L'appel système `sigsetjmp()` a le prototype suivant, déclaré dans `<setjmp.h>` :

```
| int sigsetjmp (sigjmp_buf contexte, int sauver_signaux);
```

Lorsque `sigsetjmp()` est invoqué dans le programme, il mémorise dans le buffer transmis en premier argument le contexte d'exécution et renvoie 0. Si son second argument est non nul, il mémorise également le masque de blocage des signaux dans le premier argument.

Lorsque le programme rencontre l'appel système `siglongjmp()`, dont le prototype est :

```
| void siglongjmp (sigjmp_buf contexte, int valeur);
```

l'exécution reprend exactement à l'emplacement du `sigsetjmp()` correspondant au même buffer, et celui-ci renvoie alors la valeur indiquée en second argument de `siglongjmp()`. Cette valeur permet de différencier la provenance du saut, par exemple depuis plusieurs gestionnaires de signaux d'erreur.

L'inconvénient des sauts non locaux est qu'un usage trop fréquent diminue sensiblement la lisibilité des programmes. Il est conseillé de les réserver toujours au même type de circonstances dans une application donnée, pour gérer par exemple des temporisations, comme nous le verrons ultérieurement avec le signal SIGALRM.

Nous allons pour l'instant créer un programme qui permette à l'utilisateur de saisir deux valeurs numériques entières, et qui les divise l'une par l'autre. Si un signal SIGFPE se produit (on a demandé une division par zéro), l'exécution reprendra quand même dans un contexte propre.

```
| exemple-siglongjmp.c :  
| #include <stdio.h>  
| #include <stdlib.h>  
| #include <signal.h>  
| #include <setjmp.h>
```

```
sigjmp_buf contexte;

void gestionnaire_sigfpe (int numero)
{
    siglongjmp(contexte, 1);
    /* Si l'on est ici le saut a raté, il faut quitter */
    signal(numero, SIG_DFL);
    raise(numero);
}

int main (void)
{
    int p, q, r;

    struct sigaction action;

    action.sa_handler = gestionnaire_sigfpe;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);
    sigaction(SIGFPE, & action, NULL);

    while (1) {
        if (sigsetjmp(contexte, 1) != 0) {
            /* On est arrivé ici par siglongjmp() */
            fprintf(stdout, "Aie ! erreur mathématique ! \n");
            fflush(stdout);
        }
        while (1) {
            fprintf(stdout, "Entrez le dividende p : ");
            if (fscanf(stdin, "%d", & p) == 1)
                break;
        }
        while (1) {
            fprintf(stdout, "Entrez le diviseur q : ");
            if (fscanf(stdin, "%d", & q) == 1)
                break;
        }
        r = p / q;
        fprintf(stdout, "rapport p / q = %d\n", r);
    }
    return EXIT_SUCCESS;
}
```

Voici un petit exemple d'exécution :

```
$ ./exemple-siglongjmp
Entrez le dividende p : 8
Entrez le diviseur q : 2
```

```
rapport p / q = 4
Entrez le dividende p : 6
Entrez le diviseur q : 0
Aie ! erreur mathematique !
Entrez le dividende p : 6
Entrez le diviseur q : 3
rapport p / q = 2
Entrez le dividende p :
    (Contrôle-C)
$
```

Ce genre de technique est surtout utilisée dans les interpréteurs de langages pour permettre de revenir à une boucle principale en cas d'erreur.

Les anciens appels système `setjmp()` et `longjmp()` fonctionnaient de la même manière, mais ne sauvegardaient pas le masque des signaux bloqués (comme si le second argument de `siglongjmp()` valait 0). Le masque retrouvé dans le corps du programme n'est donc pas nécessairement celui qui est attendu ; en effet, au sein d'un gestionnaire, le noyau bloque le signal concerné, ce qui n'est sûrement pas ce qu'on désire dans la boucle principale du programme.

Un signal particulier : l'alarme

Le signal `SIGALRM` est souvent utilisé comme temporisation pour indiquer un délai maximal d'attente pour des appels système susceptibles de bloquer. On utilise l'appel système `alarm()` pour programmer une temporisation avant la routine concernée, et `SIGALRM` sera déclenché lorsque le délai sera écoulé, faisant échouer l'appel bloquant avec le code `EINTR` dans `errno`. Si la routine se termine normalement avant le délai maximal, on annule la temporisation avec `alarm(0)`.

Il y a de nombreuses manières de programmer des temporisations, mais peu sont tout à fait fiables. On considérera que l'appel système à surveiller est une lecture depuis une socket réseau.

Il est évident que `SIGALRM` doit être intercepté par un gestionnaire installé avec `sigaction()` sans l'option `SA_RESTART` dans `sa_flags` (sinon l'appel bloqué redémarrerait automatiquement). Ce gestionnaire peut être vide, son seul rôle est d'interrompre l'appel système lent.

```
void gestionnaire_sigalrm (int inutilise)
{ /* Ne fait rien */
}
```


L'installation en est faite ainsi :

```
struct sigaction    action;

sigemptyset(& (action . sa_mask));
action.sa_flags = 0;
action.sa_handler = gestionnaire_sigalrm;
sigaction(SIGALRM, action, NULL);
```

Nous allons commencer par cet exemple naïf :

```
alarm(delai_maximal);
taille_lue = read(fd_socket, buffer, taille_buffer);
alarm(0);
if ((taille_lue != taille_buffer) && (errno == EINTR))
    fprintf(stderr, "delai maximal ecoule \n");
    return(-1);
}
/* ... Suite ... */
```

SUSv4 autorisant l'appel système `read()` à renvoyer soit `-1`, soit le nombre d'octets lus lors d'une interruption par un signal, nous comparerons sa valeur de retour avec la taille attendue et non avec `-1`. Cela améliore la portabilité de notre programme.

Le premier problème qui se pose est qu'un signal autre que l'alarme peut avoir interrompu l'appel système `read()`. Cela peut se résoudre en imposant que tous les autres signaux gérés par le programme aient l'attribut `SA_RESTART` validé pour faire redémarrer l'appel bloquant. Toutefois, un problème subsiste, car le redémarrage n'a généralement lieu que si `read()` n'a pu lire aucun octet avant l'arrivée du signal. Sinon, l'appel se termine quand même en renvoyant le nombre d'octets lus.

Le second problème est que, sur un système très chargé, le délai peut s'écouler entièrement entre la programmation de la temporisation et l'appel système lui-même. Il pourrait alors rester bloqué indéfiniment.

Ce qu'on aimerait, c'est disposer d'un équivalent à `sigsuspend()`, qui permette d'effectuer atomiquement le déblocage d'un signal et d'un appel système. Malheureusement, cela n'existe pas.

Nous allons donc utiliser une autre méthode, plus complexe, utilisant les sauts non locaux depuis le gestionnaire. Quel que soit le moment où le signal se déclenche, nous reviendrons au même emplacement du programme et nous annulerons alors la lecture. Bien entendu, le gestionnaire de signal doit être modifié. Il n'a plus à être installé sans l'option `SA_RESTART` puisqu'il ne se terminera pas normalement.

Cet exemple va servir à temporiser la saisie d'une valeur numérique depuis le clavier. Nous lirons une ligne complète, puis nous essayerons d'y trouver un nombre entier. En cas d'échec, nous recommencerons. Malgré tout, un délai maximal de cinq secondes est programmé, après lequel le programme abandonne.

```
exemple-alarm.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <unistd.h>

sigjmp_buf contexte_sigalrm;

void gestionnaire_sigalrm (int inutilise)
{
    siglongjmp(contexte_sigalrm, 1);
}

int main (void)
{
    char ligne [80];
    int i;

    struct sigaction action;

    action.sa_handler = gestionnaire_sigalrm;
    action.sa_flags = 0;
    sigfillset(& action.sa_mask);
    sigaction(SIGALRM, & action, NULL);

    fprintf(stdout, "Entrez un nombre entier avant 5 secondes : ");

    if (sigsetjmp(contexte_sigalrm, 1) == 0) {
        /* premier passage, installation */
        alarm(5);

        /* Lecture et analyse de la ligne saisie */
        while (1) {
            if (fgets(ligne, 79, stdin) != NULL)
                if (sscanf(ligne, "%d", & i) == 1)
                    break;
            fprintf(stdout, "Un entier svp : ");
        }

        /* Ok - La ligne est bonne */
        alarm(0);
        fprintf(stdout, "Ok !\n");
    }
}
```

```
} else {  
    /* On est arrivé par SIGALRM */  
    fprintf(stdout, "\nTrop tard !\n");  
    exit(EXIT_FAILURE);  
}  
return EXIT_SUCCESS;  
}
```

Voici quelques exemples d'exécution :

```
$ ./exemple-alarm  
Entrez un nombre entier avant 5 secondes : 6  
Ok !  
$ ./exemple-alarm  
Entrez un nombre entier avant 5 secondes : a  
Un entier svp : z  
Un entier svp : e  
Un entier svp : 8  
Ok !  
$ ./exemple-alarm  
Entrez un nombre entier avant 5 secondes :  
Trop tard !  
$
```

Nous avons ici un exemple de gestion de délai fiable, fonctionnant avec n'importe quelle fonction de bibliothèque ou d'appel système risquant de rester bloqué indéfiniment. Le seul inconvénient de ce programme est le risque que le signal SIGALRM se déclenche alors que le processus est en train d'exécuter le gestionnaire d'un autre signal (par exemple SIGUSR1). Dans ce cas, on ne revient pas au gestionnaire interrompu et ce signal est perdu.

La seule possibilité pour l'éviter est d'ajouter systématiquement SIGALRM dans l'ensemble des signaux bloqués lors de l'exécution des autres gestionnaires, c'est-à-dire en l'insérant dans chaque champ `sa_mask` des signaux interceptés :

```
struct sigaction action;  
action.sa_handler = gestionnaire_sigusr1;  
action.sa_flags = SA_RESTART;  
sigemptyset(& (action.sa_mask));  
sigaddset(& (action.sa_mask), SIGALRM);  
sigaction(SIGUSR1, & action, NULL);
```

Le signal SIGALRM n'interrompra alors jamais l'exécution complète du gestionnaire SIGUSR1.

Conclusion

Nous avons étudié dans les deux derniers chapitres l'essentiel de la programmation habituelle concernant les signaux. Certaines confusions interviennent parfois à cause d'appels système obsolètes, qu'on risque néanmoins de rencontrer encore dans certaines applications.

Des précisions concernant le comportement des signaux sur d'autres systèmes sont disponibles dans [Stevens 1993] *Advanced Programming in the Unix Environment*. La programmation portable en utilisant les fonctions Posix (SUSv4 à présent) est décrite dans [Lewine 1994] *Posix Programmer's Guide*.

Le prochain chapitre sera consacré à un aspect plus moderne des signaux, introduits dans le noyau Linux depuis sa version 2.2 : les signaux temps réel.

13

Signaux temps réel

Avec Linux 2.2 est apparue la gestion des signaux temps réel. Ceux-ci constituent une extension des signaux SIGUSR1 et SIGUSR2, qui présentaient trop de limitations pour des applications temps réel. Il faut entendre, par le terme temps réel, une classe de programmes pour lesquels le temps mis pour effectuer une tâche constitue un facteur important du résultat. Une application temps réel n'a pas forcément besoin d'être très rapide ni de répondre dans des délais très brefs, mais simplement de respecter des limites temporelles connues.

Les fonctionnalités temps réel pour les systèmes Unix furent introduites par la norme Posix.1b et sont décrites de nos jours par l'option RTS de SUSv4 ; leur support par Linux élargit le champ des applications industrielles et scientifiques utilisables sur ce système d'exploitation.

Les signaux temps réel présentent donc les caractéristiques suivantes par rapport aux signaux classiques :

- nombre plus important de signaux utilisateur ;
- empilement des occurrences des signaux bloqués ;
- délivrance prioritaire des signaux ;
- informations supplémentaires fournies au gestionnaire.

Caractéristiques des signaux temps réel

Nombre de signaux temps réel

Nous avons déjà remarqué que le fait de ne disposer que de deux signaux réservés au programmeur était une contrainte importante pour le développement d'applications utilisant cette méthode de communication.

La norme Posix.1b réclame la présence d'au moins huit signaux temps réel. Linux en propose trente-deux, ce qui est largement suffisant pour la plupart des situations.

Attention, certaines implémentations d'Unix (Solaris par exemple) sont véritablement limitées à huit signaux temps réel, il faudra donc être économe pour les applications voulues portables.

Les signaux temps réel n'ont pas de noms spécifiques, contrairement aux signaux classiques. On peut employer directement leurs numéros, qui s'étendent de SIGRTMIN à SIGRTMAX compris. Bien entendu, on utilisera des positions relatives dans cet intervalle, par exemple (SIGRTMIN + 5) ou (SIGRTMAX - 2), sans jamais préjuger de la valeur effective de ces constantes.

Il est de surcroît conseillé, pour améliorer la qualité du code source, de définir des constantes symboliques pour nommer les signaux utilisés dans le code. Par exemple, on définira dans un fichier d'en-tête de l'application des constantes :

```
#define SIGRT0    (SIGRTMIN)
#define SIGRT1    (SIGRTMIN + 1)
#define SIGRT2    (SIGRTMIN + 2)
```

ou, encore mieux, des constantes dont les noms soient parlants :

```
#define SIG_AUTOMATE_PRET    (SIGRTMIN + 2)
#define SIG_ANTENNE_AU_NORD  (SIGRTMIN + 4)
#define SIG_LIAISON_ETABLIE  (SIGRTMIN + 1)
```

On vérifiera également que le nombre de signaux temps réel soit suffisant pour l'application. Toutefois, les valeurs SIGRTMIN et SIGRTMAX peuvent être implémentées sous forme de variables, et pas nécessairement de constantes symboliques. Cette vérification doit donc avoir lieu durant l'exécution du programme, pas pendant sa compilation. On emploiera ainsi un code du genre :

```
#include <signal.h>
#include <unistd.h>
```

```
#ifndef _POSIX_REALTIME_SIGNALS
    #error "Pas de signaux temps réel disponibles"
#endif

#define SIGRT0    (SIGRTMIN)
[...]
#define SIGRT10   (SIGRTMIN + 10)

#define NB_SIGRT_UTILES    11

int main (int argc, char ** argv [])
{
    if ((SIGRTMAX - SIGRTMIN + 1) < NB_SIGRT_UTILES) {
        fprintf(stderr, "Pas assez de signaux temps réel \n");
        exit(EXIT_FAILURE);
    }
    [...]
}
```

Empilement des signaux bloqués

Nous avons vu que les signaux classiques ne sont pas empilés. Cela signifie que si deux occurrences d'un même signal arrivent alors que celui-ci est temporairement bloqué, une seule d'entre elles sera finalement délivrée au processus lors du déblocage. Rappelons que le blocage n'intervient pas nécessairement de manière explicite, mais peut aussi se produire simplement durant l'exécution du gestionnaire d'un autre signal.

Lorsqu'on veut s'assurer qu'un signal arrivera effectivement à un processus, il faut mettre au point un système d'acquiescement, compliquant sérieusement le code.

Comme un signal est automatiquement bloqué durant l'exécution de son propre gestionnaire, une succession à court intervalle de trois occurrences consécutives du même signal risque de faire disparaître la troisième impulsion. Ce comportement n'est pas acceptable dès qu'un processus doit assurer des comptages ou des commutations d'état.

Pour pallier ce problème, la norme Posix.1b a introduit la notion d'empilement des signaux bloqués. Si un signal bloqué est reçu quatre fois au niveau d'un processus, nous sommes sûrs qu'il sera délivré quatre fois lors de son déblocage.

Il existe bien entendu une limite au nombre de signaux pouvant être mémorisés simultanément. Cette limite n'est pas précisée par SUSv4. Sous Linux, on peut empiler 1 024 signaux par processus, à moins que la mémoire disponible ne soit pas suffisante. L'appel système `sigqueue()`, que nous verrons plus bas et qui remplace `kill()` pour les signaux temps réel, permet d'avoir la garantie que le signal est bien empilé.

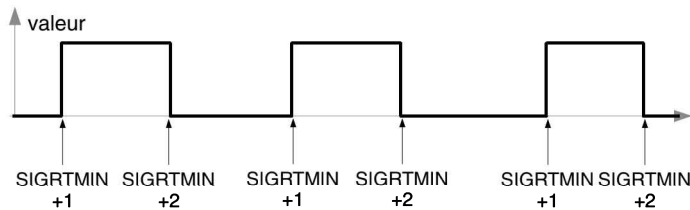
Délivrance prioritaire des signaux

Lorsque le noyau a le choix entre plusieurs signaux temps réel à transmettre au processus (par exemple lors d'un déblocage d'un ensemble complet), il délivre toujours les signaux de plus faible numéro en premier.

Les occurrences de SIGRTMIN seront donc toujours transmises en premier au processus, et celles de SIGRTMAX en dernier. Cela permet de gérer des priorités entre les événements représentés par les signaux. En revanche, SUSv4 ne donne aucune indication sur les priorités des signaux classiques. En général, ils sont délivrés avant les signaux temps réel car ils indiquent pour la plupart des dysfonctionnements à traiter en urgence (SIGSEGV, SIGILL, SIGHUP...), mais nous n'avons aucune garantie concernant ce comportement.

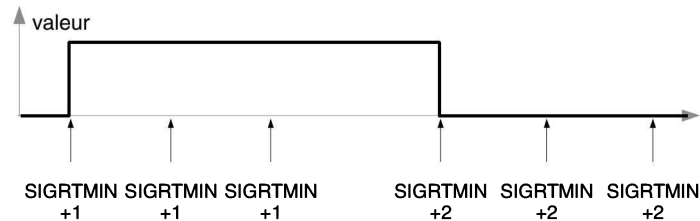
La notion de priorité entre signaux peut néanmoins présenter un inconvénient si on n'y prend pas garde. Le revers de la médaille, c'est que les signaux ne sont plus indépendants, comme l'étaient SIGUSR1 et SIGUSR2, par exemple. On pourrait vouloir utiliser deux signaux temps réel pour implémenter un mécanisme de bascule, un signal (disons SIGRTMIN+1) demandant le passage à l'état 1, et l'autre (SIGRTMIN+2) la descente au niveau 0. On aurait alors une séquence représentée sur la figure 13-1.

Figure 13-1
Séquence attendue



Malheureusement, si les signaux sont bloqués pendant un moment, ils ne seront pas délivrés dans l'ordre d'arrivée, mais en fonction de leur priorité. Toutes les impulsions SIGRTMIN+1 sont délivrées d'abord, puis toutes les impulsions SIGRTMIN+2.

Figure 13-2
Séquence obtenue



Si des événements liés doivent être transmis à l'aide des signaux temps réel, il faut se tourner vers une autre méthode, en utilisant un seul signal, mais en transmettant une information avec le signal lui-même.

Informations supplémentaires fournies au gestionnaire

Les signaux temps réel sont capables de transmettre une – petite – quantité d'information au gestionnaire associé. Ceci est valable en C mais pas en Python. Cette information est contenue dans une valeur de type union `sigval`. Cette union peut prendre deux formes :

- un entier `int`, en employant son membre `sigval_int` ;
- un pointeur `void *`, avec le membre `sigval_ptr`.

Nous avons déjà évoqué la forme du gestionnaire de signal temps réel dans le chapitre précédent, dans le paragraphe traitant de l'attribut `SA_SIGINFO` dans le champ `sa_flags` de la structure `sigaction`.

```
void gestionnaire_signal (int numero,
                        siginfo_t * info,
                        void * inutile);
```

Le troisième argument de cette routine n'est pas défini de manière portable. Certains systèmes Unix l'utilisent, mais apparemment le noyau Linux n'en fait aucun usage. Toutes les informations supplémentaires se trouvent dans la structure `siginfo_t` sur laquelle un pointeur est transmis en deuxième argument.

Pour que ce gestionnaire soit installé, il faut le placer dans le membre `sa_sigaction` de la structure `sigaction`, et non plus dans le membre `sa_handler`. De même, le champ `sa_flags` doit contenir l'attribut `SA_SIGINFO`.

L'initialisation se fait donc ainsi :

```
struct sigaction action;

action.sa_sigaction = gestionnaire_signal_temps_reel;
sigfillset(& action . sa_mask);
action.sa_flags = SA_SIGINFO;
if (sigaction(SIGRTMIN + 1, & action, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

Émission d'un signal temps réel

Bien sûr, si on désire transmettre des données supplémentaires au gestionnaire de signal, il ne suffit plus d'employer la fonction `kill()` habituelle. Il existe un nouvel appel système, nommé `sigqueue()`, défini par SUSv4 :

```
int sigqueue (pid_t pid, int numero, const union sigval valeur)
```

Les deux premiers arguments sont équivalents à ceux de `kill()`, mais le troisième correspond au membre `si_sigval` de la structure `siginfo_t` transmise au gestionnaire de signal.

Il n'y a aucun moyen dans le gestionnaire de déterminer si l'argument de type union `sigval` a été rempli, lors de l'invocation de `sigqueue()` avec une valeur entière (champ `sigval_int`) ou un pointeur (champ `sigval_ptr`). Il est donc nécessaire que l'application reste cohérente entre l'envoi du signal et sa réception. Lorsque le signal est transmis entre deux processus distincts, on ne peut bien sûr passer que des valeurs entières, un pointeur n'ayant pas de signification dans l'espace d'adressage de l'autre processus.

Il y a peu de situations où le passage d'un pointeur associé à un signal se justifie. On peut penser aux threads qui partagent leur espace mémoire, mais également aux cas où le processus s'envoie lui-même un signal différé. Nous verrons ultérieurement que l'on peut programmer une opération d'entrée-sortie asynchrone (par exemple une écriture en arrière-plan) et être averti lorsqu'elle est terminée par l'arrivée d'un signal afin de vérifier que tout s'est bien passé. Dans cette situation on peut associer au signal un pointeur sur une structure de données décrivant l'opération afin d'avoir toutes les informations nécessaires dans le gestionnaire invoqué.

Récapitulons les principaux champs de la structure `siginfo_t` reçue par le gestionnaire de signal.

Nom membre	Type	Posix.1b	Signification
<code>si_signo</code>	<code>int</code>	OK	Numéro du signal, redondant avec le premier argument de l'appel du gestionnaire.
<code>si_code</code>	<code>int</code>	OK	Voir ci-dessous.
<code>si_value.sigval_int</code>	<code>int</code>	OK	Entier de l'union passée en dernier argument de <code>sigqueue()</code> .
<code>si_value.sigval_ptr</code>	<code>void *</code>	OK	Pointeur de l'union passée en dernier argument de <code>sigqueue()</code> . Ne doit pas être employé simultanément avec le membre précédent.
<code>si_errno</code>	<code>int</code>		Valeur de la variable globale <code>errno</code> lors du déclenchement du gestionnaire. Permet de rétablir cette valeur en sortie.
<code>si_pid</code>	<code>pid_t</code>		PID du processus fils s'étant terminé si le signal est <code>SIGCHLD</code> . PID de l'émetteur si le signal est temps réel.
<code>si_uid</code>	<code>uid_t</code>		UID réel de l'émetteur d'un signal temps réel ou celui du processus fils terminé si le signal est <code>SIGCHLD</code> .
<code>si_status</code>	<code>int</code>		Code de retour du processus fils terminé, uniquement avec le signal <code>SIGCHLD</code> .

La signification du champ `si_code` varie suivant le type de signal. Pour les signaux temps réel ou pour la plupart des signaux classiques, `si_code` indique l'origine du signal.

Valeur	Provenance du signal	Posix.1b
SI_KERNEL	Signal émis par le noyau.	
SI_USER	appel système <code>kill()</code> ou <code>raise()</code> .	OK
SI_QUEUE	appel système <code>sigqueue()</code> .	OK
SI_ASYNCIO	Terminaison d'une entrée-sortie asynchrone.	OK
SI_MESGQ	Changement d'état d'une file de message temps réel (non implémenté sous Linux).	OK
SI_SIGIO	Changement d'état sur un descripteur d'entrée-sortie asynchrone.	
SI_TIMER	Expiration d'une temporisation temps réel (non implémentée sous Linux).	OK

Pour un certain nombre de signaux classiques, Linux fournit également des données (principalement utiles au débogage) dans le champ `si_code`, si le gestionnaire est installé en utilisant `SA_SIGINFO` dans l'argument `sa_flags` de `sigaction`.

Signal SIGBUS	
BUS_ADRALN	Erreur d'alignement d'adresse.
BUS_ADRERR	Adresse physique invalide.
BUS_OBJERR	Erreur d'adressage matériel.
Signal SIGCHLD	
CLD_CONTINUED	Un fils arrêté a redémarré (ce code n'est effectivement implémenté dans le noyau que depuis Linux 2.6.9).
CLD_DUMPED	Un fils s'est terminé anormalement.
CLD_EXITED	Un fils vient de se terminer normalement.
CLD_KILLED	Un fils a été tué par un signal.
CLD_STOPPED	Un fils a été arrêté.
CLD_TRAPPED	Un fils a atteint un point d'arrêt.
Signal SIGFPE	
FPE_FLTDIV	Division en virgule flottante par zéro.
FPE_FLTINV	Opération en virgule flottante invalide.
FPE_FLOVF	Débordement supérieur lors d'une opération en virgule flottante.
FPE_FLTRES	Résultat faux lors d'une opération en virgule flottante.
FPE_FLTSUB	Élévation à une puissance invalide.
FPE_FLTUND	Débordement inférieur lors d'une opération en virgule flottante.
FPE_INTDIV	Division entière par zéro.
FPE_INTOVF	Débordement de valeur entière.

Signal SIGILL	
ILL_BADSTK	Erreur de pile.
ILL_COPROC	Erreur d'un coprocesseur.
ILL_ILLADR	Mode d'adressage illégal.
ILL_ILLOPC	Code d'opération illégal.
ILL_ILLOPN	Opérande illégale.
ILL_ILLTRP	Point d'arrêt illégal.
ILL_PRVOPC	Code d'opération privilégié.
ILL_PRVREG	Accès à un registre privilégié.
Signal SIGPOLL	
POLL_ERR	Erreur d'entrée-sortie.
POLL_HUP	Déconnexion du correspondant.
POLL_IN	Données prêtes à être lues.
POLL_MSG	Message disponible en entrée.
POLL_OUT	Zone de sortie disponible.
POLL_PRI	Entrées disponibles à haute priorité.
Signal SIGSEGV	
SEGV_ACCERR	Accès interdit à la projection mémoire.
SEGV_MAPERR	Adresse sans projection mémoire.
Signal SIGTRAP	
TRAP_BRKPT	Point d'arrêt de débogage.
TRAP_TRACE	Point d'arrêt de profilage.

Attention : tous ces codes sont spécifiques à Linux et ne doivent pas être employés dans une application portable. En outre, ils sont tous déclarés dans les fichiers d'en-tête de Linux, mais ils ne sont pas tous réellement renvoyés par le noyau

À la lecture du premier tableau, concernant les champs `si_code` généraux, nous remarquons plusieurs choses :

- Il est possible d'envoyer un signal temps réel avec l'appel système `kill()`. Simple-ment, les informations supplémentaires ne seront pas disponibles. Leur valeur dans ce cas n'est pas précisée par SUSv4, mais sous Linux, le champ de type `sigval` correspondant est mis à zéro. Il est donc possible d'employer les signaux temps réel en remplacement pur et simple de `SIGUSR1` et `SIGUSR2` dans une application déjà existante, en profitant de l'empilement des signaux, mais en restant conscient du problème que nous avons évoqué, concernant la priorité de délivrance.

- Il existe un certain nombre de sources de signaux temps réel possibles, en supplément de la programmation manuelle avec `sigqueue()` ou `kill()`. Plusieurs fonctionnalités introduites par la norme Posix.1b permettent en effet à l'application de programmer un travail et de recevoir un signal lorsqu'il est accompli. C'est le cas, par exemple, des files de messages utilisant les fonctions `mq_open()`, `mq_close()`, `mq_notify()`, ou encore des temporisations programmées avec `timer_create()`, `timer_delete()` et `timer_settime()`.

Nous allons commencer par créer un programme servant de frontal à `sigqueue()`, comme l'utilitaire système `/bin/kill` pouvait nous servir à invoquer l'appel système `kill()` depuis la ligne de commande. Ce programme prendra en argument le numéro du signal,

```
exemple-sigqueue-1.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void usage(const char * nom)
{
    fprintf(stderr, "usage: %s [-signal] [valeur] pid\n", nom);
    exit(EXIT_FAILURE);
}

int main (int argc, char * argv [])
{
    int    numero;
    int    pid;
    union sigval valeur;

    switch(argc) {
        case 4:
            if (sscanf(argv[1], "%d", & numero) != 1)
                usage(argv[0]);
            if (sscanf(argv[2], "%d", & valeur.sival_int) != 1)
                usage(argv[0]);
            if (sscanf(argv[3], "%d", & pid) != 1)
                usage(argv[0]);
            if (numero >= 0)
                usage(argv[0]);
            break;
        case 3:
            if (sscanf(argv[1], "%d", & numero) != 1)
                usage(argv[0]);
            if (sscanf(argv[2], "%d", & pid) != 1)
                usage(argv[0]);
            if (numero >= 0) {
                valeur.sival_int = numero;
            }
    }
}
```

```

        numero = -SIGTERM;
    } else {
        valeur.sival_int = 0;
    }
    break;
case 2 :
    if (sscanf(argv[2], "%d", & pid) != 1)
        usage(argv[0]);
    numero = -SIGTERM;
    valeur.sival_int = 0;
    break;
default:
    usage(argv[0]);
}
if (sigqueue((pid_t) pid, -numero, valeur) < 0) {
    perror("sigqueue");
    return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}

```

À présent, nous allons créer un programme qui installe un gestionnaire de type temps réel pour tous les signaux – même les signaux classiques – pour afficher le champ `si_code` de leur argument de type `siginfo_t`.

```

exemple-siginfo.c :
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void gestionnaire (int numero, siginfo_t * info,
                  void * inutilise)
{
    fprintf(stderr, "Recu %d\n", numero);
    if (info == NULL)
        return;
    fprintf(stderr, " si_code   = %d\n", info->si_code);
    fprintf(stderr, " sival_int = %d\n", info->si_value.sival_int);
}

int main (void)
{
    int i;
    struct sigaction action;

    action.sa_sigaction = gestionnaire;
    action.sa_flags = SA_SIGINFO;
    sigfillset(& action.sa_mask);
}

```

```
fprintf(stderr, "PID=%ld\n", (long) getpid());
for (i = 1; i < NSIG; i++)
    if (sigaction(i, & action, NULL) < 0)
        fprintf(stderr, "%d non intercepté\n", i);
while (1)
    pause();
return EXIT_SUCCESS;
}
```

Finalement, nous lançons le programme `exemple-siginfo`, puis nous lui envoyons des signaux depuis une autre console (représentée en seconde colonne), en utilisant tantôt `/bin/kill`, tantôt `exemple-sigqueue-1`.

```
$ ./exemple-siginfo
PID=13840
9 non intercepté
19 non intercepté
32 non intercepté
33 non intercepté

Recu 35
  si_code   = 0
  sival_int = 2

Recu 35
  si_code   = -1
  sival_int = 10

Recu 15
  si_code   = 0
  sival_int = 10

Recu 15
  si_code   = -1
  sival_int = 20

Processus arrêté
$
```

`$ kill -35 13840`

`$./exemple-sigqueue-1 -35 10 13840`

`$ kill -TERM 13840`

`$./exemple-sigqueue-1 -15 20 13840`

`$ kill -KILL 13840`

Le champ `si_code` correspond bien à 0 (valeur de `SI_USER`) ou à -1 (valeur de `SI_QUEUE`) suivant le cas.

Attention

Si on utilise l'appel système `alarm()` pour déclencher `SIGALRM`, le champ `si_code` est rempli avec `SI_USER` et pas avec `SI_TIMER`, qui est réservé aux temporisations temps réel.

Nous voyons également qu'il est tout à fait possible d'accompagner un signal classique (SIGTERM ici) d'une valeur entière. Il faut simplement utiliser `sigqueue()` au lieu de `kill()`.

Notre second exemple va mettre en évidence à la fois l'empilement des signaux temps réel et leur respect d'une priorité. Notre programme va en effet bloquer tous les signaux, s'en envoyer une certaine quantité, et voir dans quel ordre ils arrivent. La valeur `sigval` associée aux signaux permettra de les reconnaître.

```
exemple-sigqueue-2.c :
#include <assert.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NB_SIGNAUX 10
int signal_arrive[NB_SIGNAUX];
int valeur_arrivee[NB_SIGNAUX];
int nb_signaux_arrives = 0;

void gestionnaire_signal_temps_reel (int numero, siginfo_t * info,
                                   void * inutile)
{
    assert(nb_signaux_arrives < NB_SIGNAUX);
    signal_arrive[nb_signaux_arrives] = numero - SIGRTMIN;
    valeur_arrivee[nb_signaux_arrives] = info->si_value.sival_int;
    nb_signaux_arrives ++;
}

void envoie_signal_temps_reel (int numero, int valeur)
{
    union sigval valeur_sig;

    fprintf(stdout, "Envoi signal SIRTMIN+%d, valeur %d\n",
            numero, valeur);
    valeur_sig.sival_int = valeur;
    if (sigqueue(getpid(), numero + SIGRTMIN, valeur_sig) < 0) {
        perror("sigqueue");
        exit(EXIT_FAILURE);
    }
}

int main (void)
{
    int i;
    sigset_t ensemble;
    struct sigaction action;
```



```

fprintf(stdout, "Installation gestionnaires de signaux\n");
action.sa_sigaction = gestionnaire_signal_temps_reel;
sigfillset(& action.sa_mask);
action.sa_flags = SA_SIGINFO;
if ((sigaction(SIGRTMIN + 1, & action, NULL) < 0)
    || (sigaction(SIGRTMIN + 2, & action, NULL) < 0)
    || (sigaction(SIGRTMIN + 3, & action, NULL) < 0)) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}

fprintf(stdout, "Blocage de tous les signaux\n");
sigfillset(& ensemble);
sigprocmask(SIG_BLOCK, & ensemble, NULL);

envoie_signal_temps_reel(1, 0);
envoie_signal_temps_reel(2, 1);
envoie_signal_temps_reel(3, 2);
envoie_signal_temps_reel(1, 3);
envoie_signal_temps_reel(2, 4);
envoie_signal_temps_reel(3, 5);
envoie_signal_temps_reel(3, 6);
envoie_signal_temps_reel(2, 7);
envoie_signal_temps_reel(1, 8);
envoie_signal_temps_reel(3, 9);

fprintf(stdout, "Deblocage de tous les signaux\n");
sigfillset(& ensemble);
sigprocmask(SIG_UNBLOCK, & ensemble, NULL);

fprintf(stdout, "Affichage des resultats\n");
for (i = 0; i < nb_signaux_arrives; i++)
    fprintf(stdout, "Signal SIGRTMIN+%d, valeur %d\n",
            signal_arrive[i], valeur_arrivee[i]);
return(EXIT_SUCCESS);
}

```

Notre gestionnaire stocke les signaux arrivant dans une table qui est affichée par la suite, pour éviter les problèmes de concurrence sur l'accès au flux stdout.

```

$ ./exemple-sigqueue-2
Installation gestionnaires de signaux
Blocage de tous les signaux
Envoi signal SIRTMIN+1, valeur 0
Envoi signal SIRTMIN+2, valeur 1
Envoi signal SIRTMIN+3, valeur 2
Envoi signal SIRTMIN+1, valeur 3
Envoi signal SIRTMIN+2, valeur 4

```

```
Envoi signal SIRTMIN+3, valeur 5
Envoi signal SIRTMIN+3, valeur 6
Envoi signal SIRTMIN+2, valeur 7
Envoi signal SIRTMIN+1, valeur 8
Envoi signal SIRTMIN+3, valeur 9
Déblocage de tous les signaux
Affichage des résultats
Signal SIGRTMIN+1, valeur 0
Signal SIGRTMIN+1, valeur 3
Signal SIGRTMIN+1, valeur 8
Signal SIGRTMIN+2, valeur 1
Signal SIGRTMIN+2, valeur 4
Signal SIGRTMIN+2, valeur 7
Signal SIGRTMIN+3, valeur 2
Signal SIGRTMIN+3, valeur 5
Signal SIGRTMIN+3, valeur 6
Signal SIGRTMIN+3, valeur 9
$
```

Nous vérifions ainsi que les signaux sont bien délivrés suivant leur priorité : tous les SIGRTMIN+1 en premier, suivis des SIGRTMIN+2, puis des SIGRTMIN+3. De même, au sein de chaque classe, les occurrences des signaux sont bien empilées et délivrées dans l'ordre chronologique d'émission.

Traitement rapide des signaux temps réel

La norme SUSv4 donne accès à des possibilités de traitement rapide des signaux. Ceci ne concerne que les applications qui attendent passivement l'arrivée d'un signal pour agir. Cette situation est assez courante lorsqu'on utilise les signaux comme une méthode pour implémenter un comportement multitâche au niveau applicatif.

Avec le traitement classique des signaux, nous utilisons quelque chose comme :

```
sigfillset(& tous_signaux);
sigprocmask(SIG_BLOCK, & tous_signaux, NULL);
sigemptyset(& aucun_signal);
while (! fin_programme)
    sigsuspend(& aucun_signal);
```

Lorsqu'un signal arrive, le processus doit alors être activé par l'ordonnanceur, ensuite l'exécution est suspendue, le gestionnaire de signal est invoqué, puis le contrôle revient au fil courant d'exécution, qui termine la fonction `sigsuspend()`. La boucle reprend alors.

Le problème est que l'appel du gestionnaire par le noyau nécessite un changement de contexte, tout comme le retour de ce gestionnaire. Ceci est plus coûteux qu'un simple appel de fonction usuel.

Deux appels système, définis dans la norme Posix.1b, ont donc fait leur apparition avec le noyau Linux 2.2. Il s'agit de `sigwaitinfo()` et de `sigtimedwait()`. Ce sont en quelque sorte des extensions de `sigsuspend()`. Ils permettent d'attendre l'arrivée d'un signal dans un ensemble précis. À la différence de `sigsuspend()`, lorsqu'un signal arrive, son gestionnaire n'est pas invoqué. À la place, l'appel système `sigwaitinfo()` ou `sigtimedwait()` se termine en renvoyant le numéro de signal reçu. Il n'est plus nécessaire d'effectuer des changements de contexte pour exécuter le gestionnaire, il suffit d'une gestion directement intégrée dans le fil du programme (la plupart du temps en utilisant une construction `switch-case`). Si le processus doit appeler le gestionnaire, il le fera simplement comme une fonction classique, avec toutes les possibilités habituelles d'optimisation par insertion du code en ligne.

Comme prévu, `sigtimedwait()` est une version temporisée de `sigwaitinfo()`, qui échoue avec une erreur `EAGAIN` si aucun signal n'est arrivé pendant le délai imparti :

```
int sigwaitinfo(const sigset_t * signaux_attendus, siginfo_t * info);
int sigtimedwait (const sigset_t * signaux_attendus, siginfo_t * info,
                  const struct timespec * delai);
```

De plus, ces fonctions offrent l'accès aux données supplémentaires disponibles avec les signaux temps réel.

Attention : `sigsuspend()` prenait en argument l'ensemble des signaux bloqués, `sigwaitinfo()` comme `sigtimedwait()` réclament l'ensemble des signaux attendus.

La structure `timespec` utilisée pour programmer le délai offre les membres suivants.

Type	Nom	Signification
long	tv_sec	Nombre de secondes.
long	tv_nsec	Nombre de nanosecondes.

La valeur du champ `tv_nsec` doit être comprise entre 0 et 999.999.999, sinon le comportement est indéfini.

Les appels système `sigwaitinfo()` ou `sigtimedwait()` peuvent échouer avec l'erreur `EINTR` si un signal non attendu est arrivé et a été traité par un gestionnaire. Si leur second argument est `NULL`, aucune information n'est stockée.

Il est important de bloquer avec `sigprocmask()` les signaux qu'on attend avec `sigwaitinfo()` ou `sigtimedwait()`, car cela assure qu'aucun signal impromptu n'arrivera juste avant ou après l'invocation de l'appel système.

Notre premier exemple va consister à installer un gestionnaire normal pour un seul signal, `SIGRTMIN+1`, pour voir le comportement du système avec les signaux non attendus. Ensuite, on bloque tous les signaux, puis on les attend tous, sauf `SIGRTMIN+1`.

```
exemple-sigwaitinfo.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void gestionnaire (int num, siginfo_t *info, void *inutile)
{
    fprintf(stderr, "gestionnaire : %d reçu \n", numero);
}

int main (void)
{
    sigset_t      ensemble;
    int           numero;
    struct sigaction action;

    fprintf(stderr, "PID=%ld\n", (long) getpid());

    /* Installation gestionnaire pour SIGRTMIN+1 */
    action.sa_sigaction = gestionnaire;
    action.sa_flags = SA_SIGINFO;
    sigemptyset(& action.sa_mask);
    sigaction(SIGRTMIN + 1, & action, NULL);

    /* Blocage de tous les signaux sauf SIGRTMIN+1 */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGRTMIN + 1);
    sigprocmask(SIG_BLOCK, & ensemble, NULL);

    /* Attente de tous les signaux sauf RTMIN+1 */
    sigfillset(& ensemble);
    sigdelset(& ensemble, SIGRTMIN + 1);

    while (1) {
        if ((numero = sigwaitinfo(& ensemble, NULL)) < 0)
            perror("sigwaitinfo");
        else
            fprintf(stderr, "sigwaitinfo : %d reçu\n", numero);
    }
    return EXIT_SUCCESS;
}
```

Nous ne traitons pas réellement les signaux reçus, nous contentant d'afficher leur numéro, mais nous pourrions très bien insérer une séquence switch-case au retour de `sigwaitinfo()`. Il faut bien comprendre que le signal dont le numéro est renvoyé par `sigwaitinfo()` est complètement éliminé de la liste des signaux en attente. La structure `siginfo_t` est également remplie lors de l'appel système si des informations sont disponibles.

Voici un exemple d'exécution avec, en seconde colonne, les actions saisies depuis une autre console.

```
$ ./exemple-sigwaitinfo
PID=14477
      (Contrôle-C)
sigwaitinfo : 2 reçu
                                     $ kill -TERM 14477
sigwaitinfo : 15 reçu
                                     $ kill -35 14477
gestionnaire : 35 reçu
sigwaitinfo: Interrupted system call
                                     $ kill -9 14477
Processus arrêté
$
```

Nous remarquons que lorsqu'un signal non attendu est reçu, il est traité normalement par son gestionnaire, et l'appel système `sigwaitinfo()` est interrompu et échoue avec l'erreur `EINTR`.

En Python, `signal.sigwaitinfo()` renvoie un objet `siginfo_t` dont les membres correspondent aux champs de la structure `siginfo_t` du C.

Notre dernier exemple va consister à invoquer `sigtimedwait()` sur tous les signaux :

```
exemple-sigtimedwait.c :
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main (void)
{
    sigset_t      ensemble;
    int           numero;
    struct timespec delai;

    fprintf(stderr, "PID=%ld\n", (long) getpid());
```

```

/* Blocage de tous les signaux */
sigfillset(& ensemble);
sigprocmask(SIG_BLOCK, & ensemble, NULL);

/* Attente de tous les signaux pendant 10 secondes */
delai.tv_sec = 10;
delai.tv_nsec = 0;
sigfillset(& ensemble);

while (1) {
    if ((numero = sigtimedwait(& ensemble, NULL, & delai)) < 0) {
        perror("sigtimedwait");
        break;
    }
    fprintf(stderr, "sigtimedwait : %d reçu \n", numero);
}
return EXIT_SUCCESS;
}

```

Nous allons invoquer le programme, lui envoyer quelques signaux, puis l'abandonner pendant dix secondes.

```

$ ./exemple-sigtimedwait
PID=14696
  (Contrôle-C)
sigtimedwait : 2 reçu
  (Contrôle-AltGr-\)
sigtimedwait : 3 reçu
  (attente dix secondes...)
sigtimedwait: Resource temporarily unavailable
$

```

Le message « *Resource temporarily unavailable* » correspond à une erreur EAGAIN dans `errno`.

Conclusion

Nous avons examiné dans ce chapitre une extension importante des méthodes de traitement des signaux. Ces fonctionnalités temps réel ajoutent une dimension considérable aux capacités de Linux à traiter des problèmes industriels ou scientifiques avec des délais critiques.

La norme Posix.1b, quand elle s'appelait encore Posix.4, a été étudiée en détail dans [Gallmeister 1995] *Posix.4 Programming for the real world*.

Gestion de la mémoire du processus

Nous allons nous intéresser dans ce chapitre à toutes les techniques permettant de gérer avec plus ou moins de précision l'espace mémoire d'un processus. Nous commencerons par les principes d'allocation de mémoire dynamique. Ces mécanismes sont relativement classiques, peu différents des autres systèmes d'exploitation en ce qui concerne le programmeur applicatif. Toutefois, la bibliothèque Glibc et certains utilitaires système offrent des possibilités intéressantes pour le débogage, en assurant un suivi de toutes les allocations ou en permettant d'insérer notre propre code de surveillance dans le corps même des routines de gestion de la mémoire.

Python offrant une gestion de la mémoire totalement automatique, les scripts écrits avec ce langage ne seront pas concernés par le sujet de ce chapitre. On notera qu'il existe une API pour écrire en C des extensions pour Python incluant bien sûr des fonctions d'allocation et de libération dynamiques, mais elles dépassent le cadre de ce livre.

Des fonctionnalités avancées de manipulation de la mémoire en C seront examinées dans le chapitre suivant. Indiquons rapidement que la gestion de la mémoire partagée, sujet connexe à celui de ce chapitre, sera étudiée ultérieurement avec les mécanismes de communication entre processus.

Allocation et libération de mémoire

Les variables utilisées dans un programme C peuvent être allouées de diverses manières.

- Les variables globales ou les variables déclarées statiques au sein des fonctions sont allouées une fois pour toutes lors du chargement du programme. Il existe même une différence entre les variables qui sont initialisées automatiquement au démarrage et celles qui n'ont pas de valeur initiale précise.
- Les variables locales et les arguments des fonctions voient leurs emplacements réservés dans la pile lors de l'invocation de la fonction.
- Les variables dynamiques sont allouées explicitement par l'intermédiaire des routines que nous allons étudier, à travers des pointeurs sur les zones réservées.

Le fait d'employer des variables dynamiques complique quelque peu la programmation, puisqu'il faut les manipuler au travers de pointeurs. De plus, elles doivent être allouées manuellement avant toute utilisation. Pourtant, il est nécessaire d'utiliser ces variables dans plusieurs cas.

- Lorsqu'on ne connaît pas la taille des variables lors de la compilation (par exemple une table contenant un nombre fluctuant d'éléments).
- Lorsqu'on a besoin d'allouer une zone mémoire de taille importante, principalement s'il s'agit d'une variable locale dans une fonction susceptible d'être invoquée de manière récursive, risquant un débordement de pile si on l'alloue de manière automatique.
- Lorsqu'on désire gérer la mémoire le plus finement possible, en réallouant les zones de mémoire au fur et à mesure des besoins, ou en utilisant des organisations des données telles que la liste chaînée, l'arbre binaire, la table de hachage...

Les méthodes d'allocation dynamique de mémoire n'ont rien de compliqué, mais il convient toutefois de prendre des précautions pour éviter les bogues de fuite mémoire, qui sont surtout sensibles avec des processus fonctionnant pendant de longues, voire de très longues durées.

Utilisation de malloc()

Pour allouer une nouvelle zone de mémoire, on utilise généralement la fonction `malloc()`, dont le prototype est déclaré dans `<stdlib.h>` ainsi :

```
| void * malloc (size_t taille);
```

L'argument transmis correspond à la taille, en octets, de la zone mémoire désirée. Le type `size_t` étant non signé, il n'y a pas de risque de transmettre une valeur négative. Si on demande une taille valant zéro, la version Gnu de `malloc()` renvoie un pointeur `NULL`. Sinon, le système nous accorde une zone de la taille voulue et renvoie un pointeur sur cette zone. Si la mémoire disponible ne permet pas de faire l'allocation, `malloc()` renvoie un pointeur `NULL`. Il est fortement recommandé de tester le retour de toutes les demandes d'allocation. Le code demandant d'allouer une nouvelle structure de type `ma_struct_t` serait donc :


```
ma_struct_t * ma_struct;

if ((ma_struct = malloc(sizeof(ma_struct_t))) == NULL) {
    fprintf(stderr, "Pas assez de mémoire pour la structure\n");
    exit(EXIT_FAILURE);
}
```

Pendant longtemps, il était d'usage de convertir explicitement le pointeur renvoyé par `malloc()` à l'aide d'un `cast` dans le type de pointeur désiré. Cette habitude disparaît, car elle ne présente pas d'utilité (`malloc()` renvoie un pointeur `void *` qui peut se convertir automatiquement en pointeur sur n'importe quel type de données), et peut masquer une petite erreur : si nous oublions d'inclure `<stdlib.h>`, `malloc()` est implicitement déclaré comme renvoyant une valeur de type `int`. Si nous ne faisons pas de `cast` explicite, le compilateur nous affichera un avertissement – ce qui nous permet de corriger l'oubli de `<stdlib.h>`.

Le problème qui se pose souvent au programmeur est de savoir quoi faire en cas d'échec d'allocation mémoire. En effet, il est possible que la mémoire du système se libère, lors de la terminaison d'un processus gourmand, et que l'allocation réussisse si on la tente de nouveau quelques instants plus tard, mais on peut estimer aussi que l'état de surcharge du système est tel qu'il est probablement inutilisable, et qu'il vaut mieux terminer l'application au plus vite pour redonner la main à l'utilisateur, qui devra éliminer les processus consommant trop de ressources. L'échec d'une allocation dynamique est généralement le signe d'une fuite mémoire dans l'un des processus en cours, et il est préférable dans tous les cas de le signaler à l'utilisateur.

Malheureusement, dans certains cas extrêmes, le fait qu'une allocation mémoire ait réussi ne signifie pas que le processus puisse effectivement utiliser la mémoire qu'il croit disponible. Pour comprendre ce problème, il est nécessaire de s'intéresser au mécanisme détaillé de la gestion de la mémoire virtuelle sous Linux.

Prenons l'exemple d'un ordinateur 32 bits. Ce type de processeur est encore très répandu dans le domaine des systèmes embarqués. De plus, les valeurs des pointeurs et les volumes mémoire mis en jeu sont plus faciles à appréhender que pour les modèles 64 bits, même si les concepts sont exactement semblables. Un processus dispose alors d'un espace d'adressage linéaire, s'étendant jusqu'à 3 Go environ.

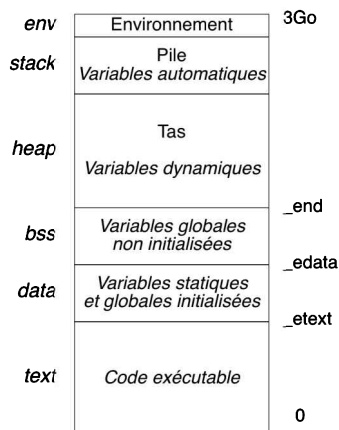
Sur les 2^{32} octets = 4 Go d'espace d'adressage théoriquement disponible (de `0x0000 0000` à `0xFFFF FFFF`), le noyau se réserve généralement le gigaoctet supérieur. Sur un processeur 64 bits actuel, l'espace utilisateur s'étend de `0x0000 0000 0000 0000` à `0x0000 7FFF FFFF FFFF` ce qui représente 128 To. Le noyau utilise la portion d'espace d'adressage de `0xFFFF 8000 0000 0000` à `0xFFFF FFFF FFFF FFFF`, de 128 To également.

Cet espace est découpé en segments ayant des rôles bien particuliers. Le processus peut connaître les limites de ses segments en utilisant des variables externes remplies par le chargeur de programmes du noyau.

- Le segment nommé *text* contient le code exécutable du processus ; il s'étend jusqu'à l'adresse contenue dans la variable `_etext`. Le début de ce segment varie selon le format de fichier exécutable. Dans le segment de code se trouvent également les routines des bibliothèques partagées utilisées par le processus.
- Le segment des données initialisées au chargement du processus et des données locales statiques des fonctions est nommé *data*. Il s'étend de l'adresse contenue dans la variable `_etext` jusqu'à celle contenue dans `_edata`.
- Le segment des données non initialisées et des données allouées dynamiquement est nommé *bss*. Il s'étend de l'adresse contenue dans `_edata` à celle contenue dans `_end`.

À l'autre bout de l'espace d'adressage se trouvent d'autres données comme les variables d'environnement et la pile du processus. Ces éléments ne nous concernent pas directement ici.

Figure 14-1
Espace d'adressage
d'un processus sur
processeur 32 bits



Lorsqu'on appelle la routine `malloc()`, de la bibliothèque C, celle-ci invoque l'appel système `brk()`, qui est déclaré par le prototype suivant :

```
| int brk (void * pointeur_end) ;
```

Cet appel système permet de positionner la variable `_end`, modifiant ainsi la taille du segment *bss*. Il existe également une fonction de bibliothèque nommée `sbrk()` et déclarée ainsi :

```
| void * sbrk (ptrdiff_t increment);
```

Celle-ci augmente ou diminue la taille du segment *bss* de l'incrément fourni en argument. En réalité, `malloc()` gère elle-même un ensemble de blocs mémoire qu'elle garde à disposition du processus, et ne fait appel à `sbrk()` qu'occasionnellement. Lorsque l'incrément est négatif, `sbrk()` sert à libérer de la mémoire. Le problème – si on peut dire – est que l'appel `sbrk()` n'échoue que très rarement. En effet, les cas d'erreur sont les suivants.

- Le processus essaye de libérer de la mémoire appartenant au segment de code *text*.
- Le processus essaye de dépasser sa limite `RLIMIT_DATA` fixée par la routine `setrlimit()` que nous avons vue dans le chapitre concernant l'exécution des programmes.
- La nouvelle zone de données va déborder sur une zone de projection mémoire telle que nous en verrons dans le prochain chapitre.
- L'allocation demandée excède la taille de la mémoire virtuelle globale du système (moins les valeurs minimales des buffers et du cache, ainsi qu'une marge de sécurité de 2 %).

Le problème principal se pose avec le dernier point. En effet, lorsque le noyau a augmenté la taille du segment de données d'un processus, il n'a pas pour autant réservé de la place effective dans la mémoire du système. Le principe de la mémoire virtuelle fait que c'est uniquement au moment où le processus tente d'écrire dans la zone nouvellement allouée que le système déclenche une faute d'accès et lui attribue une page mémoire réelle. Il est donc possible de réclamer beaucoup plus de mémoire que le système ne peut en fournir, sans pour autant que les allocations échouent.

Ce comportement est dû à un mécanisme nommé *overcommit-memory* (sur-réservation mémoire) selon lequel on considère que les applications réclament toujours plus de mémoire qu'elles n'en ont effectivement besoin. On accepte alors toutes les allocations si l'espace d'adressage (mémoire virtuelle) du processus n'est pas saturé, et si la taille de la demande n'excède pas la somme de la mémoire physique disponible et du swap.

Sans entrer dans les détails, indiquons que cela peut éviter des attaques de sécurité par déni de service. Lorsque toute la mémoire du système est saturée, un sous-ensemble du noyau nommé *OOM-Killer* (*Out Of Memory Killer*) est chargé de trouver le processus responsable de la saturation et de le tuer. Cela peut poser des problèmes de fiabilité dans des systèmes critiques, aussi est-il possible – depuis le noyau Linux 2.6 – de désactiver la sur-réservation mémoire. Pour cela, il faut écrire dans le fichier `/proc/sys/vm/overcommit-memory` la valeur 2. Pour plus de détails, on consultera le fichier `Documentation/vm/overcommit-accounting` dans les sources du noyau.

Si la machine dispose par exemple de 2 Go de mémoire virtuelle, une demande d'allocation de 2 Go en une fois échouera. En revanche, 2 048 demandes (ou plus) d'un mégaoctet chacune seront acceptées tant qu'on n'aura pas essayé d'écrire réellement dans les zones allouées.

À titre d'exemple, ma machine actuelle contient 2 Go de mémoire vive (et pas de swap). Même sans prendre en compte le fait que le système fait tourner X-Window, Gnome, un lecteur de CD audio, la suite OpenOffice sur laquelle je rédige ce texte et plusieurs Xterms, je ne peux vraiment pas compter disposer dans une application de plus de 2 Go de mémoire. Et pourtant, le code suivant fonctionne sans erreur :

```
exemple-malloc-1.c :
#include <stdio.h>
#include <stdlib.h>

#define NB_MEGA 3000

int main (void)
{
    int i;
    char * bloc[NB_MEGA];

    for (i = 0; i < NB_MEGA; i ++) {
        if ((bloc[i] = malloc(1024*1024)) == NULL) {
            fprintf(stderr, "Echec pour i = %d\n", i);
            break;
        }
    }
    fprintf(stderr, "Allocation : %d blocs de 1 Mo\n", i);

    return EXIT_SUCCESS;
}
```

En voici l'exécution :

```
$ ./exemple-malloc-1
Allocation : 3000 blocs de 1 Mo
$
```

3 Go alloués dans une mémoire virtuelle qui n'en comporte que 2 ! (et sans swap)

Si nous essayons d'utiliser les zones allouées, en revanche, le comportement est différent. Nous remplissons avec des zéros la mémoire renvoyée au fur et à mesure.

```
exemple-malloc-2.c :
[...] // Début identique à l'exemple précédent
    fprintf(stderr, "Allocation : %d blocs de 1 Mo\n", i);

    fprintf(stderr, "Utilisation des blocs...\n");

    for (i = 0; i < NB_MEGA; i ++) {
```

```
    memset(bloc[i], 1, 1024*1024);  
    fprintf(stderr, "%d ", i);  
}  
return EXIT_SUCCESS;  
}
```

Nous lançons le programme avec l'utilitaire `nice`, afin d'essayer de ne pas trop bloquer le reste du système (il faut quand même éviter de lancer ce processus sur une machine ouverte à plusieurs utilisateurs).

```
$ nice ./exemple-malloc-2  
Allocation : 3000 blocs de 1 Mo  
Utilisation des blocs...  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25  
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48  
49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71  
72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94  
95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113  
114 115 116 117 118  
[...]  
1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327  
1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341  
1342 1343 1344 1345 1346 1347 1348 1349 1350 Processus arrêté  
$
```

Cette fois-ci, nous voyons que le programme échoue effectivement lorsqu'il n'y a plus de mémoire utilisable. Toutefois, nous ne sommes pas sortis « proprement » de la boucle, `malloc()` n'a jamais renvoyé `NULL`. Il nous a toujours renvoyé un pointeur sur une partie de l'espace virtuel d'adressage. Ensuite, lors de la dernière tentative d'écriture, le *OOM-Killer* s'est déclenché au moment du `memset()` pour tuer le processus avec le signal `SIGKILL`.

Le problème, c'est donc le risque qu'une allocation semble réussir, alors qu'elle conduira par la suite à une mort du processus lorsqu'on tentera d'utiliser les zones allouées. Pour limiter au maximum cette éventualité, on essaiera toujours de réclamer uniquement la mémoire dont on a réellement besoin au moment voulu, et on utilisera systématiquement les pages mémoire allouées le plus vite possible.

La seule méthode vraiment efficace pour s'assurer la disponibilité des zones allouées est donc de toujours écrire rapidement dans les nouvelles données et d'éviter d'appeler successivement `malloc()` plusieurs fois de suite sans avoir rempli la mémoire fournie entre-temps.

On pourrait légitimement se demander si l'utilisation directe de `sbrk()` ne serait pas plus simple que celle de `malloc()`. En fait, la présentation que nous avons faite du

rôle de `malloc()` – fonction de bibliothèque – vis-à-vis de l’appel système `brk()` est largement simplificatrice. En fait, `malloc()` assure des fonctionnalités bien plus complexes que le simple agrandissement de la zone de données pour renvoyer un pointeur sur la mémoire allouée.

- **Alignement** : `malloc()` garantit que la mémoire fournie sera correctement positionnée afin de pouvoir y stocker n’importe quel type de donnée. Cela signifie que le processeur pourra manipuler directement les types entiers ou réels qu’on placera dans la mémoire allouée. Sur la plupart des machines, l’alignement des données est réalisé tous les 8 octets (taille d’un `double` ou d’un `long long int` sur les 32 bits). Sur les architectures 64 bits, l’alignement est fixé tous les 16 octets.
- **Configuration** : `malloc()` offre de nombreuses possibilités de configuration de l’algorithme d’allocation, notamment en ce qui concerne le seuil où l’on passe d’une allocation avec `sbrk()` à une projection avec `mmap()`. De plus, `malloc()` permet à l’utilisateur de fournir ses propres points d’appel qui seront invoqués dans la routine. Cela permet d’inclure notre code de débogage personnalisé au cœur même des fonctions de bibliothèque.
- **Vérification** : la fonction `malloc()` et toutes les routines associées appliquent éventuellement leurs propres vérifications aux blocs alloués. Cela permet de s’assurer que l’application ne contient pas de fuites de mémoire.
- **Optimisation** : pour éviter le supplément de travail dû à l’appel système `sbrk()`, la fonction `malloc()` réclame des blocs plus importants que nécessaire, afin de pouvoir en fournir directement une partie lors des invocations ultérieures.

De plus, `malloc()` utilise souvent l’appel système `mmap()` pour obtenir de gros blocs de données indépendants, faciles à restituer au système lors de leur libération. Le fonctionnement de `mmap()` n’a rien à voir avec `brk()`.

Enfin, ajoutons que `malloc()` doit fonctionner correctement dans le cadre d’un processus déployant de multiples threads, en évitant les conflits d’accès simultanés à la limite de la zone de données. Pour toutes ces raisons, on voit que l’implémentation de la fonction `malloc()` est loin d’être triviale, et que les personnalisations éventuelles devront de préférence être apportées en utilisant les points d’entrée fournis par la bibliothèque Glibc plutôt qu’en tentant de réécrire une version bricolée de cette fonction.

Insistons sur un dernier point, avant de passer aux autres routines d’allocation mémoire, qui concerne l’utilisation de `malloc()` avec les chaînes de caractères. La bibliothèque C terminant toujours ses chaînes de caractères par un caractère nul, il est nécessaire d’allouer un octet de plus pour la nouvelle chaîne que la longueur désirée. Voici un exemple de fonction renvoyant une copie fraîchement attribuée de la chaîne passée en argument. Il sera du ressort de la fonction appelante de libérer la mémoire occupée par la copie lorsqu’elle n’en aura plus besoin :

```
char * alloue_et_copie_chaine (char * ancienne)
{
    char * nouvelle = NULL;

    if (ancienne != NULL) {
        nouvelle = malloc(strlen(ancienne) + 1);
        if (nouvelle != NULL)
            strcpy(nouvelle, ancienne);
    }
    return nouvelle;
}
```

Utilisation de calloc()

Le prototype de `calloc()` est le suivant :

```
void * calloc (size_t nb_elements, size_t taille_element);
```

Cette fonction sert principalement à allouer des tableaux. On fournit en premier argument le nombre d'éléments à accorder, et en second la taille d'un élément. En voici un exemple extrêmement classique :

```
exemple-calloc-1.c :
#include <stdio.h>
#include <stdlib.h>

int * calcul_fibonacci (int nombre_de_valeurs)
{
    int * table = NULL;
    int i;

    if ((table = calloc(nombre_de_valeurs, sizeof(int))) == NULL) {
        fprintf(stderr, "Pas assez de memoire\n");
        exit(EXIT_FAILURE);
    }
    if (nombre_de_valeurs > 0) {
        table[0] = 1;
        if (nombre_de_valeurs > 1) {
            table[1] = 1;
            for (i = 2; i < nombre_de_valeurs; i++)
                table[i] = table[i - 2] + table[i - 1];
        }
    }
    return table;
}
```

```
int main (int argc, char * argv[])
{
    int nb_valeurs;
    int * table;
    int i;

    if ((argc != 2) || (sscanf(argv[1], "%d", & nb_valeurs) != 1)) {
        fprintf(stderr, "Syntaxe : %s nombre_de_valeurs\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    table = calcul_fibonacci(nb_valeurs);
    for (i = 0; i < nb_valeurs; i++)
        fprintf(stdout, "%d\n", table[i]);
    free(table);
    return EXIT_SUCCESS;
}
```

L'exécution affiche, on s'en doutait, le nombre désiré de termes de la suite de Fibonacci.

```
$ ./exemple-calloc-1 10
1
1
2
3
5
8
13
21
34
55
$
```

La fonction `calloc()` assure aussi, comme nous l'avons évoqué, que les zones allouées sont initialisées avec des zéros. Nous n'avons aucune garantie de ce genre avec les autres fonctions d'allocation. Elle est donc parfois préférée à `malloc()` pour s'affranchir des problèmes d'initialisation de variables, principalement lorsqu'on alloue dynamiquement des structures définies dans les fichiers d'en-tête d'autres modules, et qui sont susceptibles de posséder plus de membres que ceux qui sont utilisés par l'application. L'appel de `calloc()` permet ainsi d'initialiser toute la zone mémoire, y compris les membres dont nous n'avons pas nécessairement connaissance.

La fonction `calloc()` garantit que les zones allouées seront remplies avec des zéros, mais elle n'assure pas que cette initialisation se fera en utilisant des écritures effectives dans les zones mémoire reçues. Lorsque la taille de la zone concédée est suffisamment importante, `calloc()` utilise l'appel système `mmap()` depuis le périphérique `/dev/zero`.

Les pages allouées restent alors aussi virtuelles qu'avec `malloc()` jusqu'à ce qu'on écrive effectivement dedans. Pour plus de détails sur ce mécanisme, on se reportera au chapitre suivant. Voici un exemple qui reprend le principe de `exemple-malloc-1.c` :

```
exemple_calloc_2.c :
#include <stdio.h>
#include <stdlib.h>

#define NB_MEGA 3000

int main (void)
{
    int i;
    char * bloc[NB_MEGA];

    for (i = 0; i < NB_MEGA; i ++) {
        if ((bloc[i] = calloc(1, 1024*1024)) == NULL) {
            fprintf(stderr, "Echec pour i = %d\n", i);
            break;
        }
    }
    fprintf(stderr, "Allocation : %d blocs de 1 Mo\n", i);

    return EXIT_SUCCESS;
}
```

Le système nous alloue toujours aussi imperturbablement 3 Go sur un système ne comprenant que 2 Go :

```
$ ./exemple-calloc-2
Allocation : 3000 blocs de 1 Mo
$
```

La fonction `calloc()` n'avait donc pas réellement touché aux zones allouées.

Si, en revanche, nous utilisons des blocs de 1 Ko afin d'allouer beaucoup de petites zones, `calloc()` utilise `sbrk()` suivi de `memset()`, avec une écriture effective sur les pages allouées :

```
$ nice ./exemple-calloc-3
Processus arrêté
$
```

Le processus est tué par le OOM-Killer comme précédemment.

Nous avons beaucoup insisté sur ces détails d'implémentation concernant l'écriture ou non dans les pages allouées, mais il est important lors d'une phase de débogage de comprendre les phénomènes sous-jacents si les allocations mémoire ont un comportement étrange.

Utilisation de `realloc()`

Il est souvent nécessaire de modifier en cours de fonctionnement la taille d'une table allouée dynamiquement. Pour cela, la bibliothèque C propose la fonction `realloc()`, très polyvalente, permettant de redimensionner aisément une zone de mémoire dynamique. Son prototype est le suivant :

```
| void * realloc (void * ancien, size_t taille);
```

Cette fonction crée une nouvelle zone de la taille indiquée et y recopie le contenu de l'ancienne zone. Elle renvoie ensuite un pointeur sur la nouvelle zone mémoire. Si la taille réclamée est supérieure à celle de l'ancien bloc, celui-ci est étendu, son contenu original se retrouvant au début de la nouvelle zone. Si l'allocation échoue, `realloc()` renvoie `NULL`, mais ne touche pas à l'ancien bloc. Si, au contraire, l'allocation mémoire réussit, l'ancien pointeur n'est plus utilisable. Il faut donc employer une variable de stockage temporaire :

```
| void * nouveau;  
  
| nouveau = realloc(bloc_de_donnees, nouvelle_taille);  
| if (nouveau != NULL)  
|     bloc_de_donnees = nouveau;  
| else  
|     fprintf(stderr, "Pas assez de mémoire \n");
```

Si la taille est inférieure à celle de l'ancien bloc, il y a libération de mémoire et l'ancienne zone de données est tronquée. Normalement, une réduction ne doit pas échouer, mais cela peut quand même se produire sur certains systèmes où le nouveau bloc est alloué indépendamment de l'ancien avant d'y faire une copie. C'est surtout le cas quand des éléments de débogage importants sont ajoutés aux zones allouées.

Si la taille demandée est nulle, toute la mémoire est libérée et le pointeur renvoyé est `NULL`. Symétriquement, on peut transmettre un pointeur `NULL` en premier argument, et `realloc()` se conduit alors comme `malloc()`.

La fonction `realloc()` est particulièrement utile lorsque des données doivent être ajoutées ou supprimées au gré des actions de l'utilisateur. Imaginons un programme de dessin vectoriel où l'interface propose à l'utilisateur d'ajouter ou d'effacer des lignes. Une table stockée en variable globale et deux routines permettront de gérer proprement la mémoire :

```
static ligne_t * table_lignes = NULL;
static int      nb_lignes = 0;

int ajoute_ligne (void)
{
    // Cette routine renvoie le numéro de la nouvelle ligne
    // allouée ou -1 en cas d'échec.

    ligne_t * nouvelle;
    nouvelle = realloc(table_lignes,
                      (nb_lignes + 1) * sizeof(ligne_t));
    if (nouvelle == NULL)
        return -1;
    table_lignes = nouvelle;
    nb_lignes ++;
    return nb_lignes - 1;
}

void supprime_ligne (int numero)
{
    ligne_t * nouvelle;
    if ((numero < 0) || (numero >= nb_lignes))
        return;
    if (numero != nb_lignes - 1)
        // Si on supprime un élément autre que le dernier,
        // on va recopier celui-ci dans l'ancien emplacement.
        memcpy (& (table_lignes[numero]),
                & (table_lignes[nb_lignes - 1]),
                sizeof(ligne_t));
    nouvelle = realloc(table_lignes,
                      (nb_lignes - 1) * sizeof(ligne_t));
    if ((nouvelle != NULL) || (nb_lignes - 1 == 0))
        table_lignes = nouvelle;
    nb_lignes --;
}
```

Utilisation de free()

La plupart du temps, il faut libérer la mémoire qu'on a allouée dynamiquement. Cette libération s'effectue en invoquant la routine `free()` dont le prototype est :

```
void free (void * pointeur);
```

On transmet à `free()` un pointeur sur une zone mémoire qui a nécessairement été attribuée avec `malloc()`, `calloc()` ou `realloc()`¹. Si on passe un pointeur `NULL`, `free()` ne fait rien (mais ce n'est pas une erreur).

Une fois qu'une zone a été libérée, il ne faut sous aucun prétexte essayer d'y faire de nouveau référence. De même, il ne faut pas non plus tenter de libérer plusieurs fois de suite la même zone, même si la version Gnu peut assurer une certaine tolérance vis-à-vis de ce genre de bogue. Il faut donc se méfier de la libération naïve d'une liste chaînée ainsi :

```
for (ptr = debut; ptr != NULL; ptr = ptr->suivant)
    free(ptr);
```

C'est une erreur grave, car le troisième membre de `for` fait référence à la zone pointée par `ptr` alors même que celle-ci a déjà été libérée. Il est nécessaire en fait de passer par une variable intermédiaire :

```
for (ptr = debut; ptr != NULL; ptr = suite) {
    suite = ptr->suivant;
    free(ptr);
}
```

Règles de bonne conduite pour l'allocation et la libération de mémoire

Il y a certains cas où on peut légitimement se demander s'il faut vraiment libérer la mémoire allouée dynamiquement. Après tout, lorsqu'un processus se termine, tout son espace mémoire se libère et retourne au système d'exploitation. Si une variable est allouée à une seule reprise pour toute la durée du programme, il n'est pas indispensable de la libérer explicitement. On limitera quand même ce genre de comportement uniquement à des variables globales qui sont initialisées au démarrage du programme – par exemple en fonction de la valeur d'un argument en ligne de commande, ou suivant la valeur d'une variable d'environnement. L'idéal serait de restreindre l'allocation des blocs mémoire qu'on ne libère pas à la fonction `main()`. En effet, si le programme doit être ultérieurement modifié pour être utilisé en boucle, on verra tout de suite le problème lors de la mise à jour.

En règle générale, toute variable allouée dynamiquement devra être libérée à un moment ou à un autre. Il est essentiel, pour éviter les fuites de mémoire, d'adopter une attitude très précautionneuse lorsque l'allocation et la libération n'ont pas lieu dans la même fonction, ce qui est souvent le cas. De même, il faudra être très prudent avec les allocations dynamiques des membres de structures, elles aussi allouées dynamiquement.

Il est important de prendre de bonnes habitudes dans ces conditions. Nous présentons ici un exemple de « règles » de comportement vis-à-vis de la mémoire dyna-

1. Ou toute autre fonction de la bibliothèque C, par exemple `tempnam()`, qui appelle l'une de ces routines. Cette particularité est normalement bien indiquée dans leur documentation.

mique, mais chacun est libre d'adopter ses propres standards, du moment qu'on reste cohérent tout au long de l'application.

- À chaque déclaration d'un pointeur, on l'initialise avec `NULL`. Cela concerne également les membres des structures allouées dynamiquement s'il s'agit de pointeurs.
- Avant d'invoquer `malloc()`, on vérifie, éventuellement dans une condition `assert()`, que le pointeur à allouer est bien `NULL`.
- Après tout appel de `malloc()`, on s'assure qu'aucune erreur n'a eu lieu, sinon on gère le problème.
- Avant de libérer un pointeur, on vérifie – également dans un `assert()` – que le pointeur n'est pas `NULL`.
- Dès qu'on a libéré un pointeur avec `free()`, on le recharge immédiatement avec la valeur `NULL`.

Bien entendu, `malloc()` doit être considérée ici comme une fonction générique, et on adoptera la même attitude avec `realloc()` ou `calloc()`. Voici par exemple le genre de code qu'on peut produire :

```
typedef struct element {
    char *      nom;
    struct element * suivant;
} element_t;

element_t * table = NULL;

void insere_element (char * nom)
{
    element_t * nouveau = NULL;
    // Si on insère du code ici, entre l'initialisation
    // du pointeur et l'allocation de la variable, il est
    // bon d'effectuer la vérification suivante.
    assert(nouveau == NULL);
    nouveau = malloc(sizeof(element_t));
    if (nouveau == NULL) {
        // Traitement d'erreur
        return;
    }
    nouveau->nom = NULL;
    nouveau->suivant = NULL;

    // ... Ici on peut allouer plusieurs membres...
    if (nom != NULL) {
        nouveau->nom = malloc(strlen(nom) + 1);
        if (nouveau->nom != NULL)
            strcpy(nouveau->nom, nom);
        else {
```

```

        /* Traitement d'erreur */
        free(nouveau);
        return;
    }
}
nouveau->suivant = table;
table = nouveau;
}

void supprime_element (char * nom)
{
    element_t * elem = NULL;
    element_t * prec = NULL;

    if (nom == NULL)
        return;
    if (table == NULL)
        return;
    for (elem = table; elem != NULL; elem = elem->suivant) {
        if (strcmp(elem->nom, nom) == 0)
            break;
        prec = elem;
    }
    if (elem == NULL)
        // Pas trouvé
        return;
    if (prec == NULL)
        // Pas de précédent : premier de la table
        table = elem->suivant;
    else
        prec->suivant = elem->suivant;

    assert(elem->nom != NULL);
    free(elem->nom);
    elem->nom = NULL;
    // ... Éventuellement libération d'autres membres...
    free(elem);
    elem = NULL;
}

```

Bien sûr, c'est de la programmation paranoïaque et maniaque, mais c'est souvent ce genre de routines qui se révèlent les plus robustes à l'usage, même si on perd largement en élégance de codage. En ce qui concerne les performances du logiciel, on remarquera que tous les tests peuvent être inclus uniquement dans la version de débogage, comme c'est le cas ici avec l'utilisation de `assert()`. De plus, on peut encadrer les initialisations par des directives conditionnelles `#ifndef NDEB` et `#endif`. La version de production du logiciel n'est donc pas pénalisée par des vérifications compulsives et redondantes.

Désallocation automatique avec `alloca()`

Il existe une alternative à l'utilisation du couple `malloc()`–`free()`, constituée par la fonction `alloca()`. Celle-ci présente le même prototype que `malloc()` :

```
| void * alloca (size_t taille);
```

Le fonctionnement est identique à celui de `malloc()`, mais les zones de mémoire ne sont plus allouées dans le segment de données, mais à l'opposé dans la pile du processus. Rappelons que la mémoire du processus est constituée en bas du segment de code (*text*) et des segments de données initialisées (*data*) et non initialisées (*bss*). L'allocation avec `malloc()` fait croître ce dernier segment vers les adresses les plus élevées. À l'autre extrémité de l'espace d'adressage se trouvent les variables d'environnement et les arguments de la ligne de commande, tout en haut de l'espace virtuel réservé au processus. En dessous de cette zone se situe le segment de pile, qui croît vers le bas, vers les adresses les plus petites.

Les données allouées avec `alloca()` sont placées dans le segment de pile du processus. L'avantage principal est que les zones allouées sont automatiquement libérées lors de la sortie de la fonction ayant invoqué `alloca()`. Il n'est plus nécessaire d'appeler `free()`, le retour de la fonction remplace le pointeur de pile au-dessus des variables dynamiques, qui ne sont plus accessibles. On comprend bien d'ailleurs qu'il ne faut pas invoquer `free()` sur le pointeur renvoyé par `alloca()`, les domaines de travail de ces deux fonctions étant totalement disjoints.

Il n'est pas question d'allouer dynamiquement avec `alloca()` des variables globales ou une zone de mémoire sur laquelle la fonction doit renvoyer un pointeur. Seules les variables utilisées dans la fonction ou dans des sous-routines qu'on invoque peuvent être allouées correctement avec `alloca()`. Il ne faut pas non plus appeler directement `alloca()` dans les arguments d'une fonction qu'on invoque, car la zone allouée se trouverait, au sein de la pile, mélangée avec les arguments. Ceci est interdit :

```
| appel_fonction (i, alloca(struct element));
```

En revanche, on peut utiliser :

```
| struct element * elem;  
| elem = alloca(struct element);  
| appel_fonction(i, elem);
```

Le problème principal que pose `alloca()` est la gestion d'erreur. En effet, le système allouant automatiquement les pages nécessaires pour la pile, la seule erreur susceptible de se produire est le manque soudain de mémoire disponible. Le programme se

retrouve dans la même situation que s'il avait invoqué en boucle infinie une routine récursive. Le processus risque alors de dépasser sa limite de taille maximale de pile `RLIMIT_STACK` renvoyée par `getrlimit()`. Il y a peu de chances que la gestion d'erreur classique (retour non `NULL`) fonctionne. Au contraire, le programme va recevoir un signal `SIGSEGV` qui le tuera. Voici un exemple de ce carnage :

```
exemple-alloca.c :
#include <stdio.h>
#include <stdlib.h>

void fonction_recursive (int iteration)
{
    char * bloc;

    fprintf(stdout, "Iteration %d\n", iteration);
    fflush(stdout);
    if ((bloc = alloca(512 * 1024)) == NULL) {
        fprintf(stdout, "Echec\n");
        return;
    }
    fonction_recursive(iteration + 1);
}

int main (void)
{
    fonction_recursive(1);
    return EXIT_SUCCESS;
}
```

Avant d'appeler la fonction, nous invoquons la commande « `ulimit -s` » du shell, qui nous donne la limite de taille de pile, en kilo-octets.

```
$ ulimit -s
10240
$ ./exemple-alloca
Iteration 1
Iteration 2
Iteration 3
Iteration 4
[...]
Iteration 22
Iteration 23
Iteration 24
Erreur de segmentation (core dumped)
$
```


La limite étant de 10 240 Ko, soit 10 Mo, il est logique que notre programme ne puisse allouer correctement son vingt-quatrième bloc de 512 Ko. Toutefois, on aurait préféré que `alloca()` nous renvoie simplement une valeur d'échec, plutôt que de voir le processus arrêté par un signal.

Un autre gros avantage de `alloca()` est de permettre la libération automatique même lorsqu'il y a un saut non local depuis une sous-routine. Imaginons un interpréteur de commandes. La routine principale est celle où on revient en cas de problème de syntaxe. Lorsqu'on décompose les commandes saisies pour les analyser, on fait appel à des sous-routines d'analyse lexicale. Si l'une de ces sous-routines découvre une erreur (mauvaise utilisation d'un mot-clé réservé), elle peut déclencher un saut non local `siglongjmp()` pour revenir directement au plus haut niveau de l'interpréteur. Un problème se poserait alors pour les routines intermédiaires si elles ont alloué des données avec `malloc()`. Elles ne sont pas rappelées, car l'analyseur lexical ne revient pas, et les données allouées ne sont pas libérées. Il est alors pratique d'utiliser des allocations avec libération automatique `alloca()`. Voici un exemple très simplifié :

```
void interpreteur (void)
{
    int erreur;
    while (1) {
        erreur = sigsetjmp(environnement_saut, 1);
        if (erreur != 0) {
            // Afficher un message d'erreur
        }
        // Saisie d'une commande
        ...
        // Appel de l'analyseur syntaxique
        ...
        // Exécution des commandes
    }
}

void analyse_syntaxique (char * chaine);
{
    commande_t * table = NULL;
    commande_t * nouvelle = NULL;
    int cmd;

    // Construit une liste des commandes rencontrées
    while (1) {
        // Si fin de chaîne : retour
        ...
        // Appel de l'analyseur lexical
        ...
        nouvelle = alloca(sizeof(commande_t));
```

```
        nouvelle->suivante = table;
        table = nouvelle;
    }
}

void analyse_lexicale (char * chaine)
{
    // Extraction des mots
    ...
    // Si erreur d'entrée sortie, retour à boucle principale
    if (erreur) {
        siglongjmp(environnement_saut, 1);
    }
    // Reste du traitement
    ...
}
```

Le fait d'utiliser `alloca()` au lieu de `malloc()` permet dans ce cas une libération de la liste des commandes, car le saut non local restitue le pointeur de pile à la même position que durant l'invocation originale de `sigsetjmp()`. Même si on ne repasse pas par l'analyseur syntaxique, ses variables dynamiques sont libérées. Rappelons quand même que l'utilisation des sauts non locaux rend les programmes difficiles à lire et à déboguer, et qu'il vaut mieux les éviter au maximum.

Un dernier désagrément de `alloca()` est un léger manque de portabilité. Cette fonction est présente sur de nombreux systèmes Unix, mais elle n'est pas mentionnée dans les standards habituels.

Débogage des allocations mémoire

Dans un monde idéal, l'utilisation prudente de `malloc()` et de `free()` avec une vérification à chaque appel de l'état des pointeurs devrait suffire à éviter tout bogue de fuite de mémoire. Malheureusement, il en est rarement ainsi, et il existe toujours un risque d'erreur dans un programme où les variables sont allouées dynamiquement dans un module pour être libérées dans un autre module. C'est le cas, par exemple, pour toutes les routines utilitaires qui renvoient un pointeur sur un bloc mémoire fraîchement alloué, contenant les données désirées. Nous créerons des fonctions de ce genre dans le chapitre sur les entrées-sorties, en guise de frontaux pour `sprintf()` et `fgets()`. À chaque utilisation de ces routines, il faut penser à libérer la mémoire renvoyée. Pour entretenir la confusion, il y a d'autres routines qui renvoient un pointeur sur des données statiques, à ne surtout pas libérer.

Même si le programme semble se comporter parfaitement, on aimerait quand même avoir la certitude que la mémoire est correctement gérée. L'observation « externe » du

processus est malheureusement insuffisante, comme nous allons le vérifier. Nous allons créer un petit programme qui prend en argument deux valeurs et crée un tableau ayant le nombre d'éléments mentionnés en premier argument, chaque élément ayant la taille fournie en second argument. Ce programme invoque la commande `ps` pour afficher son propre état avant et après allocation. Ensuite, il libère tous les éléments, sauf le dernier alloué, et invoque `ps`. Puis il libère le dernier élément et affiche une dernière fois le résultat de `ps`. Nous analyserons ensuite son comportement suivant les diverses valeurs passées en argument. Bien sûr, nous remplirons les blocs alloués pour nous assurer qu'ils sont bien attribués physiquement au processus.

```
exemple-memoire.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char * argv [])
{
    int i;
    int nb_blocs;
    int taille_bloc;
    char ligne_ps[80];
    char ** table = NULL;

    if ((argc != 3)
        || (sscanf(argv[1], "%d", & nb_blocs) != 1)
        || (sscanf(argv[2], "%d", & taille_bloc) != 1)) {
        fprintf(stderr, "Syntaxe : %s Nb_blocs Taille_bloc\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }

    if ((nb_blocs < 1) || (taille_bloc < 1)) {
        fprintf(stderr, "Valeurs invalides\n");
        exit(EXIT_FAILURE);
    }

    sprintf(ligne_ps, "ps un %ld", (long) getpid());

    printf("Je demarre...\n");
    system(ligne_ps);

    printf("J'alloue %d blocs de %d octets...",
          nb_blocs, taille_bloc);
    fflush(stdout);
```

```

    table = calloc(nb_blocs, sizeof (char *));
    if (table == NULL) {
        fprintf(stderr, "Echec\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < nb_blocs; i++) {
        table[i] = malloc(taille_bloc);
        if (table[i] == NULL) {
            fprintf(stdout, "Echec\n");
            exit(EXIT_FAILURE);
        }
        memset(table[i], 1, taille_bloc);
    }
    printf("Ok\n");
    system(ligne_ps);

    printf("Je libere tous les blocs sauf le dernier\n");
    for (i = 0; i < nb_blocs - 1; i++)
        free(table[i]);
    system(ligne_ps);

    printf("Je libere le dernier bloc..\n");
    free(table[nb_blocs - 1]);
    system(ligne_ps);

    return EXIT_SUCCESS;
}

```

Nous allons faire deux expériences : tout d'abord, nous essaierons deux allocations avec un petit nombre de gros blocs, puis nous réclamerons de nombreux petits blocs. Les champs qui nous intéressent dans la commande ps sont VSZ et RSS, qui représentent respectivement la taille totale de mémoire virtuelle utilisée par le processus et la place occupée en mémoire physique.

```

$ ./exemple-memoire
Syntaxe : ./exemple-memoire Nb_blocs Taille_bloc
$ ./exemple-memoire 100 1000000
Je démarre...
USER  PID [...] VSZ   RSS [...] COMMAND
  500 28626 [...] 1820   412 [...] ./exemple-memoire 100 1000000
J'alloue 100 blocs de 1000000 octets...Ok
USER  PID [...] VSZ   RSS [...] COMMAND
  500 28626 [...] 99952 98472 [...] ./exemple-memoire 100 1000000
Je libere tous les blocs sauf le dernier
USER  PID [...] VSZ   RSS [...] COMMAND

```

```

500 28626 [...] 2932 1452 [...] ./exemple-memoire 100 1000000
Je libere le dernier bloc..
USER  PID [...] VSZ  RSS [...] COMMAND
500 28626 [...] 1952 472 [...] ./exemple-memoire 100 1000000
$ ./exemple-memoire 100 1000000
Je démarre...
USER  PID [...] VSZ  RSS TTY      STAT START  TIME COMMAND
500 28754 [...] 1820 412 [...] ./exemple-memoire 100 1000000
J'alloue 100 blocs de 1000000 octets...Ok
USER  PID [...] VSZ  RSS TTY      STAT START  TIME COMMAND
500 28754 [...] 99952 98472 [...] ./exemple-memoire 100 1000000
Je libere tous les blocs sauf le dernier
USER  PID [...] VSZ  RSS TTY      STAT START  TIME COMMAND
500 28754 [...] 2932 1452 [...] ./exemple-memoire 100 1000000
Je libere le dernier bloc..
USER  PID [...] VSZ  RSS TTY      STAT START  TIME COMMAND
500 28754 [...] 1952 472 [...] ./exemple-memoire 100 1000000
$ ./exemple-memoire 1000000 100
Je démarre...
USER  PID [...] VSZ  RSS [...] COMMAND
500 28631 [...] 1820 412 [...] ./exemple-memoire 1000000 100
J'alloue 1000000 blocs de 100 octets...Ok
USER  PID [...] VSZ  RSS [...] COMMAND
500 28631 [...] 107368 105940 [...] ./exemple-memoire 1000000 100
Je libere tous les blocs sauf le dernier
USER  PID [...] VSZ  RSS [...] COMMAND
500 28631 [...] 107368 105944 [...] ./exemple-memoire 1000000 100
Je libere le dernier bloc..
USER  PID [...] VSZ  RSS [...] COMMAND
500 28631 [...] 5860 4512 [...] ./exemple-memoire 1000000 100
$

```

Nous voyons que durant les deux premières invocations, nous réclamons 100 blocs d'un million d'octets chacun. Pourtant, les deux invocations successives ne conduisent pas à la même occupation mémoire physique. Le système avait profité de la première invocation pour libérer de la mémoire utilisée jusqu'alors par des caches divers, et il disposait alors de plus de place dès le démarrage de la seconde invocation.

Le fait de ne pas libérer le dernier bloc n'a pas de répercussions sur les libérations précédentes. La taille des blocs (1 000 000) étant plus grande que la limite `M_MAP_THRESHOLD` (128 Ko) que nous rencontrerons dans le prochain paragraphe, l'algorithme de `malloc()` utilise des projections en mémoire avec `mmap()`, indépendantes les unes des autres.

Lors de la troisième invocation, nous allouons 1 000 000 blocs de cent octets chacun. La taille mémoire est donc identique à celle des deux premières expériences. Pourtant, la taille de l'espace utilisé, tant en mémoire virtuelle que physique, est modifiée. C'est dû à l'algorithme de `malloc()`. Les blocs étant plus petits qu'une page mémoire, il les attribue par groupes, en utilisant `sbrk()`. Les blocs sont donc « empilés » les uns au-dessus des autres, le dernier se trouvant au sommet. Lorsque nous libérons toute la mémoire, sauf le dernier bloc, `malloc()` ne peut toujours pas faire redescendre la limite `_end` du segment de donnée, et nous voyons que la mémoire libérée n'est toujours pas revenue au système d'exploitation. Ce n'est qu'avec la dernière libération que la mémoire est restituée au noyau.

La conclusion de cette expérience est qu'il est difficile de vérifier depuis l'extérieur si un programme contient des fuites de mémoire. Il est nécessaire de disposer d'outils intégrés aux routines d'allocation pour surveiller le processus. C'est ce que nous verrons dans une prochaine section.

Configuration de l'algorithme utilisé par `malloc()`

La fonction `mallopt()` est déclarée ainsi dans `<malloc.h>` :

```
| int mallopt (int parametre, int valeur);
```

Cette fonction permet de préciser une valeur pour un des paramètres utilisés par les routines d'allocation et de libération. Elle renvoie 1 si elle réussit, et 0 sinon. Les paramètres qu'on peut transmettre à `mallopt()` sont définis par les constantes symboliques suivantes :

Constante	Paramètre
<code>M_MMAP_MAX</code>	Il s'agit du nombre maximal de blocs qui sont alloués en utilisant l'appel système <code>mmap()</code> et non <code>sbrk()</code> . Ce paramètre peut être mis à zéro pour empêcher toute utilisation de <code>mmap()</code> . Sur certains systèmes, la capacité de projection avec <code>mmap()</code> peut être limitée. La valeur par défaut est de 1 024.
<code>M_MMAP_THRESHOLD</code>	Il s'agit de la taille de bloc à partir de laquelle on utilise <code>mmap()</code> et non plus <code>sbrk()</code> . L'avantage de l'emploi de <code>mmap()</code> est que la mémoire ainsi allouée retourne au système d'exploitation (nous l'avons observé) dès sa libération. L'inconvénient est que certains systèmes peuvent être limités en capacité de projection avec <code>mmap()</code> . Dans un contexte multithread, il est interdit de fixer le seuil à une valeur trop grande, car on risquerait de faire croître exagérément le segment de données, ce qui poserait des problèmes d'emplacement des multiples piles. Le seuil par défaut vaut 128 Ko.
<code>M_TOP_PAD</code>	Ce paramètre précise le volume mémoire supplémentaire que <code>malloc()</code> réclame au système lorsqu'elle appelle <code>sbrk()</code> . Cette mémoire supplémentaire sera donc disponible directement dans la fonction de bibliothèque lors des prochaines allocations sans avoir besoin d'invoquer l'appel système. Par défaut, cette valeur est nulle.

Constante	Paramètre
M_TRIM_THRESHOLD	Il s'agit de la taille minimale d'un bloc à libérer pour qu'on appelle <code>sbrk()</code> avec une valeur négative. Pour éviter le surcoût d'un appel système, <code>free()</code> ne libère effectivement la mémoire que lorsque le bloc est suffisamment conséquent. La valeur par défaut est de 128 Ko.

On peut obtenir exactement les mêmes effets en définissant, avant le premier appel à l'une des fonctions de la famille `malloc()`, les variables d'environnement suivantes (éventuellement depuis le shell) :

- `MALLOC_MMAP_MAX`
- `MALLOC_MMAP_THRESHOLD`
- `MALLOC_TOP_PAD`
- `MALLOC_TRIM_THRESHOLD`

Précisons quand même que la modification des paramètres de configuration de l'algorithme utilisé par `malloc()` est rarement nécessaire. Seuls des programmes effectuant de nombreuses allocations dynamiques dans des circonstances assez critiques peuvent avoir besoin de modifier ces données. Notons également qu'il n'est pas possible de lire les valeurs en cours. Les valeurs par défaut sont codées directement dans le fichier source `malloc.c` de la bibliothèque Glibc.

Suivi des allocations et des libérations

Nous avons remarqué précédemment que l'observation externe des processus en cours d'exécution ne permettait pas de vérifier précisément si les allocations et libérations ne recelaient pas de fuites de mémoire. Les versions de `malloc()`, `calloc()`, `realloc()` et `free()` contenues dans la Glibc permettent d'enregistrer automatiquement toutes leurs actions dans un fichier externe. Ce fichier n'est pas conçu pour être lu directement par un utilisateur, mais pour être analysé automatiquement par le script `mtrace` fourni avec la Glibc. Pour activer le suivi, il faut appeler la fonction `mtrace()`, dont le prototype est déclaré dans `<mcheck.h>` :

```
| void mtrace (void);
```

Pour arrêter le suivi, on appelle :

```
| void muntrace (void);
```

Naturellement, on active souvent le suivi dès le début de la fonction `main()`, et on ne le désactive pas. Mais on peut ainsi restreindre le champ de l'analyse à une fonction parti-

culière. Lorsque `mtrace()` est appelée, elle recherche dans la variable d'environnement `MALLOC_TRACE` le nom d'un fichier sur lequel l'utilisateur a un droit d'écriture. Si le fichier existe, il est écrasé. Si `MALLOC_TRACE` n'est pas présente dans l'environnement du processus, ou si elle contient un nom de fichier invalide pour l'écriture, ou encore si le processus est installé avec les bits *Set-UID* ou *Set-GID*, `mtrace()` n'a pas d'effet.

Sinon, elle configure des routines spécifiques dans les points d'accès des routines `malloc()`, `realloc()` et `free()`, comme nous le verrons dans un prochain paragraphe. À chaque invocation de ces fonctions, des informations de débogage sont inscrites dans le fichier. À la fin du processus, on peut appeler l'utilitaire `mtrace` avec le nom de l'exécutable en argument, suivi du fichier de trace. Il présente alors les problèmes éventuels qui ont été détectés. Dans le programme de test suivant, nous introduisons une allocation à deux reprises dans le même pointeur (donc la première mémoire ne peut pas être libérée).

```
exemple-mtrace-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <mcheck.h>

int main (void)
{
    char * ptr;

    mtrace();

    if ((ptr = malloc(512)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    /* On écrase ptr, la première zone n'est plus libérable */
    if ((ptr = malloc(512)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    free(ptr);

    return EXIT_SUCCESS;
}
```

Et voici un exemple de session de débogage :

```
$ export MALLOC_TRACE=trace.out
$ ./exemple-mtrace-1
$ mtrace exemple-mtrace-1 trace.out
```



```
Memory not freed:
-----
      Address      Size      Caller
0x090aa378      0x200 at /home/cpb/Exemples/chapitre-14/exemple-
mtrace-1.c:11
$ unset MALLOC_TRACE
```

La sortie de `mtrace` indique le fichier source fautif, ainsi que le numéro de ligne. Si on ne fournit pas le nom du fichier exécutable, `mtrace` affiche des résultats moins lisibles :

```
$ mtrace trace.out

Memory not freed:
-----
      Address      Size      Caller
0x090aa378      0x200 at 0x804849e
$
```

Lorsque le programme ne présente pas de défaut, `mtrace` l'indique. Dans le programme `exemple-mtrace-2`, nous supprimons l'écrasement du pointeur observé dans l'exemple précédent.

```
$ export MALLOC_TRACE=trace.out
$ ./exemple-mtrace-2
$ mtrace exemple-mtrace-2 trace.out
No memory leaks.
$ unset MALLOC_TRACE
```

Nous avons signalé que lorsque `mtrace()` est invoquée sans que la variable d'environnement `MALLOC_TRACE` ne contienne de nom de fichier correct, la fonction ne faisait rien. Ce n'est pas pour autant une raison pour appeler systématiquement `mtrace()` au début de nos programmes. En effet, il y aura un conflit le jour où on lancera, sans y penser, le programme durant une session de débogage d'une autre application. La variable d'environnement pointant vers le même fichier de suivi, les traces des deux processus seront inextricablement mélangées. La meilleure attitude à adopter est d'encadrer l'appel à `mtrace()` par des directives de compilation conditionnelles :

```
#ifndef NDEBUG
    mtrace();
#endif
```

Ainsi, lorsque la phase de débogage sera terminée, une recompilation permettra d'éliminer automatiquement l'appel à `mtrace()` du code de distribution. Il est important de remarquer qu'une fois que `mtrace()` a été appelée, l'ensemble des fonctions de bibliothèque qui invoquent `malloc()` sont également concernées par le suivi. Il peut s'agir bien entendu de la Glibc, mais également de n'importe quelle autre bibliothèque utilisée par le programme. Certaines de ces fonctions peuvent allouer de la mémoire qu'elles libéreront en une seule fois, à la fin du processus, avec une routine enregistrée par `atexit()`. Si on utilise `muntrace()` avant que la fonction `main()` ne se termine, on risque d'obtenir des messages d'alarme excessifs de `mtrace`. On a donc intérêt à éviter au maximum l'appel de `muntrace()`, sauf lorsqu'on désire déboguer une partie précise du programme.

Notons aussi que `mtrace()` indique toutes les variables dynamiques allouées, mais qui n'ont pas été libérées explicitement. Or, il arrive fréquemment, tant dans les programmes applicatifs que dans les bibliothèques système, que des buffers soient alloués automatiquement au lancement du processus, mais qu'ils ne soient pas libérés avant que la fin du programme ne restitue toute la mémoire au système. On ne s'étonnera donc pas que ces allocations soient signalées à chaque fois. La surveillance des programmes utilisant par exemple les bibliothèques Xlib, Xt ou les fonctions réseau de la bibliothèque C est rendue un peu pénible. En règle générale, on ne s'occupera que des allocations de son propre programme, qu'on filtrera avec `grep`.

Surveillance automatique des zones allouées

Il est possible de demander aux routines d'allocation d'effectuer une surveillance des zones allouées (en s'assurant qu'on ne déborde pas avant ou après un bloc alloué) simplement en définissant la variable d'environnement `MALLOC_CHECK_`. Dès que cette variable est définie, les routines d'allocation deviennent plus tolérantes, permettant les multiples libérations d'un même bloc ou les débordements d'un octet en haut ou en bas d'un bloc. Ensuite, en fonction de la valeur de `MALLOC_CHECK_`, le comportement varie lorsqu'une erreur est rencontrée.

- Si `MALLOC_CHECK_` vaut 1, un message est inscrit sur la sortie d'erreur standard.
- Si `MALLOC_CHECK_` vaut 2, le message est inscrit, puis le processus est arrêté avec un `abort()`. Le fichier `core` créé permettra de retrouver l'endroit où l'erreur s'est produite.
- Pour toute autre valeur de `MALLOC_CHECK_`, l'erreur est silencieusement ignorée. Autant dire que ce n'est pas une méthode raisonnable pour éliminer un dysfonctionnement (quoique cela assure une certaine protection pendant une démo !).

Le programme `exemple-check.c` effectue un certain nombre d'opérations peu recommandables avec des buffers qu'il alloue dynamiquement.

```
exemple-check.c :
#include <stdio.h>
#include <stdlib.h>

#define NB_CHAR 20

int main (void)
{
    int i;
    char * table;

    fprintf(stdout, "Allocation de la table\n");
    table = malloc(NB_CHAR);

    fprintf(stdout, "On déborde vers le haut\n");
    for (i = 0; i <= NB_CHAR; i++)
        table[i] = 1;

    fprintf(stdout, "Libération de la table\n");
    free(table);

    fprintf(stdout, "Allocation de la table\n");
    table = malloc(NB_CHAR);

    fprintf(stdout, "On déborde vers le bas\n");
    i = NB_CHAR;
    while (i >= 0)
        table[--i] = 1;

    fprintf(stdout, "Libération de la table\n");
    free(table);

    fprintf(stdout, "Allocation de la table\n");
    table = malloc(NB_CHAR);

    fprintf(stdout, "Ecriture normale\n");
    for (i = 0; i < NB_CHAR; i++)
        table[i] = 0;

    fprintf(stdout, "Libération de la table\n");
    free(table);

    fprintf(stdout, "Et re-libération de la table !\n");
    free(table);

    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution de ce programme avec diverses configurations de la variable d'environnement :

```
$ unset MALLOC_CHECK_
$ ./exemple-check
Allocation de la table
On déborde vers le haut
Libération de la table
Allocation de la table
On déborde vers le bas
Libération de la table
*** glibc detected *** ./exemple-check: double free or corruption (out): 0x08d52008
***

===== Backtrace: =====
[...]
Abandon (core dumped)
$ export MALLOC_CHECK_=0
$ ./exemple-check
Allocation de la table
On déborde vers le haut
Libération de la table
Allocation de la table
On déborde vers le bas
Libération de la table
Allocation de la table
Ecriture normale
Libération de la table
Et re-libération de la table !
$ export MALLOC_CHECK_=1
$ ./exemple-check
Allocation de la table
On déborde vers le haut
Libération de la table
*** glibc detected *** ./exemple-check: free(): invalid pointer: 0x08c58008 ***
Allocation de la table
On déborde vers le bas
Libération de la table
*** glibc detected *** ./exemple-check: free(): invalid pointer: 0x08c58028 ***
Allocation de la table
Ecriture normale
Libération de la table
Et re-libération de la table !
*** glibc detected *** ./exemple-check: free(): invalid pointer: 0x08c58048 ***
$ export MALLOC_CHECK_=2
```

```
$ ./exemple-check
Allocation de la table
On déborde vers le haut
Libération de la table
Abandon (core dumped)
$
```

La première exécution (`MALLOC_CHECK_` non définie) échoue lors de la tentative de double libération du pointeur. La seconde (valeur quelconque, 0 en l'occurrence) autorise toutes les erreurs sans afficher de message. La troisième exécution (valeur 1) affiche les erreurs, mais les tolère. La troisième exécution (valeur 2) affiche les erreurs et s'arrête dès qu'une incohérence est rencontrée.

Fonctions d'encadrement personnalisées

Nous avons indiqué que les fonctionnalités de surveillance, comme `mtrace()`, utilisent des points d'entrée dans les routines d'allocation et de libération pour insérer leur code. Mais nous pouvons également utiliser ces points d'entrée pour y glisser nos propres fonctions de supervision. Ce genre de procédé de débogage est relativement pointu et n'est généralement nécessaire que pour des logiciels vraiment conséquents, où un processus particulier est, par exemple, chargé de surveiller le déroulement de ses confrères. L'insertion d'une routine de débogage se fait en utilisant les variables globales suivantes, déclarées dans `<malloc.h>` :

Variable	Utilisation
<code>__malloc_hook</code>	Pointeur sur une fonction du type <code>void *fnct(size_t sz, void *appel)</code> Cette routine sera appelée à la place de <code>malloc()</code> ; elle reçoit en argument la taille de bloc à allouer et un pointeur contenant l'adresse de retour, ce qui permet de retrouver l'emplacement de l'appel fautif si une erreur est détectée. Cette fonction doit renvoyer un pointeur sur la zone de mémoire nouvellement allouée. Nous montrerons plus bas comment invoquer l'ancienne routine d'allocation pour obtenir le bloc mémoire désiré.
<code>__realloc_hook</code>	Pointeur sur une routine de type <code>void *fnct(void *old, size_t sz, void *appel)</code> Cette fonction sera appelée lorsqu'on invoquera <code>realloc()</code> et devra s'occuper de redimensionner l'ancien bloc avec la nouvelle taille désirée. L'emplacement où on a fait appel à <code>realloc()</code> est transmis en troisième argument pour retrouver une éventuelle erreur.
<code>__free_hook</code>	Pointeur sur une routine de type <code>void fnct(void *ptr, void *appel)</code> chargée de libérer le bloc de mémoire correspondant au pointeur transmis.

Nous avons parlé de fonctions d'encadrement personnalisées, et non de fonctions d'allocation et de libération personnalisées. En effet, bien qu'il soit possible d'écrire nos propres routines de gestion complète de la mémoire, ce travail serait très difficile, et on se contente souvent d'insérer du code servant de tremplin pour l'appel des véritables routines `malloc()`, `realloc()` et `free()`. Par exemple, on peut enregistrer dans un fichier de traces tous les appels effectués. Les routines d'encadrement peuvent aussi être bien plus subtiles, en vérifiant l'intégrité des blocs mémoire par exemple, par l'ajout de petits repères supplémentaires (des *canaris* dans le vocabulaire du débogage) avant et/ou après les blocs alloués dont la présence et la cohérence sont vérifiées régulièrement.

Pour pouvoir faire appel aux routines originales `malloc()`, `realloc()` ou `free()`, il est nécessaire de stocker dans des variables globales les valeurs initiales des points d'entrée. Si on désire les invoquer, il suffira de restituer les valeurs originales des points d'entrée et de faire un appel normal à la fonction concernée. Comme nous ne connaissons pas les interdépendances entre les routines de la bibliothèque, il faut à chaque fois sauver, modifier et restituer l'ensemble des trois points d'entrée.

Naturellement, au retour d'une des véritables fonctions d'allocation, il faudrait sauver à nouveau les trois points d'entrée et réinstaller nos routines. En effet, une routine comme `malloc()` peut pointer à l'origine sur une fonction d'initialisation qui, après son exécution, modifiera son propre point d'entrée pour accéder directement au code d'allocation durant les invocations ultérieures. Une routine peut donc modifier son propre point d'entrée ou celui des autres fonctions, et on sauvera de nouveau à chaque fois les trois pointeurs.

Cela – bien que tout à fait réalisable – est vraiment compliqué à mettre en œuvre sans erreur. En outre, certains utilitaires modernes, comme *Valgrind*, peuvent nous rendre directement ce genre de services.

Utilisation de Valgrind

Valgrind est un projet composé de plusieurs outils d'instrumentation et de débogage des applications, ne nécessitant pas de modification du fichier exécutable. Pour cela, Valgrind fait tourner l'application sur un « processeur virtuel », c'est-à-dire qu'il interprète les instructions du code exécutable en vérifiant les effets de bords indésirables.

L'outil le plus souvent utilisé est *Memcheck*, un vérificateur de l'utilisation de la mémoire. Memcheck peut détecter très efficacement deux types d'erreurs qui se posent fréquemment dans l'informatique système : les débordements de buffers (mais uniquement de buffers alloués dynamiquement), et les fuites de mémoire.

L'énorme avantage de Valgrind par rapport aux `mtrace()`, `MALLOC_CHECK_`, et autres outils basés sur les `__malloc_hook` est qu'il nous prévient immédiatement lorsqu'une erreur est détectée, et pas uniquement au moment de la libération des buffers.

Voici un premier exemple, où nous allons réaliser un débordement de buffer. Notre programme attend un argument sur sa ligne de commande, qu'il copie dans une chaîne allouée dynamiquement puis qu'il affiche :

```
exemple-debordement.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char * argv[])
{
    char * chaine;
    if (argc < 2) {
        fprintf(stderr, "usage: %s argument\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "Allocation de la memoire...\n");
    chaine = malloc(strlen(argv[1]));
    fprintf(stderr, "Copie de la chaine...\n");
    strcpy(chaine, argv[1]);
    fprintf(stderr, "Affichage de la chaine\n");
    fprintf(stderr, "    %s\n", chaine);
    fprintf(stderr, "Liberation de la chaine\n");
    free(chaine);
    fprintf(stderr, "Fin du programme\n");
    return EXIT_SUCCESS;
}
```

Si nous exécutons notre programme « tel quel », il semble fonctionner correctement :

```
$ ./exemple-debordement
usage: ./exemple-debordement argument
$ ./exemple-debordement azerty
Allocation de la memoire...
Copie de la chaine...
Affichage de la chaine
    azerty
Liberation de la chaine
Fin du programme
$
```

Pourtant, si nous l'exécutons avec `MALLOC_CHECK_` initialisé, la Glibc nous signale une erreur :

```
$ export MALLOC_CHECK_=1
$ ./exemple-debordement azerty
Allocation de la memoire...
Copie de la chaine...
Affichage de la chaine
    azerty
Libération de la chaine
*** glibc detected *** ./exemple-debordement: free(): invalid pointer:
0x09a81008 ***
Fin du programme
$ unset MALLOC_CHECK_
```

L'inconvénient est que l'erreur (débordement d'un octet dû à l'oubli du caractère nul final au moment de l'allocation) n'est détectée qu'à la libération du pointeur. Dans une application en vraie grandeur, l'écrasement pourrait avoir eu lieu très longtemps auparavant et serait difficile à retrouver. Faisons donc appel à Valgrind :

```
$ valgrind ./exemple-debordement azerty
==3805== Memcheck, a memory error detector
==3805== Copyright (C) 2002-2009, and GNU GPL'd, by Julian Seward et al.
==3805== Using Valgrind-3.5.0 and LibVEX; rerun with -h for copyright
info
==3805== Command: ./exemple-debordement azerty
==3805==
Allocation de la memoire...
Copie de la chaine...
==3805== Invalid write of size 1
==3805==    at 0x4006DD7: strcpy (mc_replace_strmem.c:311)
==3805==    by 0x80485C1: main (exemple-debordement.c:16)
==3805== Address 0x402302e is 0 bytes after a block of size 6 alloc'd
==3805==    at 0x4005BDC: malloc (vg_replace_malloc.c:195)
==3805==    by 0x804857D: main (exemple-debordement.c:14)
==3805==
Affichage de la chaine
==3805== Invalid read of size 1
==3805==    at 0x1C6C91: vfprintf (in /lib/libc-2.12.so)
==3805==    by 0x1C8571: buffered_vfprintf (in /lib/libc-2.12.so)
==3805==    by 0x1C35A2: vfprintf (in /lib/libc-2.12.so)
==3805==    by 0x1CDB9E: fprintf (in /lib/libc-2.12.so)
==3805==    by 0x8048607: main (exemple-debordement.c:18)
```



```
==3805== Address 0x402302e is 0 bytes after a block of size 6 alloc'd
==3805==    at 0x4005BDC: malloc (vg_replace_malloc.c:195)
==3805==    by 0x804857D: main (exemple-debordement.c:14)
==3805==
    azerty
Liberation de la chaine
Fin du programme
==3805==
==3805== HEAP SUMMARY:
==3805==    in use at exit: 0 bytes in 0 blocks
==3805== total heap usage: 1 allocs, 1 frees, 6 bytes allocated
==3805==
==3805== All heap blocks were freed -- no leaks are possible
==3805==
==3805== For counts of detected and suppressed errors, rerun with: -v
==3805== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 12 from 8)
$
```

Les lignes préfixées par le numéro de processus (3 805 en l'occurrence) ont été écrites par Valgrind.

Nous pouvons remarquer que dès la copie de l'argument dans la chaîne trop petite, Valgrind signale un « *Invalid write of size 1* » (écriture d'un caractère en trop), en précisant les fonctions concernées et les numéros de lignes dans les fichiers source, ainsi que la position (« 0 bytes after » – juste après) d'un bloc de 6 octets en indiquant également l'emplacement du `malloc()` ayant alloué ce bloc.

Lors de l'affichage de la chaîne, un second débordement a lieu, en lecture cette fois, durant un `fprintf()` et Valgrind nous l'indique immédiatement avec les mêmes informations de débogage.

Un compte-rendu final nous annonce que deux erreurs se sont produites précédemment.

Pour mettre en relief le débogage des allocations mémoire, nous allons écrire un programme qui alloue quelques blocs de mémoire, puis écrit dans les blocs, et finalement les libère dans l'ordre inverse de l'allocation.

```
exemple-fuite.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define NB_BLOCS 10
#define TAILLE_BLOC 1024
```

```
int main(void)
{
    int i;
    char * ptr[NB_BLOCS];

    fprintf(stderr, "Allocation de %d blocs\n", NB_BLOCS);
    for (i = 0; i < NB_BLOCS; i++)
        ptr[i] = malloc(TAILLE_BLOC);
    fprintf(stderr, "Utilisation des blocs\n");
    for (i = 0; i < NB_BLOCS; i++)
        memset(ptr[i], i, TAILLE_BLOC);
    fprintf(stderr, "Liberation des blocs\n");
    for (i = NB_BLOCS - 1; i > 0; i--)
        free(ptr[i]);
    return EXIT_SUCCESS;
}
```

L'exécution normale semble correcte :

```
$ ./exemple-fuite
Allocation de 10 blocs
Utilisation des blocs
Liberation des blocs
$
```

Malgré tout, nous allons vérifier le programme avec Valgrind :

```
$ valgrind ./exemple-fuite
==7343== Memcheck, a memory error detector
[...]
==7343==
Allocation de 10 blocs
Utilisation des blocs
Liberation des blocs
==7343==
==7343== HEAP SUMMARY:
==7343==    in use at exit: 1,024 bytes in 1 blocks
==7343==   total heap usage: 10 allocs, 9 frees, 10,240 bytes allocated
==7343==
==7343== LEAK SUMMARY:
==7343==    definitely lost: 1,024 bytes in 1 blocks
==7343==    indirectly lost: 0 bytes in 0 blocks
==7343==    possibly lost: 0 bytes in 0 blocks
==7343==    still reachable: 0 bytes in 0 blocks
```

```
==7343==          suppressed: 0 bytes in 0 blocks
==7343== Rerun with --leak-check=full to see details of leaked memory
==7343==
==7343== For counts of detected and suppressed errors, rerun with: -v
==7343== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 12 from 8)
$
```

Le compte-rendu final nous indique que 1 024 octets sont « définitivement perdus ». Ce message apparemment inquiétant indique simplement qu'il existe dans la mémoire du processus des données allouées dynamiquement sur lesquelles il n'y a plus de pointeur utilisable. La mémoire sera naturellement entièrement libérée au moment de la terminaison de l'application.

Valgrind nous indique que l'on peut le relancer avec l'option `--leak-check=full` pour plus de détails. Suivons son conseil :

```
$ valgrind --leak-check=full ./exemple-fuite
==7421== Memcheck, a memory error detector
[...]
Allocation de 10 blocs
Utilisation des blocs
Libération des blocs
==7421==
==7421==  HEAP SUMMARY:
==7421==      in use at exit: 1,024 bytes in 1 blocks
==7421==    total heap usage: 10 allocs, 9 frees, 10,240 bytes
allocated
==7421==
==7421== 1,024 bytes in 1 blocks are definitely lost in loss record 1
of 1
==7421==    at 0x4005BDC: malloc (vg_replace_malloc.c:195)
==7421==    by 0x80484E5: main (exemple-fuite.c:16)
==7421==
==7421== LEAK SUMMARY:
[...]

```

L'exécution détaillée indique que le bloc perdu mesure 1 024 octets, et a été alloué ligne 16 de notre fonction `main()`. En vérifiant plus précisément nos limites, nous voyons que la boucle décroissante « oublie » de traiter la valeur 0, et qu'il faut remplacer `i>0` par `i>=0`.

Valgrind intègre de nombreuses autres possibilités, pour la surveillance des zones partagées en multithread, le suivi du tas, la prédiction des erreurs de cache, etc. Ce programme est indispensable dans la boîte à outils du développeur Linux.

Conclusion

Ce chapitre a permis de mettre en place les principes fondamentaux de la gestion de la mémoire d'un processus. Il s'agit de notions élémentaires et de routines présentes dans l'essentiel des applications courantes. Les méthodes de débogage ou de paramétrage présentées ici sont particulièrement précieuses, mais elles ne sont pas toujours portables en dehors d'un environnement de programmation Gnu. Le prochain chapitre va nous permettre d'aborder des notions plus pointues sur la manipulation de l'espace mémoire d'un processus.

Les fonctions générales d'allocation mémoire sont décrites dans [Kernighan 1994] *Le langage C*. Pour obtenir des précisions sur les rapports entre les fonctions de bibliothèque comme `malloc()` et les appels système comme `brk()` ou `mmap()`, on consultera avec profit les sources de la bibliothèque Glibc. On notera également qu'une discussion concernant les risques induits par les allocations dynamiques et que des idées pour dépister les bogues sont présentées dans le livre de [McConnell 1994] *Programmation professionnelle*.

15

Gestion avancée de la mémoire

Nous nous intéresserons dans ce chapitre à divers aspects de la gestion mémoire, plus complexes que les allocations et libérations étudiées dans le précédent chapitre. La plupart des fonctions qui permettent la gestion mémoire ont été introduites par la norme Posix.1b (avant d'être intégrées dans SUSv4) car elles concernent souvent les applications temps réel.

Nous étudierons tout d'abord le mécanisme du verrouillage de pages en mémoire, qui intéresse principalement les processus temps réel et les applications de cryptographie. Nous verrons par la suite les possibilités offertes par le noyau Linux en ce qui concerne la projection de fichier dans une zone de l'espace mémoire du processus. Nous pourrons alors comprendre le détail des techniques d'allocation de mémoire dynamique au niveau du noyau.

Nous verrons par la suite les possibilités offertes par le noyau Linux en ce qui concerne la projection de fichier dans une zone de l'espace mémoire du processus. Nous pourrons alors comprendre le détail des techniques d'allocation de mémoire dynamique au niveau du noyau. Ceci est également parfaitement utilisable en Python comme nous le verrons plus loin.

Enfin, nous observerons les protections d'une page mémoire contre les accès indésirables.

Verrouillage de pages en mémoire

La gestion de la mémoire virtuelle sous Linux permet aux processus de disposer, dans leur ensemble, de beaucoup plus de mémoire que la machine n'en contient physiquement. Nous en avons vu une illustration dans les premiers paragraphes du chapitre

précédent, où le programme `exemple-malloc-1` arrivait à disposer de 3 Go sur une machine ne comportant que 2 Go de mémoire vive.

Lorsque le noyau doit allouer une nouvelle page et qu'il n'en a plus de disponible en mémoire vive, il recherche une page inutilisée depuis longtemps et la rejette de la mémoire. Si cette page contient uniquement des données qui ont été lues depuis un fichier, par exemple le code exécutable d'un programme, elle est purement et simplement effacée. On sait où la retrouver. Si la page a été modifiée par le processus auquel elle appartient, en revanche le noyau la sauvegarde temporairement sur son périphérique de *swap*.

Un processus est donc en permanence susceptible de voir l'une de ses pages disparaître temporairement dans la mémoire secondaire, pour être rechargée lorsqu'il en demandera l'accès. Ce mécanisme très utile est transparent pour les processus, mais peut présenter néanmoins plusieurs inconvénients dans certains cas bien précis.

- Au moment où un processus tente d'accéder à une zone mémoire qui a été rejetée sur le périphérique de *swap*, le processeur déclenche une faute de page. Le noyau recharge alors la page manquante et rend le contrôle au périphérique. Pour trouver suffisamment de place pour charger la page en question, le noyau peut être obligé d'évacuer d'autres zones de données de la mémoire, quitte à les sauvegarder à leur tour sur le disque. Tout cela ralentit considérablement l'accès à la zone mémoire, et pire, le ralentissement n'est pas prévisible. Or, sur des systèmes de contrôle industriel par exemple, des processus séquencés suivant un mécanisme temps réel doivent pouvoir accéder à leur mémoire dans un temps limité, pas nécessairement de façon extrêmement rapide mais constante, quelles que soient les conditions de fonctionnement de la machine.
- Il peut arriver que des processus contiennent temporairement des données hautement confidentielles, comme des mots de passe ou des clés de cryptage. Si le noyau bascule sur le périphérique de *swap* la zone contenant ces données, elles s'y trouveront encore après le rechargement de la page en mémoire vive. Il est alors possible de démonter le périphérique de *swap* assez rapidement et de l'examiner octet par octet pour rechercher les informations secrètes.

Le noyau Linux offre une solution à ces problèmes, ou plutôt deux solutions différentes. Dans le premier cas, notre processus contient des zones critiques où il ne peut pas se permettre le moindre délai imprévu. Aucune zone de mémoire ne doit donc être projetée sur le *swap*. De plus, le processus doit disposer de suffisamment d'espace de pile libre pour les invocations des sous-routines. Rappelons que nous considérons uniquement les zones critiques de notre processus temps réel, zones dans lesquelles on ne s'amusera pas à faire des allocations de nouvelles pages de mémoire, par exemple. Nous devons donc avoir l'assurance à l'entrée dans la portion critique qu'aucune partie du processus (données, code, pile) ne sera évacuée sur le périphérique de *swap*, et ceci jusqu'à la sortie de la section critique.

Pour le second cas par contre, seules de petites parties de la mémoire doivent rester absolument en mémoire physique. Le code, la pile et les autres données du processus peuvent très bien être éjectés temporairement ; nous voulons seulement nous assurer que nos données confidentielles ne seront pas stockées, même passagèrement sur le disque. Bien entendu, ce genre de processus doit utiliser l'appel système `setrlimit()` pour mettre à zéro sa limite `RLIMIT_CORE`, comme nous l'avons vu dans le chapitre sur l'exécution des processus, afin d'être sûr de ne pas laisser une image mémoire sur le disque en cas de terminaison anormale.

L'appel système `mlock()` permet de verrouiller une zone particulière dans la mémoire centrale. Les données qui s'y trouvent ne seront pas éjectées sur le *swap* avant que le processus ne se termine, qu'il effectue un `exec()` ou qu'il invoque l'un des appels système `munlock()` et `munlockall()`.

L'appel système `mlockall()` verrouille toute la mémoire appartenant au processus, aussi bien son espace de code, de données, que sa pile, les bibliothèques partagées qu'il utilise ou les fichiers projetés que nous verrons à la prochaine section.

L'appel `munlock()` déverrouille uniquement une zone précise de mémoire alors que `munlockall()` déverrouille toute la mémoire du processus.

Le verrouillage de page en mémoire est une opération privilégiée car elle lèse nécessairement les autres processus qui disposeront de moins de mémoire physique à se partager. Ces appels système sont donc réservés aux processus s'exécutant avec l'*UID* effectif de root ou possédant la capacité `CAP_IPC_LOCK`. Notez que le nom de cette capacité évoluera peut-être dans l'avenir car le préfixe `IPC` représente habituellement les communications entre processus comme nous le verrons plus loin.

Il faut bien comprendre qu'il n'y a pas d'empilement des verrouillages. Autrement dit, une zone qui a été verrouillée plusieurs fois, par exemple en intersection de plusieurs `mlock()` et avec un `mlockall()`, sera déverrouillée avec un seul appel `munlock()`. Il ne faut donc pas utiliser des verrouillages temporaires dans des routines sans être conscient qu'on risque de déverrouiller des pages que la fonction appelante considèrerait comme fixées.

Les prototypes des fonctions de verrouillage sont déclarés dans `<sys/mman.h>` ainsi :

```
int mlock (const void * adresse_debut, size_t longueur)
int munlock (const void * adresse_debut, size_t longueur);
int mlockall (int type);
int munlockall (void);
```

Tous ces appels système renvoient 0 s'ils réussissent et -1 en cas d'échec.

On peut préciser en argument de `mlockall()` ce qu'on désire verrouiller, en effectuant un OU binaire entre les constantes suivantes :

- MCL_CURRENT permet de verrouiller les zones mémoire actuellement possédées par le processus ;
- MCL_FUTURE sert à verrouiller les zones qui ont déjà été allouée en mémoire virtuelle mais pour lesquelles la mémoire physique n'a pas encore été réservée. Ceci concerne bien entendu les nouvelles zones de mémoire allouées dynamiquement, mais également la pile par exemple.

Attention, lorsqu'on veut verrouiller l'ensemble des pages actuelles, il faut utiliser MCL_CURRENT | MCL_FUTURE et non MCL_FUTURE seule.

Avant d'entrer dans une section critique d'un processus temps réel, nous devons allouer toutes les données nécessaires, les verrouiller en mémoire, et réserver assez de place dans la pile pour tous les appels de sous-routines durant cette portion critique. On utilisera un schéma comme celui-ci :

```
// Allocation de toutes les variables dynamiques nécessaires
... malloc(...) ...

// Écriture dans les variables dynamiques pour être sûr
... memset(...) ...

// Réserve de l'espace de pile et verrouillage par
// une fonction spéciale.
if (reserve_pile_et_verrouille() != 0) {
    perror("mlockall");
    exit(EXIT_FAILURE);
}

// Maintenant nous pouvons entrer dans la section critique
...
...
// Sortie de la portion critique du code
munlockall();

// Voici à présent la routine réservant la place nécessaire dans
// la pile et verrouillant la mémoire

#define TAILLE_RESERVEE 8192

static int reserve_pile_et_verrouille (void)
{
    char * reserve;
    // Réserve dans la pile
    reserve = alloca(TAILLE_RESERVEE);
    // Écriture pour s'assurer que les pages sont bien affectées
    memset(reserve, 0, TAILLE_RESERVEE);
    // Verrouillage de la mémoire
    return mlockall(MCL_CURRENT);
}
```



```
// En revenant, la mémoire occupée par la variable "réserve"  
// est libérée mais les pages restent en mémoire physique  
// à la disposition du processus pour sa pile.  
}
```

Sous Linux, on ne peut verrouiller au maximum que la moitié de la mémoire physique du système. Les appels système `mlock()` et `mlockall()` peuvent donc échouer si on essaye de dépasser cette quantité (où s'il n'y a plus assez de place en mémoire à cause d'un autre processus ayant verrouillé d'autres zones). Cela signifie aussi que lorsqu'on utilise `MCL_FUTURE` avec `mlockall()`, des allocations mémoire à venir pourront échouer à cause de cette limitation.

Le noyau verrouillant des pages entières de mémoire, on risque des chevauchements entre des zones mémoire proches. Il est en effet possible de déverrouiller entièrement une page alors qu'elle contient des variables qu'on croit encore protégées. Dans le cas d'un processus désirant protéger des données bien précises mais pas tout son espace mémoire, il est conseillé de laisser toutes les données verrouillées jusqu'à ce qu'elles puissent toutes être relâchées sans risque.

Le verrouillage d'une zone de mémoire n'est pas hérité au cours d'un `fork()`. Les données restent verrouillées en mémoire pour le père, mais dès que l'un des deux processus veut modifier le contenu d'une page mémoire, le noyau en fait une copie à l'intention du fils, et cette copie n'est pas verrouillée. On peut donc dire qu'en cas de verrouillage complet avec `mlockall()`, le processus fils bénéficiera probablement du maintien en mémoire du code du programme jusqu'à la fin de son père, mais ce n'est pas obligatoire.

Les fonctions de verrouillage de pages en mémoire sont définies par SUSv4. Pour assurer la portabilité d'un programme, on prendra en compte dans la compilation, par des tests conditionnels, les constantes symboliques :

- `_POSIX_MEMLOCK` qui indique que `mlockall()` et `munlockall()` sont utilisables sur le système ;
- `_POSIX_MEMLOCK_RANGE` qui indique que `mlock()` et `munlock()` sont disponibles.

Projection d'un fichier sur une zone mémoire

Il est parfois intéressant, plutôt que de travailler directement sur le contenu d'un fichier, d'en projeter une image en mémoire, sur laquelle on œuvre ensuite comme avec des variables normales. Cette projection permet de manipuler le contenu du fichier au moyen d'une image en mémoire, donc beaucoup plus rapidement et simplement qu'avec de véritables lectures et écritures sur le disque.

Le noyau offre l'appel système `mmap()` pour assurer la projection d'un fichier en mémoire. Son prototype est déclaré dans `<sys/mman.h>` ainsi :

```
void * mmap (void * debut, size_t longueur,  
             int protection, int attribut,  
             int fd, off_t offset);
```

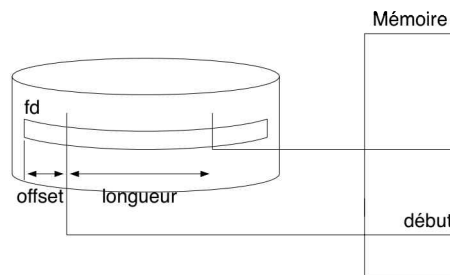
En Python, la fonction est définie ainsi :

```
mmap.mmap(fd, longueur, attributs, protections)
```

Elle renvoie un objet `mmap` assimilable à un tableau d'octets, mais sur lequel des fonctions classiques d'accès aux fichiers (`read()`, `write()`, `seek()`, etc.) sont applicables.

Nous allons voir les arguments les uns après les autres. Précisons tout de suite qu'en cas d'échec, `mmap()` renvoie la constante symbolique `MAP_FAILED`, sinon il renvoie un pointeur sur la zone de mémoire allouée pour la projection. Cette zone se trouve dans la portion de l'espace d'adressage nommée « tas ». Elle se situe au-dessus des données globales non initialisées, et en dessous de la pile.

Figure 15-1
Projection d'un fichier en
mémoire



L'argument `début` indique l'emplacement dans lequel on désire effectuer la projection. Il ne s'agit que d'un simple désir, `mmap()` pouvant utiliser n'importe quelle autre adresse s'il le préfère. En général, on ne s'occupe pas de cet argument, on transmet `NULL` pour signifier à `mmap()` de prendre l'emplacement de son choix. Notons qu'on ne doit pas allouer de mémoire pour la projection ; en fait la zone de projection ne doit même pas avoir d'intersection avec un bloc mémoire précédemment alloué. Si on désire absolument choisir un emplacement précis, il faut passer une adresse alignée sur une frontière de page. La taille d'une page est disponible en interrogeant la fonction `sysconf()` avec l'argument `_SC_PAGESIZE`. Nous ne développerons pas plus l'utilisation de l'argument `début` car il est principalement utilisé par les développeurs du noyau pour le chargement des fichiers exécutables et des bibliothèques partagées, sans intérêt immédiat pour un programmeur applicatif.

L'argument `longueur` indique, on s'en doute, la longueur de la projection en mémoire. Cet argument sera arrondi par excès pour contenir un nombre entier de pages mémoire. On peut indiquer plusieurs types de protection dans le troisième argument, en utilisant un OU binaire entre les constantes suivantes :

- `PROT_EXEC` : le processus pourra exécuter le code contenu dans la projection ainsi réalisée. Ceci est utilisé par le noyau pour implémenter l'appel système `exec()` avec les fichiers binaires exécutables ;
- `PROT_READ` : les pages obtenues seront accessibles en lecture ;
- `PROT_WRITE` : on pourra écrire dans la projection. Le noyau assurera la synchronisation avec le fichier disque dans des circonstances dont nous reparlerons plus bas.

En fait, il existe également la protection vide `PROT_NONE` grâce à laquelle on ne peut ni lire, ni écrire, ni exécuter de code dans la mémoire projetée. On peut l'utiliser de manière très spécifique pour le débogage, comme nous le décrirons dans la section traitant des protections de zones mémoire, mais c'est très rare.

Sur certaines architectures x86, le fait de demander `PROT_WRITE` entraîne obligatoirement `PROT_READ`, et qu'il n'y a pas de différence entre `PROT_READ` et `PROT_EXEC`. Ce n'est toutefois pas le cas sur d'autres processeurs, aussi positionnera-t-on bien les protections dont on a besoin.

L'argument suivant, `attribut`, est composé d'un OU binaire entre les constantes symboliques suivantes.

- `MAP_PRIVATE` : la projection n'est pas destinée à être réécrite sur le disque, et la zone de mémoire allouée ne doit pas être partagée avec d'autres processus. Si le programme effectue une modification sur la zone de projection, une copie privée de cette portion de mémoire lui sera attribuée.
- `MAP_SHARED` : au contraire de l'attribut précédent, les écritures dans la zone de projection sont destinées à être vues par le reste du monde. Dans le cas où la zone de mémoire est partagée, comme nous le verrons dans les communications entre processus, toute modification est immédiatement perceptible dans les autres programmes. En revanche, la synchronisation effective avec le fichier disque n'est pas nécessairement immédiate. Il faut être prudent si un processus concurrent tente d'accéder directement au fichier car il ne verra pas forcément les mêmes données que celles de la projection en mémoire.
- `MAP_FIXED` : permet de signaler à `mmap()` qu'on désire absolument disposer d'une projection à l'adresse de départ indiquée en premier argument. Si ce n'est pas possible, l'appel système échouera.

Ces attributs sont les seuls qui sont définis par SUSv4, mais Linux propose également d'autres attributs spécifiques, en voici quelques-uns.

- `MAP_ANON` ou `MAP_ANONYMOUS` : on ne projette pas réellement un fichier, mais on désire obtenir une zone mémoire vierge, emplie de zéros. La bibliothèque Glibc l'utilise pour implémenter `malloc()`. Lorsque cet attribut est mentionné, les deux derniers arguments `fd` et `offset` de l'appel système sont ignorés. Sur les systèmes où `MAP_ANONYMOUS` n'existe pas, on peut se servir des pseudo-fichiers `/dev/null` ou `/dev/zero` pour allouer une zone de mémoire vierge.
- `MAP_DENYWRITE` : empêche l'écriture dans le fichier correspondant tant que la projection en mémoire est valide. C'est ce qu'utilise l'appel système `exec()` pour verrouiller le fichier exécutable correspondant au processus lancé. Cela permet au noyau de supprimer des pages de code pour libérer de la mémoire tout en étant sûr de pouvoir les relire par la suite. C'est aussi ce qui déclenche une erreur `ETXTBSY` de l'éditeur de liens lorsqu'on relance une compilation sans avoir arrêté le processus résultant. Malheureusement, cet attribut est silencieusement ignoré durant les appels système, seul le noyau a la possibilité de l'utiliser.
- `MAP_GROWSDOWN` : indique qu'on utilise une projection anonyme pour allouer une zone de mémoire qui grossira par le bas. Cet attribut sert au noyau pour mettre en place la pile d'un processus.

Lors d'un appel à `mmap()`, il faut utiliser l'une des deux constantes `MAP_PRIVATE` ou `MAP_SHARED`, même si elles n'ont de sens que lorsqu'on écrit dans la zone de projection. Toutes les autres sont facultatives.

On comprend mieux à présent comment `mmap()` est utilisé par `malloc()` pour réserver de gros blocs en mémoire, et pourquoi la mémoire allouée avec `calloc()` contient des zéros sans qu'il y ait eu de véritable écriture pour l'initialiser : lorsque nous demandons une projection `PRIVATE` et `ANONYMOUS`, le noyau nous fournit des pages virtuelles, sans correspondance réelle avec la mémoire physique du système. Quand nous essayons de lire le contenu de ces zones, une faute de page est déclenchée et le noyau, s'apercevant qu'il s'agit d'une projection anonyme (ou du pseudo-fichier `/dev/zero`), sait qu'il doit nous renvoyer des zéros. Par contre, dès que nous tentons d'écrire sur cette projection, la faute de page qui se déclenche oblige le noyau à affecter une véritable zone de mémoire physique au processus, à copier le contenu de la projection (en l'occurrence, il suffit de remplir la nouvelle page avec des zéros), puis à autoriser l'écriture par le programme. Ce n'est donc qu'au moment de l'écriture dans la mémoire allouée avec `mmap()` que le stock de pages physiquement disponibles diminue.

L'argument `fd` correspond à un descripteur de fichier déjà ouvert par le processus. Bien entendu le mode d'ouverture devra correspondre, en ce qui concerne les possibilités d'écriture dans le fichier, aux protections de la zone mémoire. Dans tous les cas, le fichier devra permettre la lecture, autrement dit il faudra l'ouvrir en mode `O_RDONLY` ou `O_RDWR` (nous verrons la signification de ces modes dans le chapitre consacré à la gestion des fichiers).

Le dernier argument, `offset`, correspond à la position dans le fichier de la projection. On n'est pas obligé en effet de projeter tout le fichier, pas plus d'ailleurs que le début. Le décalage est mesuré en nombre d'octets, comme la valeur qu'on transmet à un appel système `lseek()`.

Lorsque `mmap()` échoue, la variable globale `errno` est mise à jour et contient un code indiquant l'erreur qui s'est produite. Certaines erreurs ne sont pas vraiment intuitives, aussi nous allons les détailler rapidement dans le tableau suivant.

Erreur	Problèmes
EACCES	Le fichier ne nous permet pas la lecture.
	Le fichier ne nous permet pas l'écriture, et on a réclamé un accès à la fois en <code>PROT_WRITE</code> sur la zone et un partage <code>MAP_SHARED</code> . Dans le cas d'une projection privée <code>MAP_PRIVATE</code> , ce n'est pas un problème, le fichier ne sera pas mis à jour en cas de modification de la zone de mémoire puisqu'on travaille sur une copie. En projection partagée, il faut pouvoir inscrire sur le disque les modifications pour les rendre visibles à l'extérieur.
	De même, en projection partagée, un fichier ne doit pas être ouvert en mode d'ajout uniquement en fin de fichier.
EAGAIN	Le processus a demandé un verrouillage en mémoire de ses allocations à venir avec l'argument <code>MCL_FUTURE</code> de l'appel <code>mlockall()</code> . Allouer la taille demandée dépasserait sa limite de mémoire verrouillée <code>RLIMIT_MEMLOCK</code> , accessible avec <code>getrlimit()</code> .
	Un verrouillage strict a été placé par le processus avec <code>fcntl()</code> sur le fichier, et on demande une projection partagée avec <code>MAP_SHARED</code> .
EINVAL	On a demandé une projection à une adresse invalide ou de longueur beaucoup trop grande.
	L'adresse mentionnée n'est pas alignée sur une frontière de page alors qu'on a demandé une projection figée avec <code>MAP_FIXED</code> .
	On n'a utilisé ni l'attribut <code>MAP_SHARED</code> ni <code>MAP_PRIVATE</code> , ou au contraire on les a mentionnés tous les deux.
	On a demandé une projection <code>ANONYMOUS</code> avec un attribut <code>MAP_SHARED</code> . C'est impossible car la projection anonyme nécessite de faire une copie dès une tentative d'écriture dans les pages virtuelles vierges. Attention, sur certains systèmes ce type de projection est autorisée et permet de créer de la mémoire partagée entre un processus père et son fils. Sous Linux, ce n'est pas le cas. Toutefois, on peut utiliser un fichier temporaire de la dimension voulue pour partager un bloc mémoire facilement, comme nous le ferons dans notre second exemple.
ENODEV	Le fichier mentionné ne permet pas la projection en mémoire. Ceci concerne surtout des fichiers spéciaux représentant des périphériques comme <code>/dev/tty</code> par exemple.
ENOMEM	Il y a trop de projections en mémoire. La limite <code>MAX_MAP_COUNT</code> du nombre de projections simultanées sur le système est définie dans <code><linux/sched.h></code> . Elle vaut 65536 sur les architectures x86.
	On ne peut pas obtenir une zone de mémoire de la taille demandée. Ceci peut aussi correspondre à un manque de mémoire du noyau le rendant incapable de mettre à jour ses structures internes.

L'erreur ETXTBSY devrait normalement être renvoyée si on demande la projection en mémoire avec l'attribut `MAP_DENYWRITE` d'un fichier déjà ouvert en écriture par plusieurs processus. Toutefois, cet attribut est silencieusement effacé par le noyau Linux lors de l'appel système.

Une fois que `mmap()` a réussi, il est possible de refermer le fichier, la projection en mémoire persistera jusqu'à ce qu'on la libère avec `munmap()`. Les zones projetées sont automatiquement libérées lors de la fin d'un processus ou d'un appel `exec()`. Néanmoins, les projections sont héritées au cours d'un `fork()`, nous en verrons un exemple plus loin. Les projections `PRIVATE` sont copiées au moment de la première tentative d'écriture dans la zone, aussi il n'y a aucune ambiguïté entre les processus père et fils.

L'appel système `munmap()` a le prototype suivant :

```
| int munmap (void * zone, size_t longueur);
```

Il libère la projection se trouvant dans la zone indiquée, avec la longueur passée en second argument. La seule manière raisonnable d'utiliser `munmap()` est de lui transmettre le pointeur fourni par un appel antérieur à `mmap()`, ainsi que la même longueur que celle qu'on avait passée à `mmap()`. Lorsqu'il réussit, `munmap()` renvoie 0. La zone de mémoire est alors libérée et rendue inutilisable.

On invoque rarement `munmap()`, car la plupart du temps les projections qu'on établit en mémoire sont utilisées pendant toute la durée de vie du processus, et on laisse le système les libérer automatiquement à la terminaison du programme.

Il est possible de faire une projection d'un fichier complet, dont la taille dépasse largement celle de la mémoire totale – physique et *swap* – du système. En effet, le noyau libérera des pages mémoire au fur et à mesure, en les réécrivant sur le disque si elles ont été modifiées, ou en les effaçant purement et simplement si elles sont intactes. La limitation théorique est celle de l'espace d'adressage disponible, c'est-à-dire sur architecture 32 bits 4 Go moins 1 Go réservé au noyau, moins le code du processus, sa pile et ses données, ce qui nous laisse quand même normalement plus de 2 Go. Attention, ceci n'est vrai que si le noyau peut éliminer de la mémoire les pages qui l'encombrent, autrement dit le calcul est totalement différent si on utilise une projection `PRIVATE` et si on modifie les données projetées. Dans ce cas, la limite de projection est de l'ordre de la taille de la mémoire virtuelle disponible.

Notre premier exemple va consister à écrire un programme qui retourne complètement un fichier, en échangeant successivement le premier octet et le dernier, le deuxième et l'avant-dernier, et ainsi de suite. L'intérêt d'un tel utilitaire reste encore à démontrer, mais on pourrait très bien le transformer pour effectuer du traitement d'image, par exemple.

Nous utiliserons la fonction `fstat()` – que nous étudierons ultérieurement en détail – pour connaître la taille du fichier, puis nous le projeterons en mémoire et

nous le retournerons simplement en le considérant comme un tableau de caractères. Ceci aurait été particulièrement fastidieux à écrire avec des appels système `read()` et `write()`, alors que l'implémentation se fait, avec `mmap()`, de manière très intuitive.

```
exemple-mmap-1.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main (int argc, char * argv[])
{
    int    i;
    int    fd;
    char   tmp;
    char * projection;
    long   taille_fichier;
    struct stat etat_fichier;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier_a_inverser\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open(argv[1], O_RDWR)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    if (fstat(fd, & etat_fichier) != 0) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }
    taille_fichier = etat_fichier.st_size;
    projection = (char *) mmap(NULL, taille_fichier,
                               PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (projection == (char *) MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    close(fd);
    for (i = 0; i < taille_fichier / 2; i++) {
        tmp = projection[i];
        projection[i] = projection[taille_fichier - i - 1];
        projection[taille_fichier - i - 1] = tmp;
    }
    munmap((void *) projection, taille_fichier);
    return EXIT_SUCCESS;
}
```

Nous l'utilisons pour retourner un petit fichier de texte.

```
$ cat > test.txt

AZERTYUIOP
QSDFGHJKLM
WXCVCBN
  (Contrôle-D)
$ ./exemple-mmap-1 test.txt
$ cat test.txt

NBVCXW
MLKJHGFDQS
POIUYTREZA
$ ./exemple-mmap-1 test.txt
$ cat test.txt

AZERTYUIOP
QSDFGHJKLM
WXCVCBN
$
```

Le comportement est identique avec le script Python suivant :

```
exemple-mmap-1.py :
#!/usr/bin/python
from __future__ import print_function
import mmap
import os
import sys

if len(sys.argv) < 2:
    print("usage:", sys.argv[0], "file")
    sys.exit(1)

try:
    stat = os.stat(sys.argv[1])
    size = stat.st_size;
    fd = os.open(sys.argv[1], os.O_RDWR)
    mapping = mmap.mmap(fd, size, mmap.MAP_SHARED,
                        mmap.PROT_READ | mmap.PROT_WRITE)
    for i in range(0, size / 2):
        tmp = mapping[i]
        mapping[i] = mapping[size - i - 1]
        mapping[size - i - 1] = tmp
```



```
except:
    print("Wrong file :", sys.argv[1])
    sys.exit(1)
```

À présent, nous allons vérifier que, comme nous l'annoncions plus tôt, il est possible de projeter en mémoire un fichier plus gros que la mémoire physique disponible pour notre processus si on utilise bien une projection `SHARED`, ce qui est le cas dans notre exemple. Pour cela, nous avons besoin d'un gros fichier de test. Nous allons donc écrire un petit programme créant un fichier contenant le nombre de méga-octets qu'on demande en ligne de commande. Chaque bloc d'un mégaoctet sera rempli avec une valeur différente pour vérifier que le retournement se passe bien.

```
cree-gros-fichier.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    int i;
    int fd;
    char * bloc;
    int nb_mega;

    if ((argc != 3) || (sscanf(argv[2], "%d", & nb_mega) != 1)) {
        fprintf(stderr, "Syntaxe : %s fichier nb_mega\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, 0644)) < 0) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    if ((bloc = malloc(1024*1024)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    for (i = 0 ; i < nb_mega; i++) {
        memset(bloc, i % 255, 1024*1024);
        if (write(fd, bloc, 1024*1024) < 0) {
            perror("write");
            exit(EXIT_FAILURE);
        }
    }
    close(fd);
    return EXIT_SUCCESS;
}
```

Pour observer le contenu du fichier avant et après retournement, on utilisera l'utilitaire standard `hexdump`. Lorsqu'aucune option n'est fournie, hormis le nom du fichier, `hexdump` présente en hexadécimal le contenu du fichier par lignes de huit entiers de deux octets précédées de l'*offset* à l'intérieur du fichier. Les lignes identiques ne sont pas répétées, mais remplacées par une étoile.

La mémoire totale de notre système fait 2 Go, c'est donc la valeur que nous choisirons pour notre gros fichier.

Ces programmes sont assez gourmands en ressources système, il vaut donc mieux éviter de les faire fonctionner sur une machine ayant plusieurs utilisateurs, ou alors il faut les lancer avec la commande `nice`.

```
$ ./cree-gros-fichier test.bin 2048
$ ls -l test.bin
-rw-r--r-- 1 cpb cpb 2147483647 26 juil. 06:42 test.bin
$
$ hexdump test.bin
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0100000 0101 0101 0101 0101 0101 0101 0101 0101
*
0200000 0202 0202 0202 0202 0202 0202 0202 0202
*
0300000 0303 0303 0303 0303 0303 0303 0303 0303
*
0400000 0404 0404 0404 0404 0404 0404 0404 0404
*
0500000 0505 0505 0505 0505 0505 0505 0505 0505
*
0600000 0606 0606 0606 0606 0606 0606 0606 0606
*
0700000 0707 0707 0707 0707 0707 0707 0707 0707
[...]
7fa00000 0202 0202 0202 0202 0202 0202 0202 0202
*
7fb00000 0303 0303 0303 0303 0303 0303 0303 0303
*
7fc00000 0404 0404 0404 0404 0404 0404 0404 0404
*
7fd00000 0505 0505 0505 0505 0505 0505 0505 0505
*
7fe00000 0606 0606 0606 0606 0606 0606 0606 0606
*
```

```
7ff00000 0707 0707 0707 0707 0707 0707 0707 0707
*
7fffffff 0707 0707 0707 0707 0707 0707 0707 0007
7fffffff
$ ./exemple-mmap-1 test.bin
$ hexdump test.bin
00000000 0707 0707 0707 0707 0707 0707 0707 0707
*
00ffff00 0707 0707 0707 0707 0707 0707 0707 0607
01000000 0606 0606 0606 0606 0606 0606 0606 0606
*
01ffff00 0606 0606 0606 0606 0606 0606 0606 0506
02000000 0505 0505 0505 0505 0505 0505 0505 0505
[...]
7faffff0 0505 0505 0505 0505 0505 0505 0505 0405
7fb00000 0404 0404 0404 0404 0404 0404 0404 0404
*
7fbffff0 0404 0404 0404 0404 0404 0404 0404 0304
7fc00000 0303 0303 0303 0303 0303 0303 0303 0303
*
7fcffff0 0303 0303 0303 0303 0303 0303 0303 0203
7fd00000 0202 0202 0202 0202 0202 0202 0202 0202
*
7fdffff0 0202 0202 0202 0202 0202 0202 0202 0102
7fe00000 0101 0101 0101 0101 0101 0101 0101 0101
*
7feffff0 0101 0101 0101 0101 0101 0101 0101 0001
7ff00000 0000 0000 0000 0000 0000 0000 0000 0000
*
7fffffff 0000 0000 0000 0000 0000 0000 0000 0000
7fffffff
$
```

Bien sûr, la vérification du fonctionnement de notre programme est un peu sommaire car le contenu de chaque bloc d'un méga-octet est constant, mais nous avons bien manipulé, directement dans notre espace d'adressage, un fichier dont la taille dépasse largement la mémoire que le système nous offre. L'intérêt de ce genre de projection apparaît plus clairement dans les domaines du traitement d'images ou des échantillons de son numérique, qu'on manipule ainsi comme des blocs de mémoire même si le fichier sous-jacent dépasse, de loin, la mémoire virtuelle du système.

Notre second exemple d'utilisation de `mmap()` va consister à utiliser la projection partagée d'un fichier temporaire entre un processus et son fils. Nous vérifierons ainsi que nous disposons bien d'une zone mémoire partagée entre les deux processus, de la

taille (au moins) d'un entier. Nous allons demander au système de nous donner le nom d'un fichier temporaire, puis nous allons le créer et y écrire une variable de la taille désirée. Ensuite, nous projetterons ce fichier en mémoire, et nous utiliserons l'adresse résultante comme pointeur sur une variable entière.

Les deux processus peuvent alors se séparer. Rappelons que les projections en mémoire sont héritées lors d'un `fork()`. Le processus père va incrémenter cette variable de 0 à 9, en envoyant après chaque mise à jour un signal `SIGUSR1` à son fils, et en dormant pendant une seconde (méthode peu élégante mais simple pour attendre que le fils ait affiché son résultat). Le processus fils ne fait qu'écrire à l'écran le contenu de la variable entière au sein de son gestionnaire de signal.

```
exemple_mmap_2.c :
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/mman.h>

int * entier;

void gestionnaire_sigusr1 (int num)
{
    fprintf(stdout, "Fils : * entier = %d\n", * entier);
}

int main (void)
{
    char * nom_fichier;
    int    fichier;
    pid_t pid;

    if (signal(SIGUSR1, gestionnaire_sigusr1) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    if ((nom_fichier = tmpnam(NULL)) == NULL) {
        perror("tmpnam");
        exit(EXIT_FAILURE);
    }
    fichier = open(nom_fichier, O_RDWR|O_CREAT|O_TRUNC, 0644);
    if (fichier < 0) {
        perror(nom_fichier);
        exit(EXIT_FAILURE);
    }
}
```

```
if (write(fichier, & fichier, sizeof (int)) != sizeof (int)) {
    perror("write");
    exit(EXIT_FAILURE);
}
entier = (int *) mmap(NULL, sizeof (int),
                      PROT_READ | PROT_WRITE, MAP_SHARED,
                      fichier, 0);
if (entier == (int *) MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}
close(fichier);
unlink(nom_fichier);

if ((pid = fork()) < 0) {
    perror("fork");
    exit(EXIT_FAILURE);
}
if (pid == 0) {
    while (1)
        sleep(1);
} else {
    for ((*entier) = 0; (*entier) < 10; (*entier)++) {
        fprintf(stdout, "Père : *entier = %d\n", *entier);
        kill(pid, SIGUSR1);
        sleep(1);
    }
    /* Ne pas oublier de tuer le fils qui est en attente */
    kill(pid, SIGKILL);
}
return EXIT_SUCCESS;
}
```

On notera qu'on détruit avec `unlink()` le fichier temporaire après l'avoir refermé. Le noyau ne l'effacera toutefois complètement que lorsque sa dernière référence sera refermée, en l'occurrence lorsque la fin des processus libérera la zone de projection en mémoire.

L'éditeur des liens nous signale que `tmpnam()` est dangereuse et devrait être remplacée par `mkstemp()`. C'est vrai, mais nous sommes ici dans un contexte expérimental simple, et `tmpnam()` est plus facile à utiliser que `mkstemp()`.

```
$ ./exemple-mmap-2
Père : *entier = 0
Fils : *entier = 0
Père : *entier = 1
Fils : *entier = 1
```

```
Père : * entier = 2
Fils : * entier = 2
Père : * entier = 3
Fils : * entier = 3
Père : * entier = 4
Fils : * entier = 4
Père : * entier = 5
Fils : * entier = 5
Père : * entier = 6
Fils : * entier = 6
Père : * entier = 7
Fils : * entier = 7
Père : * entier = 8
Fils : * entier = 8
Père : * entier = 9
Fils : * entier = 9
$
```

Nous voyons ainsi une méthode assez simple et amusante de partager de la mémoire entre deux processus père et fils. Bien entendu, nous observerons une autre technique, plus conventionnelle, dans le chapitre consacré aux communications entre processus.

Dans notre dernier exemple, nous ne nous sommes pas souciés du contenu effectif du fichier projeté. La seule chose qui nous intéressait était que la zone mémoire de la projection soit partagée avec le processus fils. Parfois au contraire, les modifications apportées au fichier doivent être visibles de l'extérieur, que ce soit pour la lecture directe du fichier ou pour d'autres programmes qui en effectuent une projection dans leur propre mémoire.

Par défaut, ce n'est que lorsqu'une projection SHARED est supprimée avec `munmap()` ou à la fin du processus, que le noyau nous garantit que les modifications apportées à la zone de projection seront répercutées sur le fichier. Si nous désirons nous en assurer à un autre moment, l'appel système `msync()` fournit plusieurs possibilités. Son prototype est le suivant :

```
| int msync (const void * debut, size_t longueur, int attribut);
```

Lorsqu'on invoque `msync()`, on lui transmet le pointeur sur la zone de projection qu'on désire mettre à jour, ainsi que la longueur de la zone. Naturellement, il est conseillé de ne demander que la mise à jour des portions effectivement modifiées de la projection, et pas nécessairement tout le fichier. L'attribut fourni en troisième argument peut contenir les constantes symboliques suivantes, liées par un OU binaire.

- `MS_ASYNC` : cette option demande au noyau de se préparer à mettre à jour les zones indiquées. Néanmoins, l'écriture n'a pas lieu tout de suite, l'appel système revenant immédiatement.

- **MS_SYNC** : avec cet attribut, le noyau effectue tout de suite la mise à jour. Lorsque l'appel système revient, nous savons qu'une lecture directe du fichier concerné nous renverrait les informations mises à jour. Toutefois, rien ne nous assure que les données aient été réellement écrites sur le disque, celui-ci peut gérer un cache plus ou moins important et retarder les écritures effectives.
- **MS_INVALIDATE** : qu'on utilise une mise à jour synchrone ou asynchrone, le noyau nous assure uniquement qu'une lecture directe du fichier renverra nos données mises à jour. Malgré tout, d'autres processus, totalement indépendants du nôtre, peuvent avoir effectué une projection du même fichier dans leur espace mémoire. Cet attribut garantit que leurs pages seront invalidées et que le noyau les reprendra sur le disque lors du prochain accès aux données.

On ne doit évidemment pas utiliser les options **MS_ASYNC** et **MS_SYNC** simultanément.

Il existe également sous Linux un appel système supplémentaire. Il n'est disponible qu'en utilisant l'option **_GNU_SOURCE** lors de la compilation. Il s'agit de **mremap()**, qui permet à la manière de **realloc()** d'agrandir ou de rétrécir une zone de projection en mémoire. Son prototype est le suivant :

```
void * mremap (void * zone, size_t ancienne_longueur,  
              size_t nouvelle_longueur, int attribut);
```

On transmet en argument un pointeur sur la zone de projection en cours, ainsi que sa longueur, suivi de la nouvelle longueur désirée. L'attribut peut éventuellement prendre comme unique valeur celle de la constante symbolique **MREMAP_MAYMOVE**, auquel cas **mremap()** sera autorisé à déplacer la zone de projection dans l'espace d'adressage. L'appel système renvoie un pointeur sur la nouvelle zone, ou **MAP_FAILED** en cas d'échec. Bien entendu, les relations avec le fichier sous-jacent sont conservées. Si on agrandit une zone de projection, il est possible d'accéder à une plus grande partie du fichier.

Avant de conclure ce chapitre sur la gestion de la mémoire, nous allons nous intéresser au principe de la protection des pages mémoire, sujet que nous avons effleuré sans entrer dans les détails en présentant le troisième argument d'invocation de **mmap()**.

Protection de l'accès à la mémoire

L'appel système **mprotect()** permet de limiter les possibilités d'accès à certaines pages mémoire. Son prototype est le suivant :

```
int mprotect (const void * debut_zone, size_t longueur, int protection);
```

La zone de mémoire à protéger doit être alignée sur une frontière de page. Nous détaillerons tout ceci plus bas.

La protection qu'on réclame en troisième argument est du même genre que celle de l'appel `mmap()`, avec une composition par OU binaire des constantes suivantes.

Constante	Signification
<code>PROT_NONE</code>	Aucune autorisation d'accès.
<code>PROT_READ</code>	Autorisation de lire dans la zone.
<code>PROT_WRITE</code>	Autorisation d'écrire dans la zone mémoire.
<code>PROT_EXEC</code>	Possibilité d'y exécuter du code.

Lorsque l'appel `mprotect()` réussit, il renvoie 0 et remplace complètement les protections originales de la zone mémoire, sinon il renvoie -1.

L'autorisation `PROT_EXEC` ne concerne normalement pas le programmeur applicatif. De toute manière, sur certaines architectures x86 par exemple, `PROT_EXEC` et `PROT_READ` ont exactement le même effet. Lorsque `PROT_EXEC` est réellement implémenté par le processeur, cela permet de déjouer une partie des attaques de sécurité en empêchant formellement d'exécuter des instructions en dehors du segment de code initialisé au chargement du programme. Or de nombreux piratages se basent sur le débordement de chaînes de caractères locales pour placer des instructions personnelles dans la pile des utilitaires système *Set-UID*.

Le fait d'interdire tout accès avec l'autorisation vierge `PROT_NONE` peut servir au débogage d'un programme pour contrôler tous les accès d'une application à une zone de mémoire allouée dynamiquement. Par exemple, on peut placer l'autorisation `PROT_NONE` dès l'allocation, et ne permettre la lecture qu'à partir du moment où l'initialisation a lieu. Si un autre module tente d'utiliser la variable avant la fin de l'initialisation, le processus sera tué par un signal, et nous pourrions employer `gdb` et le fichier `core` pour remonter jusqu'à l'utilisation fautive.

Sur certains processeurs, le fait de demander une autorisation d'écriture `PROT_WRITE` entraîne également la disponibilité en lecture (et donc en exécution), mais cela n'est pas nécessairement portable sur les autres architectures.

Nous avons précisé que l'adresse de début de zone doit être alignée sur une frontière de page. C'est automatiquement le cas avec les zones mémoire allouées par `mmap()` ; aussi est-ce la manière la plus simple d'allouer les zones qu'on protégera ultérieurement. La fonction `malloc()` utilise en interne l'appel système `mmap()` dans certaines conditions, mais on ne peut pas en être certain. S'il y a suffisamment de place libre dans le segment de données qui n'a pas été rendu au système, elle est employée avant toute chose.

Pour assurer la portabilité de notre application, on se conformera donc au standard SUSv4 en allouant les zones mémoire avec `mmap()` en projection `ANONYMOUS`. Il faudra simplement se méfier de la différence entre `malloc()`, qui renvoie `NULL` en cas d'échec, et `mmap()`, qui renvoie `MAP_FAILED` (qui vaut normalement `-1`). Pour éviter toute ambiguïté, on pourra se redéfinir une fonction d'allocation semblable à `malloc()`. La seule contrainte est de conserver la taille de la zone allouée car on doit la transmettre à `mprotect()`.

Lorsqu'un processus tente d'accéder de manière illégale à une zone de mémoire protégée, il est tué par le signal `SIGSEGV`. En voici une illustration :

```
exemple_mprotect_1.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

#define TAILLE_CHAINE 128

void * mon_malloc_avec_mmap (size_t taille)
{
    void * retour;
    retour = mmap(NULL, taille, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
    if (retour == MAP_FAILED)
        return NULL;
    return retour;
}

int main (void)
{
    char * chaine = NULL;

    fprintf(stdout, "Allocation de %d octets\n", TAILLE_CHAINE);
    chaine = mon_malloc_avec_mmap(TAILLE_CHAINE);
    if (chaine == NULL) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Protections par default\n");
    fprintf(stdout, "    Ecriture ...");
    strcpy(chaine, "Ok");
    fprintf(stdout, "Ok\n");
    fprintf(stdout, "    Lecture ...");
    fprintf(stdout, "%s\n", chaine);

    fprintf(stdout, "Interdiction d'écriture\n");
    if (mprotect(chaine, TAILLE_CHAINE, PROT_READ) < 0) {
        perror("mprotect");
    }
}
```

```

    exit(EXIT_FAILURE);
}

fprintf(stdout, "  Lecture ...");
fprintf(stdout, "%s\n", chaine);
fprintf(stdout, "  Ecriture ...");
strcpy(chaine, "Non");
/* Ici on doit déjà être arrêté par un signal */
return EXIT_SUCCESS;
}

```

Nous exécutons le programme, puis nous invoquons gdb pour rechercher d'où vient l'erreur :

```

$ ulimit -c unlimited
$ ./exemple-mprotect-1
Allocation de 128 octets
Protections par défaut
  Ecriture ...Ok
  Lecture ...Ok
Interdiction d'écriture
  Lecture ...Ok
Erreur de segmentation (core dumped)
$ gdb exemple-mprotect-1 core.29176
GNU gdb (GDB) Fedora (7.1-28.fc13)
[...]
Core was generated by `./exemple-mprotect-1'.
Program terminated with signal 11, Segmentation fault.
#0 0x001fd0f1 in __memcpy_ia32 () from /lib/libc.so.6
(gdb) bt
#0 0x001fd0f1 in __memcpy_ia32 () from /lib/libc.so.6
#1 0x08048776 in main () at exemple-mprotect-1.c:45
(gdb) quit
$

```

Le processus est bien tué par un signal SIGSEGV, avec création d'une image disque core. L'invocation de gdb nous apprend que l'erreur s'est produite dans la routine `__memcpy()`, et nous utilisons la commande `bt` (pour *backtrace*, suivi en arrière) qui nous permet de voir que cette fonction a été invoquée en ligne 45 du fichier `exemple-mprotect-1.c`.

Nous observons que le début de la ligne « *ecriture...* » n'a pas été transmise sur `stdout`, car le buffer n'a pas été vidé par `fflush()` ni par un retour à la ligne avant l'arrêt brutal du programme.

Le comportement du processeur face à un accès illégal à la mémoire peut paraître déroutant de prime abord. En effet, lorsqu'une faute d'accès est détectée, le noyau est prévenu. Il envoie alors un signal au processus. En revanche, le compteur d'instruction du programme n'est pas incrémenté. Cela signifie que si le processus n'est pas tué par le signal, au retour du gestionnaire de SIGSEGV, il retentera la même opération d'accès interdite. En voici une démonstration : nous avons repris le principe de l'exemple précédent, en rajoutant un gestionnaire pour SIGSEGV, qui est mis en place en début de fonction `main()` :

```
example-mprotect-2.c :
[...]
```

```
void gestionnaire_sigsegu(int numero)
{
    fprintf(stderr, "Signal SIGSEGV recu\n");
}

int main (void)
{
    char * chaine = NULL;
    if (signal(SIGSEGV, gestionnaire_sigsegu) == SIG_ERR) {
        perror("signal");
        exit(EXIT_FAILURE);
    }
    [...]
```

Cette fois nous nous limitons à une tentative de lecture de lecture sur une protection NONE.

```
$ ./exemple-mprotect-2  
Allocation de 128 octets  
Protections par défaut  
Ecriture ...Ok  
Interdiction de lecture  
Lecture ...  
  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
Signal SIGSEGV reçu  
[...]  
  
(Contrôle-C)  
  
$
```

Nous sommes obligés d'arrêter avec *Contrôle-C* le processus qui part autrement dans un cycle infini d'interception de signal et de tentative de lecture sur une zone verrouillée. Un moyen de se libérer de ce problème serait d'utiliser un saut non local depuis le gestionnaire pour revenir à un emplacement plus sûr du programme. Malgré tout, le fait de détecter une erreur d'accès mémoire doit plutôt être considéré comme un dysfonctionnement à corriger immédiatement à l'aide du fichier d'image core et d'un débogueur. Le comportement par défaut de SIGSEGV est donc plus approprié.

Conclusion

Nous avons étudié longuement la gestion de la mémoire sous Linux. Cela nous permet – espérons-le – d'avoir une vision claire des mécanismes mis en œuvre lors des allocations, projections ou accès aux zones de la mémoire tant physique que virtuelle.

Le programmeur applicatif a rarement besoin de se soucier des techniques sous-jacentes aux allocations dynamiques de mémoire. Toutefois, une bonne compréhension de ces phénomènes permet de diagnostiquer plus facilement les problèmes lorsqu'un programme se comporte de manière a priori surprenante.

Pour étudier en détail l'espace mémoire d'un processus, les meilleures informations proviennent de l'étude directe des sources du noyau Linux. Toutefois, on trouvera des éléments intéressants dans [Bach 1989] *Conception du système Unix* ou [Love 2003] *Linux Kernel Development*.

Nous reviendrons dans les prochains chapitres sur la manipulation des blocs de mémoire ainsi obtenus, notamment en ce qui concerne les chaînes de caractères.

16

Utilisation des blocs mémoire et des chaînes

L'emploi des chaînes de caractères et des blocs de mémoire sous Linux n'a rien de particulièrement original par rapport aux autres systèmes d'exploitation. En revanche, la bibliothèque Glibc propose des fonctions très intéressantes, surtout en ce qui concerne les traitements de chaînes. Certaines de ces fonctions sont assez peu connues et recouvrent pourtant des besoins pour lesquels le programmeur est souvent amené à se créer sa propre bibliothèque « maison », alors que l'implémentation de la bibliothèque C est généralement mieux optimisée.

Les fonctions présentées ici se rapportent à la gestion de chaînes de caractères spécifiques au langage C, aussi nous ne ferons pas référence à Python.

Nous nous intéresserons tout d'abord aux différentes variantes des routines permettant de manipuler des blocs de mémoire bruts sans se préoccuper de leur contenu. Ensuite, nous verrons les routines de manipulation de chaînes de caractères, notamment celles qui sont utilisées pour mesurer la longueur d'une chaîne, remplir, copier ou comparer des chaînes. Puis, nous nous pencherons sur les fonctions permettant de faire des recherches plus ou moins complexes de sous-chaînes.

Manipulation de blocs de mémoire

Les fonctions essentielles pour manipuler les blocs de mémoire commencent par le préfixe `mem` et sont déclarées dans le fichier `<string.h>`. Sauf mention explicite, les routines présentées ici sont définies dans le standard C. Les extensions Gnu sont accessibles en définissant la constante de compilation `_GNU_SOURCE` avant l'inclusion des fichiers d'en-tête.

Il existe toutefois des routines devenues quasi obsolètes de nos jours, mais qu'on peut rencontrer dans d'anciens fichiers source. Nous les mentionnerons pour information, mais il faudra éviter de les utiliser à l'avenir. Avec la Glibc, ces routines sont déclarées dans le fichier `<strings.h>` (avec un « s »).

Les blocs de mémoire qu'on peut manipuler avec la bibliothèque C sont représentés par des pointeurs `void *`. Naturellement, nous déclarerons la plupart du temps nos zones de mémoire avec un autre type, et la conversion sera automatiquement assurée lors de l'invocation des routines. Rappelons que le type générique `void *` peut recevoir n'importe quel type de pointeur sur des données, sans déclencher d'avertissement du compilateur.

Lorsqu'on désire accéder au contenu d'un bloc de mémoire octet par octet, le plus simple est de le déclarer de type `unsigned char *`. Ainsi, nous pourrions adresser directement chaque octet et comparer aisément son contenu avec un entier compris entre 0 et 255. Voici comment manipuler un tel tableau¹ :

```
int i;
unsigned char * bloc;

if ((bloc = malloc(TAILLE_BLOC)) == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}
for (i = 0; i < TAILLE_BLOC; i++) {
    bloc[i] = i & 0xFF;
}
```

La première opération qu'on effectue sur un bloc de mémoire est bien souvent de l'initialiser en le remplissant avec une valeur, par ailleurs souvent nulle. La fonction `memset()` assure un tel remplissage :

```
void * memset (void * bloc, int valeur, size_t longueur);
```

1. Bien qu'il y ait une différence conceptuelle entre `char * chaine` et `char tableau[]`, cette distinction ne nous concernera pas ici. Pour plus de détails, on pourra se reporter à la FAQ [Usenetcomp.lang.c](#).

Le premier argument pointe sur le bloc à remplir. Le second est une valeur entière qui sera convertie en `unsigned char` et qui servira à remplir le nombre d'octets indiqué en troisième argument. `memset()` renvoie la valeur du pointeur bloc. L'accès à un bloc de mémoire n'appartenant pas au processus ou le passage d'un pointeur `NULL` peuvent déclencher le signal `SIGSEGV`.

L'emploi de l'ancienne fonction `bzero()` est déconseillé. Son prototype était :

```
| void bzero (void * bloc, size_t longueur);
```

Elle ne faisait que remplir le bloc avec des zéros. On peut la remplacer par :

```
| memset (bloc, 0, longueur).
```

Il est préférable d'utiliser le plus fréquemment possible la fonction `memset()` plutôt que d'essayer d'initialiser manuellement une zone, car cette routine est optimisée, en assurant par exemple sur des processeurs 32 bits des remplissages directement avec des `long int` pour diviser par 4 le nombre de boucles à effectuer.

La seconde opération la plus répandue sur les blocs de mémoire, après leur initialisation, est probablement la copie. La fonction `memcpy()` permet de copier des blocs de mémoire disjoints. Son prototype est :

```
| void * memcpy (void * destination, void * origine,  
                 size_t longueur);
```

Elle renvoie le pointeur sur la destination, après y avoir recopié la longueur désirée de la chaîne originale.

Attention, `memcpy()` ne peut travailler que sur des blocs de mémoire disjoints. Si les deux blocs risquent de se recouvrir, il faut utiliser la fonction `memmove()` décrite plus bas. La norme *Iso-C99* a d'ailleurs ajouté au langage C un mot-clé `restrict` destiné à indiquer que les chaînes ne doivent pas se chevaucher.

La fonction `memcpy()` est par exemple très utile pour recopier tous les champs d'une structure :

```
| void ma_fonction (struct ma_structure * originale)  
{  
    struct ma_structure copie_de_travail;  
    memcpy(& copie_de_travail, originale,  
          sizeof(struct ma_structure));  
    ...  
}
```

Il est du ressort du programmeur de s'assurer qu'il y a suffisamment de place pour recevoir la copie du bloc original.

Il existe une extension Gnu, `mempcpy()`, ayant le même prototype que `memcpy()`, mais renvoyant, à la place d'un pointeur sur le début de la zone destination, un pointeur sur l'octet immédiatement après le dernier octet écrit dans la zone destination. Cette adresse est donc à nouveau utilisable pour une copie. Cela permet notamment de concaténer des objets de tailles différentes (en vue d'une écriture groupée sur le disque par exemple) :

```
void * assemble_blocs (int nb_blocs, size_t taille_bloc[], void *
bloc[])
{
    void * retour;
    void * cible;
    int     taille = 0;
    int     i;

    for (i = 0; i < nb_blocs; i++)
        taille += taille_bloc[i];
    if ((retour = malloc(taille)) == NULL)
        return NULL;
    cible = retour;
    for (i = 0; i < nb_blocs; i++)
        cible = mempcpy(cible, bloc[i], taille_bloc[i]);
    return retour;
}
```

La fonction `memccpy()` permet d'effectuer une copie jusqu'à la longueur désirée, ou jusqu'à avoir rencontré un caractère donné dans le bloc original. Son prototype est :

```
void * memccpy (void * destination, void * source,
                int octet, size_t longueur);
```

Si l'octet d'arrêt est trouvé dans le bloc source, il est copié dans le bloc destination, et `memccpy()` renvoie un pointeur sur le caractère suivant dans le bloc cible. Si la longueur maximale est atteinte durant la copie sans avoir rencontré l'octet d'arrêt, `memccpy()` renvoie un pointeur NULL.

Vous avez probablement deviné que cette routine servira avec les chaînes de caractères, pour l'implémentation de `strncpy()`. Cette fonction est aussi assez utile lorsqu'on traite des blocs de données provenant, par exemple, de dispositifs industriels ou scientifiques qui utilisent un octet précis de synchronisation pour délimiter le début d'un nouvel ensemble de données de longueur variable. Cette fonction permet ainsi de lire un bloc jusqu'à l'ensemble suivant, tout en s'assurant de ne pas déborder de la mémoire tampon prévue pour le traitement.

Nous avons bien précisé que les zones de mémoire ne doivent pas se chevaucher lors de l'utilisation des fonctions de type `memcpy()`. Sinon le résultat est indéfini, car durant la copie, on risque de rencontrer des octets déjà placés dans le bloc destination. À titre d'exemple naïf, supposons qu'on ait la situation suivante :

Source				Destination											
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

On espère, après avoir effectué `memcpy(destination, source, 12)`, obtenir :

Source				Destination											
A	B	C	D	A	B	C	D	E	F	G	H	I	J	K	L

Malheureusement, une fois les quatre premiers octets copiés, la routine va lire les quatre suivants, qui coïncident avec la chaîne destination, et au lieu de trouver EFGH, on retrouve ABCD :

Source				Destination											
A	B	C	D	A	B	C	D	I	J	K	L	M	N	O	P

La situation se reproduit de nouveau quatre octets plus loin :

Source				Destination											
A	B	C	D	A	B	C	D	A	B	C	D	M	N	O	P

Ce qui nous conduit finalement à ce résultat, assez éloigné de ce que nous espérons :

Source				Destination											
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D

Il est pourtant souvent nécessaire de déplacer une partie d'un bloc de données vers un emplacement le recouvrant partiellement. À titre d'exemple, on peut implémenter ainsi un buffer linéaire, où les données déjà traitées doivent être écrasées par celles qui restent à l'autre bout de la mémoire tampon. La fonction `memmove()` est utilisée dans ce cas. Son prototype est :

```
void * memmove (void * destination, void * source,  
                size_t longueur);
```

Cette routine se comporte exactement comme `memcpy()` lorsque les blocs sont dis-joints, mais lorsqu'ils se recouvrent, elle assure que la copie obtenue finalement sera

exactement une image de la source originale au début de l'appel. Pour cela, elle compare les pointeurs source et destination, et détermine si elle doit commencer sa copie par le début (comme nous l'avons fait précédemment) ou par la fin du bloc (comme nous aurions dû le faire pour que cela fonctionne).

Le nom de cette fonction ne doit pas engendrer de confusion, il s'agit bien d'une copie, et pas d'un déplacement. Le bloc original n'est pas modifié s'il n'y a pas de recouvrement avec le bloc destination.

La fonction `memmove()` nécessite de la part de la bibliothèque C un léger surcroît de travail par rapport à `memcpy()`, puisqu'il faut qu'elle détermine le sens de progression de la copie, mais elle est quand même beaucoup plus sécurisante que cette dernière.

Il existe une fonction obsolète nommée `bcopy()` qui fonctionnait comme `memmove()`, mais avec le prototype suivant :

```
void bcopy (void * source, void * destination, size_t longueur);
```

Elle se comporte comme `memmove()` vis-à-vis des recouvrements de blocs, mais ne renvoie pas de pointeur. Pire, ses arguments sont inversés par rapport aux routines `memcpy()` et `memmove()`. Autrement dit, il ne faut plus l'utiliser !

Il est possible de comparer deux blocs de mémoire. La fonction `memcmp()` assure ce rôle avec le prototype suivant :

```
int memcmp (const void * bloc_1, const void * bloc_2,  
            size_t taille);
```

Elle renvoie 0 si les deux blocs sont égaux sur la taille indiquée, sinon elle renvoie -1 ou 1 suivant le signe de la soustraction entre les premiers octets différents entre les deux blocs. Cette différence est calculée après avoir converti les octets sous forme de `int`. Voyons un exemple des divers cas possibles :

```
exemple-memcmp.c :  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
void affiche_resultats (unsigned char * bloc_1, unsigned char * bloc_2,  
int lg)  
{  
    int i;  
    fprintf(stdout, "bloc_1 = ");  
    for (i = 0; i < lg ; i ++)  
        fprintf(stdout, "%02d ", bloc_1[i]);  
    fprintf(stdout, "\n");
```

```
fprintf(stdout, "bloc_2 = ");
for (i = 0; i < lg ; i ++){
    fprintf(stdout, "%02d ", bloc_2[i]);
    fprintf(stdout, "\n");
    fprintf(stdout, "memcmp(bloc_1, bloc_2, %d) = %d\n",
        lg, memcmp(bloc_1, bloc_2, lg));
    fprintf (stdout, "\n");
}

int main (void)
{
    unsigned char bloc_1[4] = { 0x01, 0x02, 0x03, 0x04 };
    unsigned char bloc_2[4] = { 0x01, 0x02, 0x08, 0x04 };
    unsigned char bloc_3[4] = { 0x01, 0x00, 0x03, 0x04 };

    affiche_resultats(bloc_1, bloc_1, 4);
    affiche_resultats(bloc_1, bloc_2, 4);
    affiche_resultats(bloc_1, bloc_3, 4);

    return EXIT_SUCCESS;
}
```

Attention, `memcmp()` renvoie le signe du résultat de la soustraction des premiers octets différents, pas la valeur même de la différence.

```
$ ./exemple-memcmp
bloc_1 = 01 02 03 04
bloc_2 = 01 02 03 04
memcmp(bloc_1, bloc_2, 4) = 0

bloc_1 = 01 02 03 04
bloc_2 = 01 02 08 04
memcmp(bloc_1, bloc_2, 4) = -1

bloc_1 = 01 02 03 04
bloc_2 = 01 00 03 04
memcmp(bloc_1, bloc_2, 4) = 1

$
```

Il faut être très prudent avec les comparaisons de blocs de mémoire. En effet, on aurait tendance, à tort, à utiliser cette routine pour comparer des structures par exemple. Mais le compilateur insère fréquemment des octets de remplissage dans les structures ou dans les unions pour optimiser l'alignement des divers champs. Ces octets de remplissage n'ont pas de valeurs précisément définies et peuvent varier entre

deux structures dont les membres sont par ailleurs égaux. On ne pourra donc pas utiliser `memcmp()` pour comparer autre chose que des données binaires « brutes » où chaque octet a une signification précise.

Comme toujours, il existe une version obsolète de cette routine provenant de BSD, `bcmp()`, qui est similaire à `memcmp()` :

```
| int bcmp (const void * bloc_1, const void * bloc_2, int taille)
```

Nous nous intéresserons aux routines permettant de rechercher des sous-blocs de données au sein d'une zone de mémoire dans la section sur les recherches au cœur d'une chaîne.

Mesures, copies et comparaisons de chaînes

Nous allons dans un premier temps nous intéresser aux chaînes composées uniquement de caractères de la table Ascii standard. Cela élimine entre autres les caractères accentués francophones « é, ô, à, ï, ç » etc. que nous étudierons dans un second temps.

Avec la bibliothèque C standard, les chaînes sont représentées par une table de caractères terminée par un caractère nul permettant d'en marquer la fin. Lorsqu'on déclare une chaîne ainsi

```
| char * chaine = "Seize caracteres";
```

le compilateur crée une zone de données statique initialisée, avec dix-sept caractères:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S	e	i	z	e		c	a	r	a	c	t	e	r	e	s	\0

La fonction `strlen()` renvoie la longueur d'une chaîne, sans compter le caractère '\0' final. Cette fonction est déclarée dans `<string.h>` ainsi :

```
| size_t strlen (const char * chaine);
```

Pour poursuivre notre exemple, `strlen("Seize caracteres")` renvoie 16 puisque le caractère nul de fin n'est pas compté. Comme les tableaux sont accessibles en C à partir de l'élément d'indice 0, on retrouve toujours `chaine[strlen(chaine)] = '\0'`.

Lorsqu'on alloue dynamiquement la mémoire pour une chaîne, on la stocke dans un pointeur de type `char *`. Il importe à ce moment de ne pas oublier la place nécessaire pour le caractère nul final. La fonction `strcpy()` permet de copier une chaîne dans

une autre. Il faut avoir alloué suffisamment de place dans la chaîne réceptrice. Le prototype de `strcpy()` est le suivant :

```
| char * strcpy (char * destination, const char * origine);
```

La bonne méthode pour allouer la mémoire indispensable à la réception d'une copie d'une chaîne est la suivante :

```
| char * nouvelle;  
| if ((nouvelle = malloc(strlen(originale) + 1)) != NULL)  
|     strcpy(nouvelle, originale);  
| else  
|     perror("malloc");
```

Même une fonction aussi simple que `strlen()` peut parfois poser des problèmes. En effet, l'implémentation d'une telle routine nécessite de balayer toute la chaîne jusqu'à rencontrer un caractère nul, puis de renvoyer le nombre de caractères parcourus. La véritable implémentation est optimisée en assembleur dans la bibliothèque Glibc, mais on peut quand même en donner un équivalent fourni par Kernighan et Ritchie :

```
| size_t strlen (const char * s)  
| {  
|     char * p = s;  
|     while (* p != '\0')  
|         p++;  
|     return p - s;  
| }
```

En fait, on renvoie ici la différence arithmétique entre le pointeur sur le caractère nul et le pointeur initial. Un problème grave peut se poser si on ne trouve pas de caractère nul. Imaginons le traitement de chaînes relativement grandes provenant d'un fichier de texte par exemple. Malheureusement, l'utilisateur s'est trompé lorsqu'il nous a fourni le nom du fichier de texte à lire et il nous a transmis un fichier constitué de données binaires, une image graphique par exemple, ne contenant – comble de malheur – aucun zéro. Que se passe-t-il alors ? La fonction `strlen()` va parcourir toute la zone de mémoire à la recherche d'un zéro et, n'en trouvant pas, va déborder sur la suite de la mémoire. Il est possible que la page suivante ne soit pas attribuée, et le programme va recevoir subitement un signal `SIGSEGV` qui va le tuer.

On peut avancer qu'il suffit, avant d'appeler `strlen()`, d'écrire de force un caractère nul à une distance arbitraire, suffisamment grande pour correspondre à la plus grande chaîne qu'on puisse traiter. Dans la plupart des cas, c'est effectivement suffisant, mais nous n'avons pas toujours un accès en écriture sur la page mémoire de la chaîne à lire, la projection d'un fichier par exemple peut avoir uniquement l'autorisation `PROT_READ`.

Par ailleurs, la chaîne dont il est question peut aussi être un argument d'entrée d'une routine déclarée sous forme de `const char *`, donc non modifiable (même si le compilateur ne fournit qu'un avertissement et pas une erreur). La chaîne peut aussi être une constante statique dans un segment de données protégé en écriture. Par exemple, le code suivant déclenche une erreur de segmentation SIGSEGV :

```
void modifie_chaine (char * chaine)
{
    chaine[0] = '\0';
}

int main (void)
{
    modifie_chaine("abc");
    return EXIT_SUCCESS;
}
```

Pour éviter de déborder d'une chaîne en recherchant sa longueur, la bibliothèque Glibc offre une fonction `strnlen()` qui limite la portée de la recherche de la fin de chaîne. Elle prend tout simplement un argument supplémentaire pour indiquer la longueur maximale :

```
size_t strnlen (const char chaine, size_t longueur_maxi);
```

Dans la documentation Gnu, cette fonction est indiquée comme étant équivalente à

```
strlen(chaine) < longueur_maxi ? strlen(chaine) : longueur_maxi
```

mais elle n'est heureusement pas implémentée comme cela. Tout d'abord, il faudrait stocker dans une variable le retour de `strlen()` et ne pas la rappeler deux fois. Mais de surcroît, cette fonction n'arrangerait en rien notre problème si nous attendions le retour de `strlen()` pour limiter sa valeur. En réalité, `strnlen()` est implémentée en utilisant `memchr()`, que nous verrons dans la prochaine section, et qui recherche la première occurrence du caractère nul jusqu'à une certaine distance limite.

Pour savoir si on a atteint ou non la longueur maximale, il suffit d'examiner le dernier caractère qui doit normalement être nul :

```
taille = strnlen(chaine, TAILLE_MAXI_SEGMENT);
if (chaine[taille] != '\0') {
    /* Prévenir l'utilisateur et recommencer la saisie */
    fprintf(stderr, "La chaine fournie est trop longue \n");
    return ERREUR;
}
```

Lorsqu'on désire obtenir une copie d'une chaîne de caractères, il n'existe pas moins de huit variantes possibles dans la Glibc. La fonction la plus courante est bien entendu `strcpy()` déclarée ainsi :

```
| char * strcpy (char * destination, const char * source);
```

Elle copie tous les caractères contenus de la chaîne source, y compris le 0 final, dans la chaîne destination, et renvoie un pointeur sur cette dernière. Aucune protection n'est fournie en ce qui concerne les risques de débordement de la chaîne source. Pour cela, il faut utiliser `strncpy()` :

```
| char * strncpy (char * destination, const char * source,  
|                 size_t taille_maxi);
```

Le comportement est le suivant :

- Si la chaîne source est plus courte que la taille maximale indiquée, elle est copiée dans la chaîne destination, puis l'espace restant de la chaîne destination est rempli avec des caractères nuls jusqu'à la taille maximale. Cela sert lorsqu'on veut comparer des zones mémoire complètes, l'état des caractères inutilisés étant fixé.
- Si la chaîne source est plus longue que la taille maximale indiquée, on ne copie que cette dernière longueur. Aucun caractère nul n'est ajouté dans la chaîne destination. Nous avons vu que cela peut être une situation à risque pour l'emploi ultérieur de `strlen()`.

Par exemple, `strncpy(destination, "ABCDEFGH", 12)` remplit la chaîne de destination ainsi :

```
A      B      C      D      E      F      G      H      \0      \0      \0      \0
```

Alors que `strncpy(destination, "ABCDEFGH", 5)` remplit la chaîne de destination ainsi

```
A      B      C      D      E
```

sans qu'il y ait de caractère nul ajouté.

Le programmeur prudent pourra donc utiliser des routines du genre :

```
| char * destination;  
| size_t longueur;  
  
| if ((destination = malloc(LONGUEUR_MAXI_CHAINES + 1)) == NULL) {  
|     /* Traitement d'erreur */  
|     [...]
```

```

    }
    destination[LONGUEUR_MAXI_CHAINES] = '\0';
    strncpy(destination, source, LONGUEUR_MAXI_CHAINES);
    [...]
    longueur = strlen(destination, LONGUEUR_MAXI_CHAINES);
    if (longueur == LONGUEUR_MAXI_CHAINES) {
        /* Traitement d'erreur*/
        [...]
    }

```

Les fonctions `stpcpy()` et `stpncpy()` ont exactement la même syntaxe et la même signification que `strcpy()` et `strncpy()`, mais elles renvoient un pointeur différent :

- `stpcpy()` renvoie un pointeur sur le caractère nul final de la chaîne destination.
- `stpncpy()` renvoie un pointeur sur le caractère situé dans la chaîne destination, immédiatement après le dernier caractère écrit, si la chaîne source est plus longue que la taille maximale indiquée.
- `stpncpy()` renvoie un pointeur sur le premier caractère nul écrit dans la chaîne destination si la chaîne source est plus courte que la taille maximale. La chaîne destination est dans ce cas complétée avec des zéros jusqu'à la longueur maximale, mais on renvoie un pointeur sur le premier caractère nul ajouté.

Ces fonctions sont disponibles dans la Glibc en tant qu'extensions Gnu, même s'il s'agit probablement de routines provenant du monde Dos. En renvoyant un pointeur sur la fin de la chaîne, elles permettent de faire des concaténations successives. Nous allons créer une fonction prenant en argument une chaîne destination, une longueur maximale, suivies d'un nombre quelconque de chaînes et d'un pointeur `NULL` final. Cette routine va concaténer toutes les chaînes transmises dans la chaîne destination, en surveillant qu'il n'y ait pas de débordement, caractère nul final compris.

```

exemple_stpncpy.c :
#define _GNU_SOURCE

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void concatenation (char * destination, size_t taille_maxi, ...)
{
    va_list arguments;
    char * source;
    char * retour;
    size_t taille_chaine;

```



```
    retour = destination;
    taille_chaine = 0;

    va_start(arguments, taille_maxi);

    while (1) {
        source = va_arg(arguments, char *);
        if (source == NULL)
            /* Fin des arguments */
            break;
        retour = stpncpy(retour, source, taille_maxi - taille_chaine);
        taille_chaine = retour - destination;
        if (taille_chaine == taille_maxi) {
            /* Écraser le dernier caractère par un zéro */
            retour--;
            *retour = '\0';
            break;
        }
    }
    va_end(arguments);
}

int main (void)
{
    char chaine[20];

    concatenation(chaine, 20, "123", "456", "7890", "1234", NULL);
    fprintf(stdout, "%s\n", chaine);

    concatenation(chaine, 20, "1234567890", "1234567890", "123", NULL);
    fprintf(stdout, "%s\n", chaine);

    return EXIT_SUCCESS;
}
```

L'exécution nous permet de vérifier que notre concaténation fonctionne bien, tout en ne dépassant jamais la longueur maximale indiquée :

```
$ ./exemple-stpncpy
12345678901234
1234567890123456789
$
```

Il existe deux fonctions, `strdup()` et `strndup()`, particulièrement pratiques, car elles assurent l'allocation mémoire nécessaire pour stocker la chaîne destination. Elles sont déclarées ainsi :

```
char * strdup (const char *chaîne);  
char * strndup (const char * chaîne, size_t longueur);
```

Elles renvoient toutes deux un pointeur sur la copie nouvellement allouée de la chaîne, ou `NULL` en cas d'échec dans `malloc()`. La fonction `strndup()` ne copie au plus que la longueur indiquée, y compris le caractère nul final. La chaîne renvoyée se termine donc toujours par un zéro. Bien entendu, il faut libérer les chaînes renvoyées en invoquant `free()` une fois qu'on a fini de les utiliser.

Deux fonctions supplémentaires existent en tant qu'extensions Gnu : `strdupa()` et `strndupa()`. Elles se présentent exactement comme `strdup()` et `strndup()`, mais la copie de chaîne est allouée dans la pile en utilisant la fonction `alloca()`, et non `malloc()`. Il ne faut donc pas tenter d'appeler `free()` avec le pointeur renvoyé, car l'espace occupé par la chaîne sera automatiquement libéré au retour de la fonction (ou lors d'un saut non local `longjmp`). Pour ces deux fonctions, on prendra donc les précautions qui s'imposent vis-à-vis de l'emploi de variables allouées dynamiquement dans la pile avec `alloca()`, comme nous l'avons vu dans le chapitre traitant de la gestion de l'espace mémoire du processus.

Aucune des fonctions de copie que nous avons examinées ne permet de copier des chaînes se recouvrant partiellement. Il est pourtant utile de déplacer des parties d'une chaîne à l'intérieur d'elle-même. Cela permet, par exemple, d'éliminer les espaces en début de ligne. La seule fonction acceptable pour cela est `memmove()`, mais elle nous oblige à rechercher nous-même la fin de la chaîne. Nous verrons comment implémenter de manière assez performante une élimination des blancs en début et fin de chaîne, dans la prochaine section, car nous utiliserons les fonctions `strchr()` et `strspn()` que nous analyserons alors.

Il est également fréquent d'avoir besoin d'ajouter une portion de chaîne à la fin d'une autre. Par exemple, on prépare phrase par phrase un texte en fonction de divers paramètres, puis le texte est affiché ou transmis à une routine de sauvegarde, de présentation dans un composant d'interface graphique, etc. Plusieurs méthodes sont possibles pour concaténer des chaînes, à commencer par `strcpy()` qui utilise un pointeur sur la fin de la chaîne destination. Nous avons également écrit une routine de ce type dans l'exemple précédent, avec `stpncpy()`. Il est toujours envisageable d'employer `sprintf()`. La fonction dont on se sert le plus couramment est pourtant `strcat()`, ainsi que son acolyte `strncat()` qui permet par précaution de limiter la longueur de la chaîne réceptrice. Leurs prototypes sont déclarés ainsi :

```
char * strcat (char * destination, const char * a_ajouter);  
char * strncat (char * destination, const char * a_ajouter, size_t  
taille);
```

La taille indiquée dans l'appel de `strncat()` est celle de la portion qui peut être ajoutée à la chaîne destination. Avant l'appel de `strncat()`, la chaîne destination doit donc disposer d'une taille totale valant au moins `strlen(destination)+taille+1` (pour le caractère nul final). L'exemple suivant va nous permettre d'utiliser `strncat()` pour concaténer les arguments d'appel de la fonction, tout en limitant la taille totale. La valeur 20 est choisie arbitrairement pour imposer une limite volontairement basse.

```
exemple-strncat.c :  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
#define LG_MAXI 32      /* 20 + 12, cf. plus bas */  
  
int main (int argc, char * argv[])  
{  
    int i;  
    int taille;  
    char chaine[LG_MAXI + 1];  
  
    strcpy(chaine, "Arguments : "); /* Déjà 12 caractères */  
  
    for (i = 1; i < argc; i++) {  
        taille = strlen(chaine);  
        strncat(chaine, argv[i], LG_MAXI-taille);  
    }  
    fprintf(stdout, "%s\n", chaine);  
    return EXIT_SUCCESS;  
}
```

Lors de l'exécution, la chaîne est bien limitée à 32 caractères (20 pour les arguments, et 12 pour l'affichage de « Arguments : »), auxquels s'ajoute un caractère nul que nous avons compté lors de la déclaration de la chaîne.

```
$ ./exemple-strncat 12345 678 90 1  
Arguments : 12345678901  
$ ./exemple-strncat 123456789 01 23 45678  
Arguments : 123456789012345678  
$ ./exemple-strncat 123456789 01 23 45678 90123  
Arguments : 12345678901234567890  
$
```

Ces fonctions sont loin d'être optimales pour leur rôle, car elles nécessitent, au sein de la routine, de recalculer la longueur de la chaîne destination avant de commencer la copie. Le pire, c'est qu'avec `strncat()` nous devons disposer, avant l'appel, de la longueur actuelle de la chaîne, et nous invoquons donc `strlen()` une fois de plus. Toutes ces étapes pourraient être évitées en utilisant les fonctions `strcpy()` ou `strncpy()` et en conservant la trace du pointeur renvoyé, comme nous l'avons fait dans l'exemple de `strncpy()`.

Nous allons à présent nous intéresser aux fonctions permettant de comparer des chaînes de caractères. Ces routines peuvent bien entendu servir à des comparaisons simples, mais aussi pour trier des listes de mots par exemple. Les routines de tri que nous étudierons dans le prochain chapitre réclament en effet un pointeur sur une fonction fournissant une relation d'ordre sur l'ensemble des éléments à classer. On peut très bien utiliser les fonctions de comparaison pour implémenter cette relation d'ordre.

La routine la plus simple est `strcmp()` dont le prototype est :

```
int strcmp (const char * chaîne_1, const char * chaîne_2);
```

Si les deux chaînes sont identiques, `strcmp()` renvoie 0. Sinon, elle renvoie une valeur dont le signe correspond au résultat de la soustraction entre les premiers caractères qui diffèrent entre les deux chaînes. La comparaison des caractères est réalisée en les considérant comme des `unsigned char` (donc allant de 0 à 255), qu'on convertit en `int` pour avoir un résultat signé.

Le résultat est assez intuitif par rapport à l'ordre du dictionnaire par exemple, et correspond aux cas suivants :

- `strcmp(ch_1, ch_2) = 0` si les deux chaînes sont égales.
- `strcmp(ch_1, ch_2) < 0` si la première chaîne est à classer avant la seconde.
- `strcmp(ch_1, ch_2) > 0` dans le cas contraire.

Voici un exemple des différents résultats possibles :

chaîne 1	chaîne 2	signe de strcmp()
ABCDE	ABCDE	strcmp() = 0
ABCDE	ABCDZ	strcmp() < 0
ABCZ	ABCDE	strcmp() > 0
ABCDE	ABC	strcmp() > 0

Le caractère de fin de chaîne étant nul, il est plus petit que tous les autres caractères. Si une chaîne est plus courte qu'une autre, elle sera donc considérée comme étant inférieure, même si tous les autres caractères sont égaux.

La fonction `strncmp()` fonctionne de la même manière que `strcmp()`, mais dispose d'un argument supplémentaire :

```
int strncmp (const char * chaine_1,
             const char * chaine_2,
             size_t longueur);
```

Elle ne compare que la longueur indiquée des deux chaînes, sans aller nécessairement jusqu'à leur fin. Cette fonction est particulièrement utile lorsqu'une application autorise des saisies de mots-clés abrégés. Dans ce cas, on comparera successivement la chaîne saisie avec le vocabulaire de référence, en se limitant à la longueur de la saisie. Si on trouve une correspondance, on accepte alors l'abréviation. Voici un exemple d'une telle routine, qui explore un vocabulaire contenu dans une table de chaînes de caractères. Elle renvoie le numéro du mot saisi, ou -1 en cas d'erreur. Cette fonction va même refuser les saisies ambiguës, si deux mots peuvent servir de compléments.

```
int recherche_correspondance (const char * saisie)
{
    int i;
    int longueur;
    int trouve = -1;

    longueur = strlen(saisie);
    for (i = 0; i < Nombre_de_mots; i++) {
        if (strncmp(chaine, Table_des_mots[i], longueur) == 0) {
            if (trouve != -1) {
                fprintf(stderr, "Saisie ambiguë, complétez le mot \n");
                return -1;
            }
            trouve = i;
        }
    }
    if (trouve == -1)
        fprintf(stderr, "Saisie inconnue \n");
    return trouve;
}
```

Le problème des fonctions `strcmp()` et `strncmp()` est qu'elles sont souvent trop rigides pour les saisies effectuées avec une interface utilisateur conviviale. On aimerait par exemple offrir à l'utilisateur la possibilité de s'affranchir de la casse des caractères, c'est-à-dire des différences entre majuscules et minuscules. Même si cette différenciation est souvent importante et obligatoire (symboles du langage C, noms de fichiers sous Unix...), on peut avoir envie de relâcher la contrainte envers l'utilisateur, quitte à modifier automatiquement la saisie par la suite.

Il existe deux fonctions, `strcasemp()` et `strncasemp()`, avec les mêmes prototypes que `strcmp()` et `strncmp()`, et renvoyant des valeurs de retour similaires (0 pour l'égalité, et une valeur de même signe que la différence sinon). Toutefois, ces deux fonctions présentent l'avantage de ne pas être sensibles aux différences entre majuscules et minuscules. Par exemple, `AbC` et `aBc` sont considérées comme étant égales.

Voyons un exemple d'utilisation de `strcasemp()`.

```
exemple-strcasemp.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[])
{
    int compar;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s chaine_1 chaine_2\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    compar = strcasemp(argv[1], argv[2]);
    fprintf(stdout, "%s %c %s \n", argv[1],
        (compar > 0 ? '>' : (compar == 0 ? '=' : '<')) ,
        argv[2]);

    return EXIT_SUCCESS;
}
```

Voici un essai :

```
$ ./exemple-strcasemp abc def
abc < def
$ ./exemple-strcasemp aBc AbC
aBc = AbC
$
```

La fonction `strcasemp()` peut servir à trier des chaînes de caractères, afin de les ordonner comme dans un dictionnaire. Toutefois, elle s'appuie sur les valeurs directes représentant les caractères, et ne tient pas compte des équivalences des caractères accentués. Pour tenir compte des positions usuelles des caractères accentués par rapport aux caractères normaux, une fonction spéciale a été créée : `strcoll()` avec la même syntaxe que `strcmp()` :

```
int strcoll (const char * chaine_1, const char * chaine_2);
```

Il faut initialiser la localisation avec `setlocale()` au début du processus. La fonction `strcoll()` est particulièrement bien adaptée pour les tris lexicographiques, en classant des données suivant l'ordre alphabétique correct. Malgré tout, elle est assez coûteuse en termes de temps, car à chaque comparaison les deux chaînes doivent être copiées dans une version modifiée pour prendre en compte la localisation. Cette modification a lieu au sein de la bibliothèque C. Lorsqu'on désire ordonner un grand nombre de chaînes, chacune d'elles est comparée à plusieurs reprises avec ses voisines et, à chaque comparaison, on repasse par l'étape de modification tenant compte de la localisation.

La bibliothèque C nous offre la possibilité d'accéder directement à la routine de modification des chaînes. Ainsi, il est possible d'obtenir une copie modifiée de chaque chaîne en fonction de la localisation. On pourra ensuite utiliser la routine `strcmp()` directement sur les chaînes modifiées, et on obtiendra le même résultat final qu'en employant `strcoll()`. Dans la localisation « C » par défaut, les chaînes copiées sont exactement identiques aux originales, puisque l'ordre des caractères est celui de la table *Ascii*. Dans les autres localisations, les chaînes contiennent des caractères supplémentaires destinés à permettre le tri, mais rendant les copies modifiées illisibles. Il faut donc bien conserver la version originale. En fait, la modification remplace les caractères par des séquences plus ou moins longues permettant de retrouver l'ordre naturel de tri suivant la localisation. C'est pour cela que la chaîne copiée n'est pas directement lisible. La routine de modification est `strxfrm()`, dont le prototype est le suivant :

```
size_t strxfrm (char * destination,  
               const char * origine,  
               size_t taille_maxi);
```

Elle copie la chaîne d'origine, en la modifiant, dans la chaîne destination, en n'y plaçant que le nombre maximal de caractères indiqué, sans compter le caractère nul final. Cette fonction renvoie le nombre de caractères nécessaires pour copier la chaîne d'origine. Lorsque la taille maximale indiquée vaut zéro, la fonction ne touche pas à la chaîne de destination. On utilise donc généralement `strxfrm()` en deux fois, le premier appel avec `strxfrm(NULL, chaîne, 0)` permet de connaître le nombre de caractères nécessaires pour la destination. On effectue l'allocation (en ajoutant un octet pour le caractère nul final), et on peut appeler `strxfrm()` avec tous ses arguments à ce moment-là.

Pour effectuer le tri d'une table de caractères, on peut créer une structure contenant un pointeur sur la chaîne originale et un pointeur sur une chaîne copie (à allouer), créer une table de ces structures et demander à une routine de tri – comme `qsort()`, que nous verrons plus loin – de faire automatiquement le classement. Il faut passer, en argument à `qsort()`, un pointeur sur une fonction de comparaison. Celle-ci utilisera `strcmp()` sur les chaînes modifiées.

Caractères accentués et codage UTF-8

Les fonctions que nous avons vues précédemment fonctionnent parfaitement avec les caractères Ascii standard, représentés par des valeurs de 0 à 127. Ces caractères contiennent les lettres majuscules et minuscules non accentuées, les chiffres décimaux, les symboles de ponctuation courants et quelques caractères spéciaux supplémentaires : #, \$, @, %, etc.

La représentation des caractères accentués s'est faite pendant longtemps en utilisant une extension de la table Ascii de 128 à 255. Cette extension pouvait varier suivant les zones géographiques, aussi nommées les *localisations*. Par exemple, le caractère de code 0xC6 pouvait représenter une lettre « Æ majuscule » en localisation ouest-européenne (ISO-8859-1), un « C accent aigu » en localisation centre-européenne (ISO-8859-2), et une lettre « tsé » en localisation cyrillique (ISO-8859-5). Le choix de la localisation se faisant par l'intermédiaire de variables d'environnement.

Pour standardiser plus généralement tous les alphabets, une norme a été introduite sous le nom d'*Unicode*, qui représente sur 31 bits tous les alphabets actuels, y compris ceux des dialectes peu répandus, des langues mortes, ou d'alphabets artificiels (elfique, klingon...).

Plusieurs implémentations mémoire ont été proposées pour cette table Unicode, notamment l'UCS-2 qui représente chaque caractère par deux octets. Elle fut rejetée, car la présence involontaire (par enchaînement de deux séquences d'octets) de valeurs représentant des caractères spéciaux pour les systèmes de fichiers (notamment le *slash* « / ») la rendait dangereuse pour les utilitaires non préparés à traiter cette représentation.

De nos jours, l'encodage utilisé le plus couramment sous Linux – et sous les autres Unix également – est nommé UTF-8. Il s'agit d'une représentation des caractères sur un nombre variable d'octets (de 1 à 6). Voici quelques propriétés du codage UTF-8 :

- Les caractères Ascii classiques sont représentés par un seul octet de 0 à 127, soit en binaire : 0xxxxxxx, avec x variable.
- Les caractères Unicode inférieurs à 2 048 tiennent sur deux octets, en binaire 110xxxxx-10xxxxxx, donc le premier octet entre 0xC0 (192) et 0xDF (223) et le second entre 0x80 (128) et 0xBF (191). Et ainsi de suite jusqu'à un maximum de 6 octets pour couvrir tout l'espace Unicode de 31 bits. Les caractères les plus employés dans les communications internationales conservent donc un encombrement minimal.
- Les caractères n'appartenant pas à la table Ascii sont représentés par des séquences d'octets entièrement compris entre 128 et 253. Un caractère inférieur à 128 ne peut donc être qu'un caractère Ascii. Il n'y a donc pas d'ambiguïté, on ne risque pas d'introduire involontairement des caractères de contrôle, des séparateurs de chemin comme '/', ni surtout un caractère nul dans le corps d'une chaîne. Le standard UTF-8 est donc directement utilisable au niveau du système pour représenter des chemins d'accès, des noms de machines, etc.

Cette représentation des caractères sur un nombre variable d'octets est appelée séquence multi-octet (*multibyte*), et le fichier `<limits.h>` définit une constante symbolique `MB_LEN_MAX` qui indique le nombre maximal d'octets nécessaires pour représenter un caractère quelle que soit la localisation choisie. La variable `MB_CUR_MAX` contient le nombre maximal d'octets nécessaires pour stocker un caractère dans la localisation actuelle.

L'encodage UTF-8 est très intéressant pour échanger des données entre applications, et même entre systèmes différents. Toutefois, il ne s'agit peut-être pas de la représentation mémoire la mieux adaptée au sein même de la bibliothèque C. Celle-ci peut choisir d'organiser les octets différemment (par exemple en fonction de l'ordre poids faible/poids fort du processeur). Aussi introduit-on la notion de caractères larges (*wide char*) de type `wchar_t`, que manipuleront les routines de la bibliothèque C. On dispose en outre de fonctions de conversion dans `<stdlib.h>` :

```
// Convertir un caractère large en chaîne multi-octet
int wctomb (char * destination, wchar_t source);

// Convertir une séquence multi-octet en un caractère large
int mbtowc (wchar_t * dest, const char * source, size_t taille);
```

Ces fonctions servent à traduire une seule séquence multi-octet, et ne permettent pas d'enchaîner les appels. Pour ce faire, il serait nécessaire de stocker une valeur intermédiaire représentant la progression de l'encodage, nommée *shift state*. Pour gérer le *shift state* et convertir plusieurs séquences successives, on doit utiliser les routines :

```
size_t mbrtowc (wchar_t * destination,
               const char * source, size_t taille,
               mbstate_t * etat);
size_t wctrmb (char * destination,
               wchar_t source,
               mbstate_t * etat);
```

L'état `mbstate_t` doit être défini et initialisé auparavant ainsi :

```
mbstate_t etat;
memset (& etat, 0, sizeof (mbstate_t));
```

La fonction `mbrtowc()` remplit le caractère large sur lequel on passe un pointeur en premier argument avec le résultat de la lecture de la séquence multi-octet passée en second argument. Le nombre maximal d'octets est indiqué en troisième position. Si la séquence est correcte, la fonction renvoie le nombre d'octets utilisés pour la conversion. Si la séquence débute bien, mais que le nombre d'octets transmis est trop court, `mbrtowc()` renvoie -2, et si la séquence est définitivement invalide, elle renvoie -1. Si la conversion réussit, l'état transmis en dernier argument est mis à jour.

La routine `wcrtomb()` convertit le caractère large passé en second argument en séquence multi-octet, qu'elle écrit dans la chaîne transmise en première position. Cette chaîne doit comporter au moins `MB_CUR_MAX` octets. La conversion n'a lieu que si le caractère large a une signification dans la localisation `LC_CTYPE` en cours.

Les fonctions `btowc()` et `wctob()` ne permettent de convertir qu'un seul octet en caractère large, et inversement. En fait, elles ne sont normalement utilisables que sur l'espace Ascii. Si le caractère large nécessite plusieurs octets pour être représenté, `wctob()` échoue en renvoyant `EOF`. Ces fonctions ne sont pas intéressantes, car elles obligent l'application à déterminer si un caractère large se représente sur un ou plusieurs octets, ce qui va à l'encontre des concepts d'internationalisation.

```
wint_t btowc (int caractere);  
int wctob (wint_t caractere);
```

Les routines nommées `mbsrtowcs()` et `wcsrtombs()` convertissent des chaînes complètes en séquences multi-octets et inversement, ainsi que `mbstowcs()` et `wcstombs()`, qui ne conservent pas l'état *shift state*.

```
size_t mbsrtowcs (wchar_t * destination,  
                  const char ** chaine, size_t taille,  
                  mbstate_t * etat);  
size_t wcsrtombs (char * destination,  
                  const wchar_t * *chaine, size_t taille,  
                  mbstate_t * etat);  
size_t mbstowcs(wchar_t * destination,  
                const char * chaine, size_t taille);  
size_t wcstombs(char * destination,  
                const wchar_t * chaine, size_t taille);
```

Ces deux dernières fonctions sont généralement les plus simples à utiliser. Pour connaître le nombre de caractères larges contenus dans une séquence multi-octet, on peut appeler :

```
nb_wchar = mbstowcs(NULL, chaine_mb, 0)+1;
```

Le « +1 » permet d'inclure le caractère large nul final. De même pour connaître le nombre d'octets nécessaires au stockage du résultat de la conversion d'une chaîne de caractères larges, on utilisera :

```
nb_octets = wcstombs(NULL, chaine_wchar, 0)+1;
```

Enfin, pour pouvoir assurer des entrées-sorties employant des chaînes de caractères larges, on ajoute quelques spécifications de type aux formats employés par `printf()` et `scanf()` :

Conversion	Signification
%lc	Caractère large : <code>printf()</code> attend un argument de type <code>wchar_t</code> , alors que <code>scanf()</code> nécessite un pointeur sur un caractère large.
%ls	Chaîne de caractères larges : <code>printf()</code> comme <code>scanf()</code> demandent un argument de type <code>wchar_t *</code> .

Les conversions `%C` (comme synonyme de `%lc`) et `%S` (équivalent à `%ls`) ne doivent plus être utilisées, elles ne sont pas standardisées dans C99.

Voyons un exemple de conversion et d’affichage de chaîne multi-octet en chaîne de caractères larges :

```
exemple-mbstowcs.c :
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char * argv[])
{
    int i, j, n;
    wchar_t * warg;
    setlocale(LC_ALL, "");

    for (i = 1; i < argc; i++) {
        fprintf(stdout, "Argument %d :\n", i);
        fprintf(stdout, " Sequence multi-octets : ");
        for (j = 0; argv[i][j] != '\0'; j++)
            fprintf(stdout, "%02x ", (unsigned char) (argv[i][j]));
        fprintf(stdout, "\n");
        n = mbstowcs(NULL, argv[i], 0)+1;
        fprintf(stdout, " Nb caracteres larges : %d\n", n);
        warg = calloc(n, sizeof(wchar_t));
        if (warg == NULL) {
            perror("calloc");
            exit(1);
        }
        mbstowcs(warg, argv[i], n);
        fprintf(stdout, " Chaîne wchar : %ls\n", warg);
        free(warg);
    }
    return 0;
}
```

À l'exécution, nous voyons que les chaînes contenant des caractères accentués sont bien affichées ; nous remarquons également que leur représentation en mémoire utilise deux octets pour chaque caractère accentué :

```
$ ./exemple-mbstowcs azerty éternité déjà
Argument 1 :
  Sequence multi-octets : 61 7a 65 72 74 79
  Nb caracteres larges : 7
  Chaîne wchar : azerty
Argument 2 :
  Sequence multi-octets : c3 a9 74 65 72 6e 69 74 c3 a9
  Nb caracteres larges : 9
  Chaîne wchar : éternité
Argument 3 :
  Sequence multi-octets : 64 c3 a9 6a c3 a0
  Nb caracteres larges : 5
  Chaîne wchar : déjà
$
```

Nous voyons par exemple que 0xC3-0xA9 représente le caractère 'é' et 0xC3-0xA0 le caractère 'à'.

Il existe des correspondances « larges » à toutes les fonctions de traitement de chaînes que nous avons déjà vues, et elles fonctionnent correctement cette fois avec des caractères accentués. Elles sont déclarées dans <wchar.h> :

Fonction avec chaînes larges	Fonction équivalente avec chaînes simples
wcslen (wchar_t * chaîne);	strlen (char * chaîne);
wcsnlen (wchar_t * chaîne, size_t * maximum);	strnlen (char * chaîne, size_t maximum);
wscpy (wchar_t * cible, wchar_t * source);	strcpy (char * cible, char * source);
wscncpy (wchar_t * cible, wchar_t * source, size_t taille);	strncpy (char * cible, char * source, size_t taille);
wscat (wchar_t * cible, wchar_t * source);	strcat (char * cible, char * source);
wmemset (wchar_t * chaîne, wchar_t caractere, size_t taille);	memset (char * chaîne, char caractere, size_t taille);
wmemcmp (wchar_t * chaîne_1, wchar_t * chaîne_2, size_t taille);	memcmp (char * chaîne_1, char * chaîne_2, size_t taille);

Fonction avec chaînes larges	Fonction équivalente avec chaînes simples
wmemcpy (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);	memcpy (char * chaine_1, char * chaine_2, size_t taille);
wmemmove (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);	memmove (char * chaine_1, char * chaine_2, size_t taille);
wcsncat (wchar_t * cible, wchar_t * source, size_t taille);	strncat (char * cible, char * source, size_t taille);
wcsncmp (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);	strncmp (char * chaine_1, char * chaine_2, size_t taille);
wscasecmp (wchar_t * ch_1, wchar_t * ch_2);	strcasecmp (char * ch_1, char * ch_2);
wcsncasecmp (wchar_t * ch_1, wchar_t * ch_2, size_t lg);	strncasecmp (char * ch_1, char * ch_2, size_t lg);
wscoll (wchar_t * chaine_1, wchar_t * chaine_2);	strcoll (char * chaine_1, char * chaine_2);
wcsxfrm (wchar_t * chaine_1, wchar_t * chaine_2, size_t taille);	strxfrm (char * chaine_1, char * chaine_2, size_t taille);

Voici un exemple d'utilisation de `wscoll()` pour comparer et déterminer les positions relatives de deux chaînes dans l'ordre du dictionnaire :

```
exemple-wscoll.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <wchar.h>

int main (int argc, char * argv[])
{
    int i, n;
    wchar_t * warg[2];

    int compar;
```

```
setlocale (LC_ALL, "");

if (argc != 3) {
    fprintf(stderr, "Syntaxe : %s chaine_1 chaine_2\n", argv[0]);
    exit(EXIT_FAILURE);
}

for (i = 0; i < 2; i ++) {
    n = mbstowcs(NULL, argv[i+1], 0)+1;
    warg[i] = calloc(n, sizeof(wchar_t));
    if (warg[i] == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
    mbstowcs(warg[i], argv[i+1], n);
}

compar = wcscoll(warg[0], warg[1]);
fprintf(stdout, "%ls %c %ls \n", warg[0],
        (compar > 0 ? '>' : (compar == 0 ? '=' : '<')) ,
        warg[1]);
free(warg[1]);
free(warg[0]);

return EXIT_SUCCESS;
}
```

Voici un exemple, où nous voyons bien que les caractères accentués sont correctement placés dans l'ordre lexicographique habituel :

```
$ ./exemple-wcscoll active activé
active < activé
$ ./exemple-wcscoll activé activer
activé < activer
$ ./exemple-wcscoll décimal déçu
décimal < déçu
$ ./exemple-wcscoll déçu dédale
déçu < dédale
$
```

Une dernière remarque concernant les caractères larges : si on désire utiliser des constantes larges (pour une initialisation ou une comparaison), on indiquera au compilateur qu'elles doivent être considérées comme caractères larges, en utilisant le préfixe `L`. Ainsi on écrira :

```
wchar_t chaine[127];  
chaine[0] = L'\0' ;
```

ou

```
if (reponse_saisie[0] == L'N') || (reponse_saisie[0] == L'n')  
    return -1;
```

Nous reviendrons sur les caractères larges à plusieurs reprises, principalement pour donner les équivalences entre fonctions « classiques » et fonctions « larges » lors du traitement de chaînes de caractères.

Recherches dans une zone de mémoire ou dans une chaîne

Il arrive fréquemment qu'on ait besoin de rechercher un caractère précis dans une zone de mémoire ou dans une chaîne. Cette recherche dans une zone de mémoire peut servir, par exemple, à retrouver des délimiteurs de blocs dans un ensemble de données binaires. Au sein d'une chaîne, on cherche régulièrement le caractère nul final évidemment, mais aussi des séparateurs de mots, comme l'espace ou la tabulation. Nous allons voir plusieurs fonctions permettant ce genre d'exploration.

Recherche dans un bloc de mémoire

La fonction la plus simple est `memchr()`, dont le prototype est le suivant :

```
void * memchr (const void * bloc, int octet, size_t longueur);
```

Elle recherche la première occurrence de l'octet indiqué en second argument, dans le bloc sur lequel on fournit un pointeur et dont on précise la longueur. Le pointeur renvoyé correspond à l'octet trouvé ou est NULL si aucune correspondance n'a été trouvée dans la longueur voulue.

Son équivalent pour caractères larges ne travaille pas avec des octets, mais des caractères dont la taille dépend de la localisation :

```
wchar_t * wmemchr(wchar_t * chaine,  
                  wchar_t caractere,  
                  size_t taille);
```

Cela nous permet d'implémenter de manière efficace la fonction `strnlen()` que nous avons vue plus haut :

```
size_t strnlen (const char * chaine, size_t taille_maxi)
{
    void * fin;
    fin = memchr(chaine, 0, taille_maxi);
    if (fin == NULL)
        return(taille_maxi);
    return fin - chaine;
}
```

Précisons tout de suite que l'implémentation interne de `memchr()` dans la bibliothèque C est loin d'être triviale. Il ne s'agit pas d'un « bête » :

```
for (i = 0; i < longueur; i++)
    if (bloc[i] == octet)
        return & (bloc[i]);
return NULL;
```

En réalité, non seulement cette routine est optimisée en assembleur, mais de plus elle emploie un algorithme astucieux permettant de faire la recherche directement dans des blocs de 4 ou 8 octets suivant la machine. Tout comme les autres fonctions d'accès à la mémoire ou aux chaînes de caractères, l'optimisation de cette routine poussera le programmeur à y avoir recours le plus souvent possible et à éviter toute implémentation personnelle d'une fonction existante.

Il existe une extension Gnu, nommée `rawmemchr()`, fonctionnant comme `memchr()` mais sans indiquer de longueur maximale. Étant donné l'effet dévastateur d'une telle routine quand on ne trouve pas l'octet recherché, nous nous abstiendrons de l'utiliser.

Mis à part quelques cas précis que nous avons évoqués plus haut, la recherche de données dans un bloc de mémoire est rarement limitée à un seul octet. On a souvent besoin de déterminer la position d'un sous-ensemble d'un bloc. Pour cela, la bibliothèque Glibc fournit une extension `memmem()` intéressante :

```
void * memmem (const char * bloc, size_t lg_bloc,
               const char * sous_bloc, size_t lg_sous_bloc);
```

Cette fonction renvoie la position de la première occurrence du sous-bloc au sein du bloc complet, ou `NULL` s'il n'a pas été trouvé. Il faut être très prudent avec `memmem()` et ne jamais lui transmettre un sous-bloc de taille nulle, car le comportement est différent suivant les implémentations de la bibliothèque C. Une attitude sage consiste à considérer le comportement de cette routine comme indéfini si le sous-bloc est vide.

Les fonctions `memchr()` et `memmem()` constituent donc les deux routines clés pour le travail sur les blocs de mémoire. Il existe toutefois de très nombreuses autres fonctions, permettant cette fois-ci de travailler sur des chaînes de caractères.

Recherche de caractères dans une chaîne

La fonction `strchr()` est semblable dans son principe à `memchr()`, la limite de la recherche étant évidemment la fin de la chaîne, sans qu'on ait besoin de la préciser explicitement :

```
| char * strchr (const char * chaîne, int caractère);
```

Cette fonction renvoie un pointeur sur le premier caractère correspondant trouvé, ou `NULL` en cas d'échec. On peut rechercher n'importe quel caractère, y compris le caractère nul final. Cela peut être intéressant dans le cas où on voudrait disposer d'un pointeur sur la fin de la chaîne, pour y ajouter quelque chose ou pour la parcourir vers l'arrière (élimination des sauts de lignes, espaces, tabulations en fin de chaîne, par exemple).

Au lieu d'écrire quelque chose comme

```
| char * suite;  
| suite = & (origine[strlen(origine)]);
```

ou à la limite

```
| suite = origine + strlen(origine);
```

qui présente les dangers de toutes les manipulations arithmétiques de pointeurs, on peut utiliser

```
| suite = strchr(origine, '\0');
```

qui évite un calcul inutile.

Rappelons que dans la Glibc, les routines de recherche de caractères sont parfaitement optimisées pour parcourir la chaîne par blocs de 4 ou 8 octets, et on a tout intérêt à y faire appel plutôt que de tenter de balayer la chaîne directement. On peut employer `strchr()` pour rechercher des séparateurs dans des enregistrements de données se présentant sous forme de texte, comme les deux-points dans les lignes du fichier `/etc/passwd` par exemple, mais nous verrons un peu plus loin des fonctions mieux adaptées à ce type de travail.

L'équivalent de `strchr()` pour chaînes de caractères larges est :

```
| wchar_t * wcschr (wchar_t * chaîne,  
|                  wchar_t caractère);
```

La fonction `strrchr()` présente le même prototype que `strchr()` :

```
char * strrchr (const char * chaîne, int caractere);
```

et son équivalent pour caractères larges également :

```
wchar_t * wcsrchr(wchar_t * chaîne,  
                  wchar_t car);
```

mais elles s'intéressent à la dernière occurrence du caractère dans la chaîne. Elle peut servir par exemple à rechercher le dernier caractère '/' dans un chemin d'accès, pour ne conserver que le nom d'un fichier. Il existe une fonction `basename()` dans la Glibc qui effectue ce travail, mais elle n'est pas toujours définie, car il y a un conflit avec une autre fonction `basename()` du groupe XPG. L'implémentation Gnu est en substance la suivante :

```
char * basename (const char * nom_de_fichier)
{
    char * retour;

    retour = strrchr(nom_de_fichier, '/');
    if (p == NULL)
        // Le nom de fichier n'a pas de préfixe
        return nom_de_fichier;
    // Renvoyer un pointeur sur le nom situe immédiatement
    // après le dernier /
    return p + 1;
}
```

Il existe deux fonctions obsolètes, `index()` et `rindex()`, qui sont respectivement des synonymes exacts de `strchr()` et `strrchr()`. On risque toujours de les rencontrer dans d'anciens fichiers source, mais il ne faut plus les employer, car non seulement elles sont amenées à disparaître, mais pire, les noms de ces fonctions sont mal choisis et peu révélateurs de leur rôle.

Recherche de sous-chaînes

À l'instar de `memchr()` qui est souvent moins utile que `memmem()`, les fonctions `strchr()` et `strrchr()` ont besoin d'être complétées par une routine de recherche de sous-chaîne entière. Il existe plusieurs variantes, la plus courante étant, on s'en doute, appelée `strstr()` :

```
char * strstr (const char * chaîne, const char * sous_chaine);
wchar_t * wcsstr (const wchar_t * chaîne,  
                  const wchar_t * sous_chaine);
```

Ces fonctions retournent un pointeur sur la première occurrence de la sous-chaîne recherchée au sein de la chaîne mentionnée. Si aucune correspondance n'est trouvée, elles renvoient un pointeur NULL. Si la sous-chaîne est vide, le pointeur renvoyé correspond au début de la chaîne. Toutefois, si on désire assurer la portabilité d'un programme, on évitera ce comportement extrême, comme avec `memmem()`, car d'autres bibliothèques C peuvent avoir un résultat différent.

L'utilisation de `strstr()` est simple :

```
exemple-strstr.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[])
{
    int    i;
    char * chaine;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe: %s chaine sous-chaine\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strlen(argv[2]) == 0) {
        fprintf(stderr, "La sous-chaine recherchee est vide !\n");
        exit(EXIT_FAILURE);
    }
    i = 0;
    chaine = argv[1];
    while (1) {
        chaine = strstr(chaine, argv[2]);
        if (chaine == NULL)
            break;
        // On saute la sous-chaine trouvée
        chaine += strlen(argv[2]);
        i ++;
    }
    if (i == 0)
        fprintf(stdout, "%s ne se trouve pas dans %s\n",
            argv[2], argv[1]);
    else
        fprintf(stdout, "%s a ete trouvee %d fois dans %s\n",
            argv[2], i, argv[1]);

    return EXIT_SUCCESS;
}
```

```
$ ./exemple-strstr abcdabcdefgabc abc
abc a ete trouvee 3 fois dans abcdabcdefgabc
$ ./exemple-strstr abcdabcdefgabc abcd
abcd a ete trouvee 2 fois dans abcdabcdefgabc
$ ./exemple-strstr abcdabcdefgabc abcde
abcde a ete trouvee 1 fois dans abcdabcdefgabc
$ ./exemple-strstr abcdabcdefgabc abcdf
abcdf ne se trouve pas dans abcdabcdefgabc
$
```

Il existe également une extension Gnu, nommée `strcasestr()`, dont le fonctionnement est le même que celui de `strstr()`, mais qui ne fait pas de distinction entre minuscules et majuscules. Elle est également sensible à la localisation. Pour créer le programme `exemple-strcasestr.c`, on recopie le programme `exemple-strstr.c`, en ajoutant une définition `_GNU_SOURCE` avant les inclusions d'en-têtes, pour accéder aux extensions Gnu. On insère également une ligne `setlocale()` pour tenir compte de la localisation et, bien entendu, on remplace `strstr()` par `strcasestr()`. Voici l'exécution qui en résulte :

```
$ ./exemple-strcasestr AbcaBcABC abc
abc a ete trouvee 3 fois dans AbcaBcABC
$ ./exemple-strcasestr AeaE ae
ae a ete trouvee 2 fois dans AeaE
$ ./exemple-strcasestr AeaE ad
ad ne se trouve pas dans AeaE
$
```

Une autre variante des fonctions de recherche consiste à s'occuper des caractères appartenant à un ensemble donné. Par exemple, la fonction `strspn()`, dont le prototype est le suivant :

```
| size_t strspn (const char * chaine, const char * ensemble);
```

et la fonction pour caractères larges :

```
| size_t wcsspn (const wchar_t * chaine, const wchar_t * ensemble);
```

renvoie la longueur de la sous-chaine initiale constituée uniquement de caractères compris dans l'ensemble fourni en second argument. On peut utiliser cette routine pour éliminer les caractères blancs en début de ligne :

```
void elimine_blancs_en_tete (char * chaine)
{
    size_t debut;
    size_t longueur;

    debut = strspn(chaine, " \t\n\r");
    if (debut != 0) {
        longueur = strlen(chaine + debut);
        memmove(chaine, chaine + debut, longueur + 1);
        // Longueur + 1 pour avoir le caractère nul final
    }
}
```

L'ordre des caractères dans l'ensemble n'a pas d'importance. Il existe une fonction inverse, `strcspn()`, renvoyant la longueur du segment initial ne contenant aucun caractère de l'ensemble transmis. Son prototype est équivalent à `strspn()` :

```
size_t strcspn (const char *   chaine, const char *   ensemble);
size_t wcscspn (const wchar_t * chaine, const wchar_t * ensemble);
```

Il en existe également une variante, `strpbrk()`, qui retourne un pointeur sur le premier caractère appartenant à l'ensemble :

```
char *      strpbrk (const char *   chaine,
                    const char *   ensemble);
wchar_t *   wcpbrk (const wchar_t * chaine,
                    const wchar_t * ensemble);
```

Lorsque cette fonction ne trouve pas de caractère contenu dans l'ensemble indiqué, elle renvoie `NULL`. Son implémentation pourrait être :

```
char * strpbrk (const char * chaine, const char * ensemble)
{
    size_t longueur;
    longueur = strcspn(chaine, ensemble);
    if (chaine[longueur] == '\0')
        return NULL;
    return chaine + longueur + 1;
}
```

On peut utiliser cette routine pour éliminer les sauts de ligne et retours chariot en fin de chaîne, mais également pour ignorer tous les commentaires se trouvant à la suite d'un caractère particulier, comme '#' ou '%' :

```
void elimine_commentaires_et_sauts_de_ligne (char * chaîne)
{
    char * rejet;
    rejet = strpbrk(chaîne, "\\n\\r#%");
    if (rejet != NULL)
        rejet[0] = '\\0';
}
```

Analyse lexicale

Un programme peut parfois avoir besoin d'implémenter un petit analyseur lexical. Nous insistons sur le mot *petit* car, dès que la complexité d'un tel analyseur augmente, on a intérêt à se tourner vers des outils spécialisés, comme `lex` et `yacc`, dont les versions Gnu sont nommées `flex` et `bison`. Pour des décompositions lexicales simples, la bibliothèque Glibc offre donc une fonction nommée `strtok()`. Le terme anglais *token*, qui signifie « jeton » en français, désigne des éléments d'analyse lexicale (par exemple les mots-clés, mais aussi les caractères de synchronisation comme « ; » en langage C).

La fonction `strtok()` est déclarée avec le prototype suivant :

```
char * strtok (char * chaîne, const char * separateurs);
```

On passe en premier argument un pointeur sur la chaîne à analyser, mais uniquement lors du premier appel. Ce pointeur est mémorisé par `strtok()` dans une variable statique. Lorsqu'on rappellera ensuite cette fonction, on lui transmettra un premier argument `NULL`, à moins de vouloir analyser une nouvelle chaîne.

Le second argument est une chaîne de caractères contenant ce qu'on considère comme des séparateurs. Pour extraire les mots d'une phrase, on pourra ainsi employer une chaîne de séparateurs comme « `\\t,;:!?-` ».

Lors de l'appel à `strtok()`, cette fonction modifie la chaîne transmise à l'origine en premier argument. Cette chaîne ne doit donc pas être une constante ni une variable statique susceptible d'être modifiée par d'autres fonctions de la bibliothèque. Dans de telles situations, il convient d'allouer une copie de la chaîne, avec `strdup()` ou `strdupa()` par exemple, qu'on transmettra à `strtok()`.

La fonction `strtok()` renvoie un pointeur sur le premier élément lexical, après avoir éliminé les éventuels séparateurs en début de chaîne. Lors de l'appel suivant, `strtok()` renvoie un pointeur sur le second élément lexical, et ainsi de suite jusqu'à la fin de la chaîne, où elle renvoie `NULL`.

En fait, le fonctionnement de `strtok()` est relativement simple. Elle dispose d'une variable statique initialement nulle où elle stocke le pointeur sur le début de la chaîne. Lors d'une invocation, `strtok()` recherche le premier caractère n'appartenant pas à l'ensemble des séparateurs en utilisant `strspn()`. Elle mémorise ce pointeur, car ce sera la valeur qu'elle renverra. Ensuite elle recherche, en appelant `strpbrk()`, le premier caractère qui soit un séparateur – donc le caractère suivant la fin du mot –, puis elle le remplace par un `'\0'` et stocke le pointeur sur l'octet suivant pour reprendre son travail lors de sa future invocation.

Nous allons écrire un programme simple qui analyse les champs des lignes transmises sur son entrée standard, en utilisant les caractères blancs comme séparateurs :

```
exemple-strtok.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LG_MAXI 256

int main (void)
{
    char * ligne;
    char * champs;
    int n, c;

    if ((ligne = malloc(LG_MAXI)) == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    n = 1;
    while (fgets(ligne, LG_MAXI, stdin) != NULL) {
        fprintf(stdout, "Ligne %d\n", n);
        c = 1;
        champs = strtok(ligne, " \t");
        while (champs != NULL) {
            fprintf(stdout, " champs %d : %s\n", c, champs);
            champs = strtok(NULL, " \t");
            c ++;
        }
        n ++;
    }
    return EXIT_SUCCESS;
}
```

Nous pouvons utiliser ce programme pour analyser le fichier `/etc/fstab` par exemple, où les champs sont séparés par des tabulations ou des espaces :

```
$ ./exemple-strtok < /etc/fstab
```

```
Ligne 1
```

```
champs 1 : tmpfs
champs 2 : /dev/shm
champs 3 : tmpfs
champs 4 : defaults
champs 5 : 0
champs 6 : 0
```

```
Ligne 2
```

```
champs 1 : devpts
champs 2 : /dev/pts
champs 3 : devpts
champs 4 : gid=5,mode=620
champs 5 : 0
champs 6 : 0
```

```
Ligne 3
```

```
champs 1 : sysfs
champs 2 : /sys
champs 3 : sysfs
champs 4 : defaults
champs 5 : 0
champs 6 : 0
```

```
Ligne 4
```

```
champs 1 : proc
champs 2 : /proc
champs 3 : proc
champs 4 : defaults
champs 5 : 0
champs 6 : 0
```

```
[...]
```

Le fait que `strtok()` garde une variable statique globale entre deux appels le rend non réentrant. En d'autres termes, cette fonction ne doit pas être utilisée au sein d'un gestionnaire de signal et doit être évitée dans le cadre d'un programme multithread. Pour pallier ce problème, la bibliothèque Glibc fournit une fonction supplémentaire où le pointeur doit être transmis en argument à chaque appel.

La fonction `strtok_r()` est calquée sur `strtok()` avec juste un argument supplémentaire lui permettant d'être réentrante :


```
char * strtok_r (char * chaîne,  
                const char * separateurs,  
                char ** pointeur);
```

C'est également ce format qui a été choisi pour l'implémentation avec caractères larges :

```
wchar_t * wcstok (wchar_t * chaîne,  
                  const wchar_t * separateurs,  
                  wchar_t ** pointeur);
```

Son fonctionnement est exactement identique à celui de `strtok()`, mais il faut donc lui fournir un pointeur supplémentaire, dont on n'a toutefois pas spécialement besoin de se soucier. Dans notre exemple précédent, il suffisait de modifier le programme en ajoutant une variable

```
char * pointeur;
```

et d'utiliser les appels

```
champs = strtok_r (ligne, " \t", & pointeur);  
champs = strtok_r (NULL, " \t", & pointeur);
```

Le programme `exemple-strtok-r` fonctionne alors exactement comme `exemple-strtok`.

Une autre possibilité pour le même résultat est d'utiliser `strsep()`, qui vient de l'univers BSD. Son prototype est le suivant :

```
char * strsep (char ** pointeur, const char * separateur);
```

Globalement, elle fonctionne comme `strtok()`, mais il est du ressort du programmeur d'initialiser le pointeur fourni en premier argument pour qu'il soit dirigé vers la chaîne à traiter.

Cette routine se comporte toutefois différemment lorsqu'elle rencontre plusieurs séparateurs successivement, puisqu'elle renvoie à ce moment-là une chaîne vide alors que `strtok()` sautait les occurrences successives de séparateurs. Il faut donc ajouter un test supplémentaire à notre programme, dont la boucle principale devient :

```
exemple_strsep.c :  
...  
/* Début identique à strtok_r.c */  
...
```

```
while (fgets(ligne, LG_MAXI, stdin) != NULL) {
    fprintf(stdout, "Ligne %d\n", n);
    c = 1;
    pointeur = ligne;
    while (1) {
        champs = strsep(& pointeur, " \t");
        if (champs == NULL)
            break;
        if (champs[0] == '\0')
            continue;
        fprintf(stdout, " champs %d : %s\n", c, champs);
        c ++;
    }
    n ++;
}

return EXIT_SUCCESS;
}
```

L'exécution présente bien entendu les mêmes résultats.

Conclusion

Nous achevons ainsi ce chapitre consacré à la gestion des chaînes de caractères et des blocs de mémoire. Nous y avons étudié en détail les routines classiques de traitement des chaînes de caractères ainsi que leurs extensions pour caractères larges. Le prochain chapitre présentera des traitements importants dans de nombreuses applications : les tris.

17

Tris, recherches et structuration des données

Les algorithmes mis en œuvre lors des opérations de tri ou de recherche de données peuvent être très simples ou au contraire extrêmement compliqués. La bibliothèque C nous propose plusieurs variantes que nous choisirons au gré des caractéristiques de l'application.

Nous allons tout d'abord étudier les routines de comparaison entre éléments que l'utilisateur doit fournir aux routines de la bibliothèque C. Puis nous examinerons les recherches linéaires dans une table, ainsi qu'une amélioration intéressante concernant les données auto-organisatrices. Nous verrons par la suite comment trier rapidement une table, afin de pouvoir utiliser une recherche dichotomique plus rapide. Une section sera consacrée à l'étude des arbres binaires, qui sont une structure de données permettant des opérations de recherche rapides.

Nous analyserons enfin les fonctions que la bibliothèque Glibc met à notre disposition pour manipuler des tables de hachage, des structures particulièrement précieuses pour gérer des listes de mots ou de symboles, par exemple.

Fonctions de comparaison

L'essentiel des fonctions présentées dans ce chapitre nécessite de pouvoir comparer des données, afin de les ordonner ou simplement pour identifier l'élément recherché. Cette comparaison peut être effectuée sur plusieurs critères variant en fonction du

type de données stockées. Lorsqu'il s'agit simplement de valeurs numériques, on peut très bien utiliser les comparaisons classiques `<`, `=`, ou `>`. Lorsqu'on doit comparer des mots, on peut imaginer employer `strcmp()`, quoique ce ne soit pas forcément le meilleur choix. Lorsqu'on veut comparer des enregistrements d'une base de données, il est nécessaire de définir un champ comme clé de recherche pour que la comparaison entre deux enregistrements s'effectue sur leurs membres ainsi choisis.

Pour homogénéiser les opérations, les routines de tri et de recherche reçoivent en argument un pointeur sur une fonction écrite par l'utilisateur, et qui devra prendre en arguments deux éléments et renvoyer une valeur négative, nulle ou positive selon que le premier argument est considéré comme inférieur, égal ou supérieur au second.

Le prototype d'une routine de comparaison sera donc :

```
| int comparaison (const void * element_1, const void * element_2);
```

Bien sûr, celle-ci devra être adaptée au type de données manipulées par le programme. Les arguments doivent être déclarés comme des pointeurs `void *`, mais la bibliothèque C l'invoque en utilisant des pointeurs sur les objets à comparer. Voici quelques exemples :

```
| int compare_entiers (const void * arg_1, const void * arg_2)
| {
|     int entier_1 = * ((int *) arg_1);
|     int entier_2 = * ((int *) arg_2);
|     return entier_1 - entier_2;
| }
```

ou pour des caractères (non larges) :

```
| int compare_chaines_ascii (void * arg_1, void * arg_2)
| {
|     return strcmp((const char *) arg_1, (const char *) arg_2);
| }
```

Nous voyons que, dans ce dernier cas, on aurait pu prendre directement un pointeur sur la fonction `strcmp()`, à condition de forcer son type par un *cast* explicite. Naturellement, dans le cas de chaînes susceptibles de contenir des caractères accentués, on utiliserait plutôt `wstrcmp()`.

Voici un exemple de comparaison de données structurées :

```
| typedef struct {
|     char * nom;
|     char * prenom;
|     time_t date_naiss;
```

```
    char * lieu_naiss;
    /* Champs non pris en compte dans la comparaison */
    time_t date_d_inscription;
    long    livre_emprunte;
    /* ... Etc ... */
} individu_t;

int compare_identites (const void * element_1, const void * element_2)
{
    individu_t * individu_1 = (individu_t *) element_1;
    individu_t * individu_2 = (individu_t *) element_2;
    int    comparaison;

    comparaison = strcasecmp(individu_1->nom, individu_2->nom);
    if (comparaison != 0)
        return comparaison;

    comparaison = strcasecmp(individu_1->prenom, individu_2->prenom);
    if (comparaison != 0)
        return comparaison;

    comparaison = individu_1->date_naiss - individu_2->date_naiss;
    if (comparaison != 0)
        return comparaison;

    comparaison = strcasecmp(individu_1->lieu_naiss,
                              individu_2->lieu_naiss);
    return comparaison;
}
```

Nous faisons ici une comparaison successive sur les quatre critères d'identification. Nous prendrons donc une clé primaire représentée par le nom, puis des clés secondaires constituées successivement par le prénom, la date et le lieu de naissance. Nos enregistrements peuvent bien entendu contenir des champs qui ne sont pas pris en compte lors de la comparaison.

Remarquons au passage que, dans certains cas, l'emploi de la comparaison `strcasecmp()` n'est pas le plus approprié. Lors des recherches sur les noms de famille par exemple, il existe toujours une marge assez importante d'incertitude concernant les fautes de frappe, les inversions de lettres dans un nom épilé au téléphone (je peux en témoigner personnellement...), ou simplement des problèmes de mauvaise compréhension dus à un accent prononcé. Pour assurer une certaine tolérance, on peut utiliser un algorithme de phonétisation. Il en existe de nombreux, comme *Soundex* ou *Métaphone*, qui sont largement employés dans les logiciels de généalogie ou les recherches en annuaire par exemple, et qui permettent de réduire un nom à ses consonnes les plus importantes. Les règles de phonétisation varient suivant la langue, il est conseillé de rechercher un algorithme adapté aux noms à comparer.

Pour simplifier les prototypes des routines de recherche et de tri, la bibliothèque Glibc définit un type spécial, nommé `comparison_fn_t`, sous forme d'extension Gnu, qui correspond à une fonction de comparaison. Ce type est défini, en substance, ainsi :

```
| typedef int (* comparison_fn_t) (const void *, const void *);
```

Lorsque nous trouverons, dans une liste d'arguments, une déclaration :

```
| int fonction_de_tri (... , comparison_fn_t compare, ...);  
| cela signifiera que la fonction compare() est du genre :  
| int compare (void * element_1, void * element2);
```

La plupart des routines permettent de trier des tables contenant des éléments de taille constante. Le tri des données de taille variable (chaînes de caractères par exemple) ne pose pas de problème puisqu'il suffit d'ajouter un niveau d'indirection supplémentaire en triant en réalité une table de pointeur sur les données de taille variable. Bien entendu, il faudra tenir compte correctement de cette indirection dans la routine de comparaison.

Recherche linéaire, données non triées

Les premières fonctions que nous étudierons permettent de faire une recherche linéaire dans une table, aussi appelée recherche séquentielle. Il s'agit simplement de parcourir toute la table jusqu'à trouver l'élément correspondant à la clé recherchée. Cette méthode n'a d'intérêt que si la table n'est pas ordonnée car, dans le cas contraire, nous verrons des routines beaucoup plus rapides pour accéder aux données.

On peut s'interroger sur l'intérêt de conserver une table de données non ordonnée, alors qu'il existe des routines de tri simples et performantes. En fait, la recherche dans une table triée n'est intéressante que si le nombre d'éléments est suffisamment grand et si la table ne subit que peu de modifications. En effet, l'insertion ou la suppression de données sont obligatoirement plus coûteuses dans une table triée que dans une table non ordonnée, puisqu'il faut faire appel à des routines spécialisées pour placer l'enregistrement au bon endroit.

Si notre table est « petite » (au maximum quelques dizaines d'enregistrements), et si la routine de comparaison est simple et rapide, il est plus commode de laisser la table non ordonnée et d'utiliser une recherche séquentielle. Ce choix sera également plus judicieux si la table change beaucoup. Cela signifie qu'on renouvelle le contenu de la table en permanence, et qu'un enregistrement donné n'est pas recherché plus de deux ou trois fois durant son existence. Ainsi, j'ai employé une recherche séquentielle dans

un logiciel dans lequel on reçoit en permanence des positions d'avions en approche finale sur un aéroport. Chaque enregistrement n'existant dans notre liste que pendant une durée assez courte, alors qu'il y a des ajouts et des suppressions pratiquement toutes les secondes, l'utilisation d'une liste triée ne se justifiait pas.

Les deux routines de recherche linéaire offertes par la bibliothèque Glibc sont nommées `lfind()` et `lsearch()`. Leurs prototypes sont les suivants :

```
void * lfind (const void * cle, const void * base,
              size_t * nb_elements, size_t taille,
              comparison_fn_t compare);
et :
void * lsearch (const void * cle, void * base,
                size_t * nb_elements, size_t taille,
                comparison_fn_t compare);
```

Elles sont déclarées dans le fichier `<search.h>`.

La première routine recherche l'élément qui correspond à la clé fournie en premier argument dans la table commençant à l'adresse passée en second argument contenant `*nb_elements`, et chaque élément ayant la taille indiquée en quatrième position. Pour chercher la donnée correspondant à la clé, la fonction de comparaison fournie en dernière position est employée. On doit passer un pointeur sur le nombre d'éléments, et non la véritable valeur, même si son contenu n'est pas modifié par `lfind()`. Si la routine trouve un élément correspondant à la clé, elle renvoie un pointeur dessus. Sinon, elle renvoie `NULL`.

Voici un exemple d'utilisation avec la structure de données `individu_t` que nous avons définie plus haut :

```
static individu_t * table_individus = NULL;
static size_t      nb_individus = 0;

individu_t * donne_individu (const char * nom, const char * prenom,
                             time_t date, const char * lieu)
{
    individu_t cle;
    individu_t * retour;

    cle.nom = nom; /* On copie le pointeur, pas la chaîne */
    cle.prenom = prenom;
    cle.date_naiss = date;
    cle.lieu_naiss = lieu;

    retour = lfind(&cle, table_individus, &nb_individus,
                  sizeof (individu_t), compare_identites);
```

```

    if (retour != NULL)
        return retour;
    /* On ne l'a pas trouvée, on va en créer un nouveau */
    table_individus = realloc(table_individus,
                              sizeof(individu_t) * (nb_individus + 1));
    if (table_individus == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    table_individus[nb_individus].nom = strdup(nom);
    table_individus[nb_individus].prenom = strdup(prenom);
    table_individus[nb_individus].date_naiss = date;
    table_individus[nb_individus].lieu = strdup(lieu);
    time (& (table_individus[nb_individus].date_inscription));
    table_individus[nb_individus].livre_emprunte = -1;

    nb_individus ++;
    return (& (table_individus[nb_individus - 1]));
}

```

Nous avons utilisé dans cet exemple les structures déjà définies plus haut, mais on remarquera par ailleurs que la gestion d'un fichier de ce type – probablement les inscriptions dans une bibliothèque – n'est justement pas adaptée à une organisation séquentielle, puisque les données varient peu et que le nombre d'enregistrements est certainement assez conséquent.

La fonction `lsearch()` recherche également l'enregistrement de manière séquentielle dans la table fournie, mais si elle ne le trouve pas, elle ajoute un enregistrement à la fin, et incrémente l'argument `nb_elements`, sur lequel on doit passer un pointeur, comme avec `lfind()`. Cela signifie qu'il faut être sûr avant d'appeler `lsearch()` de disposer d'au moins un emplacement supplémentaire libre dans la table. On l'utilise parfois en effectuant des allocations « par blocs », afin de réduire le nombre d'appels à `realloc()`. Voici un exemple :

```

static individu_t * table_individus = NULL;
static size_t      nb_individus = 0;
static size_t      contenance_table = 0;

#define NB_BLOCS_AJOUTES 64

individu_t * donne_individu (const char * nom, const char * prenom,
                             time_t date, const char * lieu)
{
    individu_t  cle;
    individu_t * retour;

```



```

    if (contenance_table == nb_individus) {
        contenance_table += NB_BLOCS_AJOUTES;
        table_individus = realloc(table_individus,
                                   contenance_table * sizeof(individu_t));
        if (table_individus == NULL) {
            perror("realloc");
            exit(EXIT_FAILURE);
        }
    }

    cle.nom = nom;    /* On copie le pointeur, pas la chaîne */
    cle.prenom = prenom;
    cle.date_naiss = date;
    cle.lieu_naiss = lieu;

    retour = lsearch(& cle, table_individus,
                    & nb_individus, sizeof(individu_t),
                    compare_identites);

    return retour;
}

```

On notera que, dans le cas d'une recherche séquentielle, la fonction de comparaison doit simplement renvoyer 0 si les éléments concordent, et une autre valeur sinon. On n'a pas besoin d'indiquer si la première est inférieure à la seconde ou non.

En fait, il est très commode d'appeler les routines `lfind()` ou `lsearch()` si on désire implémenter une table non triée pour débiter le développement d'une application, quitte à se tourner ensuite vers une implémentation plus structurée si le besoin s'en fait sentir. Les routines de recherche dans les tables ordonnées ou dans les arbres binaires ont une interface quasi identique, et la modification d'implémentation est facile.

Toutefois, si on désire conserver une table non triée, et si les fonctionnalités de recherche dans cette table sont critiques pour l'application, on peut envisager de réimplémenter ses propres routines à la place de celles de la bibliothèque Glibc. C'est l'un des rares cas où une réécriture de fonctions existantes peut apporter quelque chose de sensible à une application, sans risque d'erreur, vu la simplicité de l'algorithme utilisé.

La fonction `lfind()` est implémentée, en substance, dans la Glibc ainsi :

```

void * lfind (const void * cle, const void * base,
              size_t * nb_elem, size_t taille, comparison_fn_t compare)
{
    const void * retour = base;
    size_t      compteur = 0;

    while ((compteur < *nb_elem)
          && ((*compare)(cle, retour) != 0)) {

```

```
        retour += taille;
        compteur ++;
    }
    return (compteur < *nb_elem ? retour : NULL);
}
```

On peut reprocher deux choses à cette fonction :

- Elle appelle pour chaque enregistrement la routine de comparaison, alors qu'on pourrait éviter la surcharge de code due à une invocation de fonction en intégrant directement le code de comparaison dans la recherche séquentielle.
- Elle effectue deux tests à chaque itération, en vérifiant à la fois si le compteur a atteint le nombre d'éléments dans la table et si la comparaison a réussi.

En fait, pour éviter de dupliquer le test à chaque itération, il suffit d'ajouter un élément fictif à la fin de la table, dans lequel on copie la clé recherchée. On ne fait plus que la comparaison à chaque itération. Lorsqu'on sort de la boucle, on vérifie alors si on avait atteint le dernier élément ou non. Cette méthode oblige à toujours disposer d'un emplacement supplémentaire en fin de table, mais il suffit d'allouer un élément de plus à chaque appel `realloc()`.

Nous pouvons alors écrire une routine spécialisée pour nos données. Par exemple, pour rechercher un entier dans une table non triée :

```
int * recherche_entier (int cle, int * table, int nb_entiers)
{
    int * resultat = table;

    /* on sait qu'on dispose d'un élément supplémentaire */
    table[nb_entiers] = cle;
    while (cle != *resultat)
        resultat += sizeof(int);
    if (resultat == & (table[nb_entiers]))
        return NULL;
    return resultat;
}
```

Ceci nous permet d'augmenter les performances de cette recherche, en l'adaptant à nos données. Une autre amélioration peut parfois être apportée en utilisant une organisation automatique des données.

La recherche séquentielle balaye tous les enregistrements jusqu'à trouver celui qui convient. Lorsque le nombre d'enregistrements croît, la durée de la recherche augmente dans la même proportion. On dit que la complexité de cet algorithme s'exprime en $O(N)$, N étant le nombre de données dans la base.

Lorsque tous les enregistrements présents ont la même probabilité d'être recherchés, le parcours séquentiel balaye, en moyenne, $N/2$ éléments. Toutefois, ceci n'est vrai que si

les données sont équiprobables, c'est-à-dire si tous les enregistrements font l'objet d'une recherche le même nombre de fois.

Or, dans de très nombreuses situations, certaines données sont beaucoup plus sollicitées que d'autres. À titre d'exemple, les secteurs d'un disque dur ou les mots d'un lexique obéissent plutôt à une loi dite *80-20*, c'est-à-dire que 20 % des données font l'objet de 80 % des recherches. Et au sein de ces 20 %, la même loi peut se répéter.

Autrement dit, on a tout intérêt à trouver un moyen de placer en tête de table les données les plus fréquemment recherchées. Pour cela, il existe une méthode simple : à chaque fois qu'une recherche aboutit, l'élément retrouvé est permuté avec celui qui le précède dans la table. Les données les plus demandées vont donc remonter automatiquement au cours des recherches successives, afin de se trouver aux places de choix, celles qui nécessitent un balayage minimal. De même, les enregistrements qu'on ne réclame jamais vont descendre en fin de table, là où la recherche dure le plus longtemps.

La table s'organisant au fur et à mesure des demandes de l'utilisateur, les résultats sont parfois étonnamment bons, surtout si on remarque que la plupart des recherches successives ne sont pas indépendantes et sont dictées par un centre d'intérêt commun qui réclame parfois le même élément à plusieurs reprises. C'est un phénomène un peu similaire à celui des mémoires cache en lecture, qui permettent d'améliorer sensiblement les performances d'un disque dur.

On peut, par exemple, implémenter ce mécanisme en ajoutant à la suite d'un appel à `lfind()` :

```
element_t * retour;  
element_t  echange;  
  
retour = lfind(cle, table, & nb_elements,  
              sizeof (element_t), compare_elements);  
if ((retour != NULL) && (retour != table)) {  
    memcpy(& echange, retour, sizeof(element_t));  
    memcpy(retour, retour-sizeof(element_t), sizeof(element_t));  
    memcpy(retour-sizeof(element_t), & echange, sizeof(element_t));  
}
```

Bien entendu, avec des types entiers ou réels par exemple, l'échange est bien plus simple. Attention, répétons que les tables auto-organisatrices fonctionnent uniquement si les données n'ont vraiment pas la même probabilité d'être recherchées. Ceci nécessite donc d'analyser précisément les éléments dont on dispose lors de l'implémentation de l'application.

Nous avons ainsi vu les mécanismes les plus simples pour rechercher des données non organisées, ainsi que quelques astuces pouvant améliorer les performances. Malgré tout, dans la majeure partie des cas, il est préférable d'essayer de trier le contenu de notre ensemble d'informations afin d'obtenir des recherches beaucoup plus rapides.

Recherches dichotomiques dans une table ordonnée

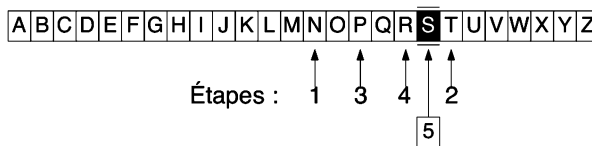
Lorsqu'on dispose d'une table où les données sont triées, une recherche est bien plus rapide. Il suffit en effet d'utiliser un algorithme de recherche dichotomique pour obtenir des performances très intéressantes. Le principe de la recherche dichotomique est simple :

- On choisit un élément au centre de la table triée et on le compare avec la clé recherchée.
- Si la clé est égale à l'élément, on a fini avec succès.
- Si la clé est plus petite que l'élément, on réitère le processus sur la moitié inférieure de la table.
- Sinon, on recommence en partant de la moitié supérieure de la table.
- Si la clé recherchée ne se trouve pas dans la table, on finira par se retrouver avec une portion réduite à un seul élément, auquel cas on finira l'algorithme en échec.

Cet algorithme nous garantit une complexité en $O(\log(N))$, ce qui signifie que lorsque le nombre N de données augmente, la durée de la recherche croît proportionnellement à $\log(N)$. Or, pour des valeurs suffisamment grandes, $\log(N)$ est très inférieur à N . Cette recherche dichotomique est donc largement plus rapide qu'une recherche séquentielle.

Figure 17-1

Recherche dichotomique dans une table triée



Si la recherche dichotomique paraît simple à première vue, sa programmation l'est beaucoup moins. Cet exercice classique révèle des subtilités de mise en œuvre, et la première tentative d'implémentation est rarement exempte de défauts. Aussi, on se limitera autant que possible à utiliser directement la routine `bsearch()` de la bibliothèque C, qui implémente la recherche dichotomique de manière exacte et optimisée, définie dans `<stdlib.h>` :

```
void * bsearch (const void *cle, const void * table,
                size_t nb_elements, size_t taille,
                comparison_fn_t compare);
```

La seule différence avec le prototype de `lsearch()` est qu'on passe le nombre d'éléments du tableau et pas un pointeur sur ce nombre.

Cette fonction renvoie un pointeur sur l'élément recherché, ou NULL en cas d'échec. La fonction de comparaison doit renvoyer une valeur positive ou négative en fonction de la position des deux éléments comparés, et non plus simplement une valeur nulle ou non nulle comme dans la recherche séquentielle. Si plusieurs éléments de la table sont égaux, l'algorithme ne précise pas celui qui sera trouvé (contrairement à `lsearch()` qui rencontre toujours le premier d'abord).

Avant de pouvoir utiliser `bsearch()`, il faut ordonner les données. Pour cela, il existe de nombreux algorithmes plus ou moins performants, et la bibliothèque C implémente deux, le choix étant effectué au moment de l'appel en fonction de la taille des données et de la disponibilité mémoire.

L'algorithme le plus efficace garantit une complexité en $O(N \cdot \log(N))$, c'est-à-dire que la durée du tri croît proportionnellement à $N \cdot \log(N)$ lorsque le nombre N d'éléments augmente. Indiquons que les algorithmes de tri « évidents », comme le célèbre tri à bulles, ont une complexité en $O(N^2)$, et rappelons que lorsque N est déjà moyennement grand – quelques centaines d'éléments – $\log(N)$ est nettement inférieur à N .

Cet algorithme nécessite un espace de stockage de taille identique à la table à trier. Le principe consiste à séparer récursivement la partition à trier en deux ensembles de taille identique à un élément près. Ces deux ensembles sont alors triés séparément par appel récursif de l'algorithme. Ensuite, on mélange les données triées dans la mémoire temporaire en les parcourant une seule fois, par comparaisons successives. La mémoire temporaire est alors recopiée dans la table à présent ordonnée.

Cet algorithme est optimal en termes de rapidité, mais il nécessite un espace de stockage temporaire qui peut parfois être exagéré si on trie par exemple le contenu d'un fichier projeté en mémoire avec `mmap()`. Aussi, la bibliothèque Glibc en propose un second qui est presque aussi efficace et qui ne nécessite pas de mémoire auxiliaire : le *quicksort* (tri rapide). Ce tri, décrit par C. Hoare en 1952, est particulièrement célèbre, et la Glibc implémente de surcroît des améliorations pour augmenter encore ses performances.

Le *quicksort* repose sur la division successive de la table à trier en partitions de taille de plus en plus réduite. Le principe consiste à choisir dans la partition à ordonner une valeur médiane, dite *pivot*, et à scinder la partition en deux sous-ensembles distincts, l'un contenant uniquement des valeurs inférieures au pivot, et l'autre comprenant seulement des valeurs supérieures au pivot. Pour effectuer ce découpage rapidement, on utilise deux pointeurs : l'un partant du bas de la partition et remontant progressivement jusqu'à rencontrer une valeur plus grande que le pivot, et l'autre partant symétriquement du haut de la partition pour descendre jusqu'à trouver un élément inférieur au pivot. Si les deux pointeurs se sont croisés, la séparation en deux sous-partitions est finie, sinon on échange les deux éléments rencontrés et on continue. Le processus est alors répété sur les deux nouvelles partitions, jusqu'à avoir des sous-ensembles ne contenant que trois éléments ou moins, et la table originale est alors entièrement triée.

L'éventuelle complication avec le *quicksort* réside dans le choix du pivot. Dans l'algorithme original, on propose d'utiliser comme pivot le premier élément de la table, ce qui simplifie la suite des opérations puisqu'il suffit de placer le pointeur bas sur le deuxième élément et le pointeur haut sur le dernier, sans se soucier de rencontrer le pivot lui-même. Toutefois, cela pose un grave problème sur les tables déjà ordonnées. En effet, le découpage obtenu est alors catastrophique puisqu'il contient une sous-partition ne comprenant qu'un seul élément, et une seconde comportant les $N-1$ autres. La complexité de l'algorithme n'est plus $O(N \cdot \log(N))$ mais approche au contraire $O(N^2)$.

Pour éviter ce problème, la Glibc choisit comme pivot une valeur médiane entre le premier élément du tableau, le dernier et un élément placé au milieu. Même si la table est déjà ordonnée, la performance du tri reste intacte. De même, la bibliothèque Glibc évite d'utiliser le *quicksort* lorsque la taille des partitions devient petite (quatre éléments en l'occurrence), et elle se tourne alors vers un tri par insertion qui est plus efficace dans ce cas. Enfin, les performances de l'implémentation sont encore améliorées en évitant d'utiliser la récursivité naturelle de l'algorithme, mais en gérant directement une liste des partitions à traiter.

La Glibc emploie donc autant que possible le tri avec une mémoire auxiliaire, sinon elle se tourne vers le *quicksort*. La routine `qsort()`, déclarée dans `<stdlib.h>`, qui tire son nom du *quicksort* utilisé dans l'implémentation traditionnelle sous Unix, est très simple d'utilisation :

```
void qsort (void * table,
            size_t nb_elements, size_t taille_element,
            comparison_fn_t compare);
```

Voici un exemple de programme qui crée une table de valeurs aléatoires, puis qui invoque `qsort()` pour les trier.

```
exemple-qsort.c :
#include <stdio.h>
#include <stdlib.h>

int compare_entiers (const void * elem_1, const void * elem_2)
{
    return * ((int *)elem_1) - * ((int *)elem_2);
}

#define NB_ENTIERS 100

int main (void)
{
    int table_entiers[NB_ENTIERS];
    int i;
```

```

for (i = 0; i < NB_ENTIERS; i++) {
    /* On limite un peu la taille des entiers pour l'affichage */
    table_entiers[i] = rand() & 0xFFFF;
    fprintf(stdout, "%05d ", table_entiers[i]);
}
fprintf(stdout, "\n\n");

qsort(table_entiers, NB_ENTIERS, sizeof(int), compare_entiers);

for (i = 0; i < NB_ENTIERS; i++)
    fprintf(stdout, "%05d ", table_entiers[i]);
fprintf(stdout, "\n");
return EXIT_SUCCESS;
}

```

Voici un exemple d'exécution.

\$./exemple-qsort

```

17767 09158 39017 18547 56401 23807 37962 22764 07977 31949 22714
55211 16882 07931 43491 57670 00124 25282 02132 10232 08987 59880
52711 17293 03958 09562 63790 29283 49715 55199 50377 01946 64358
23858 20493 55223 47665 58456 12451 55642 24869 35165 45317 41751
43096 23273 33886 43220 48555 36018 53453 57542 30363 40628 09300
34321 50190 07554 63604 34369 62753 48445 36316 61575 06768 56809
51262 54433 49729 63713 44540 09063 33342 24321 50814 10903 47594
19164 54123 30614 55183 42040 22620 20010 17132 31920 54331 01787
39474 52399 36156 36692 35308 06936 32731 42076 63746 18458 30974
47939

```

```

00124 01787 01946 02132 03958 06768 06936 07554 07931 07977 08987
09063 09158 09300 09562 10232 10903 12451 16882 17132 17293 17767
18458 18547 19164 20010 20493 22620 22714 22764 23273 23807 23858
24321 24869 25282 29283 30363 30614 30974 31920 31949 32731 33342
33886 34321 34369 35165 35308 36018 36156 36316 36692 37962 39017
39474 40628 41751 42040 42076 43096 43220 43491 44540 45317 47594
47665 47939 48445 48555 49715 49729 50190 50377 50814 51262 52399
52711 53453 54123 54331 54433 55183 55199 55211 55223 55642 56401
56809 57542 57670 58456 59880 61575 62753 63604 63713 63746 63790
64358
$

```

La routine `qsort()` de la bibliothèque Glibc étant pratiquement optimale, il est fortement conseillé d'y faire appel aussi souvent que possible, et de n'implémenter sa propre routine de tri que pour des applications vraiment spécifiques.

L'inconvénient que pose la fonction `bsearch()` est qu'il n'est pas facile d'ajouter simplement un élément si on ne le trouve pas. Pour cela, il faut insérer l'élément à la fin de la table par exemple, et invoquer `qsort()` pour la trier à nouveau. C'est intéressant si on peut grouper de multiples ajouts, mais peu efficace pour des ajouts isolés et fréquents. Voici donc une routine qui peut servir à ajouter un seul élément si on ne le trouve pas. Elle suppose que la table contient suffisamment de place pour adjoindre au moins une donnée.

```
void * b_insert (const void * cle, const void * table,
                size_t * nb_elements, size_t taille_element,
                int (* compare) (const void * lm1, const void * lm2))
{
    const char * element;
    int          comparaison;
    size_t       bas = 0;
    size_t       haut = (* nb_elements);
    size_t       milieu;

    while (bas < haut) {

        milieu = bas + (haut - bas) / 2;
        element = (void *) (((const char *) table)
                             + (milieu * taille_element));
        comparaison = compare(cle, element);
        if (comparaison < 0)
            haut = milieu;
        else if (comparaison > 0)
            bas = milieu + 1;
        else
            return ((void *) element);
    }
    // Ici, haut=bas, on n'a pas trouvé l'élément,
    // on va l'ajouter, mais il faut vérifier de quel
    // côté de l'élément "haut".
    if (haut >= (* nb_elements)) {
        element = (void *) (((const char *) table)
                             + ((* nb_elements) * taille_element));
    } else {
        element = (void *) (((const char *) table)
                             + (haut * taille_element));
        if (compare (cle, element) > 0) {
            element += taille_element;
            haut ++;
        }
        memmove((void *) element + taille_element,
                (void *) element,
                (* nb_elements) - haut);
    }
}
```



```
    }  
    memcpy((void *) element, cle, taille_element);  
    (* nb_elements) ++;  
    return (void *) element;  
}
```

La première partie de cette fonction est calquée sur la routine `bsearch()`, implémentée dans la Glibc. Ensuite, au lieu d'échouer et de renvoyer `d`, elle ajoute l'élément dans la table, en le positionnant au bon endroit.

Voici un programme qui utilise cette routine pour ajouter un caractère dans une chaîne lue en argument de ligne de commande. Nous ne répétons pas l'implémentation de la routine `b_insert()`.

```
exemple-qsort-2.c :  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
[...] // b_insert vu plus haut  
  
int compare_char (const void * l1, const void * l2)  
{  
    return ((char *)l1)[0] - ((char *)l2)[0];  
}  
  
int main (int argc, char * argv[])  
{  
    char * table = NULL;  
    size_t longueur;  
  
    if (argc != 3) {  
        fprintf(stderr, "syntaxe: %s table element\n", argv[0]);  
        exit(EXIT_FAILURE);  
    }  
  
    longueur = strlen(argv[1]);  
    if ((table = malloc(longueur + 2)) == NULL) {  
        perror("malloc");  
        exit(EXIT_FAILURE);  
    }  
    strcpy(table, argv[1]);  
  
    fprintf(stdout, "tri avec qsort ...\n");  
    qsort(table, strlen(table), 1, compare_char);  
    fprintf(stdout, "%s\n", table);  
  
    fprintf(stdout, "recherche / insertion de %c\n", argv[2][0]);
```

```
b_insert((void *) argv[2], table, & longueur, 1, compare_char);
table[longueur] = '\0';
fprintf(stdout, "%s\n", table);

return EXIT_SUCCESS;
}
```

Le premier argument est une chaîne de caractères, qu'on trie avec `qsort()` après l'avoir recopiée en réservant une place supplémentaire pour l'insertion.

La routine `b_insert()` ajoute ensuite le caractère se trouvant au début du second argument, s'il ne se trouve pas déjà dans la chaîne. Voici les différents cas de figure possibles.

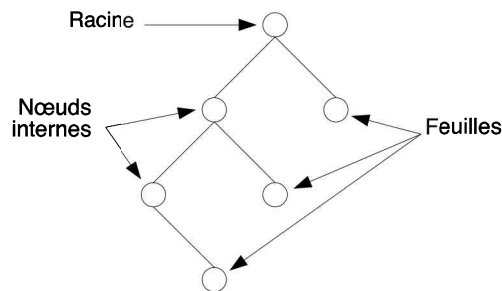
```
$ ./exemple-qsort-2 ertyuiop a
tri avec qsort ...
eioprtuy
recherche / insertion de a
aeioprtuy
$ ./exemple-qsort-2 ertyuiop z
tri avec qsort ...
eioprtuy
recherche / insertion de z
eioprtuyz
$ ./exemple-qsort-2 ertyuiop q
tri avec qsort ...
eioprtuy
recherche / insertion de q
eiopqrtuy
$ ./exemple-qsort-2 ertyuiop i
tri avec qsort ...
eioprtuy
recherche / insertion de i
eioprtuy
$
```

Il peut parfois être gênant de réimplémenter sa propre routine pour effectuer des ajouts, et on préférerait que cette fonctionnalité soit directement incorporée dans la bibliothèque C. On peut alors se tourner vers une autre structure de données, qui est entièrement gérée par des fonctions internes de la Glibc, et qui fournit des performances remarquables en termes de complexité : les arbres binaires.

Manipulation, exploration et parcours d'un arbre binaire

Un arbre binaire est une organisation de données très répandue en algorithmique. Il s'agit d'une représentation des éléments sous forme de nœuds, chacun d'eux pouvant avoir 0, 1, ou 2 nœuds fils. On représente généralement les arbres binaires avec, au sommet, un nœud particulier nommé racine, qui n'a pas de père. Les fils d'un nœud lui sont rattachés par un lien. Un nœud sans fils est nommé feuille.

Figure 17-2
Arbre binaire

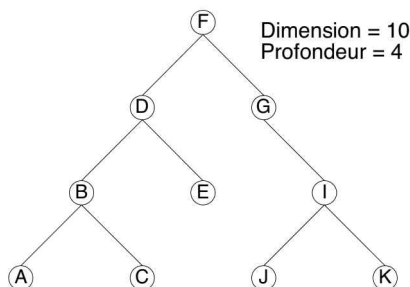


La dimension d'un arbre est égale au nombre de nœuds qui le composent, tandis que sa profondeur correspond à la plus grande distance qui sépare la racine d'un nœud feuille. Toutes ces notions sont assez intuitives dès qu'on a assimilé que notre arbre – tel ses congénères généalogiques – pousse la tête en bas...

Les arbres binaires ordonnés présentent de surcroît la particularité suivante :

- le fils gauche d'un nœud contient une valeur inférieure ou égale à celle de son père ;
- le fils droit d'un nœud comprend une valeur supérieure ou égale à celle de son père.

Figure 17-3
Arbre ordonné

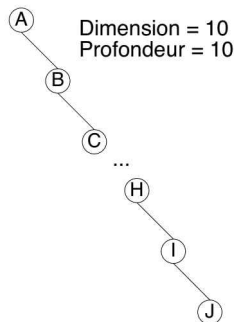


La recherche d'un élément dans un tel arbre nécessite donc, au maximum, un nombre de comparaisons égal à la profondeur de l'arbre, soit $\log(N)$, N étant sa dimension si l'arbre est correctement équilibré.

À première vue, il est aisé d'insérer une donnée, puisqu'il suffit de créer un nouveau nœud qu'on rattachera en tant que fils du dernier nœud feuille qu'on a rencontré lors de la vérification de l'existence de cet élément.

Malheureusement, cette technique n'est pas exploitable en réalité, car lors de l'ajout successif d'éléments déjà ordonnés, on va créer un arbre constitué uniquement de nœuds ayant un seul fils. L'arbre aura une profondeur égale à sa dimension, et la recherche d'un élément sera équivalente à une recherche séquentielle !

Figure 17-4
Insertion naïve d'éléments déjà
ordonnés



Pour éviter cela, il est nécessaire d'équilibrer l'arbre à chaque ajout ou suppression de nœud. La bibliothèque Glibc l'effectue automatiquement en utilisant un algorithme assez compliqué fondé sur un « coloriage » des nœuds en rouge ou en noir, et vérifiant à chaque modification l'équilibre de la structure complète. L'arbre restant donc équilibré, la recherche d'un élément croît donc suivant $\log(N)$ lorsque N augmente, ce qui est presque idéal. De plus, il est possible de parcourir automatiquement tout l'arbre suivant diverses méthodes, afin de le sauvegarder de manière ordonnée par exemple.

L'arbre est représenté en interne par des structures qui ne nous concernent pas. Pour l'utiliser, nous lui transmettrons simplement des pointeurs sur nos données, convertis en pointeur `void *`. La racine de l'arbre est aussi représentée par un pointeur de type `void *`, qu'on initialise à l'origine à `NULL` avant d'insérer des éléments. Cette insertion se fait en employant la routine `tsearch()`, déclarée ainsi dans `<search.h>` :

```
void * tsearch (const void * cle, void ** racine,
               comparison_fn_t compare);
```

Cette routine recherche la clé transmise et, si elle ne la trouve pas, l'insère dans l'arbre. La fonction renvoie un pointeur sur l'élément trouvé ou créé, ou `NULL` si un

problème d'allocation mémoire s'est présenté. On notera qu'on doit transmettre un pointeur sur la racine de l'arbre, elle-même définie comme un pointeur `void *`. En effet, la fonction peut à tout moment modifier cette racine pour réorganiser l'arbre.

Il existe une fonction `tfind()` permettant de rechercher un élément sans le créer s'il n'existe pas :

```
void * tfind (const void * cle, void ** racine,  
             comparison_fn_t compare);
```

Si la clé n'est pas rencontrée dans l'arbre, la fonction renvoie `NULL`.

Pour supprimer un élément, on utilise la fonction `tdelete()`, qui assure également le rééquilibrage de l'arbre :

```
void * tdelete (const void * cle, void ** racine,  
               comparison_fn_t compare);
```

L'élément est supprimé mais sa valeur est renvoyée par la fonction, sauf dans le cas où la clé n'a pas été trouvée, la routine retournant alors `NULL`.

Enfin, si on veut supprimer complètement un arbre, on peut utiliser la routine `tdestroy()`, qui est une extension *Gnu* déclarée ainsi :

```
void tdestroy (void * racine,  
              void (* liberation) (void * element));
```

La routine `liberation()` sur laquelle on passe un pointeur est invoquée sur chaque nœud de l'arbre, avec en argument la valeur du nœud.

Nous avons vu les fonctions d'insertion, de recherche et de suppression d'éléments dans un arbre binaire. Nous allons à présent examiner la routine `twalk()` qui permet de parcourir l'ensemble de l'arbre en appelant une fonction de l'application sur chaque nœud. Cette fonction doit être définie ainsi :

```
void action (const void * noeud, const VISIT methode,  
            const int profondeur);
```

Lorsqu'elle sera invoquée, elle recevra en premier argument un pointeur sur le nœud. Autrement dit, pour accéder aux données proprement dites, il faudra utiliser `**noeud`. Le second argument contient l'une des valeurs du type `enum VISIT` suivantes : `leaf`, `preorder`, `postorder` ou `endorder`, en fonction du moment où la fonction a été appelée. Nous détaillerons tout cela ci-après. Enfin, le troisième argument comprend la profondeur du nœud.

Pour chaque nœud interne, la routine d'action est donc appelée trois fois, et une fois pour chaque feuille de l'arbre. La fonction peut choisir d'agir ou non en fonction de la méthode avec laquelle elle a été appelée.

The diagram shows a tree structure with nodes labeled 'preorder', 'endorder', 'postorder', and 'leaf'. The root node is labeled 'preorder' and 'endorder'. It has two children: a left child labeled 'postorder' and a right child labeled 'preorder' and 'endorder'. The left child 'postorder' has a self-loop labeled 'leaf'. The right child 'preorder' and 'endorder' has two children: a left child labeled 'postorder' and a right child labeled 'leaf'. The left child 'postorder' has a self-loop labeled 'leaf'.

Puis, nous emploierons `twalk()` avec, à chaque fois, une sélection suivant une méthode particulière. Dans tous les cas, les feuilles `leaf` sont affichées.

```
example-tsearch.c :
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int compare_char (const void * l1, const void * l2)
```

```
{
    return strcmp(lm1, lm2);
}

static VISIT type_parcours;

void parcours (const void * noeud, const VISIT methode,
               const int profondeur)
{
    if (methode == type_parcours)
        fprintf(stdout, "%s ", * (char **) noeud);
    else if (methode == leaf)
        fprintf(stdout, "(%s) ", * (char **) noeud);
}

int main (void)
{
    int i;
    void * racine = NULL;
    char * chaines [] = {
        "A", "Z", "E", "R", "T", "Y", "U", "I", "O", "P",
        "Q", "S", "D", "F", "C", "H", "J", "K", "L", "M",
        "W", "X", "N", "V", "B", "G", "I", "N", NULL,
    };

    /* Insertion des chaînes dans l'arbre binaire */
    for (i = 0; chaines[i] != NULL; i++)
        if (tsearch(chaines[i], & racine, compare_char) == NULL) {
            perror("tsearch");
            exit(EXIT_FAILURE);
        }

    for (i = 0; chaines[i] != NULL; i++)
        if (tfind(chaines[i], & racine, compare_char) == NULL) {
            fprintf(stderr, "%s perdue ?\n", chaines[i]);
            exit(EXIT_FAILURE);
        }

    fprintf(stdout, "Parcours preorder (+ leaf) : \n ");
    type_parcours = preorder;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours postorder (+ leaf) : \n ");
    type_parcours = postorder;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours endorder (+ leaf) : \n ");
    type_parcours = endorder;
```

```

    twalk(racine, parcours);
    fprintf(stdout, "\n");

    fprintf(stdout, "Parcours leaf : \n ");
    type_parcours = leaf;
    twalk(racine, parcours);
    fprintf(stdout, "\n");

    return EXIT_SUCCESS;
}

```

Voici le résultat de l'exécution de ce programme.

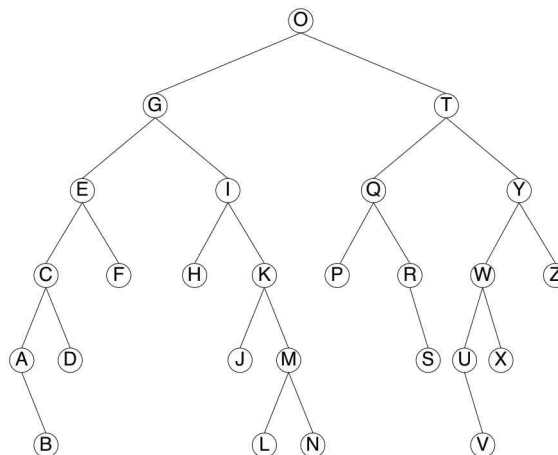
```

$ ./exemple-tsearch
Parcours preorder (+ leaf) :
  O G E C A (B) (D) (F) I (H) K (J) M (L) (N) T Q (P) R (S) Y W U (V) (X) (Z)
Parcours postorder (+ leaf) :
  A (B) C (D) E (F) G (H) I (J) K (L) M (N) O (P) Q R (S) T U (V) W (X) Y (Z)
Parcours endorder (+ leaf) :
  (B) A (D) C (F) E (H) (J) (L) (N) M K I G (P) (S) R Q (V) U (X) W (Z) Y T O
Parcours leaf :
  B D F H J L N P S V X Z
$

```

Nous vérifions bien que l'affichage ordonné des éléments s'obtient avec les méthodes `leaf` et `postorder`. On peut d'ailleurs, en observant ces résultats, retrouver la structure de l'arbre binaire interne géré par la bibliothèque C.

Figure 17-6
Arbre obtenu par insertion des
lettres de l'alphabet



Nous avons donc vu ici une structure de données souple donnant de bons résultats, tant en termes d'insertion de nouveaux éléments qu'en recherche de données. Nous allons à présent étudier les routines permettant de gérer des tables de hachage, puisqu'elles peuvent en théorie offrir une complexité constante, c'est-à-dire une durée de recherche n'augmentant pas quand le nombre de données croît de manière raisonnable.

Gestion d'une table de hachage

Une table de hachage est une structure de données particulière, dans laquelle on accède directement aux éléments en calculant leur adresse à partir de leur clé (c'est le principe des *dictionnaires* en Python). Cette organisation est particulièrement intéressante car le temps d'accès aux éléments ne dépend pas de la taille de la table. Elle est toutefois soumise à plusieurs contraintes.

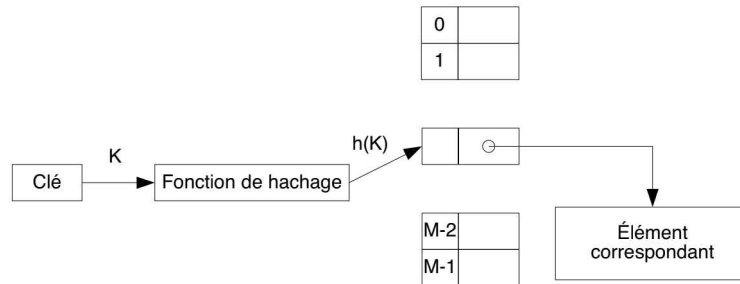
- Il faut indiquer le nombre maximal d'éléments dans la table dès sa création. Cette taille ne peut être modifiée ultérieurement.
- L'accès aux données est très efficace tant que le taux de remplissage de la table est assez faible (disons inférieur à 50 %). Les performances se dégradent par la suite et deviennent mauvaises à partir de 80 % de remplissage environ.
- Il n'est pas possible de supprimer un élément de la table. Si cette fonctionnalité est nécessaire, il faudra utiliser un indicateur dans le corps même des données, pour marquer l'élément comme « détruit ».

La table de hachage est donc une organisation idéale pour les éléments qu'on ajoute une seule fois, et dont le nombre maximal est connu dès le début. Par exemple, on utilise fréquemment une table de hachage dans les compilateurs pour stocker les mots-clés d'un langage, dans les éditeurs de liens pour mémoriser les adresses associées aux symboles, ou dans les vérificateurs orthographiques pour accéder au lexique d'une langue.

Le principe d'une table de hachage repose sur une fonction permettant de transformer la clé associée à un événement en une valeur pouvant servir d'adresse, d'indice dans une table. Cette fonction doit donc répartir les clés, le plus uniformément possible, dans l'intervalle compris entre 0 et le nombre maximal M d'éléments dans la table.

Les clés utilisées dans les tables de hachage gérées par la bibliothèque Glibc sont des chaînes de caractères. La bibliothèque appelle une fonction permettant de transformer la chaîne de caractères en un `unsigned int`. Elle utilise ensuite une première fonction de hachage constituée simplement de l'opération « modulo M ». La valeur résultante est employée alors comme adresse dans la table.

Figure 17-7
Gestion d'une table de hachage



Plusieurs chaînes de caractères différentes peuvent malheureusement se transformer en une valeur identique à l'issue de ce hachage. On dit alors qu'il y a une collision dans la table. Pour résoudre les collisions, plusieurs méthodes sont possibles.

- On utilise, pour chaque entrée dans la table, une liste de tous les éléments correspondants. Cette méthode est appelée chaînage séparé, elle augmente sensiblement la taille mémoire requise et le temps d'accès aux éléments. Toutefois, la taille initiale de la table n'est plus une limite stricte.
- Lors de l'insertion d'un élément, si sa place est déjà occupée, on vérifie l'emplacement suivant et, s'il est libre, on le prend. Sinon, on passe au suivant, et ainsi de suite, en revenant au début une fois la fin de la table atteinte, jusqu'à explorer toute la table. Cette méthode nommée hachage linéaire nécessite parfois de parcourir toute la table, et est donc inefficace lorsque le taux de remplissage est élevé.
- On peut utiliser le même principe que le hachage linéaire, mais en employant une seconde fonction de hachage pour parcourir la table, plutôt que de se déplacer d'un seul cran à chaque fois. Cette méthode dite de l'adressage ouvert permet de séparer plus facilement des chaînes qui donnaient le même résultat avec la première fonction de hachage. Elle conserve toutefois les mêmes défauts lorsque le taux de remplissage augmente.

La bibliothèque Glibc est implémentée en utilisant un adressage ouvert, avec une première fonction de hachage donnant simplement le modulo M de la valeur numérique obtenue avec la clé, et une seconde fonction de hachage valant $(1 + \text{la valeur de hachage précédente}) \text{ modulo } (M - 2)$. Cette fonction est suggérée par le paragraphe 6.4 de [KNUTH 1973c].

Pour que ces fonctions donnent des résultats satisfaisants, il est nécessaire que M soit un nombre premier. Lors de la création de la table, la bibliothèque C augmente donc silencieusement la valeur transmise au nombre premier le plus proche. Il ne faut donc pas s'étonner de pouvoir exceptionnellement dépasser la taille maximale de la table sans pour autant déclencher une erreur.

Nous comprenons également à présent pour quelle raison il n'est pas possible de détruire un élément d'une table de hachage, car sa présence peut avoir obligé un autre

élément à se placer plus loin dans la table à la suite d'une collision. Si on libère l'emplacement, la prochaine recherche aboutira à une adresse vide, et on en conclura que la clé recherchée n'existe pas.

Pour créer une table de hachage, on utilise la fonction `hcreate()`, déclarée dans `<search.h>` :

```
| int hcreate (size_t taille_maximale);
```

Cette routine crée une nouvelle table de hachage vide dans un espace de mémoire global et renvoie une valeur non nulle en cas de réussite. Si une table existe déjà ou s'il n'y a pas assez de mémoire disponible, la fonction échoue et renvoie 0.

Le fait que la table soit allouée dans une zone de mémoire globale pose deux problèmes :

- il n'est possible d'utiliser qu'une seule table de hachage à la fois ;
- l'emploi de la table par un programme multithreads n'est pas sûr.

Aussi, la bibliothèque Glibc propose-t-elle, sous forme d'extension Gnu, de gérer des tables distinctes en utilisant un pointeur transmis par le programme appelant, dans lequel les éléments nécessaires au stockage sont réunis. Le type correspondant à une table de hachage est `struct hsearch_data`, et on passe aux routines un pointeur sur cette variable. Les champs internes de la structure `hsearch_data` ne sont pas documentés, aussi utilisera-t-on `memset()` pour initialiser l'ensemble de la structure à zéro.

Les routines `hcreate_r()`, `hsearch_r()` et `hdestroy_r()`, permettant de traiter des tables de hachage explicitement passées en arguments, sont donc utilisables en programmation multithread.

```
| int hcreate_r (size_t taille_maxi, struct hsearch_data * table);
```

On l'utilisera donc ainsi :

```
| hsearch_data table;  
  
| memset (& table, 0, sizeof (table));  
| if (hcreate_r (nb_elements_maxi, & table) == 0) {  
|     perror ("hcreate_r");  
|     exit (1);  
| }
```

Lorsqu'on a fini d'utiliser une table, on la détruit avec `hdestroy()` :

```
| void hdestroy (void);
```

pour la table globale, ou :

```
| void hdestroy_r (struct hsearch_data * table);
```

pour les tables indépendantes.

Les éléments qu'on stocke dans une table de hachage sont du type ENTRY. Cette structure contient deux champs.

Type	Nom	Signification
char *	key	La clé qu'on utilise pour le hachage, consistant en une chaîne de caractères terminée par un zéro.
char *	data	Un pointeur sur une chaîne de caractères (ou sur tout autre type de données, avec la conversion char * pour l'initialisation). Ce pointeur est copié lorsqu'une entrée de la table est créée. Lorsqu'un élément est recherché et trouvé dans la table, un pointeur sur la structure ENTRY correspondante est renvoyé, contenant donc le champ data initial.

L'insertion ou la recherche se font avec la même fonction hsearch() :

```
| ENTRY * hsearch (ENTRY element, ACTION action);
```

ou :

```
| int hsearch_r (ENTRY element, ACTION action,  
                ENTRY ** retour, hsearch_data * table);
```

pour la version réentrante.

ACTION est un type énuméré pouvant prendre les valeurs :

- FIND : pour rechercher simplement l'élément correspondant à la clé. hsearch() renvoie un pointeur sur l'élément de la table ayant la clé indiquée, ou NULL s'il n'y a pas d'élément enregistré avec cette clé. hsearch_r() fournit cette même information dans l'argument retour, qui doit être l'adresse d'un pointeur sur un enregistrement. Si la clé n'est pas trouvée, hsearch_r() renvoie une valeur nulle ;
- ENTER : pour enregistrer l'élément ou mettre à jour son champ data s'il existe déjà. La fonction hsearch() renvoie un pointeur sur l'élément ajouté ou mis à jour, ou NULL en cas d'échec à cause d'un manque de mémoire. hsearch_r() fonctionne de la même manière et renvoie une valeur nulle en cas d'échec par manque de mémoire.

Nous allons mettre en pratique ces mécanismes avec un premier exemple qui va construire une table de hachage dont les enregistrements utilisent en guise de clé le libellé des jours de la semaine et des mois de l'année en français, et dont la partie data contient une chaîne équivalente en anglais.

```
exemple-hsearch.c :
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void ajoute_entree (char * francais, char * anglais)
{
    ENTRY entree;

    entree.key = strdup(francais);
    entree.data = strdup(anglais);
    if (hsearch(entree, ENTER) == NULL) {
        perror("hsearch");
        exit(EXIT_FAILURE);
    }
}

int main (int argc, char * argv[])
{
    int i;
    ENTRY entree;
    ENTRY * trouve;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s [mois | jour]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    /* 12 mois + 7 jours */
    if (hcreate(19) == 0) {
        perror("hcreate");
        exit(EXIT_FAILURE);
    }
    ajoute_entree("janvier", "january");
    ajoute_entree("fevrier", "february");
    ajoute_entree("mars", "march");
    ajoute_entree("avril", "april");
    ajoute_entree("mai", "may");
    ajoute_entree("juin", "june");
    ajoute_entree("juillet", "july");
    ajoute_entree("aout", "august");
    ajoute_entree("septembre", "september");
    ajoute_entree("octobre", "october");
    ajoute_entree("novembre", "november");
    ajoute_entree("decembre", "december");
    ajoute_entree("lundi", "monday");
    ajoute_entree("mardi", "tuesday");
    ajoute_entree("mercredi", "wednesday");
    ajoute_entree("jeudi", "thursday");
    ajoute_entree("vendredi", "friday");
}
```

```

ajoute_entree("samedi", "saturday");
ajoute_entree("dimanche", "sunday");

for (i = 1; i < argc; i++) {
    entree.key = argv[i];
    fprintf(stdout, "%s -> ", argv[i]);
    trouve = hsearch(entree, FIND);
    if (trouve == NULL)
        fprintf(stdout, "pas dans la liste \n");
    else
        fprintf(stdout, "%s\n", (char *)trouve->data);
}
hdestroy();
return EXIT_SUCCESS;
}

```

L'exécution du programme est conforme à nos attentes.

```

$ ./exemple-hsearch jeudi
jeudi -> thursday
$ ./exemple-hsearch janvier juillet dimanche samstag
janvier -> january
juillet -> july
dimanche -> sunday
samstag -> pas dans la liste
$

```

Nous allons également mettre en œuvre les extensions Gnu réentrantes, pour vérifier le fonctionnement de ces routines. L'exemple suivant fonctionne comme le précédent, avec la liste des départements français métropolitains. Nous forçons le pointeur `char * data` de la structure `ENTRY` à être manipulé comme un `int` représentant le numéro du département.

```

exemple-hsearch-r.c :
#define _GNU_SOURCE
#include <search.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void ajoute_entree (char * nom, long numero, struct hsearch_data * table)
{
    ENTRY entree;
    ENTRY * retour;

```

```
entree.key = strdup(nom);
entree.data = (char *) numero;
if (hsearch_r(entree, ENTER, & retour, table) == 0) {
    perror("hsearch_r");
    exit(EXIT_FAILURE);
}
}

int main (int argc, char * argv[])
{
    struct hsearch_data table;
    int i;
    ENTRY entree;
    ENTRY * trouve;

    if (argc < 2) {
        fprintf(stderr, "Syntaxe : %s nom-dept\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    memset(& table, 0, sizeof(table));
    if (hcreate_r(100, & table) == 0) {
        perror("hcreate");
        exit(EXIT_FAILURE);
    }

    ajoute_entree("ain", 1, & table);
    ajoute_entree("aisne", 2, & table);
    ajoute_entree("allier", 3, & table);
    ajoute_entree("alpes-de-haute-provence", 4, & table);
    [...]
    ajoute_entree("seine-saint-denis", 93, & table);
    ajoute_entree("val-de-marne", 94, & table);
    ajoute_entree("val-d'oise", 95, & table);

    for (i = 1; i < argc; i++) {
        entree.key = argv[i];
        fprintf(stdout, "%s -> ", argv[i]);
        if (hsearch_r(entree, FIND, & trouve, & table) == 0)
            fprintf(stdout, "pas dans la liste \n");
        else
            fprintf(stdout, "%ld\n", (long) (trouve->data));
    }
    hdestroy_r(& table);
    return EXIT_SUCCESS;
}
```

Le programme s'exécute de la façon suivante.

```
$ ./exemple-hsearch-r essonne val-de-marne seine gironde
essonne -> 91
val-de-marne -> 94
seine -> pas dans la liste
gironde -> 33
$
```

Récapitulatif sur les méthodes d'accès aux données

En définitive, nous avons vu plusieurs méthodes de structuration des informations avec l'assistance de la bibliothèque C. Nous allons essayer de dégager les avantages et les inconvénients de chacune de ces techniques. Il faut toutefois garder à l'esprit qu'une application bien conçue ne devra pas être tributaire de telle ou telle organisation, mais pourra au contraire évoluer pour utiliser une structure mieux adaptée si le besoin s'en fait sentir. Il est aisé de regrouper dans un module distinct les appels aux routines de la bibliothèque C, en les encadrant dans des fonctions générales d'initialisation de l'ensemble des données, d'ajout, de suppression et de recherche des éléments. Il est alors possible de changer de structure pour essayer d'améliorer les performances sans avoir besoin d'intervenir sur le reste de l'application. Voici donc un récapitulatif des méthodes étudiées ici.

Recherche linéaire, table non triée

- Organisation extrêmement simple. Ajout et suppression d'éléments très faciles.
- Performances intéressantes lorsque les données sont peu nombreuses (quelques dizaines d'éléments au maximum), et lorsqu'il y a de fréquentes modifications du contenu de la table.
- La fonction de comparaison ne doit pas nécessairement fournir une relation d'ordre, mais simplement une égalité entre les éléments ; l'implémentation peut donc parfois être optimisée en ce sens.
- En contrepartie, les performances sont très mauvaises lorsque la taille de la table augmente, puisqu'il faut en moyenne balayer la moitié des éléments présents. Une amélioration est toutefois possible si certaines données, peu nombreuses, sont réclamées très fréquemment.

Recherche dichotomique, table triée

- Accès très rapide aux données, on n'effectue au maximum que $\log_2(N)$ comparaisons pour trouver un élément dans une table en contenant N .
- L'ajout d'élément est compliqué, puisqu'il faut utiliser une routine personnalisée ou retrier la table après chaque insertion.
- Les performances sont donc optimales lorsque le nombre d'éléments est important (plusieurs centaines), et si l'insertion de données est un phénomène rare. L'idéal est de pouvoir regrouper plusieurs insertions en un lot qu'on traite en une seule fois avant de retrier les données.

Arbre binaire

- Accès rapide aux données, de l'ordre de $\log_2(N)$. L'insertion ou la suppression peuvent toutefois nécessiter des réorganisations importantes de l'arbre pour conserver l'équilibre.
- Les possibilités de parcours automatique de l'arbre peuvent permettre d'implémenter automatiquement des algorithmes divers nécessitant une exploration en profondeur d'abord. On peut aussi accéder aux données triées, dans le but de les sauvegarder dans un fichier par exemple.
- Cette méthode est donc une alternative intéressante par rapport aux tables triées et à la recherche dichotomique. La recherche peut être légèrement plus longue si l'arbre n'est pas parfaitement équilibré, mais il est facile d'ajouter ou de supprimer des éléments.

Table de hachage

- Principalement utilisée pour gérer des tables de chaînes de caractères qu'on désire retrouver rapidement.
- La taille de la table est fixée dès sa création et ne peut pas être augmentée. Pour que cette méthode soit vraiment efficace, la table doit être suffisamment grande pour éviter de dépasser un taux de remplissage de 60 à 80 %. L'occupation mémoire peut donc être importante.
- Il n'est pas nécessaire de fournir une fonction de comparaison, les clés sont des chaînes de caractères confrontées grâce à `strcmp()`.
- Il n'est pas possible de supprimer des éléments dans une table de hachage ni de la balayer pour en sauvegarder le contenu dans un fichier par exemple.

Conclusion

Nous verrons à nouveau des mécanismes d'organisation plus ou moins similaires dans le chapitre consacré aux bases de données conservées dans des fichiers.

Pour les lecteurs désireux d'approfondir le sujet des algorithmes de tri et voulant implémenter eux-mêmes des versions modifiées, la référence reste probablement [Knuth 1973c] *The Art of Computer Programming volume 3*. On trouvera dans [Bentley 1989] *Programming Pearls*, des exemples montrant l'importance du choix d'un bon algorithme dans ce type de routines.

Les concepts fondamentaux sont présentés dans [Hernert 2003] *Les algorithmes*. On pourra également trouver des idées intéressantes dans [Minoux 1986] *Graphes, algorithmes, logiciels*.

Routines avancées de traitement des blocs mémoire

Nous avons déjà observé un grand nombre de routines permettant d'accomplir les tâches les plus courantes du traitement de blocs ou de chaînes de caractères. Nous allons analyser ici deux types de traitements plus rares, mais également précieux : les expressions rationnelles, qui permettent de rendre une manipulation de chaînes beaucoup plus généraliste, et les techniques de cryptage plus ou moins élaborées des blocs de données.

Utilisation des expressions rationnelles

Ce qu'on appelle *expression rationnelle* (*regular expression* en anglais, aussi traduit par *expression régulière*) est en fait un motif contenant par exemple des caractères génériques (comme '*' ou '?' dans les commandes du shell), qu'on peut mettre en correspondance avec des chaînes de caractères précises. La syntaxe des expressions rationnelles peut être très compliquée, en gérant des répétitions, des OU logiques, etc.

La bibliothèque C nous offre des fonctions permettant de vérifier si une chaîne donnée correspond à un motif ou non. Les applications de ce principe sont nombreuses, de la recherche de noms de fichiers (`/usr/include/*.h`) à l'extraction d'une chaîne particulière dans un fichier de texte (comme avec `grep`).

Une bonne partie des applications courantes conservent, sous une forme ou une autre, une liste d'objets qu'elles manipulent. Ces objets sont souvent étiquetés à destination de l'utilisateur. Offrir à celui-ci la possibilité d'afficher, de sélectionner et de rechercher tous les objets dont le nom correspond à un motif donné peut améliorer sensiblement les performances d'une application.

Nous traiterons des fonctions permettant spécifiquement de rechercher les fichiers dont le nom correspond à un certain motif dans le chapitre consacré aux accès aux répertoires.

Les fonctions génériques de traitement des expressions rationnelles sont déclarées dans le fichier `<regex.h>`. Certaines de ces fonctions sont définies par SUSv4 (ayant été introduite auparavant par Posix.2), d'autres sont bien plus anciennes et spécifiques aux applications Gnu. Si on désire utiliser uniquement les fonctionnalités portables, il suffit de définir la constante `_POSIX_SOURCE` avant l'inclusion.

Il est difficile de donner une définition des expressions rationnelles sans entrer dans une description formelle et rébarbative. Aussi, nous laisserons le lecteur se reporter à la page de manuel `regex(7)` qui, à défaut d'être un modèle de clarté, présente l'avantage d'une exhaustivité quasi totale. On peut aussi examiner la documentation de l'utilitaire `grep`, qui est probablement le programme le plus populaire pour manipuler les expressions rationnelles.

Heureusement pour nous, le programmeur n'a aucunement besoin de connaître en détail la syntaxe des expressions rationnelles, puisque justement la bibliothèque C nous offre une interface avec ce format. Seul l'utilisateur final devra se pencher sur les arcanes de ces expressions. En fait, le programmeur devra s'y intéresser un minimum, ne serait-ce que pour rédiger la documentation de son application, mais nous échapperons à la description détaillée et formelle des expressions rationnelles. En fait, nous allons à la fin de ce paragraphe fournir un programme généraliste détaillant chaque option des routines à utiliser, mais sans avoir besoin de décrire précisément les mécanismes syntaxiques mis en œuvre.

Le principe adopté pour mettre en correspondance une chaîne avec un motif donné consiste en une première étape de compilation de l'expression rationnelle. Cette compilation permet de créer une représentation interne de l'expression afin de rendre possible une comparaison rapide par la suite. Le détail de la compilation n'est pas spécifié, il s'agit d'un choix d'implémentation de la bibliothèque C. La fonction de compilation est `regcomp()`, dont le prototype est :

```
int regcomp (regex_t * motif_compile, const char * motif,  
             int attributs);
```

Cette fonction prend en deuxième argument une chaîne de caractères contenant le motif à compiler, et remplit une structure de données opaque, de type `regex_t`, qu'on

passé en premier argument. On pourra ensuite utiliser le motif compilé représenté par la structure de type `regex_t` pour vérifier rapidement la correspondance avec une chaîne donnée.

Le troisième argument peut contenir un ou plusieurs attributs, représentés par des constantes symboliques qu'on associe avec un OU binaire :

Constante	Signification
REG_EXTENDED	Le motif doit être considéré comme une expression rationnelle au format étendu. Ceci correspond à l'option <code>-E</code> de <code>grep</code> . Dans les expressions rationnelles étendues, les caractères <code>?</code> , <code>+</code> , <code>{</code> , <code> </code> , <code>(</code> , et <code>)</code> ont une signification spéciale, alors que dans les expressions simples, il faut les préfixer avec « <code>\</code> » pour obtenir le même comportement.
REG_ICASE	Ignorer les différences entre minuscules et majuscules lors de la mise en correspondance.
REG_NOSUB	On ne désire pas conserver le contenu des sous-expressions mises en correspondance. Dans ce cas, on s'intéresse uniquement à la correspondance ou non d'un motif avec une chaîne, sans avoir besoin de savoir comment les sous-expressions sont remplies. Nous détaillerons ce mécanisme un peu plus loin.
REG_NEWLINE	Le caractère de saut de ligne rencontré dans une chaîne ne sera pas considéré comme un caractère ordinaire, mais prendra sa signification normale. En conséquence, les caractères spéciaux « <code>\$</code> » et « <code>^</code> » contenus dans un motif pourront être mis en correspondance respectivement avec les parties suivant et précédant le saut de ligne. Le caractère « <code>.</code> » ne peut plus correspondre au saut de ligne.

Lorsque la compilation réussit, `regcomp()` renvoie 0. Sinon elle renvoie une valeur d'erreur qu'on peut transmettre à la fonction `regerror()` dont le prototype est le suivant :

```
size_t regerror (int erreur, regex_t * motif_compile,  
                char * libelle, size_t taille_maxi);
```

Cette fonction analyse le code d'erreur passé en premier argument, ainsi que le pointeur sur le motif compilé (ou plutôt sur le motif dont la compilation a échoué) rempli par `regcomp()`. Elle en déduit un message d'erreur – malheureusement ne prenant pas encore en compte la localisation – dont elle copie, dans la chaîne passée en troisième argument, le nombre d'octets indiqué en dernier argument, caractère nul final compris. Si le message d'erreur n'a pas pu être copié en entier, il est tronqué. La fonction `regerror()` renvoie le nombre d'octets nécessaires pour stocker le message d'erreur, caractère nul compris. Il est donc possible de l'invoquer en deux passes, la première pour déterminer la longueur à allouer avec un libellé valant `NULL` et une taille maximale à zéro, la seconde pour remplir le message.

Lorsqu'on n'a pas précisé l'option `REG_NOSUB`, la bibliothèque C nous fournit des détails sur les correspondances effectuées, sans se contenter de nous dire si les chaînes concor-

dent. Ces informations sont stockées dans des structures de type `regmatch_t`, qu'il faut allouer avant la vérification. Au sein de ces structures, deux champs nous permettent de savoir quelle portion de la chaîne correspond à chaque sous-expression.

Le nombre de sous-expressions détectées est fourni dans le champ `re_nsub` du motif compilé, de type `regex_t`, après la réussite de `regcomp()`. Toutefois, il faut allouer un élément de plus, car la fonction de comparaison nous indique aussi la portion de chaîne correspondant à l'expression complète.

Une fois que la compilation est terminée, qu'on a alloué éventuellement un tableau de structures `regmatch_t` de la taille indiquée par le champ `re_nsub+1`, on peut appeler la fonction de comparaison `regexexec()`. Celle-ci a le prototype suivant :

```
int regexexec (regex_t * motif_compile, char * chaîne,  
               size_t nb_sous_expr, regmatch_t sous_expr [], int attribut);
```

Cette fonction compare la chaîne et le motif compilé, et renvoie zéro s'ils concordent. Sinon, elle renvoie une valeur pouvant être :

- `REG_NOMATCH` : pas de correspondance.
- `REG_ESPACE` : pas assez de mémoire pour traiter l'expression compilée. Ceci peut se produire à cause de récurrence dans les sous-expressions. Ce cas est très rare et doit quasiment être considéré comme une erreur fatale.

Lorsque la mise en correspondance réussit, `regexexec()` remplit `nb_sous_expr` éléments du tableau `sous_expr[]` avec les informations permettant de savoir quelles portions de la chaîne correspondent aux sous-expressions entre parenthèses du motif.

Les éléments du tableau `sous_expr[]` sont de structures `regmatch_t`, possédant deux champs qui nous intéressent :

- `rm_so` correspond à la position du premier caractère de la portion de chaîne mis en correspondance.
- `rm_eo` correspond à la position de la fin de la portion de chaîne mise en correspondance.

Le premier élément (d'indice 0) dans le tableau `sous_expr[]` correspond en fait à la portion équivalant à l'expression complète. Les éléments suivants concernent les sous-expressions successives.

Le dernier argument de `regexexec()` contient un attribut constitué d'un OU binaire entre les constantes suivantes :

- `REG_NOTBOL` : ne pas considérer le début de la chaîne comme un début de ligne. Le caractère spécial `$` ne s'appliquera donc pas à cet endroit.
- `REG_NOTEOL` : ne pas considérer la fin de la chaîne comme une fin de ligne. Le caractère spécial `^` ne s'y appliquera donc pas.

Enfin, une fois qu'on a terminé de traiter une expression rationnelle, il faut bien entendu libérer la table des sous-expressions qu'on a allouée, mais il faut également invoquer la fonction `regfree()` en lui passant en argument le pointeur sur le motif compilé. Cela permet à la bibliothèque de libérer toutes les données qu'elle a allouées dans cette structure lors de la compilation. Bien sûr, ces libérations ne sont importantes que si on souhaite à nouveau compiler une autre expression régulière, mais c'est quand même une bonne habitude à prendre pour éviter les fuites de mémoire.

Nous allons écrire un programme qui prend en argument une expression rationnelle, et qui tente de la mettre en correspondance avec les lignes qu'il lira successivement sur son entrée standard. De plus, ce programme acceptera un certain nombre d'options, qui seront transmises dans les attributs des fonctions `regcomp()` et `regexexec()`. Ces options sont :

Option	Argument équivalent	Fonction concernée
-e	REG_EXTENDED	regcomp()
-i	REG_ICASE	regcomp()
-s	REG_NOSUB	regcomp()
-n	REG_NEWLINE	regcomp()
-d	REG_NOTBOL	regexexec()
-f	REG_NOTEOL	regexexec()

Lorsque la correspondance réussit, le programme affiche les expressions et sous-expressions reconnues. Nous traitons toutes les fonctions décrites ci-dessus.

```
exemple-regcomp.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <regex.h>

void affiche_syntaxe (char * nom_prog)
{
    printf("Syntaxe : %s [options] motif\n", nom_prog);
    printf(" Options :\n");
    printf(" -e : expressions rationnelles etendues\n");
    printf(" -i : pas de differences majuscule/minuscule\n");
    printf(" -s : ne pas memoriser les sous-expressions\n");
    printf(" -n : gerer les sauts de lignes\n");
    printf(" -d : debut de chaine sans saut de ligne\n");
    printf(" -f : fin de chaine sans saut de ligne\n");
}

#define LG_MAXI 256
```

```
int main (int argc, char * argv[])
{
    int    option;
    char * liste_options = "eisndf";

    int    option_regcomp = 0;
    int    option_regexec = 0;
    regex_t motif_compile;

    int    erreur;
    char * message_erreur;
    size_t lg_message;

    size_t nb_sous_chaines = 0;
    regmatch_t * sous_chaines = NULL;

    char    ligne[LG_MAXI];
    char    sous_chaine[LG_MAXI];
    size_t lg_sous_chaine;
    int     i;

    opterr = 0; // Pas de message d'erreur de getopt()
    while ((option = getopt(argc, argv, liste_options)) != -1 ){
        switch (option) {
            case 'e' :
                option_regcomp |= REG_EXTENDED;
                break;
            case 'i' :
                option_regcomp |= REG_ICASE;
                break;
            case 's' :
                option_regcomp |= REG_NOSUB;
                break;
            case 'n' :
                option_regcomp |= REG_NEWLINE;
                break;
            case 'd' :
                option_regexec |= REG_NOTBOL;
                break;
            case 'f' :
                option_regexec |= REG_NOTEOL;
                break;
            case '?' :
                affiche_syntaxe (argv [0]);
                exit (1);
        }
    }
    if (argc - optind != 1) {
        // Il manque le motif
    }
}
```



```
    affiche_syntaxe(argv[0]);
    exit(EXIT_FAILURE);
}

erreur = regcomp(& motif_compile, argv[argc - 1],
                option_regcomp);
if (erreur != 0) {
    lg_message = regerror(erreur, & motif_compile, NULL, 0);
    message_erreur = malloc(lg_message);
    if (message_erreur == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    regerror(erreur, & motif_compile, message_erreur, lg_message);
    fprintf(stderr, "%s\n", message_erreur);
    free(message_erreur);
    exit(EXIT_FAILURE);
}

if ((option_regcomp & REG_NOSUB) == 0) {
    nb_sous_chaines = motif_compile.re_nsub + 1;
    sous_chaines = calloc(nb_sous_chaines, sizeof (regmatch_t));
    if (sous_chaines == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }
}

while (fgets(ligne, LG_MAXI, stdin) != NULL) {

    erreur = regexec(& motif_compile, ligne, nb_sous_chaines,
                    sous_chaines, option_regexec);
    if (erreur == REG_NOMATCH) {
        fprintf(stdout, "Pas de correspondance\n");
        continue;
    }
    if (erreur == REG_ESPACE) {
        fprintf(stderr, "Pas assez de memoire\n");
        exit(EXIT_FAILURE);
    }

    fprintf(stdout, "Correspondance Ok\n");

    if ((option_regcomp & REG_NOSUB) != 0)
        continue;

    for (i = 0; i < nb_sous_chaines; i++) {
        lg_sous_chaine=sous_chaines[i].rm_eo-sous_chaines[i].rm_so;
        strncpy (sous_chaine, ligne+sous_chaines[i].rm_so,
                lg_sous_chaine);
    }
}
```

```

        sous_chaine[lg_sous_chaine] = '\0';
        if (i == 0)
            fprintf (stdout, "expression : %s\n", sous_chaine);
        else
            fprintf (stdout, "ss-expr %02d : %s\n", i, sous_chaine);
    }
}

// Ces libérations seraient indispensables si l'on voulait
// compiler un nouveau motif
free(sous_chaines);
sous_chaines = NULL;
nb_sous_chaines = 0;
regfree(& motif_compile);
return EXIT_SUCCESS;
}

```

Voici quelques exemples d'exécution, mais nous encourageons le lecteur à expérimenter lui-même les différentes options des routines regcomp() et regex().

```

$ ./exemple-regcomp "a\(b*\)c\(de\)"
abcdefg
Correspondance Ok
expression : abcde
ss-expr 01 : b
ss-expr 02 : de
acdef
Correspondance Ok
expression : acde
ss-expr 01 :
ss-expr 02 : de
abbbbbbbbcdefg
Correspondance Ok
expression : abbbbbbbbcde
ss-expr 01 : bbbbbbbb
ss-expr 02 : de
acdf
Pas de correspondance
  (Contrôle-D)
$

```

Rappelons que dans les expressions rationnelles '*' signifie « zéro, une ou plusieurs répétitions du caractère précédent » et n'a donc pas son sens habituel avec le shell. Vérifions la non-différenciation majuscules/minuscules :

```
$ ./exemple-regcomp -i "a\(b*\)c\(de\)"
ABBBBCDEF
Correspondance Ok
expression : ABBBCDE
ss-expr 01 : BBB
ss-expr 02 : DE
  (Contrôle-D)
$
```

Avec l'option REG_NOSUB, on ne veut pas savoir comment la mise en correspondance se fait, mais juste avoir un résultat *Vrai* ou *Faux* :

```
$ ./exemple-regcomp -s "a\(b*\)c\(de\)"
abcdefg
Correspondance Ok
  (Contrôle-D)
$
```

Voyons un message d'erreur transmis par `regerror()` lors d'une erreur de compilation :

```
$ ./exemple-regcomp "a\(b*\)c\(de"
Unmatched ( or \
$
```

Enfin, avec l'option REG_EXTENDED, les expressions rationnelles sont étendues, ce qui signifie que les métacaractères prennent leur signification sans avoir besoin d'être précédés de `'\'` :

```
$ ./exemple-regcomp -e "a(b)*c(de)"
abbcdeff
Correspondance Ok
expression : abbcde
ss-expr 01 : b
ss-expr 02 : de
  (Contrôle-D)
$
```

Nous voyons que ces fonctions sont très puissantes puisqu'elles facilitent l'accès à des performances améliorées pour une application, sans nécessiter de développement complexe. Ces fonctionnalités sont en fait une extension naturelle des comparaisons de chaînes qu'on a pu étudier précédemment.

Il existe un équivalent BSD quasi obsolète puisqu'il utilise une zone de mémoire statique pour mémoriser le motif compilé. Cet ensemble est constitué par les routines `re_comp()` et `re_exec()`, déclarées dans `<re_comp.h>` :

```
char * re_comp (const char * motif);  
int   re_exec (const char * chaîne);
```

Ces fonctions n'étant pas utilisables dans un environnement multithread par exemple, il vaut mieux les éviter dorénavant.

Cryptage de données

Pour terminer cet ensemble de chapitres traitant de la manipulation des blocs de mémoire et des chaînes, nous allons consacrer un moment aux routines permettant le cryptage plus ou moins complexe de données.

Cryptage élémentaire

Tout d'abord, notons rapidement l'existence de la fonction `strfry()` :

```
char * strfry (char * chaîne);
```

Cette fonction est une extension Gnu qui utilise le générateur aléatoire `rand()` pour modifier la chaîne transmise et en créer un anagramme. Elle renvoie ensuite un pointeur sur cette même chaîne. L'utilité d'une telle fonction ne me saute pas vraiment aux yeux. Peut-être pour créer automatiquement des mots de passe ou des jeux de lettres ?

```
exemple-strfry.c :  
#define _GNU_SOURCE  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main (int argc, char * argv[])  
{  
    char * chaîne;  
  
    if (argc != 2) {  
        fprintf(stderr, "Syntaxe : %s chaîne\n", argv[0]);  
        exit(EXIT_FAILURE);  
    }  
}
```

```
    chaine = strdup(argv[1]);  
    strfry(chaine);  
    fprintf(stdout, "%s\n", chaine);  
    return EXIT_SUCCESS;  
}
```

```
$ ./exemple-strfry linux  
xnïlu  
$ ./exemple-strfry linux  
inïxu  
$ ./exemple-strfry linux  
nxuïi  
$
```

`memfrob()` peut être une fonction un petit peu plus utile. Cette extension Gnu dispose du prototype suivant :

```
void * memfrob (void * bloc, size_t taille);
```

Elle parcourt le bloc indiqué et effectue un *OU EXCLUSIF* binaire octet par octet avec la valeur magique 42 (en hommage, je suppose, à Douglas Adams). Bien sûr, lorsqu'on repasse la fonction une seconde fois sur le bloc, on retrouve exactement les données d'origine.

L'intérêt de cette routine est de dissimuler grossièrement des blocs de texte qu'on pourrait sinon trouver dans le fichier exécutable (par exemple, les listes de mots-clés et de commentaires dans les jeux).

Cryptage simple et mots de passe

La fonction de cryptage la plus simple disponible dans la bibliothèque C se nomme `crypt()`. Elle est utilisée pour la transformation des mots de passe. Son prototype est déclaré dans `<crypt.h>` :

```
char * crypt (const char * mot_passe, const char * prefixe);
```

Elle prend deux chaînes de caractères : le mot de passe lui-même, et un préfixe que nous préciserons ci-après. Elle renvoie une chaîne de caractères, allouée de manière statique, contenant le mot de passe crypté.

Le principe des mots de passe sous Unix consiste à utiliser un algorithme non réversible, transformant la chaîne claire en une bouillie illisible, mais reflétant le mot de passe initial. Lors d'une tentative de connexion, le mot de passe saisi est lui aussi

passé dans cet algorithme de cryptage et les deux bouillies sont alors comparées. Si elles sont égales, la connexion est acceptée. Cette méthode permet de ne conserver sur le système que des mots de passe déjà cryptés par l'intermédiaire d'un algorithme dont on ne connaît pas de fonction inverse.

La seule manière théorique d'attaquer le système est alors de se procurer un dictionnaire, de passer tous les mots dans la moulinette de cryptage, et de comparer les mots de passe cryptés avec chacun des résultats du dictionnaire. Cela pourrait être facilement exécutable, sans le préfixe qu'on ajoute. Ce préfixe a deux rôles. Tout d'abord, il permet de sélectionner entre deux types de cryptage, *MD5* ou *DES*, et il sert ensuite à perturber le cryptage. On veut éviter qu'un pirate puisse une fois pour toutes chiffrer à l'avance tout le dictionnaire et comparer les résultats avec les mots de passe cryptés. L'introduction d'un préfixe occupant au minimum deux caractères alphanumériques l'obligerait à crypter au minimum 4 096 dictionnaires. En fait, de plus en plus, le préfixe contiendrait plutôt 8 caractères imprimables aléatoires, ce qui nécessiterait de préparer 64^8 , c'est-à-dire 2^{48} , ou encore 200 000 milliards de dictionnaires.

De plus, sur les distributions Linux courantes, ce mécanisme est encore renforcé par l'utilisation des *shadow passwords*, grâce auxquels la liste des mots de passe cryptés n'est plus accessible à tous, mais uniquement à root.

Le hachage utilisant *MD5* est préférable à celui utilisant *DES* car il s'agit réellement d'une fonction à sens unique, ne permettant en aucun cas de retrouver le mot original à partir de la version cryptée. L'algorithme d'encryptage *MD5* est décrit en détail dans la *RFC 1321*, datant d'avril 1992. Ce document présente non seulement l'algorithme mais aussi des exemples de code d'implémentation. Pour utiliser le cryptage *MD5*, le préfixe à fournir doit obligatoirement commencer par les caractères « \$1\$ ». Ensuite, on trouve jusqu'à 8 caractères, de préférence aléatoires, choisis dans l'ensemble constitué des chiffres '0' à '9', des lettres 'A' à 'Z' et 'a' à 'z', ainsi que des caractères '.' et '/'. On peut éventuellement ajouter un '\$' à la fin du préfixe. Sinon, la fonction `crypt()` le rajoutera elle-même.

Pour utiliser le cryptage *DES*, on fournit un préfixe constitué de deux caractères seulement, pris dans l'ensemble décrit plus haut. Ce cryptage nécessite également que la bibliothèque Glibc ait été compilée avec un complément particulier. Si ce n'est pas le cas, lors de l'exécution du programme, la fonction `crypt()` renvoie une chaîne vide, et la variable globale `errno` contient le code `EOPNOTSUPP`.

La chaîne renvoyée par `crypt()` contient donc le préfixe fourni, intact, éventuellement complété d'un '\$' pour le MD5, suivi de la « bouillie » correspondant au cryptage du mot de passe.

Lors de l'emploi de la fonction `crypt()` sur un système acceptant le mécanisme *DES*, il faut utiliser la bibliothèque `libcrypt.so` au moment de l'édition des liens en ajoutant l'option `-lcrypt` sur la ligne de commande du compilateur.

Notre premier exemple va consister à crypter le mot de passe et le préfixe passés en arguments sur la ligne de commande, et à afficher le résultat (tel qu'on pourrait le trouver dans un fichier `/etc/passwd` ou `/etc/shadow`).

```
exemple-crypt.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <crypt.h>

int main (int argc, char * argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s mot_passe prefixe\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "%s\n", crypt(argv[1], argv[2]));
    exit(EXIT_SUCCESS);
}
```

Nous utilisons un préfixe arbitraire, qui aurait dû normalement être choisi aléatoirement. Nous créons un cryptage *MD5*, puis un cryptage *DES*.

```
$ ./exemple-crypt linux '$1$abcdefgh$'
$1$abcdefgh$4zfe08Fu5aAm8Xw6jDqzS/
$ ./exemple-crypt linux ab
abbw1AAT5v/7k
$
```

Dans le premier exemple, les apostrophes protègent la chaîne préfixe pour éviter que le shell n'interprète `$1` ou `$a`.

Notre second exemple va consister à vérifier si le mot de passe transmis en premier argument correspond bien au cryptage fourni en second argument. Nous pouvons directement passer à la fonction `crypt()` le mot de passe crypté en guise de préfixe, elle ne prendra en considération que les caractères qui la concernent.

```
exemple-crypt-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <crypt.h>
```

```

int main (int argc, char * argv[])
{
    char * cryptage;
    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s mot_passe bouillie\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    cryptage = crypt(argv[1], argv[2]);
    if (strcasecmp(cryptage, argv[2]) == 0)
        fprintf(stdout, "Verification Ok\n");
    else
        fprintf(stdout, "Mauvais mot de passe\n");
    return EXIT_SUCCESS;
}

```

Nous allons vérifier un cryptage MD5 et un DES provenant de l'exemple précédent, puis nous modifierons la dernière lettre du mot de passe crypté afin de faire échouer la comparaison.

```

$ ./exemple-crypt-2 linux '$1$abcdefgh$4zfe08Fu5aAm8Xw6jDqzS/'
Verification Ok
$ ./exemple-crypt-2 linux 'abbwIAAT5v/7k'
Verification Ok
$ ./exemple-crypt-2 linux '$1$abcdefgh$4zfe08Fu5aAm8Xw6jDqzT/'
Mauvais mot de passe
$

```

La fonction `crypt()`, utilisant une chaîne de caractères statique pour renvoyer son résultat, n'est pas utilisable dans un environnement multithread ou au sein d'un gestionnaire de signaux. Pour pallier ce problème, la Glibc offre une extension Gnu nommée `crypt_r()`, dont le prototype est le suivant :

```

char * crypt_r (const char * mot_de_passe, const char * prefixe,
                 struct crypt_data * cryptage);

```

Le dernier argument est un pointeur sur une structure contenant suffisamment de place pour stocker le mot de passe crypté. Avant d'appeler cette routine, il faut mettre à zéro le champ `initialized` de cette structure ainsi :

```

struct crypt_data    cryptage;

cryptage.initialized = 0;

resultat = crypt_r(mot_passe, prefixe, & cryptage);

```


On notera bien que la fonction `crypt()` ne peut servir qu'à un cryptage de mot de passe. La fonction n'étant pas réversible, on ne peut pas récupérer les données initiales.

Cryptage de blocs de mémoire avec DES

La bibliothèque Glibc offre la possibilité de chiffrer des blocs de mémoire en utilisant l'algorithme *DES*. Ce système, mis au point par IBM dans les années soixante-dix, fonctionne sur le principe d'une clé privée. Il a été décrit dans le document *FIPS 46-1*, publié en 1988 par le gouvernement américain, et est équivalent à l'algorithme décrit sous le nom *DEA Ansi X3.92-1981*.

DES fonctionne en cryptant des blocs de données de 64 bits en utilisant une clé longue de 56 bits. Cette clé comportant des bits de parité, elle s'étend également sur une longueur totale de 64 bits.

Il n'est pas question de construire une véritable application cryptographique en utilisant ces fonctions. Tout d'abord, *DES* se servant d'un système de clé privée, il est nécessaire de disposer d'un canal de communication sûr pour transmettre la clé de décodage à son interlocuteur, ce qui est souvent aussi compliqué que d'envoyer tout le message de manière sécurisée. De plus, *DES* fonctionne avec des clés de 56 bits, et la plupart des services de renseignements disposent d'ores et déjà de machines capables de décrypter un message en employant la force brute (essayer toutes les clés possibles jusqu'à obtention d'un texte lisible), et cela dans un temps raisonnable. Ce système cryptographique repose en effet sur l'idée qu'un décodage par force brute nécessite un investissement informatique et un temps de calcul rédhibitoires. Bien entendu, la validité de ces deux paramètres est difficile à estimer.

Pour éviter ces désagréments, on emploiera dans des applications cryptographiques des bibliothèques fonctionnant avec d'autres systèmes plus sûrs comme *AES* qui n'est pas implémenté dans la Glibc, mais est disponible dans plusieurs bibliothèques sous licences libres.

Tout en étant conscient de toutes les limitations de sécurité inhérentes à l'emploi de *DES*, nous pouvons toutefois vouloir l'employer dans une application pour, par exemple, crypter le contenu d'un fichier de données dans une application comptable, masquer l'identité des patients dans les dossiers d'un système d'aide au diagnostic médical, ou encore dissimuler le contenu d'une application d'agenda électronique.

En fait, de par sa nature de système à clé privée, *DES* est surtout utilisable dans des environnements où le même utilisateur cryptera et décryptera les données. Son intérêt principal réside dans le verrouillage de fichiers qui restent ainsi illisibles, même pour l'administrateur root. *DES* est peu recommandé lorsque les données doivent être transmises à un correspondant, à cause du problème posé par la communication de la clé.

Comme nous l'avons déjà expliqué, l'algorithme *DES* utilise une clé privée de 64 bits et chiffre un bloc de 64 bits pour produire un nouveau bloc de 64 bits. Il existe dans la bibliothèque Glibc des fonctions de bas niveau, `setkey()`, `encrypt()`, `setkey_r()` et `encrypt_r()`, dont l'utilisation est particulièrement pénible car elles manipulent les blocs de 64 bits sous forme de tables de 64 caractères, chaque caractère représentant un seul bit à la fois.

Heureusement, il existe deux fonctions de plus haut niveau qui nous simplifient le travail, `ecb_crypt()` et `cbc_crypt()`. ECB signifie « *Electronic Code Book* » et CBC « *Cipher Block Chaining* ». Il s'agit de modes opératoires différents pour la normalisation de *DES*. Ces fonctions servent toutes deux à chiffrer ou à déchiffrer un bloc, mais `cbc_crypt()` assure un niveau de plus de chiffrage. Cette fonction effectue en effet un *OU EXCLUSIF* sur les blocs avant de les chiffrer, en changeant la valeur à chaque bloc. Il existe donc une chaîne de 8 octets supplémentaires à conserver avec les données, mais l'algorithme est beaucoup moins sensible à une cryptanalyse si plusieurs blocs originaux sont semblables.

La fonction `ecb_crypt()` est déclarée dans `<rpc/des_crypt.h>` ainsi :

```
int ecb_crypt (char * cle, char * bloc, unsigned longueur,
               unsigned mode)
```

La clé est transmise en premier argument sous forme de bloc de 8 octets. Les bits de parité de la clé doivent être positionnés correctement. Ceci est assuré par une fonction supplémentaire que nous verrons plus bas. La fonction chiffre ou déchiffre les blocs situés à partir de l'adresse transmise en second argument, jusqu'à la longueur indiquée. Cette longueur doit être un multiple de 8. Les blocs chiffrés remplacent les blocs originaux.

Le mode indiqué en dernier argument est constitué par un OU binaire entre les constantes symboliques suivantes :

Constante	Signification
DES_ENCRYPT	On désire crypter les données.
DES_DECRYPT	On désire décrypter les données. Bien entendu, une seule de ces deux constantes doit être indiquée.
DES_HW	Essayer d'utiliser un coprocesseur de chiffrement DES s'il en existe un sur la machine. Ceci peut améliorer sensiblement la vitesse du cryptage. Si aucun coprocesseur n'est disponible, le chiffrement sera fait de manière logicielle.
DES_SW	Ne pas utiliser de coprocesseur de cryptage, même s'il en existe un sur le système. Cette option peut servir à garantir que les données ne pourront pas être interceptées avec un coprocesseur truqué installé par un administrateur peu scrupuleux.

En retour, `ecb_crypt()` renvoie l'une des constantes symboliques suivantes :

Code	Signification
<code>DESERR_NONE</code>	Cryptage réussi.
<code>DESERR_NOHWDEVICE</code>	Cryptage réussi de manière logicielle, pas de coprocesseur disponible.
<code>DESERR_HWERROR</code>	Échec de cryptage dû au coprocesseur ou à l'absence du supplément « crypt » lors de la compilation de la bibliothèque C.
<code>DESERR_BADPARAM</code>	Échec de cryptage dû à de mauvais paramètres, notamment si la longueur indiquée n'est pas un multiple de 8.

Pour éviter d'avoir à tester plusieurs cas, la macro `DES_FAILED(int erreur)` prend une valeur non nulle si l'erreur est l'une des deux dernières constantes symboliques.

Le fonctionnement de `cbc_crypt()` est exactement le même, mais avec un argument supplémentaire :

```
int cbc_crypt (char * cle, char * bloc, unsigned longueur,  
              unsigned mode, char * vecteur);
```

Le vecteur est un bloc de 8 octets qui sera associé par un *OU EXCLUSIF* au premier bloc avant son chiffrement. Ensuite, le premier bloc crypté est utilisé à nouveau dans un *OU EXCLUSIF* avec le second bloc avant son chiffrement, et ainsi de suite. Le phénomène inverse a lieu lors du décryptage des blocs. On emploie souvent un bloc composé de 8 octets choisis aléatoirement en guise de vecteur initial. Il faut alors conserver ce vecteur avec les données cryptées, afin de pouvoir les décoder. Une autre solution consiste à utiliser une valeur constante, par exemple 8 octets à zéro, mais à employer un premier bloc rempli aléatoirement. Bien entendu, ces deux méthodes sont équivalentes, mais dans la seconde, le vecteur aléatoire fait partie intégrante des données cryptées.

Nous avons indiqué que la clé de chiffrage devait disposer de bits de parité correctement positionnés. Pour cela, il existe une fonction d'aide, `des_setparity()`, dont le prototype est :

```
void des_setparity (char * cle);
```

On transmet à cette fonction la clé qui nous a été donnée par l'utilisateur par exemple, et elle s'occupe de placer comme il le faut les 8 bits de parité en fonction des 56 bits efficaces de la clé. Les parités sont représentées par les bits de poids faibles de chaque octet.

L'exemple que nous allons présenter ici utilisera la fonction `ecb_crypt()` pour ne pas compliquer inutilement le code avec la gestion du vecteur initial. Le programme que nous allons créer sera appelé de deux manières différentes, avec l'aide d'un lien sym-

bolique. Il étudiera son argument numéro 0 pour savoir s'il a été invoqué sous le nom `des-crypte` ou `des-decrypte`, et adaptera alors l'argument `mode` de `ecb_crypt()`.

Notre programme prend n'importe quelle clé passée sur la ligne de commande, en la limitant à 8 caractères. Nous savons que `strncpy()` complétera la clé avec des zéros si elle a moins de 8 caractères. Nous ne recommandons pas l'utilisation directe de la clé fournie par l'utilisateur, car les clés saisies par un être humain restent dans le domaine des caractères imprimables et ont une entropie bien plus faible qu'un choix aléatoire dans l'espace allant de 0 à 255. Pour une application importante, il faudrait résoudre ce problème.

Nous arrondissons la taille du fichier à crypter au multiple de 8 supérieur. Puis, nous projetons ce fichier en mémoire avec `mmap()`. Nous pouvons alors appeler `ecb_crypt()` pour effectuer le codage.

```
exemple-ecb-crypt.c :
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <rpc/des_crypt.h>

int main (int argc, char * argv[])
{
    char * nom_programme;
    int    fichier;
    struct stat estat_fichier;
    long   taille_fichier;
    char *  projection;
    char    cle [8];
    unsigned mode;
    int      retour;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe %s : fichier cle\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    nom_programme = basename(argv[0]);
    if (strcasecmp(nom_programme, "des-decrypte") == 0)
        mode = DES_DECRYPT;
    else
        mode = DES_ENCRYPT;
```

```

if ((fichier = open(argv[1], O_RDWR)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
if (stat(argv[1], & etat_fichier) != 0) {
    perror("stat");
    exit(EXIT_FAILURE);
}
taille_fichier = etat_fichier.st_size;
taille_fichier = ((taille_fichier + 7) >> 3) << 3;
projection = (char *) mmap(NULL, taille_fichier,
                           PROT_READ | PROT_WRITE,
                           MAP_SHARED, fichier, 0);

if (projection == (char *) MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}
close(fichier);

strncpy(cle, argv[2], 8);
des_setparity(cle);

retour = ecb_crypt(cle, projection, taille_fichier, mode);
if (DES_FAILED(retour)) {
    perror("ecb_crypt");
    exit(EXIT_FAILURE);
}
munmap(projection, taille_fichier);
return EXIT_SUCCESS;
}

```

Voici l'utilisation du programme :

```

$ make
ln -sf exemple-ecb-crypt des-crypte
ln -sf exemple-ecb-crypt des-decrypte
cc -Wall -g -lcrypt   exemple-ecb-crypt.c   -o exemple-ecb-crypt
$ cat > fichier_a_crypter
Voici un fichier
que nous allons
crypter avec la
bibliotheque DES
(Contrôle-D)
$ ./des-crypte fichier_a_crypter cle-linux
$ cat fichier_a_crypter
##??## ?ik-?? ?m?-?|,?Ip?.Rg*??34Xl#?)8##&Hs#?Vj?mK#?#?-"?J$

```

```
$ ./des-decrypte fichier_a_crypter cle-linux
$ cat fichier_a_crypter
Voici un fichier
que nous allons
crypter avec la
bibliotheque DES
$
```

Nous voyons bien que le fichier crypté était totalement illisible. Ce petit programme souffre beaucoup du manque d'efficacité dans la création de la clé, mais ce défaut mis à part, il pourrait servir de modèle pour la création d'un utilitaire permettant de dissimuler le contenu de fichiers personnels, même pour l'administrateur root.

Conclusion

Ce chapitre nous a permis de détailler l'emploi de routines particulièrement puissantes de la Glibc pour manipuler des expressions régulières ou crypter des données.

La cryptographie est un domaine complexe, nécessitant de solides connaissances mathématiques. On en trouvera une présentation claire dans [Beckett 1990] *Introduction aux méthodes de la cryptologie*.

19

Types de données et conversions

Types de données génériques

Les types de données connus par le compilateur C sous Linux sont les suivants : char, short int, int, long int, long long int, float, double, long double et void*. On peut y ajouter les variantes unsigned des types entiers, mais elles ont la même taille que leur équivalent signed. Le type long long int est une extension par rapport au C Ansi.

La taille nécessaire pour stocker les données est déterminée à l'aide de la fonction sizeof(). On notera qu'il ne s'agit pas d'une fonction de bibliothèque, mais d'un opérateur du langage C appartenant à la liste de ses mots-clés, au même titre que for, if, switch...

Voici la taille des données génériques sur un PC sous Linux, avec les options standards du compilateur :

Type	x86 32 bits	x86 64 bits
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8

Type	x86 32 bits	x86 64 bits
float	4	4
double	8	8
long double	12	16
void *	4	8

Pour certains types entiers, les valeurs minimale et maximale sont définies sous forme de constantes symboliques dans `<limits.h>`. Il y a cette fois une différence entre les types entiers signés et non signés. Bien entendu, les types non signés commencent tous à 0. Voici les noms des constantes symboliques représentant les limites, ainsi que leurs valeurs sur architecture x86.

Les valeurs entre parenthèses s'appliquent à l'architecture x86 64 bits lorsqu'il y a une différence par rapport au x86 32 bits.

Type	Nom limite	Valeur limite
signed char	SCHAR_MIN	-128
	SCHAR_MAX	127
unsigned char	UCHAR_MAX	255
signed short int	SHRT_MIN	-32 768
	SHRT_MAX	32 767
unsigned short int	USHRT_MAX	65 535
signed int	INT_MIN	-2 147 483 648
	INT_MAX	2 147 483 647
unsigned int	UINT_MAX	4 294 967 295
signed long int	LONG_MIN	-2 147 483 648 (-9 223 372 036 854 775 808)
	LONG_MAX	2 147 483 647 (9 223 372 036 854 775 807)
unsigned long int	ULONG_MAX	4 294 967 295 (18 446 744 073 709 551 615)

À partir de ces types, la bibliothèque C définit, par typedef ou #define, tous les types spécifiques qu'on peut rencontrer, comme `size_t`, `time_t`, etc. Certains d'entre eux sont des structures, comme `rusage` que nous avons vue dans le chapitre 9, ou des unions, comme `sigval` que nous avons rencontrée dans le chapitre 13.

Types de tailles définies

Lorsqu'on doit assurer une bonne portabilité d'un programme entre systèmes d'exploitation différents, et que le format de stockage des données importe peu, les types décrits précédemment conviennent parfaitement. Néanmoins, il y a des situations où la taille des données doit être prise en compte. On préférera alors les types décrits dans `<stdint.h>` :

Taille (octets)	Type signé	Type non signé
1	<code>int8_t</code>	<code>uint8_t</code>
2	<code>int16_t</code>	<code>uint16_t</code>
4	<code>int32_t</code>	<code>uint32_t</code>
8	<code>int64_t</code>	<code>uint64_t</code>

En outre, ce fichier d'en-tête contient des types spéciaux pour permettre des *casts* entre pointeurs et valeurs d'adresses entières ainsi que des constantes symboliques définissant les bornes d'utilisation d'un type (par exemple, `INT32_MIN` et `INT32_MAX`).

Catégories de caractères

Les caractères représentent le bloc fondamental sur lequel repose tout le dialogue avec l'utilisateur. Un programme peut manipuler en interne des entiers, des réels, ou même des objets structurés complexes, mais dans tous les cas les saisies et les affichages se feront par l'intermédiaire de caractères. Il est donc normal qu'il existe une quinzaine de fonctions faisant partie du C Ansi, permettant de préciser l'appartenance d'un caractère à une ou plusieurs catégories bien définies. Ces fonctions permettent par exemple de s'assurer qu'un caractère est bien une majuscule, un chiffre, un symbole affichable, etc.

Le prototype général de ces routines, déclarées dans `<ctype.h>`, est le suivant :

```
| int is<TYPE> (int caractere);
```

La valeur passée en argument doit correspondre à celle d'une donnée de type `char` ou à la rigueur à la valeur `EOF`. Cela permet de traiter directement la sortie d'une routine comme `getchar()`.

Nom	Type de caractères
<code>isalnum()</code>	Caractère alphanumérique, lettre ou chiffre.
<code>isalpha()</code>	Caractère alphabétique. Dans la localisation C par défaut, il s'agit uniquement des lettres A-Z et a-z sans accentuation.

Nom	Type de caractères
<code>isascii()</code>	Caractère appartenant au standard Ascii (compris entre 0 et 127). La table Ascii est rappelée en annexe.
<code>isblank()</code>	Caractère blanc, c'est-à-dire une espace ou une tabulation.
<code>iscntrl()</code>	Caractère de contrôle non imprimable.
<code>isdigit()</code>	Chiffre décimal.
<code>isgraph()</code>	Caractère imprimable ayant un symbole non blanc.
<code>islower()</code>	Lettre minuscule. Dans la localisation C par défaut, les minuscules accentuées ne sont pas comprises dans cette catégorie.
<code>isprint()</code>	Caractère imprimable, c'est-à-dire un caractère graphique ou une espace.
<code>ispunct()</code>	Caractère de ponctuation. Cela recouvre les caractères graphiques non alphanumériques.
<code>isspace()</code>	Caractère d'espacement comprenant par exemple les tabulations horizontale et verticale, le saut de ligne, le retour chariot ou le saut de page.
<code>isupper()</code>	Caractère majuscule.
<code>isxdigit()</code>	Chiffre hexadécimal.

Une routine de cet ensemble est particulièrement précieuse pour analyser le résultat d'une fonction de saisie ou pour afficher correctement des données binaires, comme nous le verrons dans le chapitre 18.

Attention, les routines `is<TYPE>()` comme les trois routines `to<TYPE>()`, que nous verrons dans la prochaine section, peuvent être implémentées – dans d'anciennes bibliothèques C – sous forme de macros définies dans `<ctype.h>`, évaluant plusieurs fois leurs arguments.

Il faut donc éviter tout effet de bord, comme dans

```
while (i < strlen(saisie))
    if (! isdigit(saisie[i++]))
        return -1;
```

qui risque de ne vérifier qu'un caractère sur deux si `isdigit()` est implémentée ainsi :

```
#define isdigit(x) ((x >= '0') && (x <= '9'))
```

Le programme suivant permet d'examiner les caractéristiques des caractères saisis en entrée :

```
exemple-is.c :
#include <ctype.h>
#include <locale.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

void affiche_caracteristiques (int c)
{
    fprintf(stdout, "%02X : ", (unsigned char) c);
    if (isalnum(c)) fprintf(stdout, "alphanumerique ");
    if (isalpha(c)) fprintf(stdout, "alphabetique ");
    if (isascii(c)) fprintf(stdout, "ascii ");
    if (iscntrl(c)) fprintf(stdout, "controle ");
    if (isdigit(c)) fprintf(stdout, "chiffre ");
    if (isgraph(c)) fprintf(stdout, "graphique ");
    if (islower(c)) fprintf(stdout, "minuscule ");
    if (isprint(c)) fprintf(stdout, "imprimable ");
    if (ispunct(c)) fprintf(stdout, "ponctuation ");
    if (isspace(c)) fprintf(stdout, "espace ");
    if (isupper(c)) fprintf(stdout, "majuscule ");
    if (isxdigit(c)) fprintf(stdout, "hexadecimal ");
    fprintf(stdout, "\n");
}

int main (void)
{
    int caractere;
    setlocale(LC_ALL, "");
    while ((caractere = getchar()) != EOF)
        affiche_caracteristiques(caractere);
    return EXIT_SUCCESS;
}
```

Nous allons observer les résultats de ces fonctions :

```
$ ./exemple-is
az 1      é
61 : alphanumerique alphabetique ascii graphique minuscule imprimable
hexadecimal
7A : alphanumerique alphabetique ascii graphique minuscule imprimable
20 : ascii imprimable espace
31 : alphanumerique ascii chiffre graphique imprimable hexadecimal
09 : ascii controle espace
C3 :
A9 :
0A : ascii controle espace
```

Nous remarquons que le caractère a (61) est considéré comme une lettre, mais aussi comme un chiffre hexadécimal, ce qui n'est pas le cas de z (7A). L'espace (20) est

imprimable, alors que la tabulation (09) entre le 1 et le é est considérée comme un caractère de contrôle, au même titre que le retour chariot (0A) en fin de saisie.

Le caractère é est codé sur deux octets (C3 / A9). Cela est une propriété de l'encodage UTF-8 que nous avons observé au chapitre 16. Comme nous l'avons vu, on utilise alors une conversion dans une représentation interne spéciale : les caractères larges `wchar_t`.

On notera qu'en Python, les méthodes `isalnum()`, `isalpha()`, `isdecimal()`, `isdigit()`, `islower()`, `isnumeric()`, `isprintable()` et `isupper()` s'appliquent sur des chaînes de caractères et renvoient une valeur vraie si tous les caractères de la chaîne correspondent au type indiqué.

Nous disposons des fonctions suivantes, déclarées dans `<wctype.h>` jouant le même rôle que les précédentes pour les caractères larges :

```
int iswalnum (wint_t wc);
int iswalpha (wint_t wc);
int iswcntrl (wint_t wc);
int iswdigit (wint_t wc);
int iswgraph (wint_t wc);
int iswlower (wint_t wc);
int iswprint (wint_t wc);
int iswpunct (wint_t wc);
int iswspace (wint_t wc);
int iswupper (wint_t wc);
int iswxdigit (wint_t wc);
```

On peut noter que ces routines ne travaillent pas sur un type `wchart_t`, mais sur un `wint_t`. Ce type, introduit pour jouer un rôle symétrique au `int` par rapport au `char`, peut contenir n'importe quel caractère large ainsi que la valeur spéciale `WEOF` représentant la fin d'un fichier.

Voyons un exemple d'utilisation, calqué sur le précédent :

```
exemple-isw.c :
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

void affiche_caracteristiques (wint_t c)
{
    fprintf(stdout, "%lc : ", c);
    if (iswalnum(c)) fprintf(stdout, "alphanumerique ");
    if (iswalpha(c)) fprintf(stdout, "alphabetique ");
```

```

    if (iswcntrl(c)) fprintf(stdout, "controle ");
    if (iswdigit(c)) fprintf(stdout, "chiffre ");
    if (iswgraph(c)) fprintf(stdout, "graphique ");
    if (iswlower(c)) fprintf(stdout, "minuscule ");
    if (iswprint(c)) fprintf(stdout, "imprimable ");
    if (iswpunct(c)) fprintf(stdout, "ponctuation ");
    if (iswspace(c)) fprintf(stdout, "espace ");
    if (iswupper(c)) fprintf(stdout, "majuscule ");
    if (iswxdigit(c)) fprintf(stdout, "hexadecimal ");
    fprintf(stdout, "\n");
}

int main (void)
{
    wint_t caractere;
    setlocale(LC_ALL, "");
    while ((caractere = getwchar()) != WEOF)
        affiche_caracteristiques(caractere);
    return EXIT_SUCCESS;
}

```

Naturellement, le fonctionnement est correct cette fois, même avec les caractères accentués :

```

$ ./exemple-isw
À à Ê ê Î î
A : alphanumerique alphanetique graphique imprimable majuscule
hexadecimal
a : alphanumerique alphanetique graphique minuscule imprimable
: imprimable espace
É : alphanumerique alphanetique graphique imprimable majuscule
e : alphanumerique alphanetique graphique minuscule imprimable
hexadecimal
Î : alphanumerique alphanetique graphique imprimable majuscule
ù : alphanumerique alphanetique graphique minuscule imprimable

: controle espace
$

```

Conversions entre catégories de caractères

Les conversions de caractères entre différentes catégories sont très limitées. Il existe trois fonctions permettant de modifier la classe d'un caractère classique, `tolower()`, `toupper()` et `tolower()`, dont les prototypes sont déclarés dans `<ctype.h>` :

```
int toascii (int caractere);  
int toupper (int caractere);  
int tolower (int caractere);
```

La fonction `toascii()` supprime purement et simplement le huitième bit du caractère transmis afin de renvoyer une valeur comprise entre 0 et 127. On comprend bien que le caractère résultant de cette modification n'a que très peu de chance d'avoir quelque chose à voir avec la lettre originale. En particulier, un caractère accentué comme `é` n'est pas transformé en `e` mais en un caractère quelconque de la table `Ascii`. Cela explique les modifications parfois étranges des textes contenant des caractères accentués lorsqu'ils franchissent des passerelles de courrier électronique mal configurées.

Les fonctions `toupper()` et `tolower()` permettent respectivement de passer un caractère en majuscule et en minuscule.

Attention, dans les bibliothèques C courantes, `toupper()` ne modifie pas le caractère passé en argument si ce n'est pas une minuscule. Mais dans des versions plus anciennes, cette routine renvoyait un caractère erroné, car elle modifiait toujours le sixième bit de la lettre. Cela est également vrai avec `tolower()` et les caractères non majuscules. On emploie donc systématiquement une vérification du genre :

```
if (isupper(c))  
    c = tolower(c);
```

ou

```
if (islower(c))  
    c = toupper(c);
```

Comme précédemment, on dispose des fonctions symétriques pour les caractères larges :

```
wint_t towlower (wint_t wc);  
wint_t towupper (wint_t wc);
```

La conversion en *Ascii* n'existe évidemment pas pour les caractères larges.

Voici un exemple d'appel de `towupper()` :

```
exemple-towupper.c :  
#include <locale.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <wchar.h>
```

```
#include <wctype.h>

int main(int argc, char * argv[])
{
    int i, j, n;
    wchar_t * warg = NULL;

    setlocale(LC_ALL, "");
    for (i = 1; i < argc; i++) {
        n = mbstowcs(NULL, argv[i], 0);
        warg = calloc(n, sizeof(wchar_t));
        if (warg == NULL) {
            perror("calloc");
            exit(EXIT_FAILURE);
        }
        mbstowcs(warg, argv[i], n);
        for (j = 0; j < n; j++)
            if (iswlower(warg[j]))
                fprintf(stdout, "%lc", towupper(warg[j]));
            else
                fprintf(stdout, "%lc", (wint_t) warg[j]);
        free(warg);
        fprintf(stdout, "\n");
    }
    return 0;
}
```

Vérifions le fonctionnement :

```
$ ./exemple-towupper aBcDéÊËÏÔöÇ
ABCDÉÊËÏÔöÇ
$
```

Conversions entre différents types

Les conversions qui nous intéressent ici sont celles qui permettent de passer d'une valeur numérique entière ou réelle à une chaîne de caractères, et inversement ainsi que les conversions mathématiques entre réels et entiers.

Il est toujours possible d'utiliser `sprintf()` ou `sscanf()` pour présenter les résultats d'un calcul ou examiner le contenu d'une chaîne. Toutefois, le surcoût imposé par l'énorme machine que représente `sscanf()` ne se justifie pas lorsqu'on veut juste convertir une chaîne de trois caractères en une valeur numérique comprise entre 1 et 100. Si la conversion n'a lieu qu'une seule fois avant un gros calcul et une fois après pour afficher

le résultat, mieux vaut probablement employer `sscanf()` et `sprintf()`, dont on maîtrise généralement mieux l'interface. Néanmoins, si on doit convertir à répétition les coordonnées de 200 000 points contenues dans des chaînes de caractères alors que l'utilisateur attend le résultat, il est sûrement préférable d'employer des routines optimisées.

Pour ce genre d'opération, il existe des fonctions spécialisées très efficaces. Les plus simples sont `atoi()`, `atol()`, `atof()`, ainsi que `atoll()`. Déclarées dans `<stdlib.h>`, ces routines convertissent les chaînes de caractères passées en arguments dans les types correspondant à leurs noms :

```
int      atoi (const char * chaîne); // Ascii TO Integer
long     atol (const char * chaîne); // Ascii TO Long
long long atoll (const char * chaîne); // Ascii TO Long Long
double   atof (const char * chaîne); // Ascii TO Float
```

Le problème que posent ces routines réside dans l'impossibilité de déterminer si une erreur s'est produite, comme le montre le programme suivant :

```
exemple-atoi.c
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char chaîne [128];
    while (fgets(chaîne, 128, stdin) != NULL)
        fprintf(stdout, "Lu : %d \n", atoi(chaîne));
    return EXIT_SUCCESS;
}
```

La routine ne permet pas de faire la différence entre 0 et une chaîne invalide :

```
$ ./exemple-atoi
4767
Lu : 4767
-101325
Lu : -101325
-2147483648
Lu : -2147483648
-2147483649
Lu : -2147483648
-2200000000
Lu : -2147483648
0
Lu : 0
```



```
azertyuiop
Lu : 0
  (Contrôle-D)
$
```

On remarque que la fonction plafonne les valeurs à la limite du type de données correspondant. Toutefois, le fait de ne pas pouvoir détecter des situations d'erreur est très dangereux, aussi évitera-t-on au maximum d'employer ces routines, à moins d'avoir auparavant vérifié entièrement le contenu de la chaîne.

Il est souvent préférable de se tourner vers les fonctions `strtol()`, `strtoul()`, `strtoll()` et `strtoull()`, déclarées dans `<stdlib.h>` :

```
long int          strtol (const char * chaîne,
                          char ** fin, int base);
unsigned long int strtoul(const char * chaîne,
                          char ** fin, int base);
long long int     strtoll(const char * chaîne,
                          char ** fin, int base);
unsigned long long strtoull(const char * chaîne,
                            char ** fin, int base);
```

Les équivalents pour caractères larges sont :

```
long int          wcstol (const wchar_t * chaîne,
                          wchar_t ** fin, int base);
unsigned long int wcstoul(const wchar_t * chaîne,
                          wchar_t ** fin, int base);
long long int     wcstoll(const wchar_t * chaîne,
                          wchar_t ** fin, int base);
unsigned long long wcstoull(const wchar_t * chaîne,
                            wchar_t ** fin, int base);
```

Ces fonctions analysent la chaîne de caractères passée en premier argument et en extraient une variable entière qu'elles retournent. Le second argument, s'il n'est pas nul, est un pointeur qui est mis à jour pour être dirigé vers le premier caractère non utilisé par la conversion. Finalement, le dernier argument représente la base employée pour la lecture. La base peut s'étendre de 2 à 36 ou prendre la valeur spéciale 0. Alors, la lecture est effectuée en base 10, sauf si la chaîne commence par 0x, cas où la conversion sera en hexadécimal, ou par un 0, cas où la lecture se fera en octal.

Pour les bases supérieures à 10, on emploie les lettres dans l'ordre alphabétique pour compléter les chiffres manquants. Ainsi, on utilise A, B, C, D, E et F en hexadécimal, et toutes les lettres jusqu'à Z en base 36. Il n'y a pas de différences entre les majuscules et les minuscules. Tous les caractères d'espacement en début de chaîne sont ignorés.

Le pointeur fourni en second argument permet de savoir si la conversion a pu avoir lieu. En effet, si aucun chiffre n'est lu, *fin est égal à chaine. En cas de débordement supérieur ou inférieur, la valeur renvoyée est plafonnée à la limite maximale ou supérieure du type de donnée, et errno vaut ERANGE. Voici un exemple d'utilisation de strtol() :

```
exemple-strtol.c :
#include <errno.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char chaine[128];
    char * fin;
    long retour;
    while (fgets(chaine, 128, stdin) != NULL) {
        retour = strtol(chaine, & fin, 0);
        if (fin == chaine) {
            fprintf(stdout, "Erreur \n");
            continue;
        }
        if (((retour == LONG_MAX) || (retour == LONG_MIN))
            && (errno == ERANGE)) {
            fprintf(stdout, "Débordement ! \n");
            continue;
        }
        fprintf(stdout, "Lu : %ld \n", retour);
    }
    return EXIT_SUCCESS;
}
```

Nous pouvons observer que, cette fois, la détection d'erreur est parfaitement gérée :

```
$ ./exemple-strtol
4767
Lu : 4767
0xFFFF
Lu : 65535
-2147483648
Lu : -2147483648
-2147483649
Débordement !
0
Lu : 0
```

```
azertyui
Erreur
  (Contrôle-D)
$
```

Pour lire des valeurs réelles, il existe trois fonctions `strtod()`, `strtof()` et `strtold()` :

```
float      strtof (const char * chaine, char ** fin);
double     strtod (const char * chaine, char ** fin);
long double strtold (const char * chaine, char ** fin);
```

ainsi que les fonctions pour caractères larges :

```
float      wcstof (const wchar_t * chaine, wchar_t ** fin);
double     wcstod (const wchar_t * chaine, wchar_t ** fin);
long double wcstold (const wchar_t * chaine, wchar_t ** fin);
```

Ces routines fonctionnent comme leurs consœurs entières, à la différence qu'il n'y a pas de notion de base ici, toutes les représentations étant considérées comme décimales. De plus, les routines de conversion de réels sont sensibles à la localisation pour tout ce qui concerne le séparateur décimal. Dans l'exemple précédent, nous n'avons pas utilisé la possibilité de lire successivement plusieurs valeurs au sein de la même chaîne. Dans le programme suivant, nous allons nous y employer :

```
exemple-strtof.c :
#define _GNU_SOURCE
#include <errno.h>
#include <limits.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void)
{
    char  chaine[128];
    char * debut;
    char * fin;
    float retour;

    setlocale(LC_ALL, "");
    while (fgets(chaine, 128, stdin) != NULL) {
        if (chaine[strlen(chaine) - 1] == '\n')
            chaine[strlen(chaine) - 1] = '\0';
```

```

    for (fin = debut = chaine; * fin != '\0'; debut = fin) {
        errno = 0;
        retour = strtod(debut, & fin);
        if (fin == debut) {
            fprintf(stdout, "Erreur \n");
            break;
        }
        if (errno == ERANGE)
            fprintf(stdout, "Débordement ! \n");
        else
            fprintf(stdout, "Lu : %f \n", retour);
    }
}
return EXIT_SUCCESS;
}

```

Commençons par vérifier le comportement vis-à-vis de la localisation :

```

$ unset LC_ALL
$ unset LANG
$ ./exemple-strtod
1.5
Lu : 1.500000
1,8
Lu : 1.000000
Erreur
(Contrôle-D)
$ export LANG=fr_FR
$ ./exemple-strtod
1.5
Lu : 1,000000
Erreur
1,8
Lu : 1,800000
(Contrôle-D)
$

```

Nous observons au passage que `printf()` est également sensible à la localisation pour ce qui concerne l'affichage de la valeur réelle. À présent, vérifions le problème du débordement :

```

$ ./exemple-strtod
9999999999
Lu : 10000000000,000000

```

```

1e10
Lu : 10000000000,000000
1e20
Lu : 100000002004087734272,000000
1e30
Lu : 1000000015047466219876688855040,000000
1e40
Débordement !
1e39
Débordement !
1e38
Lu : 99999996802856924650656260769173209088,000000
(Contrôle-D)
$

```

Nous constatons par la même occasion que la précision d'une variable `float` est assez limitée. Nous pouvons aussi examiner le fonctionnement des lectures successives dans la même chaîne :

```

$ ./exemple-strtof
4 7 6 7
Lu : 4,000000
Lu : 7,000000
Lu : 6,000000
Lu : 7,000000
18 4 22
Lu : 18,000000
Lu : 4,000000
Lu : 22,000000
(Contrôle-D)
$

```

Parallèlement à ces fonctions de lecture de chaîne, il existe des routines spécialisées dans la conversion de variables réelles en chaînes de caractères. Étant peu portables et compliquées à utiliser, on les déconseille en général. Il est souvent préférable d'employer `sprintf()`.

Les routines `ecvt()`, `fcvt()` et `gcvt()` sont héritées de Système V. Leurs prototypes sont déclarés dans `<stdlib.h>` ainsi :

```

char * ecvt (double nombre, size_t nb_chiffres,
             int * position_point, int * signe);

```

```
char * fcvt (double nombre, size_t nb_chiffres,  
             int * position_point, int * signe);  
char * gcvt (double nombre, size_t nb_chiffres,  
             char * buffer);
```

La fonction `ecvt()` convertit la valeur passée en premier argument en une chaîne de caractères contenant au maximum le nombre de chiffres indiqué en second argument. La chaîne renvoyée est allouée dans la mémoire statique – écrasée à chaque appel – et ne comprend pas de point décimal. En contrepartie, le troisième argument comportera en retour la position du premier chiffre après ce point décimal. Enfin, le dernier argument sera rempli avec une valeur nulle si le chiffre est positif.

La routine `fcvt()` fonctionne de la même manière, mais le second argument indique le nombre de décimales désirées.

La fonction `gcvt()` écrit le nombre de chiffres significatifs indiqué en second argument dans le buffer qui est passé en troisième argument, et renvoie un pointeur sur celui-ci.

L'utilisation de `ecvt()` et `fcvt()` est loin d'être intuitive. En voici un exemple :

```
exemple-ecvt.c :  
#include <stdio.h>  
#include <stdlib.h>  
  
int main (int argc, char * argv[])  
{  
    double valeur;  
    int    nb_chiffres;  
    int    position;  
    int    signe;  
    char * retour;  
  
    if ((argc != 3)  
        || (sscanf(argv[1], "%lf", & valeur) != 1)  
        || (sscanf(argv[2], "%d", & nb_chiffres) != 1)) {  
        fprintf(stderr, "Syntaxe: %s valeur nb_chiffres\n", argv[0]);  
        exit (EXIT_FAILURE);  
    }  
  
    retour = ecvt(valeur, nb_chiffres, & position, & signe);  
    fprintf(stdout, "ecvt() = %s \n", retour);  
    fprintf(stdout, "    position = %d \n", position);  
    fprintf(stdout, "    signe = %d \n", signe);  
  
    retour = fcvt(valeur, nb_chiffres, & position, & signe);  
    fprintf(stdout, "fcvt() = %s \n", retour);  
    fprintf(stdout, "    position = %d \n", position);  
    fprintf(stdout, "    signe = %d \n", signe);  
  
    return EXIT_SUCCESS;  
}
```

Les exécutions suivantes montrent bien que prévoir le résultat de ces routines nécessite une bonne dose de concentration :

```
$ ./exemple-ecvt 100 3
ecvt() = 100
    position = 3
    signe = 0
fcvt() = 100000
    position = 3
    signe = 0
$ ./exemple-ecvt 100 2
ecvt() = 10
    position = 3
    signe = 0
fcvt() = 10000
    position = 3
    signe = 0
$ ./exemple-ecvt 1.5 3
ecvt() = 150
    position = 1
    signe = 0
fcvt() = 1500
    position = 1
    signe = 0
$ ./exemple-ecvt -1.5 2
ecvt() = 15
    position = 1
    signe = 1
fcvt() = 150
    position = 1
    signe = 1
$
```

La bibliothèque Gnu ajoute en extensions les fonctions `qecvt()`, `qfcvt()` et `qgcvt()`, qui ont un comportement similaire, mais en utilisant des valeurs `long double` (*quad*).

```
char * qecvt (long double nombre, size_t nb_chiffres,
              int * position_point, int * signe);
char * qfcvt (long double nombre, size_t nb_chiffres,
              int * position_point, int * signe);
char * qgcvt (long double nombre, size_t nb_chiffres,
              char * buffer);
```

Enfin, toutes ces fonctions renvoyant leurs valeurs dans des zones de mémoire statique, elles ne sont pas utilisables dans un contexte multithread. Il existe donc quatre autres extensions Gnu, `ecvt_r()`, `fcvt_r()`, `qecvt_r()` et `qfcvt_r()`, auxquelles on transmet un buffer personnel à remplir en indiquant sa taille maximale.

```
char * ecvt_r(double nombre, size_t nb_chiffres,
              int * position_point, int * signe,
              char * buffer, size_t longueur);
char * fcvt_r(double nombre, size_t nb_chiffres,
              int * position_point, int * signe,
              char * buffer, size_t longueur);
char * qecvt_r(long double nombre, size_t nb_chiffres,
               int * position_point, int * signe,
               char * buffer, size_t longueur);
char * qfcvt_r(long double nombre, size_t nb_chiffres,
               int * position_point, int * signe,
               char * buffer, size_t longueur);
```

Les routines `gcvt()` et `qgcvt()` emploient déjà un buffer transmis par l'application et ne nécessitent donc pas d'équivalences réentrantes. Rappelons que ces fonctions sont difficiles à employer et qu'il vaut généralement mieux se tourner vers `sprintf()` qui peut offrir les mêmes résultats. Pour cette raison, d'ailleurs, `ecvt()`, `fcvt()` et `gcvt()` ont été supprimées du standard ISO C9X.

Types et conversions mathématiques

Nombres complexes

Depuis la norme C9X, un nouveau type de données est disponible pour représenter les nombres complexes. Il faut pour cela inclure l'en-tête `<complex.h>`. Nous pouvons dès lors définir des variables de type `float complex`, `double complex`, ou `long double complex` en fonction de l'étendue désirée, et utiliser des fonctions spécifiques pour les manipuler.

Nous disposons de la constante symbolique `I` pour représenter le nombre imaginaire pur i tel que $i^2 = -1$. Voici un exemple simple dans lequel nous initialisons un nombre complexe, puis affichons ses divers paramètres, calculons son conjugué¹ et affichons les caractéristiques de celui-ci :

```
exemple-complexe.c :
#include <complex.h>
```



```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    double complex z;

    z = 0.5 + I * (sqrt(3)/2);
    fprintf(stdout, "Z : \n");
    fprintf(stdout, " Partie réelle      : %f\n", creal(z));
    fprintf(stdout, " Partie imaginaire   : %f\n", cimag(z));
    fprintf(stdout, " Module          : %f\n", cabs(z));
    fprintf(stdout, " Argument         : %f\n", carg(z));

    z = conj(z);
    fprintf(stdout, "\nConjugue de Z : \n");
    fprintf(stdout, " Partie réelle      : %f\n", creal(z));
    fprintf(stdout, " Partie imaginaire   : %f\n", cimag(z));
    fprintf(stdout, " Module          : %f\n", cabs(z));
    fprintf(stdout, " Argument         : %f\n", carg(z));

    return EXIT_SUCCESS;
}
```

L'exécution donne :

```
$ ./exemple-complexe
Z :
  Partie réelle      : 0.500000
  Partie imaginaire   : 0.866025
  Module          : 1.000000
  Argument         : 1.047198

Conjugue de Z :
  Partie réelle      : 0.500000
  Partie imaginaire   : -0.866025
  Module          : 1.000000
  Argument         : -1.047198
$
```

La valeur 0.866025 correspond à $\frac{\sqrt{3}}{2}$ et 1.047198 à $\frac{\pi}{2}$.

1. Le conjugué d'un nombre complexe est le nombre de même partie réelle, mais dont la partie imaginaire est opposée.

Toutes les routines usuelles de la bibliothèque mathématique existent en différentes versions suivant le type d'arguments, y compris les arguments complexes. Les arguments par défaut sont de type `double`, la fonction avec le suffixe 'f' prend un argument de type `float`, celle avec le suffixe 'l' prend un argument `long double`, enfin celles avec un préfixe 'c' prennent des arguments complexes. Par exemple, pour la fonction *cosinus* :

```
double      cos (double x);
float       cosf(float x);
long double cosl(long double x);
double complex ccos(double complex z);
float complex ccosf (float complex z);
long double complex ccosl (long double complex z);
```

Ce schéma se reproduit pour les fonctions mathématiques suivantes : `sin()`, `tan()`, `acos()`, `asin()`, `atan()`, `cosh()`, `sinh()`, `tanh()`, `acosh()`, `asinh()`, `atanh()`, `exp()`, `log()`, `pow()`, `sqrt()`.

Conversions de réels en entiers

Il est fréquent de devoir transformer le résultat de calculs réels en valeurs entières. Toutefois, il y a plusieurs fonctions disponibles, et un mauvais choix peut conduire à des erreurs de conversion assez déroutantes.

La conversion la plus simple est celle qui est implicite lorsqu'on transfère le contenu d'une variable réelle dans une variable entière. Cette conversion consiste simplement à supprimer la partie décimale du nombre réel. Ainsi, 4.5 devient 4, et -3.2 devient -3. Fréquemment employée, cette méthode est pourtant rarement celle qui est conseillée en pratique. Dans un affichage cartographique par exemple, les polygones sont généralement représentés par des listes de points dont les coordonnées sont réelles. Ainsi, il est possible de réaliser des opérations de zoom, translation ou rotation avec une bonne précision. Toutefois, lors de l'affichage, une conversion en valeurs entières doit avoir lieu pour obtenir les coordonnées des pixels. Si on utilise une conversion implicite des variables du langage C, on risque de voir deux polygones adjacents mal raccordés par leur sommet commun.

Pour éviter ce problème, on emploie plutôt la fonction `rint()` qui arrondit à l'entier le plus proche :

```
double      rint(double x);
float       rintf (float x);
long double rintl (long double x);
```

On observe alors que `rint(4.8)=5`, `rint(4.2)=4`, `rint(-3.1)=-3` et `rint(-0.9)=-1`.

Il existe des fonctions équivalentes :

```
double    nearbyint(double x);
float     nearbyintf (float x);
long double nearbyintl (long double x);
```

Elles sont légèrement moins portables, mais ne risquent pas de déclencher d'exception de débordement. Quoi qu'il en soit, ces exceptions ne peuvent pas déclencher sur les systèmes actuels, aussi je préfère utiliser `rint()` et ses consœurs qui me semblent plus répandues.

Parfois, on peut également préférer utiliser la véritable fonction mathématique « partie entière », qui arrondit à l'entier immédiatement inférieur ou égal. Cette fonction est nommée `floor()` :

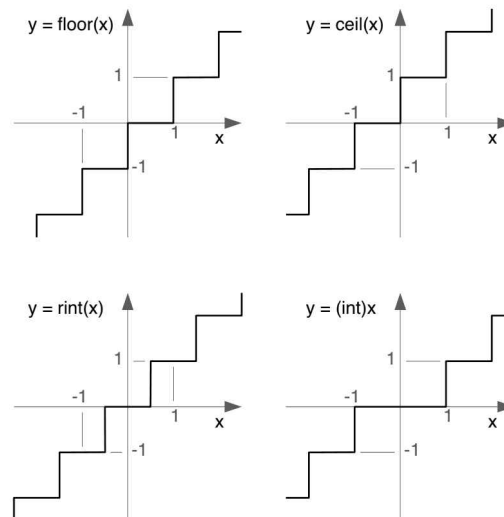
```
double    floor(double x);
float     floorf (float x);
long double floorl (long double x);
```

Cette fois, `floor(1.2)=1`, et `floor(1.9)=1`, mais également `floor(-0.9)=-1`. On notera qu'il existe un synonyme de `floor()` nommé `trunc()`.

Enfin, la fonction `ceil()` arrondit symétriquement par excès à l'entier immédiatement supérieur :

```
double    ceil(double x);
float     ceilf (float x);
long double ceill (long double x);
```

Figure 19-1
Fonctions d'arrondi



Voyons les différences de comportement de ces quatre routines autour de zéro.

```
exemple-arrondis.c :
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    double d;
    double arrondi_inf;
    double arrondi_sup;
    double arrondi_proche;
    int    converti;

    printf("reel    floor() ceil() rint() (int)\n");
    for (d = -1.8; d < 1.9; d += 0.2) {
        arrondi_inf = floor(d);
        arrondi_sup = ceil(d);
        arrondi_proche = rint(d);
        converti = (int) d;
        printf("% 4.1f    % 4.1f    % 4.1f    % 4.1f    % 2d\n",
            d, arrondi_inf, arrondi_sup, arrondi_proche, converti);
    }
    return EXIT_SUCCESS;
}
```

L'exécution correspond à ce qu'on attendait :

```
$ ./exemple-arrondis
reel    floor() ceil() rint() (int)
-1.8    -2.0    -1.0    -2.0    -1
-1.6    -2.0    -1.0    -2.0    -1
-1.4    -2.0    -1.0    -1.0    -1
-1.2    -2.0    -1.0    -1.0    -1
-1.0    -2.0    -1.0    -1.0    -1
-0.8    -1.0    -0.0    -1.0    0
-0.6    -1.0    -0.0    -1.0    0
-0.4    -1.0    -0.0    -0.0    0
-0.2    -1.0    -0.0    -0.0    0
-0.0    -1.0    -0.0    -0.0    0
 0.2     0.0     1.0     0.0     0
 0.4     0.0     1.0     0.0     0
 0.6     0.0     1.0     1.0     0
 0.8     0.0     1.0     1.0     0
```

1.0	0.0	1.0	1.0	0
1.2	1.0	2.0	1.0	1
1.4	1.0	2.0	1.0	1
1.6	1.0	2.0	2.0	1
1.8	1.0	2.0	2.0	1
\$				

Infinis et erreurs

Les fonctions mathématiques ont des domaines de définition bien précis. Essayer d'invoquer une fonction, par exemple `log()`, pour une valeur interdite (disons -5) doit renvoyer une erreur. Toutefois, la routine `log()` ne peut pas se contenter de renvoyer -1 en cas d'erreur, comme le font d'autres fonctions de bibliothèque habituellement. Cette valeur, en effet, est tout à fait légitime pour $x = \frac{1}{e}$.

Valeur non numérique

Pour signaler une erreur, les routines renvoient une valeur spéciale, nommée NaN, ce qui signifie *Not a Number*. De plus, elles positionnent la variable globale `errno` (avec l'erreur `EDOM` en général). Pour vérifier le résultat, il existe une fonction nommée `isnan()`, déclarée ainsi :

```
| int isnan (double valeur);
```

Il s'agit en réalité d'une macro, aussi peut-on utiliser la même routine pour les différents types de réels (`float`, `double` et `long double`).

Elle renvoie 0 si son argument est numérique et une valeur non nulle sinon. On peut donc employer le code suivant :

```
| double cosinus;  
| double angle;  
  
| angle = acos(cosinus);  
| if (isnan(angle)) {  
|     perror("acos");  
|     exit(EXIT_FAILURE);  
| }  
| ...
```

Attention, il n'existe pas de constante symbolique NaN avec laquelle on pourrait faire la comparaison. Nous verrons dans la représentation binaire des réels qu'il n'y a pas une unique valeur non numérique, mais qu'on en trouve une multitude.

Infinis

Parfois, une fonction réelle, par exemple $f(x) = \frac{1}{x}$, peut renvoyer une valeur infinie sur un point précis de son intervalle de définition, en $x=0$ en l'occurrence. Comme la précision de la représentation des réels en virgule flottante est limitée, il existe nécessairement une certaine zone de « flou » autour de ce point. On ne peut donc pas simplement renvoyer une erreur, mais la bibliothèque C doit permettre de traiter les infinis. Elle utilise donc deux valeurs supplémentaires spéciales, indiquant $-\infty$ et $+\infty$. Pour les détecter, il existe une fonction `isinf()` :

```
| int isinf (double valeur);
```

Cette routine renvoie 0 si la valeur est finie, -1 s'il s'agit de $-\infty$, et +1 s'il s'agit de $+\infty$. Il existe également une routine `finite()`, ayant le fonctionnement contraire :

```
| int finite (double valeur);
```

Elle renvoie une valeur non nulle si la valeur transmise est numérique (pas NaN) et finie. Le paramètre indiqué ici est de type double, mais on utilise les mêmes routines `isinf()` et `finite()` quel que soit le type de données réelles.

Voici un exemple qui va nous permettre de voir les divers cas traités par la bibliothèque mathématique :

```
exemple-infinis.c :
#define _XOPEN_SOURCE 600

#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void affiche_nombre (const char * chaine, double d)
{
    fprintf(stdout, "%s", chaine);
    if (isnan(d))
        fprintf(stdout, "Indefini\n");
    else if (isinf(d) == 1)
        fprintf(stdout, "+Infini\n");
    else if (isinf(d) == -1)
        fprintf(stdout, "-Infini\n");
    else
```

```
        fprintf(stdout, "%f\n", d);
    }

    int main (void)
    {
        double d;

        d = +0.0;
        d = 1.0 / d;
        affiche_nombre("1 / +0 = ", d);
        d = -0.0;
        d = 1.0 / d;
        affiche_nombre("1 / -0 = ", d);
        d = 0.0 / 0.0;
        affiche_nombre("0 / 0 = ", d);
        d = log(0.0);
        affiche_nombre("log(0) = ", d);
        d = log(-1.0);
        affiche_nombre("log(-1)= ", d);
        d = MAXFLOAT;
        affiche_nombre("MAXFLOAT = ", d);
        d = exp(MAXFLOAT);
        affiche_nombre("exp(MAXFLOAT)= ", d);
        return EXIT_SUCCESS;
    }
```

Pour des raisons de compatibilité avec les standards antérieurs, il faut définir la constante `_XOPEN_SOURCE` avec la valeur 600 pour obtenir `MAXFLOAT` et `isinf()`.

Nous remarquons que la bibliothèque distingue `+0` de `-0`, ce qui peut paraître surprenant à première vue, mais qui s'explique par la représentation des nombres que nous examinerons plus bas.

```
$ ./exemple-infinis
1 / +0 = +Infini
1 / -0 = -Infini
0 / 0 = Indefini
log(0) = -Infini
log(-1)= Indefini
MAXFLOAT = 340282346638528859811704183484516925440.000000
exp(MAXFLOAT)= +Infini
$
```

Si on construit une bibliothèque mathématique complémentaire, il peut être nécessaire de renvoyer des valeurs infinies ou non numériques en cas d'erreur. Il ne serait pas très élégant d'être obligé de les obtenir avec des artifices du genre `(-1.0/0.0)` ou

$\log(-1.0)$. Il faut donc avoir un moyen d'accéder directement à ces valeurs. Les infinis sont représentés par `HUGE_VAL` ou `-HUGE_VAL`. Il n'y a pas de constante symbolique permettant de renvoyer directement NaN, mais il existe une routine BSD non définie par SUSv4 pouvant servir à renvoyer une erreur.

```
double infnan (int erreur);
```

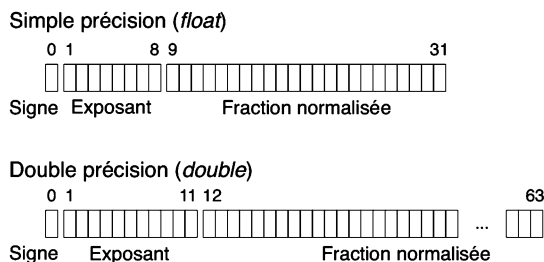
Si l'argument `erreur` vaut `EDOM`, `infnan()` renvoie NaN, s'il vaut `ERANGE` ou `-ERANGE`, `infnan()` renvoie respectivement `HUGE_VAL` ou `-HUGE_VAL`.

Représentation des réels en virgule flottante

Pour stocker un réel en mémoire, la plupart des ordinateurs utilisent un format normalisé, connu sous le nom de *IEEE 754*. Ce format peut donc convenir pour transférer des valeurs numériques entre ordinateurs. Toutefois, il est important de savoir éventuellement décoder « manuellement » les données si le système destinataire ne respecte pas le même format.

Le format IEEE 754 utilise 32 bits pour les valeurs de type `float`, 64 bits pour les réels de type `double`. Dans les deux cas, le réel est stocké sous forme d'un bit de signe, suivi d'un exposant (sur 8 bits dans un cas, et 11 bits dans l'autre), suivi de la fraction normalisée sur 23 et 52 bits respectivement.

Figure 19-2
Représentations binaires
des nombres réels



Le format `long double` est défini par IEEE 854 (en complément de IEEE 754). Il s'agit d'un codage sur 12 octets, soit 96 bits, composés d'un bit de signe, suivi par 15 bits d'exposant, 16 bits vides, puis 64 bits de fraction normalisée.

Le bit de signe vaut 0 si le nombre est positif, et 1 s'il est négatif. Cela nous explique pourquoi la bibliothèque distingue `+0.0` et `-0.0`.

L'exposant est compris entre 0 et 255 pour les `float`, 2 047 pour les réels de type `double`, ou 32 767 pour les `long double`.

Si l'exposant vaut 255 (2 047 ou 32 767) et si la fraction normalisée n'est pas nulle, alors le nombre représente NaN. Si l'exposant vaut 255 (2 047 ou 32 767) et si la fraction est nulle, le nombre correspond à $+\infty$ ou $-\infty$ en fonction du bit de signe. On comprend alors qu'il existe une grande quantité de valeurs pouvant correspondre à NaN, puisqu'il suffit que la fraction normalisée ne soit pas nulle. C'est pour cela qu'il n'existe pas de constante symbolique NaN avec laquelle on pourrait comparer une valeur.

Si l'exposant n'est pas nul et s'il n'a pas sa valeur maximale (255, 2 047 ou 32 767), alors le nombre représenté vaut :

$(-1)^{\text{signe}} \times 2^{(\text{exposant} - 127)} \times (1.0 + \text{fraction})$ pour les réels de type float,

$(-1)^{\text{signe}} \times 2^{(\text{exposant} - 1023)} \times (1.0 + \text{fraction})$ pour les réels de type double, et

$(-1)^{\text{signe}} \times 2^{(\text{exposant} - 16383)} \times (1.0 + \text{fraction})$ pour les réels de type long double.

La valeur de la fraction est calculée en additionnant les puissances négatives successives de 2, en commençant par $\frac{1}{2}$ et en se terminant par $\frac{1}{2^{23}} = \frac{1}{8388608}$, $\frac{1}{2^{52}}$, $\frac{1}{2^{64}}$.

Par exemple, la fraction normalisée 11010010 (en se limitant à 8 bits) vaut :

$$\frac{1}{2} + \frac{1}{4} + 0 + \frac{1}{16} + 0 + 0 + \frac{1}{128} + 0 = 0.83203125.$$

Finalement, un cas particulier se présente si l'exposant vaut zéro.

Si la fraction est nulle, le nombre correspond à $+0$ ou -0 en fonction du bit de signe, sinon, la valeur est :

$(-1)^{\text{signe}} \times 2^{(-126)} \times \text{fraction}$ pour un float,

$(-1)^{\text{signe}} \times 2^{(-1022)} \times \text{fraction}$ pour un double,

$(-1)^{\text{signe}} \times 2^{(-16382)} \times \text{fraction}$ pour un long double.

Si nous avons présenté ce format ici, c'est qu'il est largement utilisé dans les ordinateurs modernes et permet normalement un échange assez facile des données. On peut ainsi sur 4, 8, ou même 12 octets, transférer des valeurs réelles avec une très bonne précision sur un réseau ou dans un fichier. La connaissance détaillée des formats IEEE 754 et 854 nous permet de conserver dans un tiroir des routines d'encodage et de décodage s'il faut porter une application sur un système représentant différemment les nombres en virgule flottante.

Conclusion

Nous avons examiné dans ce chapitre quelques spécificités des types de données actuellement utilisés par Linux et la Glibc, ainsi que les conversions particulières entre eux. L'évolution actuelle des systèmes Linux conduit à une généralisation des processeurs 64 bits dans le domaine des serveurs et des postes de travail, toutefois les

processeurs à 32 bits sont encore très présents dans les systèmes embarqués, aussi est-il important lors du développement d'une application d'être conscient des variations de formats des données suivant les architectures.

Entrées-sorties simplifiées

Flux standard d'un processus

Les entrées-sorties sous Linux sont uniformisées par l'intermédiaire de fichiers. Nous verrons, dans la partie consacrée à la gestion des fichiers, qu'on peut y accéder grâce à des primitives de bas niveau (des appels système) gérant des *descripteurs* ou par des fonctions de haut niveau (de la bibliothèque C) manipulant des *flux*.

Les flux sont une abstraction ajoutant automatiquement aux descripteurs de fichiers des tampons d'entrée-sortie, des verrous, ainsi que des rapports d'état et d'erreur plus fins. Les flux sont du type opaque `FILE`, défini dans `<stdio.h>` (ou plus précisément dans `<libio.h>`, inclus par ce dernier). On ne doit pas tenter d'accéder aux membres internes de la structure `FILE`, pas plus qu'on ne doit utiliser d'objets de type `FILE`, mais uniquement des pointeurs sur ces objets. Les allocations et libérations de mémoire nécessaires sont entièrement gérées par les fonctions de la bibliothèque C.

En Python, la construction qui se rapproche le plus de ces flux est l'objet `file` que l'on peut obtenir avec la fonction standard `open()` (à ne pas confondre avec `os.open()` qui fournit un descripteur et que nous verrons au chapitre 22). Son comportement est décrit dans de nombreuses documentations et tutoriels sur Python, nous n'en verrons donc qu'un aperçu très léger.

Lorsqu'on désire accéder à un fichier par l'intermédiaire d'un flux C, on invoque la fonction `fopen()`, que nous décrirons plus en détail dans le chapitre 21. Cette fonction prend en argument le nom du fichier désiré, ainsi qu'une chaîne de caractères

indiquant le mode d'accès voulu, et renvoie un pointeur sur un flux de type `FILE *`. On l'utilise ainsi

```
FILE * fp;  
  
fp = fopen ("mon_fichier.txt", "r");
```

pour ouvrir le fichier en lecture seule (mode *rread*). Le pointeur de flux renvoyé peut alors être utilisé pour lire des données. Si on ouvrait notre fichier en mode écriture *wwrite*, on pourrait alors y écrire des informations. En Python, on utilisera :

```
fp = sys.open("mon_fichier.txt", "r")
```

mais on peut également se contenter de `sys.open("mon_fichier.txt")` car le mode par défaut est la lecture seule.

Nous détaillerons toutes ces notions plus loin, mais ce qui nous intéresse pour l'instant c'est que tout programme s'exécutant sous Linux dispose de trois flux ouverts automatiquement lors de son démarrage.

Ces trois flux sont déclarés dans `<stdio.h>` (module `sys` en Python) :

- **stdin** en C ou `sys.stdin` en Python : flux d'entrée standard. Ce flux est ouvert en lecture seule ; il s'agit par défaut du clavier. Le processus peut y recevoir ses données.
- **stdout** ou `sys.stdout` : flux de sortie standard. Ouvert en écriture seule, le processus y affiche ses résultats normaux. Par défaut, il s'agit de l'écran de l'utilisateur.
- **stderr** ou `sys.stderr` : flux d'erreur standard. Ce flux, ouvert en écriture seule, sert à afficher des informations concernant le comportement du processus ou ses éventuels problèmes. Par défaut, ces informations sont également affichées sur l'écran de l'utilisateur.

Nous arrivons ici à l'un des principes de la conception même des systèmes Unix. Dans cet environnement, une grande partie des outils et des commandes de base est vue comme des filtres. Ils reçoivent des données en entrée, les transforment, et fournissent leurs résultats en sortie. En cas de problème, l'utilisateur est averti par un message qui s'affiche sur un flux de sortie distinct.

Il est possible, au niveau du shell, de rediriger les flux d'entrée ou de sortie d'un processus à volonté. On peut rediriger par exemple la sortie d'un programme vers un fichier en utilisant l'opérateur « > » :

```
$ mon_programme > sortie.txt
```

À ce moment, toutes les données écrites sur `stdout` seront envoyées dans le fichier concerné. Les informations relatives à `stderr` resteront sur l'écran.

On peut également rediriger l'entrée standard pour lire les données depuis un fichier avec l'opérateur « < » ;

```
| $ mon_programme < entree.txt
```

On peut orienter la sortie standard d'un processus directement vers l'entrée standard d'un autre en utilisant l'opérateur « | » :

```
| $ programme_1 | programme_2
```

Cette opération est souvent effectuée pour renvoyer les données vers un utilitaire de pagination comme `more` ou `less`. Dans ce dernier cas également, les informations de diagnostic provenant de `programme_1` et envoyées sur `stderr` sont affichées à l'écran, et ne sont pas mêlées aux données de `stdout`, qui sont dirigées vers `programme_2`.

Pareillement, il est possible de procéder à d'autres interventions, comme ajouter la sortie standard en fin de fichier sans écrasement, regrouper la sortie d'erreur et la sortie standard, lire des données directement depuis la ligne de commande ou le script shell utilisés. Ces opérateurs peuvent varier suivant le shell employé. On se reportera, pour plus de détails, à un ouvrage traitant de la programmation sous shell ou à la page de manuel de l'interpréteur concerné.

En fait, les notions d'entrée standard, de sortie standard et de sortie d'erreur sont des concepts de l'univers Unix qui n'ont pas de réelle signification au niveau du noyau. Il s'agit simplement d'une convention instituée par les shells historiques (et qui risque fort de tomber peu à peu en désuétude avec l'avènement des environnements uniquement graphiques).

Dans ce chapitre, nous allons nous intéresser à la présentation ou à la lecture de données sous forme de textes, lisibles par un être humain. Toutefois, les flux standard d'un processus peuvent également être utilisés pour transporter des données binaires. Cela permet de construire une série de petits outils indépendants qu'on regroupe ensuite. Lorsque nous aborderons la programmation réseau, nous étudierons un programme capable de recevoir des données sur un port réseau *UDP/IP* et de les écrire sur sa sortie standard. De même, un autre programme lira son entrée standard et enverra les informations vers le port *UDP/IP* d'une autre application. Ces programmes s'appelleront respectivement `udp_2_stdout` et `stdin_2_udp` (le 2 doit se lire « *two* », ou plutôt « *to* », c'est-à-dire « vers » ; c'est une tradition de nommer ainsi les programmes de filtrage servant à changer le format de leurs données).

Imaginons qu'on ait un autre programme nommé « convertisseur », qui modifie les informations reçues sur son entrée standard pour les renvoyer en sortie. Avec ces trois outils, nous pouvons obtenir toutes les possibilités suivantes :

- enregistrement de données provenant d'une application serveuse :

```
| $ udp_2_stdout > fichier_1
```

- relecture des données et émission vers l'application cliente :

```
| $ stdin_2_udp < fichier_1
```

- passerelle entre deux réseaux, par exemple :

```
| $ udp_2_stdout | stdin_2_udp
```

- conversion des données « à froid » :

```
| $ convertisseur < fichier_1 > fichier_2
```

- conversion « au vol » des données :

```
| $ udp_2_stdout | convertisseur | stdin_2_udp
```

- on peut les insérer dans un filtre entre deux programmes :

```
| $ programme_1 | programme_2
```

qu'on transforme en :

```
| $ programme_1 | stdin_2_udp
```

sur une machine, et :

```
| $ udp_2_stdout | programme_2
```

sur une autre machine.

Nous voyons la puissance des redirections des flux standard des processus. Ces exemples ne sont pas artificiels, je les ai personnellement utilisés dans une application industrielle pour convertir au vol des données provenant d'un radar et les rendre compatibles avec une application de visualisation se trouvant sur une autre machine. Le fait de pouvoir enregistrer des données ou intercaler un programme d'affichage hexadécimal des valeurs en modifiant simplement le script shell de lancement aide grandement à la mise au point du système.

Écritures dans un flux

L'une des tâches premières des programmes informatiques est d'afficher des messages lisibles par les utilisateurs sur un périphérique de sortie, généralement l'écran. La preuve en est donnée dans le célèbre `hello.c` [Kernighan 1994], dont l'unique rôle est d'afficher « `hello world!` » et de se terminer normalement¹.

```
#include <stdio.h>
main()
{
    printf("Hello, world \n");
}
```

Équivalent en Python :

```
from __future__ import print_function
print("Hello world")
```

La ligne d'importation permet d'avoir un comportement cohérent entre les versions 2 de Python pour lesquelles `print` était une procédure et les versions 3 où elle est devenue une fonction.

Écritures formatées

Lorsqu'il s'agit d'une chaîne de caractères constante, comme dans ce fameux exemple, le travail est relativement simple – il suffit d'envoyer les caractères l'un après l'autre sur le flux de sortie –, mais les choses se compliquent nettement quand il faut afficher des valeurs numériques. La conversion entre la valeur 2010, contenue dans une variable de type `int`, et la série de caractères '2', '0', '1' et '0' n'est déjà pas une tâche simple. Ce qui se présente comme un exercice classique des premiers cours d'assembleur se corse nettement lorsqu'il faut gérer les valeurs signées, puis différentes bases d'affichage (décimal, hexa, octal). Imaginez alors la complexité du travail qui est nécessaire pour afficher le contenu d'une variable en virgule flottante, avec la multitude de formats possibles et le nombre de chiffres significatifs adéquat.

Heureusement, la bibliothèque C standard nous offre les fonctions de la famille `printf()`, qui permettent d'effectuer automatiquement les conversions requises pour afficher les données. Ces routines sont de grands classiques depuis les premières versions des bibliothèques standard du langage C, aussi nous ne détaillerons pas en pro-

1. L'absence de `return 0` n'est pas un oubli, mais est due à la volonté de reproduire exactement l'exemple original de Kernighan et de Ritchie.

fondeur chaque possibilité de conversion. On pourra, pour avoir plus de renseignements, se reporter à la page de manuel `printf(3)`.

En Python, on pourra se reporter à la méthode `format()` des chaînes de caractères, qui remplace avantageusement l'essentiel des fonctions `printf()` du C.

Il existe quatre variantes sur le thème de `printf()`, chacune d'elles étant disponible en deux versions, suivant la présentation des arguments.

La fonction la plus utile est bien souvent `fprintf()`, dont le prototype est déclaré dans `<stdio.h>` ainsi :

```
| int fprintf (FILE * flux, const char * format, ...);
```

Les points de suspension indiquent qu'on peut fournir un nombre variable d'arguments à cet emplacement. Le premier argument est le flux dans lequel on veut écrire ; il peut s'agir bien entendu de `stdout` ou `stderr`, mais nous verrons ultérieurement qu'il peut s'agir aussi de n'importe quel fichier préalablement ouvert avec la fonction `fopen()`.

Le second argument est une chaîne de caractères qui sera envoyée sur le flux indiqué, après avoir remplacé certains caractères spéciaux qu'elle contient. Ceux-ci indiquent la conversion à apporter aux arguments situés à la fin de l'appel avant de les afficher. Par exemple, la séquence « `%d` » dans le format sera remplacée par la représentation décimale de l'argument de type entier situé à la suite du format. On peut bien entendu placer plusieurs arguments à afficher, en indiquant dans la chaîne de format autant de caractères de conversion.

Les conversions possibles avec la Glibc sont les suivantes.

Conversion	But
<code>%d</code>	Afficher un nombre entier sous forme décimale signée.
<code>%i</code>	Synonyme de <code>%d</code> .
<code>%u</code>	Afficher un nombre entier sous forme décimale non signée.
<code>%o</code>	Afficher un nombre entier sous forme octale non signée.
<code>%x</code>	Afficher un entier non signé sous forme hexa avec des minuscules.
<code>%X</code>	Afficher un entier non signé sous forme hexa avec des majuscules.
<code>%f</code>	Afficher un nombre réel en notation classique (3.14159).
<code>%e</code>	Afficher un réel en notation ingénieur (1.602e-19).
<code>%E</code>	Afficher un réel en notation ingénieur avec E majuscule.
<code>%g</code>	Afficher un réel le plus lisiblement possible entre <code>%f</code> et <code>%e</code> suivant sa valeur.
<code>%G</code>	Comme <code>%g</code> , mais en choisissant entre <code>%f</code> et <code>%E</code> .
<code>%a</code>	Afficher un réel avec la mantisse en hexa et l'exposant de 2 en décimal.

Conversion	But
%A	Comme %a, mais le « P » indiquant l'exposant de 2 est en majuscule.
%C	Afficher un simple caractère.
%C	Afficher un caractère large (on préfère toutefois %l c).
%s	Afficher une chaîne de caractères.
%S	Afficher une chaîne de caractères larges (mais on préfère %l s).
%p	Afficher la valeur d'un pointeur.
%n	Mémoriser le nombre de caractères déjà écrits (voir plus bas).
%m	Afficher la chaîne de caractères décrivant le contenu de <code>errno</code> .
%%	Afficher le caractère de pourcentage.

Notons tout de suite que %m est une extension Gnu, qui correspond à afficher la chaîne de caractères `strerror(errno)`. Signalons également que la conversion %n est très particulière puisqu'elle n'écrit rien en sortie, mais stocke dans l'argument correspondant, qui doit être un pointeur de type `int *`, le nombre de caractères qui a déjà été envoyé dans le flux de sortie. Cette fonctionnalité n'est pas couramment employée ; on peut imaginer l'utiliser avec `sprintf()`, que nous verrons ci-après, pour mémoriser l'emplacement des champs de données successifs si on désire y accéder à nouveau par la suite.

La conversion %n peut introduire des failles de sécurité par écrasement de mémoire. Si l'utilisateur peut fixer lui-même le format de sortie de `printf()`, il dispose d'un moyen d'écrire dans la mémoire du processus et de lui faire ainsi exécuter n'importe quel code. Ceci est très dangereux pour un programme *Set-UID* ou un démon réseau. L'affichage avec `printf()` d'une chaîne de caractères fournie par l'utilisateur se fera avec `printf("%s", chaine)` et jamais `printf(chaine)`.

Les autres conversions sont très classiques en langage C et ne nécessitent pas plus de détails ici. Voyons un exemple d'utilisation des diverses conversions.

```
exemple-fprintf-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int main (void)
{
    int          d = INT_MAX;
    unsigned int u = UINT_MAX;
    unsigned int o = INT_MAX;
    unsigned int x = UINT_MAX;
    unsigned int X = UINT_MAX;
```

```

double      f = 1.04;
double      e = 1500;
double      E = 101325;
double      g = 1500;
double      G = 0.00000101325;
double      a = 1.0/65536.0;
double      A = 0.125;
char        c = 'a';
char        * s = "chaîne";
void        * p = (void *) main;
fprintf(stdout, " d=%d \n u=%u \n o=%o \n x=%x \n X=%X \n"
               " f=%f \n e=%e \n E=%E \n g=%g \n G=%G \n"
               " a=%a \n A=%A \n c=%c \n s=%s \n p=%p \n",
               d, u, o, x, X, f, e, E, g, G, a, A, c, s, p);
return EXIT_SUCCESS;
}

```

Bien sûr, les valeurs INT_MAX et UINT_MAX définies dans <limits.h> peuvent varier avec l'architecture de la machine.

```

$ ./exemple-fprintf-1
d=2147483647
u=4294967295
o=1777777777
x=ffffffff
X=FFFFFFFF
f=1.040000
e=1.500000e+03
E=1.013250E+05
g=1500
G=1.01325E-06
a=0x1p-16
A=0X1P-3
c=a
s=chaîne
p=0x80483e4
$

```

On peut incorporer, comme dans n'importe quelle chaîne de caractères en langage C, des caractères spéciaux comme \n, \r, \t..., qui seront interprétés par le terminal au moment de l'affichage.

Entre le symbole % et le caractère indiquant la conversion à effectuer, on peut insérer plusieurs indications permettant de modifier la conversion ou de préciser le formatage, en termes de largeur minimale ou maximale d'affichage.

Le premier indicateur qu'on peut ajouter concerne le formatage. Il sert principalement à spécifier sur quel côté le champ doit être aligné. À la suite de cet indicateur peut se trouver un nombre signalant la largeur minimale du champ. On justifie ainsi des valeurs en colonne. On peut encore inclure un point, suivi d'une deuxième valeur marquant la précision d'affichage de la valeur numérique. Un dernier modificateur peut être introduit afin de préciser comment la conversion de type doit être effectuée à partir du type effectif de la variable transmise en argument.

Pour les conversions entières (%d, %i, %u, %o, %x, %X) ou réelles (%f, %e, %E, %g, %G), on peut utiliser – en premier caractère – les indicateurs de formatage suivants.

Caractère	Formatage
+	Toujours afficher le signe dans les conversions signées.
-	Aligner les chiffres à gauche et non à droite.
espace	Laisser un espace avant les chiffres positifs d'une conversion signée.
0 (zéro)	Compléter le chiffre par des zéros au début plutôt que par des espaces à la fin.
#	Préfixer par 0x ou 0X les conversions hexadécimales, et par 0 les conversions octales. Le résultat peut ainsi être relu automatiquement.

Avec les conversions affichant un caractère (%c), une chaîne (%s) ou un pointeur (%p), seul l'indicateur « - » peut être utilisé, afin d'indiquer une justification à gauche du champ.

À la suite de ce modificateur, on indique éventuellement la largeur minimale du champ. Si la valeur à afficher est plus longue, elle débordera. En revanche, si elle est plus courte, elle sera alignée à droite ou à gauche, et complétée par des espaces ou par des zéros suivant le formatage vu précédemment.

Après la largeur minimale du champ, on peut placer un point suivi de la précision de la valeur numérique. La précision correspond au nombre minimal de chiffres affichés dans le cas d'une conversion entière et au nombre de décimales lors des conversions de nombres réels. Voici quelques exemples de formatage en colonne.

```
exemple-fprintf-2.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int d;

    fprintf(stdout, "| %%6d | % %+6d | % %-6d | % % -+6d | % % 6d | % %06d | \n");
    fprintf(stdout, "+-----+-----+-----+-----+-----+-----+ \n");
    d = 0;
    fprintf(stdout, "| %6d | % +6d | % -6d | % -+6d | % 6d | %06d | \n", d, d, d, d, d, d);
```

```

d = 1;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
d = -2;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
d = 100;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
d = 1000;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
d = 10000;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
d = 100000;
fprintf(stdout, "|%6d|+%6d|%-6d|%-+6d|% 6d|%06d|\n", d, d, d, d, d, d);
return EXIT_SUCCESS;
}

```

```

$ ./exemple-fprintf-2
| %6d | +%6d | %-6d | %-+6d | % 6d | %06d |
+-----+-----+-----+-----+-----+-----+
|      0 |      +0 | 0      | | +0      |      0 | 000000 |
|      1 |      +1 | 1      | | +1      |      1 | 000001 |
|     -2 |     -2 | -2     | | -2     |     -2 | -00002 |
|     100 |    +100 | 100    | | +100    |    100 | 000100 |
|    1000 |   +1000 | 1000   | | +1000   |   1000 | 001000 |
|   10000 |  +10000 | 10000  | | +10000  |  10000 | 010000 |
| 100000 | +100000 | 100000 | | +100000 | 100000 | 000000 |
$

```

Nous voyons que l'indication de largeur du champ correspond bien à une largeur minimale, un débordement pouvant se produire, comme c'est le cas sur la dernière ligne si le signe est affiché. L'exemple suivant montre l'effet de l'indicateur de précision sur des conversions entières et réelles.

```

exemple-fprintf-3.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int d;
    double f;

    fprintf(stdout, "| %%8.0d | %%8.2d | %%8.0f "
               "| %%8.2f | %%8.2e | %%8.2g |\n");
    fprintf(stdout, "+-----+-----+-----+
               +-----+-----+-----+\n");
}

```

```

d = 0;
f = 0.0;
fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
           d, d, f, f, f, f);

d = 1;
f = 1.0;
fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
           d, d, f, f, f, f);

d = -2;
f = -2.0;
fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
           d, d, f, f, f, f);

d = 10;
f = 10.1;
fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
           d, d, f, f, f, f);

d = 100;
f = 100.01;
fprintf(stdout, "|%8.0d|%8.2d|%8.0f|%8.2f|%8.2e|%8.2g|\n",
           d, d, f, f, f, f);

return EXIT_SUCCESS;
}

```

```
$ ./exemple-fprintf-3
```

%8.0d	%8.2d	%8.0f	%8.2f	%8.2e	%8.2g
0	00	0	0.00	0.00e+00	0
1	01	1	1.00	1.00e+00	1
-2	-02	-2	-2.00	-2.00e+00	-2
10	10	10	10.10	1.01e+01	10
100	100	100	100.01	1.00e+02	1e+02

Notons, là encore, que la largeur indiquée peut être dépassée au besoin (comme avec -2 en notation exponentielle). La précision correspond bien au nombre minimal de chiffres affichés pour les entiers et au nombre de décimales pour les réels.

Enfin, le dernier indicateur est un modificateur qui précise le type réel de l'argument transmis, avant sa conversion. Avec les conversions entières, les modificateurs suivants sont autorisés.

Modificateur	Effet
h	L'argument est un short int ou un unsigned short int.
hh	L'argument est un char ou un unsigned char.

Modificateur	Effet
l	L'argument est un caractère large (si suivi de c), un <code>long int</code> (si suivi de d), un <code>unsigned long int</code> (si suivi de u) ou une chaîne de caractères larges (si suivi de s).
ll, L, ou q	L'argument est un <code>long long int</code> , parfois nommé « <i>quad</i> » sur d'autres systèmes.
t	L'argument est de type <code>ptrdiff_t</code> .
z	L'argument est de type <code>size_t</code> ou <code>ssize_t</code> .

Le choix entre le type signé ou non dépend du type de conversion qui suit le modificateur (%d ou %u, par exemple).

`ptrdiff_t` sert lorsqu'on effectue manuellement des opérations arithmétiques sur les pointeurs. Le type `size_t` ou sa version signée `ssize_t` servent à mesurer la taille des données.

Avec les conversions réelles, tout type de donnée est promu au rang de `double` avant d'être affichée. On peut éventuellement utiliser le modificateur `L`, qui indique que l'argument est de type `long double`. Il n'y a pas d'autre modificateur pour les conversions réelles.

Nous allons à présent observer quelques particularités moins connues de `fprintf()` : la largeur de champ variable et la permutation des arguments. Si on remplace la largeur minimale du champ ou la précision numérique par un astérisque, la valeur sera lue dans l'argument suivant de `fprintf()`. Cela permet de fixer la largeur d'un champ de manière dynamique. En voici une démonstration.

```
exemple-fprintf-4.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int largeur;
    int precision;

    for (largeur = 1; largeur < 10; largeur++)
        fprintf(stdout, "|%*d|\n", largeur, largeur);
    for (precision = 0; precision < 10; precision++)
        fprintf(stdout, "|%.*d|\n", precision, precision);
    return EXIT_SUCCESS;
}
```

```
$ ./exemple-fprintf-4
```

```
|1|
| 2|
| 3|
|   4|
|   5|
|   6|
|   7|
|   8|
|   9|
||
|1|
|02|
|003|
|0004|
|00005|
|000006|
|0000007|
|00000008|
|000000009|
$
```

L'intérêt de cette caractéristique est principalement de pouvoir fixer la largeur des colonnes d'un tableau pendant l'exécution du programme (éventuellement après avoir vérifié que la plus grande valeur y tient).

La permutation de paramètre est une deuxième particularité peu connue de `fprintf()`. On peut indiquer en tout début de conversion, juste après le symbole %, le numéro du paramètre qu'on désire convertir, suivi du signe \$. Ce numéro doit être supérieur ou égal à 1, et inférieur ou égal au rang du dernier argument transmis. Si on utilise cette possibilité, il faut nécessairement le faire pour toutes les conversions lors de l'appel de `fprintf()`, sinon le comportement est indéfini. L'utilité de cette fonctionnalité est de permettre de préciser l'ordre des arguments au sein même de la chaîne de formatage. Une application évidente est d'ordonner correctement les jours, mois et année de la date en fonction des désirs de l'utilisateur, uniquement en sélectionnant la bonne chaîne de formatage.

```
exemple-fprintf-5.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int main (void)
{
    int i;
    time_t timer;
    struct tm * date;
    char * format[2] =
        { "La date est %3$02d/%2$02d/%1$02d\n",
          "Today is %1$02d %2$02d %3$02d\n" };

    time(& timer);
    date = localtime(& timer);
    for (i = 0; i < 2; i++)
        fprintf (stdout, format[i],
                 date->tm_year % 100,
                 date->tm_mon + 1,
                 date->tm_mday);
    return EXIT_SUCCESS;
}
```

```
$ ./exemple-fprintf-5
La date est 04/08/10
Today is 10 08 04
$
```

On voit la puissance de cette fonctionnalité, qui permet de profiter de la phase de traduction des messages pour réordonner correctement les champs suivant la localisation.

Autres fonctions d'écriture formatée

Toutes les fonctions de la famille `printf()` renvoient le nombre de caractères écrits en sortie, ou une valeur négative en cas d'erreur. Cette valeur est rarement utilisée, aussi certains programmeurs préfixent tous leurs appels à ces fonctions d'un `(void)` destiné à indiquer aux outils de vérification statique de code, comme `splint`, que la valeur de retour est volontairement ignorée. Sous Linux, tout cela n'est pas nécessaire car `Splint`, l'outil standard de vérification de code, reconnaît les fonctions de la famille `printf()` et sait que leurs valeurs de retour peuvent être ignorées.

La fonction `printf()`, dont le prototype est :

```
int printf (const char * format, ...);
```

Elle est exactement équivalente à `fprintf(stdout, format, ...)`. Personnellement, je préfère utiliser systématiquement `fprintf()` et indiquer explicitement, à chaque

écriture, dans quel flux (`stdout` ou `stderr`) les données doivent être dirigées. C'est une simple question d'habitude.

La fonction `sprintf()` est déclarée ainsi :

```
| int sprintf (char * buffer, const char * format, ...);
```

Elle permet d'écrire les données formatées dans la chaîne fournie en premier argument, en ajoutant un caractère nul « `\0` » à la fin. Ce caractère nul n'est pas compté dans la valeur renvoyée par `sprintf()`.

Avec le développement des applications graphiques dans des environnements fenêtrés, l'utilité de `d` sur le flux de sortie `stdout` est de plus en plus réduite. Les programmes préfèrent envoyer leur sortie sur des composants graphiques (*widgets*) effectuant l'affichage de manière beaucoup plus esthétique. Il est alors courant de mettre les données en forme dans une chaîne de caractères, qu'on transmet ensuite à la bibliothèque graphique.

La chaîne de caractères envoyée en premier argument doit être assez grande pour contenir toutes les données affichées, y compris le caractère nul final. Il est alors nécessaire de dimensionner correctement cette chaîne, ce qui peut se révéler difficile.

Le fait de déborder d'une chaîne lors d'une écriture est l'une des pires choses qui puisse arriver à un programme : non seulement son comportement sera erroné, mais en plus les dysfonctionnements se produiront intempestivement, et les symptômes seront variables. Le programme peut essayer d'écrire en dehors de ses limites d'adressage autorisées, ce qui le conduit à se terminer à cause du signal `SIGSEGV`. Il peut aussi corrompre les données se trouvant au-delà de la chaîne et avoir alors un comportement indéfini. Mais, le plus grave, c'est que cette erreur peut être employée volontairement par un pirate pour créer une faille de sécurité dans le système. Nous reparlerons de ce problème dans le paragraphe consacré à la saisie de chaînes de caractères.

Ce problème ne se pose pas avec `printf()` ou `fprintf()`, car l'écriture dans un flux n'est pas limitée (ou du moins les limites sont gérées par le noyau lors de l'écriture effective, et la fonction échoue proprement).

Il existe une fonction `snprintf()`, avec le prototype suivant, permettant de régler en partie le problème :

```
| int snprintf (char * buffer, size_t taille,  
               const char * format, ...);
```

Cette fonction écrira au maximum `taille` caractères, y compris le nul final. Comme le caractère nul n'est pas compté dans la valeur de retour, cette valeur doit toujours être strictement inférieure à `taille`. Si le retour est supérieur ou égal à `taille`, `snprintf()`

nous indique alors que la chaîne est trop petite et que le formatage a été tronqué. Sur d'autres systèmes, `snprintf()` renvoie -1 en cas de dépassement.

Comme il est difficile de dimensionner au départ la chaîne correctement, on peut procéder en plusieurs étapes. Nous allons construire une routine utilitaire, qui va allouer automatiquement une chaîne de caractères de la dimension nécessaire. La libération de cette chaîne après emploi est sous la responsabilité du programme appelant.

Pour construire cette routine, nous appellerons la routine `vsnprintf()`, dont l'emploi est plus simple dans notre cas. Les routines `vprintf()`, `vfprintf()`, `vsprintf()` et `vsnprintf()` fonctionnent exactement comme leurs homologues sans « v » initial, mais reçoivent les arguments à afficher dans une table de type `va_list`, et pas sous forme de liste variable d'arguments. Le type `va_list` est défini dans `<stdarg.h>`, ainsi que des macros qui permettent de passer d'une liste variable d'arguments à une table qu'on peut parcourir. Les prototypes de ces quatre autres fonctions de la famille `printf()` sont :

```
int vprintf (const char * format, va_list arguments);
int vfprintf(FILE * flux, const char * format,
             va_list arguments);
int vsprintf(char * buffer, const char * format,
             va_list arguments);
int vsnprintf (char * buffer, size_t taille,
              const char * format, va_list arguments);
```

Au sein d'un programme, il est beaucoup plus simple d'invoquer les routines `printf()` que `vprintf()` lorsqu'on connaît le nombre d'arguments à transmettre lors de l'écriture du programme. Ce cas est bien entendu le plus fréquent. Par contre, il peut arriver qu'on ne sache pas à l'avance quels seront les arguments à transmettre, ni même leur nombre. Cette situation se présente par exemple lorsque la mise en forme et les données à afficher sont choisies dynamiquement par l'utilisateur. Un autre exemple est celui d'une routine qui sert de frontal à `printf()`, en offrant une interface assez similaire pour le programmeur qui l'invoque, mais qui effectue des tâches supplémentaires (comme une vérification des données) avant d'appeler effectivement `printf()`. Il faut alors invoquer la version `v` de cette fonction, en lui passant un tableau construit dynamiquement. Nous avons déjà rencontré la même dualité entre tableaux et listes variables d'arguments dans le chapitre 3, avec les fonctions de la famille `exec()`.

Voici un exemple d'implémentation d'une routine semblable à `sprintf()`, mais qui allouera automatiquement l'espace nécessaire pour écrire toutes les données. Elle tente de faire son allocation par étapes successives de 64 caractères (valeur purement arbitraire). Elle renvoie un pointeur `NULL` en cas d'échec d'allocation mémoire.

```
exemple-vsnprintf.c :
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * alloc_printf (const char * format, ...);

int main (void)
{
    char * chaine;
    char * seizecars = "0123456789ABCDEF";
    chaine = alloc_printf(" %s %s",
                          seizecars, seizecars);
    if (chaine != NULL) {
        fprintf(stdout, "Chaine de %d caracteres\n%s\n",
                strlen(chaine), chaine);
        free(chaine);
    }
    chaine = alloc_printf(" %s %s %s %s",
                          seizecars, seizecars, seizecars, seizecars);
    if (chaine != NULL) {
        fprintf(stdout, "Chaine de %d caracteres\n%s\n",
                strlen(chaine), chaine);
        free(chaine);
    }
    return EXIT_SUCCESS;
}

char * alloc_printf (const char * format, ...)
{
    va_list arguments;
    char * retour = NULL;
    int taille = 64;
    int nb_ecrits;
    va_start(arguments, format);
    while (1) {
        retour = realloc(retour, taille);
        if (retour == NULL)
            break;
        nb_ecrits = vsnprintf(retour, taille, format, arguments);
        if ((nb_ecrits >= 0) && (nb_ecrits < taille))
            break;
        taille = taille + 64;
    }
    va_end(arguments);
    return retour;
}
```

Nous appelons deux fois cette routine. La première, avec deux chaînes de 16 caractères, plus deux caractères blancs, ce qui tient nettement dans la tentative d'allocation initiale de 64 octets. En revanche, lors du second appel, on dépasse volontairement ces 64 caractères pour forcer une réallocation automatique.

Après utilisation de la chaîne renvoyée, on prend soin de libérer la mémoire.

```
$ ./exemple-vsnprintf
Chaîne de 34 caracteres
0123456789ABCDEF 0123456789ABCDEF
Chaîne de 71 caracteres
0123456789ABCDEF 0123456789ABCDEF 0123456789ABCDEF 0123456789ABCDEF
$
```

Cette routine peut être très utile pour remplir une chaîne de caractères avant de la transmettre à une boîte de dialogue d'une bibliothèque graphique. On est assuré qu'aucun débordement de chaîne ne risque de se produire.

Écritures simples de caractères ou de chaînes

La fonction la plus simple pour écrire un unique caractère dans un flux est `fputc()`, déclarée ainsi dans `<stdio.h>` :

```
| int fputc (int c, FILE * flux);
```

Lorsqu'on passe un caractère en argument à `fputc()`, il est tout d'abord converti en `int` avant l'appel, puis à nouveau transformé en `unsigned char` (prenant donc n'importe quelle valeur entre 0 et `UCHAR_MAX`, normalement 255). Il est ensuite envoyé dans le flux indiqué. Si une erreur se produit, `fputc()` renvoie `EOF`, sinon elle renvoie la valeur du caractère émis.

Il existe une version pour caractères larges définie dans `<wchar.h>` :

```
| wint_t fputwc (wchar_t wc, FILE * flux);
```

L'exemple suivant va mettre en relief les divers comportements de `fputc()` en fonction de ses arguments.

```
| exemple-fputc.c :
| #include <stdio.h>
| #include <stdlib.h>
| #include <limits.h>
```

```
void test_fputc (int valeur, FILE * fp);

int main (void)
{
    test_fputc('A', stdout);
    test_fputc(65, stdout);
    test_fputc(UCHAR_MAX, stdout);
    test_fputc(-1, stdout);
    test_fputc('A', stdin);
    return EXIT_SUCCESS;
}

void test_fputc (int valeur, FILE * fp)
{
    int retour;
    retour = fputc(valeur, fp);
    fprintf(stdout, "\n Ecrit : %d, ", valeur);
    fprintf(stdout, "retour = %d ", retour);
    if (retour == EOF)
        fprintf(stdout, "(EOF)");
    fprintf(stdout, "\n");
}
```

Voici le résultat de l'exécution.

```
$ ./exemple-fputc
A
Ecrit : 65, retour = 65
A
Ecrit : 65, retour = 65
?
Ecrit : 255, retour = 255
?
Ecrit : -1, retour = 255

Ecrit : 65, retour = -1 (EOF)
$
```

Lors du premier appel, le caractère A est transformé en sa valeur Ascii entière et est affiché normalement. Dans le second cas, la transformation avait été effectuée manuellement auparavant, il n'y a donc pas de différence. Dans le troisième appel, la valeur UCHAR_MAX vaut 255, qui se traduit par un caractère Unicode inexistant (affiché sous forme de point d'interrogation en négatif) .

Dans le quatrième exemple, nous voyons que la valeur entière signée -1 est traduite en son équivalent en caractère non signé, c'est-à-dire UCHAR_MAX, qui est affiché également. Nous notons ici que la valeur de retour de fputc() est 255, c'est-à-dire la valeur en carac-

tère non signé retransformée en `int`. Nous observons alors une chose importante : toutes les valeurs que `fputc()` renvoie, lorsqu'il réussit, sont comprises entre 0 et `UCHAR_MAX`.

Dans le dernier exemple, nous demandons à `fputc()` d'écrire dans le flux `stdin` qui est exclusivement ouvert en lecture. La fonction échoue donc. Mais par contre, elle nous renvoie -1, qui est la valeur attribuée à la constante symbolique `EOF` dans `<stdio.h>`. Comme nous avons observé qu'en cas de réussite la valeur renvoyée est toujours comprise entre 0 et 255, il n'y a pas d'ambiguïté possible. Nous retrouverons ce comportement dans la fonction de lecture d'un caractère, qui renvoie aussi une valeur négative en cas d'échec.

La seconde fonction de sortie de caractère, `putc()`, correspond au prototype suivant :

```
| int putc (int valeur, FILE * flux);
```

Elle se comporte exactement comme `fputc()`, mais peut être implémentée sous forme de macro. Elle est donc optimisée, mais peut évaluer plusieurs fois ses arguments. On l'utilisera donc de préférence à `fputc()`, mais en faisant attention à ne pas placer en argument des expressions ayant des effets de bord, comme `putc(table[i++])`. L'équivalent pour caractères larges est :

```
| wint_t putwc (wchar_t wc, FILE * flux);
```

Lorsque la sortie se fait sur le flux `stdout`, on peut utiliser les fonctions :

```
| int putchar(int valeur);  
| wint_t putwchar (wchar_t c);
```

qui sont équivalentes à `putc(valeur, stdout)` et `putwc(c, stdout)` ou, sans avoir besoin d'évaluer le second argument, et qui est donc encore mieux optimisée.

Lorsqu'on désire écrire une chaîne de caractères complète, on utilise les fonctions :

```
| int fputs(const char * s, FILE * fp);  
| int fputws (const wchar_t * ws, FILE *fp);
```

Ces fonctions envoient dans le flux mentionné la chaîne de caractères transmise, sans le caractère nul « `\0` » final et sans ajouter non plus de retour à la ligne.

```
| exemple-fputs.c :  
| #include <stdio.h>  
| #include <stdlib.h>  
  
| int main (int argc, char * argv[])  
| {  
|     int i;
```

```
if (argc == 1) {  
    fputs("Pas d'argument\n", stdout);  
} else {  
    fputs("Arguments : ", stdout);  
    for (i = 1; i < argc; i++)  
        fputs(argv[i], stdout);  
    fputs("\n", stdout);  
}  
return EXIT_SUCCESS;  
}
```

Cet exemple montre que les arguments vont être écrits les uns à la suite des autres, sans séparation.

```
$ ./exemple-fputs  
Pas d'argument  
$ ./exemple-fputs aze rty uiop  
Arguments : azertyuiop  
$
```

Le comportement est assez semblable à celui de `fprintf()`, mais une confusion possible vient souvent du fait que l'argument flux est le dernier et non plus le premier.

Pour écrire un message sur `stdout`, on peut utiliser la fonction `puts()` qui n'a pas d'équivalents pour caractères larges :

```
int puts (const char * message);
```

Elle écrit le message sur la sortie standard, sans le caractère nul final, mais en ajoutant automatiquement un retour à la ligne «`\n`». En remplaçant tous les `fputs(..., stdout)` par `puts(...)` dans l'exemple précédent, on obtient l'exécution suivante.

```
$ ./exemple-puts aze rty uiop  
Arguments :  
aze  
rty  
uiop  
  
$
```

Nous y trouvons évidemment des retours à la ligne supplémentaires car nous avons laissé tous les «`\n`» déjà présents dans le code précédent.

Saisie de caractères

Lorsqu'on désire lire un caractère depuis un flux, `fgetc()` fonctionne exactement à l'inverse de `fputc()`. Cette fonction est déclarée ainsi dans `<stdio.h>` :

```
int fgetc (FILE * flux);
```

Elle lit un unique caractère comme un `unsigned char` et le renvoie une fois qu'il est converti en `int`. La valeur renvoyée est donc comprise entre 0 et `UCHAR_MAX`. En cas d'échec, la valeur renvoyée est `EOF`. Cette constante symbolique est généralement définie comme égale à -1. Quoi qu'il en soit, cette constante n'est jamais située dans l'intervalle 0 à `UCHAR_MAX`. Il est donc important de lire le résultat de `fgetc()` dans une variable de type `int` et de le comparer avec `EOF` avant de le convertir en `char` si la lecture a réussi.

```
exemple-fgetc.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int i;
    while ((i = fgetc(stdin)) != EOF)
        fprintf(stdout, " %02X\n", i);
    return EXIT_SUCCESS;
}
```

Nous allons lire les données depuis le terminal. En voici un premier exemple :

```
$ ./exemple-fgetc
a
61
0A
b
62
0A
c
63
0A
abc
61
62
63
0A
  (Contrôle-D)
$
```


Nous voyons que le programme affiche d'abord la valeur correspondant au caractère a, suivie du retour à la ligne `\n` (0x0A). En effet, le terminal sous Linux n'envoie les données qu'après la validation de toute la ligne avec la touche Entrée. Nous en voyons un exemple lorsque la chaîne abc est saisie en entier, puis validée avant que les données ne soient envoyées sur le flux `stdin` du programme.

Pour terminer le programme, il faut faire échouer la lecture en lui envoyant le code EOF. Pour cela, on utilise la touche *Contrôle-D*. Ceci est configuré à l'aide de la commande `stty`. On voit, à la fin de la deuxième ligne d'affichage des résultats, que le caractère de contrôle eof est attribué à la touche *Contrôle-D*.

```
$ stty -a
speed 38400 baud; rows 34; columns 125; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?;
eol2 = M-^?; swtch = M-^?; start = ^Q; stop = ^S;
susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1;
time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
[...]
$
```

Pour pouvoir lire directement les données sans attendre le retour chariot, il faut modifier le comportement du terminal. Nous en verrons une description détaillée dans le chapitre sur la gestion des terminaux. On peut quand même agir au niveau du shell pour modifier le mode de lecture du terminal :

```
$ stty -icanon
$ ./exemple-fgetc
a 61
b 62
c 63
(Contrôle-D) 04
(Contrôle-C)
$ stty sane
$
```

La commande `stty -icanon` modifie la gestion du terminal. La fonction `fgetc()` permet alors de lire immédiatement les caractères, sans attendre leur validation par la touche Entrée. Toutefois, la touche *Contrôle-D* ne fait plus échouer la lecture, elle renvoie simplement le code normal de la touche. On arrête le programme en utilisant *Contrôle-C*, qui envoie le signal `SIGINT`. La commande `stty sane` permet de rétablir le terminal dans un état normal.

Nous reviendrons dans le chapitre consacré à la gestion des terminaux sur le moyen de modifier la configuration du terminal directement depuis l'application, et d'utiliser des lectures non bloquantes pour capturer des caractères au vol, afin que l'application puisse continuer à s'exécuter même si l'utilisateur n'a pas appuyé sur des touches.

L'équivalent pour caractères larges est :

```
wint_t fgetwc (FILE * flux);
```

On peut employer également `getc()` ou `getwc()` qui sont fonctionnellement équivalentes à `fgetc()` et `fgetwc()` mais qui peut être implémentées sous forme de macros, évaluant plusieurs fois leur argument `flux`.

Enfin, les routines `getchar()` et `getwchar()` sont respectivement équivalentes à `getc(stdin)` et `getwc(stdin)`.

Nous allons employer pour écrire une application qui affiche le contenu de son entrée standard en hexadécimal et sous forme de caractères. Ce genre d'utilitaire est souvent employé pour le débogage, pour analyser le contenu de fichiers de données binaires.

```
exemple-getchar.c :
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

int main (void)
{
    int lu;
    char caracteres[17];
    int emplacement = 0;
    int rang = 0;

    caracteres[16] = '\0';
    while ((lu = getchar()) != EOF) {
        if ((rang = emplacement % 16) == 0)
            fprintf(stdout, "%08X ", emplacement % 0xFFFFFFFF);
        fprintf(stdout, "%02X", lu);
        if (rang == 7)
            fprintf(stdout, "-");
        else
            fprintf(stdout, " ");
        if (isprint(lu))
            caracteres[rang] = lu;
        else
            caracteres[rang] = ' ';
        if (rang == 15)
            fprintf(stdout, " %s\n", caracteres);
        emplacement ++;
    }
```

```

    }
    while (rang < 15) {
        fprintf(stdout, " ");
        caracteres[rang] = '\\0';
        rang ++;
    }
    fprintf(stdout, " %s\\n", caracteres);
    return EXIT_SUCCESS;
}

```

Ce genre de programme peut être utilisé tant sur des fichiers binaires que sur des fichiers de texte.

```

$ ./exemple-getchar < exemple-getchar.c
00000000 23 69 6E 63 6C 75 64 65-20 3C 73 74 64 69 6F 2E #include <stdio.
00000010 68 3E 0A 23 69 6E 63 6C-75 64 65 20 3C 73 74 64 h> #include <std
[...]
000002B0 65 73 29 3B 0A 09 72 65-74 75 72 6E 20 45 58 49 es); return EXI
000002C0 54 5F 53 55 43 43 45 53-53 3B 0A 7D 0A          T_SUCCESS; }
$ ./exemple-getchar < exemple-getchar
00000000 7F 45 4C 46 01 01 01 00-00 00 00 00 00 00 00 00 ELF
00000010 02 00 03 00 01 00 00 00-10 84 04 08 34 00 00 00 4
00000020 B8 13 00 00 00 00 00 00-34 00 20 00 08 00 28 00 4 (
[...]
000020D0 5F 69 36 38 36 2E 67 65-74 5F 70 63 5F 74 68 75 _i686.get_pc_thu
000020E0 6E 6B 2E 62 78 00 6D 61-69 6E 00 5F 69 6E 69 74 nk.bx main _init
000020F0 00
$

```

Nous avons bien entendu éliminé de nombreuses lignes pour présenter les résultats du programme exécuté sur son propre fichier binaire exécutable. Le fonctionnement est correct *a priori*, mais au final il n'est pas très satisfaisant si le fichier contient des caractères accentués. Prenons comme exemple le fichier suivant.

```

$ cat fichier-test-accentués.txt
Ceci est un fichier de test
qui contient des caractères accentués.
À à Ç ç É é Î î Ö ö Ù ù

$

```

Si nous le soumettons à `exemple-getchar`, nous voyons les codes binaires UTF-8 représentant les caractères accentués, mais pas les caractères eux-mêmes.

```
$ ./exemple-getchar < fichier-test-accents.txt
00000000 43 65 63 69 20 65 73 74-20 75 6E 20 66 69 63 68 Ceci est un fich
00000010 69 65 72 20 64 65 20 74-65 73 74 0A 71 75 69 20 ier de test qui
00000020 63 6F 6E 74 69 65 6E 74-20 64 65 73 20 63 61 72 contient des car
00000030 61 63 74 C3 A8 72 65 73-20 61 63 63 65 6E 74 75 act res accentu
00000040 C3 A9 73 2E 0A C3 80 20-C3 A0 20 C3 87 20 C3 A7 s.
00000050 20 C3 89 20 C3 A9 20 C3-8E 20 C3 AE 20 C3 96 20
00000060 C3 B6 20 C3 99 20 C3 B9-0A 0A
$
```

Par exemple les octets 0xC3 et 0xA9 à la position 00000040 représentent le caractère 'é'. Pour obtenir une interprétation correcte de l'UTF-8, il faudra employer la fonction :

```
wint_t getwchar (void);
```

Il devient nécessaire dans ce cas d'initialiser correctement la localisation et d'utiliser l'ensemble des éléments que nous avons vu dans les chapitres 16 et 19.

```
exemple-getwchar.c :
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <wctype.h>

int main (void)
{
    wint_t lu;
    wchar_t wcaracteres[17];
    int emplacement = 0;
    int rang = 0;
    setlocale(LC_ALL, "");
    wcaracteres[16] = '\0';
    while ((lu = getwchar()) != WEOF) {
        if ((rang = emplacement % 16) == 0)
            fprintf(stdout, "%08X ", emplacement % 0xFFFFFFFF);
        fprintf(stdout, "%02X", lu);
        if (rang == 7)
            fprintf(stdout, "-");
        else
            fprintf(stdout, " ");
        if (iswprint(lu))
            wcaracteres[rang] = lu;
        else
            wcaracteres[rang] = L' ';
        if (rang == 15)
```

```

        fprintf(stdout, " %ls\n", wcaracteres);
        emplacement ++;
    }
    while (rang < 15) {
        fprintf(stdout, " ");
        wcaracteres[rang] = L'\0';
        rang ++;
    }
    fprintf(stdout, " %ls\n", wcaracteres);
    return EXIT_SUCCESS;
}

```

Cette fois-ci l’affichage des caractères est correct (même si la représentation hexadécimale est celle des caractères Unicode lus, pas celles des octets du fichier).

```

$ ./exemple-getwchar < fichier-test-accents.txt
00000000 43 65 63 69 20 65 73 74-20 75 6E 20 66 69 63 68 Ceci est un fich
00000010 69 65 72 20 64 65 20 74-65 73 74 0A 71 75 69 20 ier de test qui
00000020 63 6F 6E 74 69 65 6E 74-20 64 65 73 20 63 61 72 contient des car
00000030 61 63 74 E8 72 65 73 20-61 63 63 65 6E 74 75 E9 actères accentué
00000040 73 2E 0A C0 20 E0 20 C7-20 E7 20 C9 20 E9 20 CE s. À à Ç ç É é Î
00000050 20 EE 20 D6 20 F6 20 D9-20 F9 0A 0A          î Ö ö Ù ù
$

```

Réinjection de caractère

Il peut arriver qu’on veuille en quelque sorte annuler la lecture d’un caractère. Imaginons une routine qui doit lire des caractères uniquement numériques et qui s’arrête dès qu’elle rencontre un caractère ne se trouvant pas dans l’intervalle ‘0’ - ‘9’. La suite du traitement sera prise en charge par une autre routine, qui agira en fonction du nouveau caractère lu. Une des éventualités serait de toujours conserver le caractère lu dans une variable globale, la lecture ayant toujours un caractère d’avance sur le traitement proprement dit. C’est d’ailleurs la méthode généralement employée par les analyseurs lexicaux, qui fonctionnent avec des mots complets (*token*).

Une autre possibilité serait de replacer dans le flux d’entrée le dernier caractère lu, pour que la prochaine lecture le renvoie à nouveau. Comme les flux fonctionnent en utilisant des mémoires tampons, il ne s’agit pas d’une véritable écriture dans le fichier associé, mais simplement d’un ajout en tête de buffer. Les routines assurant cette tâche sont `ungetc()` déclarée dans `<stdio.h>` et `ungetwc()` dans `<wchar.h>` :

```

int    ungetc(int caractere_lu, FILE * flux);
wint_t ungetwc (wint_t caractere, FILE * flux);

```

Cette routine remplace le caractère transmis dans le flux. Le premier argument est de type `int` ou `wint_t`, car on peut également lui transmettre la constante symbolique `EOF` ou `WEOF`. Cela permet au besoin de transmettre directement à `ungetc(w)c()` le résultat de `fgetc(w)c()`. Il n'est possible de remplacer dans le flux qu'un seul caractère, et il est inutile d'invoquer plusieurs fois de suite `ungetc(w)c()`. Le comportement est indéfini pour ce qui est de savoir si le dernier caractère transmis écrasera les précédents. Notons également que le caractère qu'on remplace dans le flux n'est pas nécessairement celui qu'on vient de lire.

L'exemple que nous allons construire est un peu artificiel : deux routines sont chargées de lire caractère par caractère l'entrée standard. L'une s'occupe des caractères numériques, l'autre des caractères alphabétiques. La routine `main()` centrale lit un caractère puis, s'il correspond à l'une des deux classes de caractères définies précédemment, elle réinjecte le caractère lu dans le flux d'entrée, et invoque la routine spécialisée correspondante. Ces routines sont construites de manière à lire tout ce qui arrive puis, dès qu'un caractère ne leur convient pas, elles le replacent dans le flux, et reviennent à la fonction `main()`.

```
exemple-ungetc.c :
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

void lecture_numerique (FILE * fp);
void lecture_alphabetique(FILE * fp);

int main (void)
{
    int c;
    while ((c = getc(stdin)) != EOF) {
        if (isdigit(c)) {
            ungetc(c, stdin);
            lecture_numerique(stdin);
        } else if (isalpha(c)) {
            ungetc(c, stdin);
            lecture_alphabetique(stdin);
        }
    }
    return EXIT_SUCCESS;
}

void lecture_numerique (FILE * fp)
{
    int c;
    fprintf(stdout, "Lecture numerique : ");
    while (1) {
        c = getc(fp);
```

```
        if (! isdigit(c))
            break;
        fprintf(stdout, "%c", c);
    }
    ungetc(c, fp);
    fprintf(stdout, "\n");
}

void lecture_alphabetique (FILE * fp)
{
    int c;
    fprintf(stdout, "Lecture alphabetique : ");
    while (1) {
        c = getc(fp);
        if (! isalpha(c))
            break;
        fprintf(stdout, "%c", c);
    }
    ungetc(c, fp);
    fprintf(stdout, "\n");
}
```

Voici un exemple d'exécution.

```
$ ./exemple-ungetc
AZE123 ABCDEF9875XYZ
Lecture alphabetique : AZE
Lecture numerique : 123
Lecture alphabetique : ABCDEF
Lecture numerique : 9875
Lecture alphabetique : XYZ
$
```

Saisie de chaînes de caractères

Pour lire une chaîne de caractères Ascii, il existe deux fonctions : `gets()` et `fgets()`. Le prototype de `gets()` est le suivant :

```
char * gets (char * buffer);
```

Cette fonction lit l'entrée standard `stdin` et place les caractères dans la chaîne passée en argument. Lorsqu'elle rencontre le caractère EOF ou un retour chariot, elle les remplace par le caractère nul de fin de chaîne `'\0'`, et renvoie le pointeur sur la chaîne. Si le caractère EOF est rencontré avant qu'elle ait pu lire un seul caractère, `gets()` renvoie le pointeur `NULL`.

Important

Il ne faut jamais utiliser `gets()` !

En effet, `gets()` ne permet pas de préciser la longueur maximale de la chaîne à saisir. En conséquence, si le nombre de caractères reçus excède la taille de la zone qu'on a allouée, `gets()` continuera joyeusement à écrire en mémoire en provoquant un débordement de buffer.

`gets()` servant généralement à lire une chaîne de caractères tapée au clavier par l'utilisateur, on pourrait croire qu'allouer un buffer suffisamment grand éviterait tout problème. Malheureusement, il suffit de rediriger l'entrée standard du processus en provenance d'un fichier pour que la saisie puisse prendre n'importe quelle longueur.

Dans le meilleur des cas, le programme ira écrire en dehors de son espace d'adressage autorisé par le noyau et sera alors terminé par un signal `SIGSEGV`. Mais un grave problème de sécurité peut aussi survenir si le programme est installé avec un bit *Set-UID* ou *Set-GID*. En C, les données automatiques des fonctions (celles qui ne sont pas déclarées statiques) sont allouées dans la pile. Lors de l'entrée dans une fonction, l'adresse de retour et les arguments sont empilés. Ensuite, on réserve dans la pile la place nécessaire aux variables automatiques. Par exemple, lors de l'entrée dans la routine suivante :

```
int fonction (int x)
{
    int x1;
    char chaine_1[128];
    ...
}
```

on stocke successivement sur la pile l'argument `x`, l'adresse de retour, puis on réserve 4 octets pour `x1`, et 128 octets pour la chaîne. Sous Linux x86, la pile croît vers le bas, signifiant que l'adresse de `chaine_1[0]` est plus petite que celle de `x1`, qui est elle-même inférieure à l'adresse de retour et à l'adresse de `x`. Voici un exemple pour clarifier la situation :

```
exemple-pile.c :
#include <stdio.h>
#include <stdlib.h>

int fonction (int x);

int main (void)
{
    return fonction(1);
}
```



```

int fonction (int x)
{
    int x1;
    char chaine[128];

    printf("& x    = %p lg = %d\n", & x, sizeof(x));
    printf("& x1   = %p lg = %d\n", & x1, sizeof(x1));
    printf("chaine = %p lg = %d\n", chaine, sizeof(chaine));
    if (x > 0)
        return fonction(x - 1);
    return EXIT_SUCCESS;
}

```

La fonction s'appelle elle-même une fois, pour pouvoir déduire la position de l'adresse de retour par rapport à l'argument. Lors de l'exécution, nous obtenons ceci :

```

$ ./exemple-pile
& x    = 0xbf8e8a50 lg = 4
& x1   = 0xbf8e8a3c lg = 4
chaine = 0xbf8e89bc lg = 128
& x    = 0xbf8e89a0 lg = 4
& x1   = 0xbf8e898c lg = 4
chaine = 0xbf8e890c lg = 128
$

```

Il est clair que lors de la deuxième invocation de `fonction()`, la pile est structurée ainsi.

Adresse début	Adresse fin	Taille	Contenu
0xbf8e8a50	0xbf8e8a54	4	Argument x (valant 1)
0xbf8e8a4C	0xbf8e8a4f	4	Adresse de retour
0xbf8e8a40	0xbf8e8a4b	12	Sauvegarde %ebp et réserve pour alignement
0xbf8e8a3c	0xbf8e8a3f	4	Variable automatique x1
0xbf8e89bc	0xbf8e8a3b	128	Variable automatique chaine[]
0xbf8e89a4	0xbf8e89bb	24	Remplissage pour alignement ?
0xbf8e89a0	0xbf8e89a3	4	Second argument x (valant 0)
0xbf8e899C	0xbf8e899f	4	Seconde adresse de retour
0xbf8e8990	0xbf8e899b	12	Sauvegarde %ebp et réserve pour alignement
0xbf8e898c	0xbf8e898f	4	Seconde variable automatique x1
0xbf8e890c	0xbf8e898b	128	Seconde variable automatique chaine[]

Si, lors d'une lecture avec `gets()`, nous débordons de la chaîne allouée dans la pile, nous allons écraser d'abord `x1` – ce qui n'est pas très grave –, mais également l'adresse de retour. Lorsque la fonction va se terminer, le programme va essayer de revenir à une adresse erronée et va avoir un comportement incohérent, difficile à déboguer.

Si le programme est *Set-UID*, la situation est encore pire car un pirate peut l'exploiter en faisant volontairement déborder la chaîne (en fournissant des données depuis un fichier binaire). Il s'arrangera pour glisser du code valide dans la pile et fera pointer l'adresse de retour sur ce code. Le programme *Set-UID* exécutera alors exactement ce que veut son utilisateur mais avec l'identité du propriétaire du fichier exécutable. Obtenir un *shell root* est alors très simple. On comprend mieux à présent l'intérêt de limiter les privilèges d'une application *Set-UID* en diminuant son ensemble de capacités.

Une grande partie des failles de sécurité découvertes dans les programmes *Set-UID* sont dues à ce genre de problèmes. Il ne faut donc jamais utiliser `gets()`. D'ailleurs, l'éditeur de lien Gnu « `ld` » signale aussi qu'il ne faut pas utiliser cette fonction.

```
exemple-gets.c :
#include <stdio.h>

int main (void)
{
    char chaine[128];

    return (gets(chaine) != NULL);
}
```

Lors de la compilation, on obtient le message suivant :

```
$ cc -Wall exemple-gets.c -o exemple-gets
/tmp/ccEdej0J.o: In function 'main':
exemple-gets.c:(.text+0x14): warning: the 'gets' function is
dangerous and should not be used.
$
```

Il n'existe pas d'équivalent pour caractères larges de `gets()`, et c'est une très bonne chose ! Au contraire, `fgets()` est bien plus robuste puisqu'elle permet de limiter la taille de la saisie. Son prototype est le suivant :

```
char * fgets (char * chaine, int taille, FILE * flux);
```

Cette fonction lit les caractères sur le flux indiqué et les place dans la chaîne transmise en argument. En aucun cas, elle ne dépassera `taille-1` caractères lus. Elle

s'arrêtera également si elle rencontre un caractère de retour à la ligne '\n' ou une fin de fichier EOF. Le caractère '\n' éventuel est écrit dans le buffer. Ensuite, fgets() termine la chaîne par un caractère nul.

L'équivalent pour caractères larges est :

```
wchar_t * fgetws (wchar_t * chaine, int taille, FILE * flux);
```

Ces routines renvoient le pointeur sur la chaîne passée en argument lorsqu'elles réussissent. Au contraire, si elles ont rencontré le caractère EOF (ou WEOF) avant d'avoir pu lire quoi que ce soit, elles renvoient un pointeur NULL. Si on désire lire une ligne en entier, quelle que soit sa longueur, il est possible d'écrire une routine qui encadre fgets() et qui alloue de la mémoire jusqu'à la fin de la ligne. Il s'agit du même genre de fonctionnalité que celle que nous avons créée pour sprintf().

```
exemple-fgets.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * alloc_fgets (FILE * fp);

int main (void)
{
    char * chaine;
    while (1) {
        chaine = alloc_fgets(stdin);
        if (chaine == NULL)
            // Pas assez de mémoire
            break;
        if ((chaine[0] == '\n') || (chaine[0] == '\0'))
            // Chaîne vide... on quitte
            break;
        fprintf(stdout, "%d caracteres\n", strlen(chaine));
        free(chaine);
    }
    return EXIT_SUCCESS;
}

char * alloc_fgets (FILE * fp)
{
    char * retour = NULL;
    char * a_ecrire = NULL;
    int taille = 64;

    retour = malloc (taille);
    retour[0] = '\0';
    a_ecrire = retour;
```

```

while (1) {
    if (fgets(a_ecrire, 64, fp) == NULL)
        break;
    if (a_ecrire[strlen(a_ecrire) - 1] == '\n')
        a_ecrire[strlen(a_ecrire) - 1] = '\0';
    if (strlen(a_ecrire) < 63)
        break;
    // On se place sur le caractère nul final
    a_ecrire = a_ecrire + 63;
    // Et on agrandit également le buffer de 63 caractères
    taille += 63;
    retour = realloc(retour, taille);
    if (retour == NULL)
        break;
}
return (retour);
}

```

Le programme lit des chaînes de caractères et en affiche la longueur jusqu'à ce qu'il reçoive une chaîne vide, puis il se termine.

```

$ ./exemple-fgets
ABCDEFGHJKLMNOPQRSTUVWXYZ
27 caracteres
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz12345678901234567890
73 caracteres
ABC
4 caracteres
(Entrée)
$

```

On remarquera que dans les longueurs affichées, le caractère de retour à la ligne induit par la touche de validation Entrée est comptabilisé. On voit bien qu'avec la chaîne de 73 caractères, la saisie a effectué l'allocation en deux étapes et a bien renvoyé tous les caractères (26 lettres + 26 lettres + 20 chiffres + retour chariot = 73).

La bibliothèque Glibc offre une routine assez semblable, mais qui a l'inconvénient d'être une extension Gnu, donc de ne pas être disponible sur d'autres systèmes. Il est quand même conseillé de s'en servir, et on pourra toujours la redéfinir en utilisant la même méthode que pour notre exemple précédent si le programme doit être porté sur un système différent. Cette routine est nommée `getline()`, et elle est déclarée ainsi dans `<stdio.h>` :

```

| ssize_t getline (char ** chaine, size_t * taille, FILE * flux);

```

Son utilisation est légèrement moins intuitive, puisqu'elle prend en argument un pointeur sur un pointeur de chaîne de caractères et un pointeur sur une valeur de longueur. Elle tente tout d'abord d'effectuer la lecture dans la chaîne existante, qui doit avoir (*taille) octets au moins. Si cela suffit, elle renvoie le nombre de caractères lus, sans compter le caractère nul final qu'elle ajoute. Sinon, elle réalloue de la mémoire, en modifiant le pointeur chaîne et la taille, jusqu'à ce que la ligne lue tienne en entier dans la chaîne. On peut également l'invoquer avec un pointeur *chaîne valant NULL, et *taille valant zéro, elle assurera l'allocation initiale.

La fin de la ligne est déterminée par EOF ou par le retour chariot. Si EOF arrive dès le début de la lecture ou si une autre erreur se produit, `getline()` renvoie -1 (ce qui explique le type `ssize_t` de la fonction, c'est-à-dire `signed size_t`).

L'avantage de cette routine c'est qu'elle renvoie le nombre de caractères lus. Dans la routine que nous avons écrite, ce nombre ne pouvait être défini qu'à l'aide de `strlen()`. Malheureusement, si la chaîne lue contient un caractère nul, `strlen()` s'arrêtera à ce niveau. Cela peut parfois poser des problèmes lors de la redirection d'un fichier binaire en entrée.

Voyons un exemple d'utilisation de `getline()`, dans le même genre que le précédent :

```
exemple-getline.c :
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void)
{
    char * chaîne;
    size_t taille;
    ssize_t retour;
    while (1) {
        taille = 0;
        chaîne = NULL;
        retour = getline(& chaîne, & taille, stdin);
        if (retour == -1)
            break;
        fprintf(stdout, "%d caracteres lus\n", retour);
        fprintf(stdout, "%d caracteres alloues\n", taille);
        free(chaîne);
    }
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, on arrête le programme en tapant directement sur *Contrôle-D* (EOF) en début de ligne pour provoquer un échec. Nous affichons également la taille du buffer alloué par la routine, afin de pouvoir le dépasser volontairement lors de la seconde saisie.

```
$ ./exemple-getline
ABCDEFGHJKLMNOPQRSTUVWXYZ
27 caracteres lus
120 caracteres alloues
ABCDEFGHJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOP
QRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF
GHIJKLMNOPQRSTUVWXYZ
157 caracteres lus
240 caracteres alloues
(Contrôle-D)
$
```

Nous avons vu comment lire des caractères ou des chaînes. Nous allons à présent nous intéresser à la manière de recevoir des informations correspondant à d'autres types de données.

Lectures formatées depuis un flux

La saisie formatée de données se fait avec les fonctions de la famille `scanf()`. Comme pour la famille `printf()`, il existe six versions ayant les prototypes suivants :

```
int scanf (const char * format, ...);
int vscanf(const char * format, va_list arguments);

int fscanf(FILE * flux, const char * format, ...);
int vfscanf (FILE * flux, const char * format,
             va_list arguments);

int sscanf(const char * chaine, const char * format, ...)
int vsscanf (const char * chaine, const char * format,
             va_list args);
```

Pareillement, il existe en fait trois types de fonctions, chacune disponible en deux versions, avec un nombre variable d'arguments ou avec une table d'arguments. Ce dernier type nécessite l'inclusion du fichier d'en-tête `<stdarg.h>`.

- `scanf()` et `vscanf()` lisent les données en provenance de `stdin`.
- `fscanf()` et `vfscanf()` analysent les informations provenant du flux qu'on transmet en premier argument.
- `sscanf()` et `vsscanf()` effectuent la lecture formatée depuis la chaîne de caractères transmise en premier argument.

L'argument de format se présente comme une chaîne de caractères semblable à celle qui est employée avec `printf()`, mais avec quelques différences subtiles.

Les arguments fournis ensuite sont des pointeurs sur les variables qu'on désire remplir. Les fonctions renvoient le nombre de variables qu'elles ont réussi à remplir correctement.

Contrairement à `printf()`, qui est assez tolérante avec le formatage demandé puisqu'elle assure de toute manière une conversion de type, il faut indiquer ici, dans la chaîne de format, le bon type de donnée correspondant au pointeur à remplir. Si on demande par exemple à `scanf()` de lire un réel double et qu'on lui transmette un pointeur sur un char, le compilateur fournira un avertissement, mais rien de plus. Lors de l'exécution du programme, l'écriture débordera de la place mémoire réservée au caractère.

Voici un exemple de programme met en relief ce type de débordement. Une variable `int` puis une variable `char` sont réservées dans la pile. Puis on remplit avec `scanf()` la variable `char` en la considérant comme un entier. Naturellement, nous allons écraser le contenu de la variable `int` juxtaposée.

```
exemple-scanf-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (void)
{
    int i;
    char c;

    i = 2010;
    printf("i = %d\n", i);
    printf("Entrez un nombre : ");
    scanf("%d", & c);
    printf("i = %d\n", i);

    return 0;
}
```

Et voici le résultat, dont il n'y a pas lieu d'être fier... Le contenu de la variable `i` est difficile à prévoir après l'écrasement, car c'est l'octet de poids fort de la valeur saisie qui vient l'écraser.

```
$ ./exemple-scanf-1
i = 2010
Entrez un nombre : 1
i = 0
$ ./exemple-scanf-1
i = 2010
Entrez un nombre : 65535
i = 255
$
```

Voyons donc à présent quels sont les bons indicateurs à fournir dans la chaîne de format, en correspondance avec le type de donnée à utiliser.

Type	Format
char	%c
char *	%s
short int	%hd %hi
unsigned short int	%du %do %dx %dX
int	%d %i
unsigned int	%u %o %x %X
long int	%ld %li
unsigned long int	%lu %lo %lx %lX
long long int	%lld %Ld %lli %Li
unsigned long long int	%llu %Lu %llo %Lo %llx %Lx %llX %LX
float	%e %f %g
double	%le %lf %lg
long double	%lle %Le %llf %Lf %llg %Lg
void *	%p

Nous voyons qu’il y a en définitive quelques indicateurs principaux et des modificateurs de type. Les indicateurs généraux sont les suivants.

Indicateur	Type
d	Valeur entière signée sous forme décimale.
i	Valeur entière signée exprimée comme les constantes en C (avec un préfixe 0 pour l’octal, 0x pour l’hexadécimal...).
u	Valeur entière non signée sous forme décimale.
o	Valeur entière non signée en octal.
x ou X	Valeur entière non signée en hexadécimal.
e, f ou g	Valeur réelle.
s	Chaîne de caractères sans espace.
c	Un ou plusieurs caractères.

À cela s’ajoutent les modificateurs h pour short, l pour long (dans le cas des entiers) pour double (dans le cas des réels) ou large pour les caractères et chaînes, et ll ou L pour long long ou pour long double.

Notons qu'on peut insérer entre le caractère % et l'indicateur de conversion une valeur numérique représentant la taille maximale à accorder à ce champ. Ce détail est précieux avec la conversion %s pour éviter un débordement de chaîne. Il est également possible de faire précéder cette longueur d'un caractère 'a', qui demandera à scanf() d'allouer automatiquement la mémoire nécessaire pour la chaîne de caractères à lire. Cela n'a de sens qu'avec une conversion de type %s. Dans ce dernier cas, il faut transmettre un pointeur de type char **.

L'indicateur de conversion 'c' est précédé d'une valeur numérique : il indique le nombre de caractères qu'on désire lire. Par défaut, on lit un seul caractère, mais il est ainsi possible de lire des chaînes de taille quelconque. Contrairement à la conversion s, la lecture ne s'arrête pas au premier caractère blanc. On peut ainsi lire des chaînes contenant n'importe quel caractère d'espacement. En revanche, scanf() n'ajoute pas de caractère nul à la fin de la chaîne, il faut le placer soi-même.

Lorsqu'un caractère non blanc est présent dans la chaîne de format, il doit être mis en correspondance avec la chaîne reçue depuis le flux de lecture. Cela permet d'analyser facilement des données provenant d'autres programmes, si le format d'affichage est bien connu. En voici un exemple :

```
exemple-scanf-2.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int i, j, k;

    if (fscanf(stdin, "i = %d j = %d k = %d", & i, & j, & k) == 3)
        fprintf(stdout, "Ok (%d, %d, %d)\n", i, j, k);
    else
        fprintf(stdout, "Erreur\n");

    return EXIT_SUCCESS;
}
```

Ce programme réussit lorsqu'on lui fournit une ligne construite sur le modèle prévu, mais il échoue sinon.

```
$ ./exemple-scanf-2
i=1 j=2 k=3
Ok (1, 2, 3)
$ ./exemple-scanf-2
i = 4   j =   5 k = 006
Ok (4, 5, 6)
```

```
$ ./exemple-scanf-2
45 67 89
Erreur
$
```

Ici, les caractères blancs dans la chaîne de format servent à éliminer tous les caractères blancs éventuels présents dans la ligne lue. Les concepteurs de la bibliothèque `stdio` devaient être d'humeur particulièrement facétieuse le jour où ils ont défini le comportement de `scanf()` vis-à-vis des caractères blancs et de la gestion d'erreur. En effet, lorsque `scanf()` reçoit un caractère qu'elle n'arrive pas à mettre en correspondance avec sa chaîne de format, elle le réinjecte dans le flux de lecture avec la fonction `ungetc()`. Ceci se produit par exemple lorsqu'on attend un caractère particulier et qu'un autre arrive, ou lorsqu'on attend un entier et qu'on reçoit un caractère alphabétique. De nombreux débutants en langage C se sont arrachés les cheveux sur le comportement a priori incompréhensible de programmes comme celui-ci.

```
exemple-scanf-3.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int i;
    do {
        printf("Veuillez entrer un entier : ");
    } while (scanf("%d", & i) != 1);
    printf ("Ok, vous avez saisi : %d\n", i);
    return EXIT_SUCCESS;
}
```

La saisie se passe très bien tant que l'utilisateur ne commet pas d'erreur.

```
$ ./exemple-scanf-3
Veuillez entrer un entier : 123
Ok, vous avez saisi : 123
$
```

En revanche, si on entre un caractère alphabétique à la place d'un chiffre, `scanf()` le refuse, le réinjecte dans le flux et indique qu'elle n'a pu faire aucune conversion. Toute notre belle gestion d'erreur s'effondre alors, car à la tentative suivante nous allons relire à nouveau le même caractère erroné ! Cela se traduit alors par une avalanche de messages d'erreur que seul un *Contrôle-C* peut interrompre.

```
$ ./exemple-scanf-3
Veuillez entrer un entier : A
Veuillez entrer un entier : Veuillez entrer un entier : Veuillez
entrer un entier : Veuillez entrer un entier : Veuillez entrer un
entier : Veuillez entrer un entier : Veuillez entrer un entier :
Veuillez entrer un entier : Veuillez entrer un entier : Veuillez
entrer un entier : Veuillez entrer un
[...]
(Contrôle-C)
$
```

Le seul moyen simple de gérer ce genre de problème est de passer par une étape de saisie intermédiaire de ligne, à l'aide de la fonction `fgets()`.

```
exemple-scanf-4.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char ligne[128];
    int i;
    do {
        printf("Veuillez entrer un entier : ");
        if (fgets(ligne, 128, stdin) == NULL)
            return EXIT_FAILURE;
    } while (sscanf(ligne, "%d", & i) != 1);
    printf ("Ok, vous avez saisi : %d\n", i);
    return EXIT_SUCCESS;
}
```

Cette fois-ci, le comportement est celui qu'on attend.

```
$ ./exemple-scanf-4
Veuillez entrer un entier : 123
Ok, vous avez saisi : 123
$ ./exemple-scanf-4
Veuillez entrer un entier : A
Veuillez entrer un entier : Z
Veuillez entrer un entier : E
Veuillez entrer un entier : 456
Ok, vous avez saisi : 456
$
```

L'autre piège classique de `scanf()` c'est qu'un caractère blanc dans la chaîne de format élimine tous les caractères blancs présents dans le flux en lecture. Lorsqu'on parle de caractères blancs, il s'agit de l'espace et de la tabulation bien sûr, mais également des retours à la ligne. En fait, il s'agit des caractères correspondant à la fonction `isspace()` que nous avons vue dans le chapitre 19, c'est-à-dire l'espace, les tabulations verticales et horizontales `'\t'` et `'\v'`, le saut de ligne `'\n'`, le retour chariot `'\r'`, et le saut de page `'\f'`.

Cela a des conséquences inattendues sur un programme aussi simple que celui-ci.

```
exemple-scanf-5.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int i;
    fprintf(stdout, "Entrez un entier : ");
    if (scanf("%d", & i) == 1)
        fprintf(stdout, "Ok i=%d\n", i);
    else
        fprintf(stdout, "Erreur\n");
    return EXIT_SUCCESS;
}
```

Tout se passe correctement avec ce programme.

```
$ ./exemple-scanf-5
Entrez un entier : 123
Ok i=123
$ ./exemple-scanf-5
Entrez un entier : A
Erreur
$
```

À présent, supposons qu'on introduise un caractère blanc supplémentaire à la fin de la chaîne de format. Par exemple, on pourrait en croyant bien faire y ajouter un retour à la ligne `'\n'` pour marquer la fin de la saisie. La ligne de `scanf()` deviendrait :

```
if (scanf ("%d\n", & i) == 1)
```

Mais le comportement serait particulièrement surprenant.

```
$ ./exemple-scanf-6
Entrez un entier : 123 (Entrée)
(Entrée)
(Entrée)
(Tabulation)(Entrée)
(Entrée)
A(Entrée)
Ok i=123
$
```

Nous avons appuyé trois fois sur la touche « *Entrée* » à la suite de notre saisie, pressé la tabulation, la barre d'espace, puis en désespoir de cause, nous avons retapé une lettre quelconque (A) suivie de « *Entrée* ». Et c'est à ce moment seulement que notre saisie initiale a été validée !

Pourtant ce fonctionnement est tout à fait normal. Comme nous avons mis un '\n' en fin de chaîne de format – mais le résultat aurait été le même avec n'importe quel caractère blanc – `scanf()` élimine tous les caractères blancs se trouvant à la suite de notre saisie décimale. Seulement, pour pouvoir éliminer tous les caractères blancs, elle est obligée d'attendre d'en recevoir un qui ne soit pas blanc. Tout ceci explique l'inefficacité de nos multiples pressions sur la touche « *Entrée* », et qu'il ait fallu attendre un caractère non blanc, en l'occurrence A, pour que `scanf()` se termine. Notons que ce caractère non blanc est remplacé dans le flux pour la lecture suivante.

Le comportement de `scanf()` est parfois déroutant lorsqu'elle agit directement sur les flux. Pour cela, il est souvent préférable de faire la lecture ligne par ligne grâce à `fgets()` ou à `getline()`, et d'analyser ensuite le résultat avec `sscanf()`. Celle-ci aurait en effet, dans notre dernier exemple, rencontré la fin de la chaîne, qu'elle aurait traitée comme un EOF, ce qui lui aurait permis d'arrêter la recherche d'un caractère non blanc. En voici la preuve avec le programme suivant (le test d'erreur sur `fgets()` a été supprimé pour simplifier l'exemple).

```
exemple-scanf-7.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char ligne[128];
    int i;
    fprintf(stdout, "Entrez un entier : ");
    fgets(ligne, 128, stdin);
```

```
if (sscanf(ligne, "%d\n", & i) == 1)
    fprintf(stdout, "Ok i=%d\n", i);
else
    fprintf(stdout, "Erreur\n");
return EXIT_SUCCESS;
}
```

```
$ ./exemple-scanf-7
Entrez un entier : 123
Ok i=123
$ ./exemple-scanf-7
Entrez un entier : ABC
Erreur
$
```

Les fonctions de la famille `scanf()` offrent également quelques possibilités moins connues que nous allons voir rapidement.

- La saisie de l'adresse d'un pointeur, avec la directive `%p` : ceci ne doit être utilisé qu'avec une extrême précaution, le programme étant prêt à capturer un signal `SIGSEGV` dès qu'il va essayer de lire le contenu du pointeur si l'utilisateur a fait une erreur.
- La lecture d'un champ sans stockage dans une variable, en insérant un astérisque juste après le caractère `%`. Le champ est purement et simplement ignoré, sans stockage dans un pointeur ni incrémentation du nombre de champs correctement lus. Ceci est surtout utilisé pour ignorer des valeurs lors de la relecture de la sortie d'autre programme. Imaginons par exemple un programme de dessin vectoriel qui affiche les coordonnées `X` et `Y` de tous les points qu'il a en mémoire. Lors d'une relecture de ces données, le numéro du point ne présente pas d'intérêt, aussi préfère-t-on l'ignorer avec une lecture du genre :

```
scanf (" point %*d : X = %lf Y = %lf", & x, & y).
```

- La directive `%n` n'effectue pas de conversion mais stocke dans le pointeur correspondant, qui doit être de type `int *`, le nombre de caractères lus jusqu'à présent. Cela peut servir dans l'analyse d'une chaîne contenant plusieurs champs. Supposons par exemple que le premier champ indique de manière numérique le type du champ suivant (0 = entier, 1 = réel). Il est alors commode de stocker la position atteinte après cette première lecture, pour reprendre ensuite l'extraction avec le format approprié dans un second `sscanf()`. En voici une illustration :

```
exemple-scanf-8.c :
#define _GNU_SOURCE /* Pour avoir getline() */

#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char * ligne;
    size_t taille;

    int position;
    int type_champ;
    int entier;
    float reel;

    while (1) {
        fprintf(stdout, "<type> <valeur> :\n");
        ligne = NULL;
        taille = 0;
        if (getline(& ligne, & taille, stdin) == -1)
            break;
        if (sscanf(ligne, "%d %n", & type_champ, & position) != 1) {
            fprintf(stdout, "Entrez le type (0=int, 1=float) "
                "suivi de la valeur\n");
            free(ligne);
            continue;
        }
        if (type_champ == 0) {
            if (sscanf(& (ligne[position]), "%d", &entier) != 1)
                fprintf(stdout, "Valeur entiere attendue\n");
            else
                fprintf(stdout, "Ok : %d\n", entier);
        } else if (type_champ == 1) {
            if (sscanf(& (ligne[position]), "%f", & reel) != 1)
                fprintf(stdout, "Valeur reellevue attendue\n");
            else
                fprintf(stdout, "Ok : %e\n", reel);
        } else {
            fprintf(stdout, "Type inconnu (0 ou 1)\n");
        }
        free(ligne);
    }
    return EXIT_SUCCESS;
}
```

On arrête la boucle principale de ce programme en faisant échouer `getline()`, en lui envoyant EOF (*Contrôle-D*) en début de ligne. Voici un exemple d'exécution.

```
$ ./exemple-scanf-8
<type> <valeur> :
(Entrée)
Entrez le type (0=int, 1=float) suivi de la valeur
<type> <valeur> :
0 12345
Ok : 12345
<type> <valeur> :
1 23.45
Ok : 2.345000e+01
<type> <valeur> :
0 A
Valeur entiere attendue
<type> <valeur> :
1 B
Valeur reelle attendue
<type> <valeur> :
2
Type inconnu (0 ou 1)
<type> <valeur> :
$
```

Il est également possible de restreindre le jeu de caractères utilisables lors d'une saisie de texte, en utilisant une directive `%[` à la place de `%s`, et en indiquant à l'intérieur des crochets les caractères autorisés. On peut signaler des intervalles du genre `%[A-Za-z]`, des négations avec le signe `^` en début de directive, comme `%[^0-9]` pour refuser les chiffres. Si on veut mentionner le caractère `']`, il faut le placer en premier, et pour indiquer `'['`, on le place en dernier, comme dans `%[(){}[]`, qui regroupe les principaux symboles d'encadrement. On notera que cette conversion ne saute pas automatiquement les espaces en tête, contrairement à `%s`. Comme pour cette dernière conversion, il y a lieu d'être prudent pour éviter les débordements de chaînes, soit en mentionnant une taille maximale `%5[A-Z]` qui convertit au plus cinq majuscules, soit en demandant à la bibliothèque d'allouer la mémoire nécessaire (en lui passant un pointeur sur un pointeur sur une chaîne).

Avec toutes leurs possibilités, les fonctions de la famille `scanf()` sont très puissantes. Toutefois, elles réclament beaucoup d'attention lors de la lecture des données si plusieurs champs sont présents sur la même ligne. Lorsque la syntaxe d'une ligne est très compliquée et qu'une lecture champ par champ comme dans notre dernier exemple est vraiment rébarbative, il est possible de se tourner vers un analyseur syntaxique qu'on pourra construire à l'aide de `flex` et `bison`, par exemple.

Conclusion

Nous avons examiné ici les différentes fonctions d'entrée-sortie simples pour un programme.

Comme nous l'avons déjà indiqué avec `printf()`, l'évolution actuelle des interfaces graphiques conduit les utilisateurs à se détourner des applications dont les données sont saisies depuis un terminal classique. À moins de construire un programme qui, à la manière d'un filtre, reçoive sur son entrée standard des données provenant d'une autre application, il est de plus en plus rare d'utiliser `scanf()` ou `fscanf()` sur `stdin`. Toutefois, l'emploi de `sscanf()` est toujours d'actualité. En effet, la saisie de données par l'intermédiaire d'une interface graphique se fait souvent dans une boîte de dialogue, dont les composants de saisie renvoient leur contenu sous forme de chaîne de caractères. Il est alors du ressort du programme appelant de convertir ces chaînes dans le format de donnée qu'il désire (entier, réel, voire pointeur). Il peut utiliser à ce moment `sscanf()` ou d'autres fonctions de conversion que nous avons déjà vues, comme `strtol()`, `strtod()` ou `strtoul()`.

Les commandes de redirection des entrées-sorties standards sont présentées, par exemple, dans [Blaess 2012] *Scripts Shell Unix/Linux par la pratique*. La plupart des fonctions de la bibliothèque C Ansi et principalement `stdio` sont décrites dans [Kernighan 1994] *Le langage C*, qui reste une référence incontournable.

21

Flux de données

Nous avons déjà abordé la notion de flux de données lors de la présentation des opérations simplifiées d'entrée-sortie dans le chapitre précédent. À présent, nous allons étudier plus en détail la relation entre les flux de données et les fichiers qui leur sont associés.

Nous verrons successivement les fonctions utilisées pour ouvrir ou fermer des flux, ainsi que les routines permettant d'y écrire des données ou de s'y déplacer. Par la suite, nous examinerons la configuration des buffers associés à un flux, ainsi que les variables indiquant leur état.

Différences entre flux et descripteurs

Une certaine confusion existe parfois dans l'esprit des programmeurs débutants sous Unix en ce qui concerne les rôles respectifs des flux de données et des descripteurs de fichiers. Il s'agit pourtant de deux notions complémentaires mais distinctes.

Les *descripteurs de fichiers* sont des valeurs de type `int`, que le noyau associe à un fichier à la demande d'un processus. Ces entiers sont en réalité des indices dans des tables propres à chaque processus, que le noyau est le seul à pouvoir modifier. Les descripteurs fournis par le noyau peuvent bien entendu être associés à des fichiers réguliers, mais aussi à d'autres éléments du système, comme des répertoires, des périphériques accessibles par un fichier spécial, des moyens de communication comme les tubes (*pipe*) ou les files (*FIFO*) que nous étudierons ultérieurement, ou encore des sockets utilisées pour établir la communication dans la programmation en réseau.

Les *flux de données* sont des objets dont le type est opaque. C'est une structure aux champs de laquelle l'application n'a pas accès. On manipule uniquement des variables pointeurs de type `FILE *`. Un flux est associé, en interne, à un descripteur de fichier, mais tout est masqué au programmeur applicatif. Un flux dispose en plus du descripteur de fichier d'une mémoire tampon, ainsi que de membres permettant de mémoriser l'état du fichier, notamment les éventuelles erreurs survenues lors des dernières opérations. Les flux de données sont les équivalents des objets de type `file` de la bibliothèque standard Python.

Il faut comprendre que les descripteurs de fichiers appartiennent à l'interface du noyau. Les fonctions `open()`, `close()`, `read()`, `write()`, `fcntl()` par exemple sont des appels système qui dialoguent donc directement avec le noyau Linux. Les flux par contre sont une couche supérieure ajoutée aux descripteurs et qui n'appartient qu'à la bibliothèque C. Les fonctions `fopen()`, `fclose()`, `fread()` ou `fwrite()` ne sont implémentées que dans la bibliothèque C. Le noyau n'a aucune connaissance de la notion de flux.

Ceci explique d'ailleurs que la bibliothèque d'entrée-sortie du C Ansi standard ne comporte aucune indication concernant les descripteurs. Ceux-ci sont à l'origine spécifiques aux systèmes d'exploitation de type Unix et ne peuvent pas être pris en considération dans une normalisation générale. Pour assurer la portabilité d'un programme, on utilisera les flux, même si la plupart des systèmes d'exploitation courants implémentent les fonctions d'accès aux descripteurs de fichiers (pas nécessairement sous forme de primitives système d'ailleurs).

La plupart du temps, le programmeur se tournera vers les flux de données pour manipuler des fichiers, ceci pour plusieurs raisons.

- **Portabilité** : en effet, nous l'avons indiqué, les flux de données seront disponibles sur toutes les machines supportant le C standard. Ce n'est pas nécessairement vrai pour les descripteurs de fichiers.
- **Performance** : les flux utilisant des buffers pour regrouper les opérations de lecture et d'écriture, le surcoût dû à l'appel système sur le descripteur sous-jacent est plus rare qu'avec une gestion directe du descripteur.
- **Simplicité** : la large panoplie de fonctions d'entrée et de sortie disponibles pour les flux n'existe pas pour les descripteurs de fichiers. Ceux-ci ne permettent que des lectures ou écritures de blocs mémoire complets. Il n'existe pas l'équivalent par exemple de la fonction `fgets()`, qui permet de lire une ligne de texte depuis un flux.

Les fonctions d'entrée-sortie formatées, que nous avons déjà rencontrées, comme `fprintf()` ou `fscanf()`, fonctionnent directement sur des flux. Toutefois, on peut également les employer dans un programme traitant uniquement les descripteurs de fichiers, en utilisant une chaîne de caractères intermédiaire et en appelant `sscanf()` ou `sprintf()`.

Certaines fonctionnalités sont vraiment spécifiques aux descripteurs de fichiers et ne peuvent pas être appliquées directement sur les flux. C'est le cas de la fonction

`fcntl()`, qui permet de paramétrer des notions comme la lecture non bloquante, les fichiers conservés à travers un `exec()`, etc.

Il est toujours possible d'obtenir le numéro de descripteur associé à un flux, tout comme il est possible d'ouvrir un nouveau flux autour d'un descripteur donné. Le passage de l'une à l'autre des représentations des fichiers est donc possible, bien qu'à éviter pour prévenir les risques de confusion.

Nous étudierons donc en premier lieu les fonctions permettant de manipuler les flux, puisque, en général, nous les préférerons aux descripteurs.

Ouverture et fermeture d'un flux

Nous avons déjà vu un bon nombre de fonctions permettant d'échanger des données avec des flux, comme `fgetc()`, `fgets()`, `fprintf()`, etc. Lorsque nous les avons rencontrées, nous n'utilisions que les trois flux prédéfinis `stdin`, `stdout` et `stderr`, ouverts par le système avant l'exécution d'un processus.

Ouverture normale d'un flux

La fonction `fopen()`, déclarée dans `<stdio.h>`, permet d'ouvrir un nouveau flux à partir d'un fichier du disque :

```
| FILE * fopen (const char * nom, const char * mode);
```

Cette fonction ouvre un flux à partir du fichier dont le nom est mentionné en premier argument, avec les autorisations de lecture et/ou d'écriture décrites dans une chaîne de caractères passée en second argument. En Python, la fonction équivalente est :

```
| open(nom, mode, buffer)
```

Le premier argument est le nom du fichier, c'est le seul obligatoire. Le deuxième argument est le mode d'accès (comme en C), le troisième une valeur entière représentant le type de buffer de sortie. Nous reviendrons sur ce sujet plus loin mais généralement, on ne précise pas cet argument.

Le nom du fichier peut contenir un chemin d'accès complet ou relatif. Un chemin commençant par le caractère `'/'` est pris en compte à partir de la racine du système de fichiers. Sinon, le chemin d'accès commence à partir du répertoire en cours. Il faut noter que le caractère `'~'` représentant le répertoire personnel d'un utilisateur, est un métacaractère du shell qui n'a aucune signification pour `fopen()`. Si on désire situer

un fichier à partir du répertoire personnel de l'utilisateur, il faut interroger la variable d'environnement HOME.

Notons également tout de suite qu'il n'est pas possible de retrouver le nom d'un fichier ouvert à partir du pointeur sur l'objet FILE (ni d'ailleurs à partir du descripteur sous-jacent). Si on désire garder une trace de ce nom, il faut le mémoriser au moment de l'ouverture.

Le mode indiqué en second argument permet de préciser le type d'accès désiré. Le mode peut prendre l'une des valeurs suivantes.

Mode	Type d'accès
r	Lecture seule, le fichier doit exister bien entendu.
w	Écriture seule. Si le fichier existe déjà, sa taille est ramenée à zéro, sinon il est créé.
a	Écriture seule en fin de fichier. Si le fichier existe, son contenu n'est pas modifié. Sinon, il est créé.
r+	Lecture et écriture. Le contenu précédent du fichier n'est pas modifié, mais les lectures et écritures démarreront au début, écrasant les données déjà présentes.
w+	Lecture et écriture. Si le fichier existe, sa taille est ramenée à zéro, sinon il est créé.
a+	Ajout et lecture. Le contenu initial du fichier n'est pas modifié. Les lectures commenceront au début du fichier, mais les écritures se feront toujours en fin de fichier.

Sur certains systèmes non Unix, on peut rencontrer des lettres supplémentaires comme 'b' pour indiquer que le flux contient des données binaires, et non du texte. Cette précision n'est d'aucune utilité sous Linux et n'a pas d'influence sur l'ouverture du flux. Il existe également sur de nombreux systèmes des restrictions de fonctionnement pour les flux ouverts en lecture et écriture. Ces limitations n'ont pas cours sous Linux, mais nous les détaillerons, par souci de portabilité des applications, dans la section consacrée au positionnement au sein d'un flux.

Il existe une extension Gnu de `fopen()` qui permet d'ajouter un caractère 'x' à la fin du mode pour indiquer qu'on veut absolument créer un nouveau fichier. L'ouverture échouera si le fichier existe déjà. Cette fonctionnalité n'est pas portable, mais elle peut parfois être indispensable lorsque deux processus concurrents risquent de créer simultanément le même fichier (un verrou par exemple). Le principe consistant à tenter une ouverture en lecture seule pour vérifier si le fichier existe, suivie d'une réouverture en écriture seule s'il n'existe pas ne fonctionne pas. En effet, ces deux opérations doivent être faites de manière atomique, en un seul appel système, sous peine de voir le noyau interrompre le processus entre les deux opérations pour autoriser l'exécution d'un processus concurrent qui créera également le même fichier. Nous verrons qu'il y a un moyen d'ouvrir un descripteur de fichier en écriture, uniquement si le fichier n'existe pas. On pourra alors utiliser cette méthode pour ouvrir un flux autour du descripteur obtenu, afin d'implémenter l'équivalent de l'extension Gnu 'x' de `fopen()`.

La fonction `fopen()` renvoie un pointeur sur un flux, qu'on pourra ensuite utiliser dans toutes les fonctions d'entrée-sortie. En cas d'échec, `fopen()` renvoie `NULL`, et la variable globale `errno` contient le type d'erreur, qu'on peut afficher avec `perror()`. Voici quelques exemples d'ouverture de fichiers.

```
exemple-fopen.c :
#include <stdio.h>
#include <stdlib.h>

void ouverture (char * nom, char * mode)
{
    FILE * fp;
    fprintf(stderr, "fopen(%s, %s) : ", nom, mode);
    if ((fp = fopen(nom, mode)) == NULL) {
        perror("");
    } else {
        fprintf(stderr, "Ok\n");
        fclose(fp);
    }
}

int main (void)
{
    ouverture("/etc/inittab", "r");
    ouverture("/etc/inittab", "w");
    ouverture("essai.fopen", "r");
    ouverture("essai.fopen", "w");
    ouverture("essai.fopen", "r");
    return EXIT_SUCCESS;
}
```

La lecture du fichier `/etc/inittab` est autorisée pour tous les utilisateurs sur les distributions Linux classiques, par contre l'écriture est réservée à `root`.

Attention !

Il ne faut surtout pas exécuter cet exemple en étant connecté avec les droits `root`, au risque de vider le fichier `/etc/inittab`.

Le fichier `essai.fopen` n'existe pas avant l'exécution du programme. Il est créé lors de l'ouverture en mode « `w` », ce qui explique que la seconde tentative d'ouverture en lecture réussisse.

```
$ ./exemple-fopen
fopen(/etc/inittab, r) : Ok
fopen(/etc/inittab, w) : Permission denied
```

```
fopen(essai.fopen, r) : No such file or directory
fopen(essai.fopen, w) : Ok
fopen(essai.fopen, r) : Ok
$ rm essai.fopen
$
```

On obtient le même résultat avec le script Python suivant :

```
exemple-open.py :
#!/usr/bin/python
from __future__ import print_function

def open_file(name, mode):
    print("open({}, {}) :".format(name, mode), end='')
    try:
        f = open(name, mode)
        f.close()
        print("Ok")
    except IOError:
        print("Error")

open_file("/etc/inittab", "r")
open_file("/etc/inittab", "w")
open_file("essai.fopen", "r")
open_file("essai.fopen", "w")
open_file("essai.fopen", "r")
```

On notera que le nombre de fichiers simultanément ouverts par un processus est limité. Cette restriction est décrite par une constante symbolique `OPEN_MAX`. Celle-ci inclut les trois flux prédéfinis : `stdin`, `stdout` et `stderr`. Sous Linux, avec la *Glibc*, elle vaut 1024.

Fermeture d'un flux

La fermeture d'un flux s'effectue à l'aide de la fonction `fclose()`, dont le prototype est :

```
| int fclose (FILE * flux);
```

En Python, on utilise la méthode `close()` de l'objet `File` obtenu avec `open()`. Une fois que le flux est fermé, une tentative d'écriture ou de lecture ultérieure échouera. Le buffer alloué par la bibliothèque C lors de l'ouverture est libéré. Par contre, le buffer qu'on peut avoir explicitement installé avec la fonction `setbuf()` ou ses déri-

vées, que nous verrons plus bas, n'est pas libéré. La fonction `fclose()` renvoie 0 si elle réussit ou EOF si une erreur s'est produite.

Il est important de vérifier la valeur de retour de `fclose()`, au même titre que toutes les écritures dans un fichier. En effet, avec le principe des écritures différées, le buffer associé à un flux n'est réellement écrit dans le fichier qu'au moment de sa fermeture. Une erreur peut alors se produire si le disque est plein ou si une connexion réseau est perdue (système de fichiers *NFS* par exemple). Une autre erreur peut se produire avec certains types de systèmes de fichiers, comme `ext3`, si un problème d'entrée-sortie apparaît sur une partition, qui est alors remontée automatiquement en lecture seule. Notre flux initialement ouvert en écriture renvoie une erreur « disque plein » au moment de la fermeture.

Si la fonction `fclose()` signale une erreur, le flux n'est plus accessible, mais il est toujours possible de prévenir l'utilisateur qu'un problème a eu lieu et qu'il peut réitérer la sauvegarde après avoir arrangé la situation. On peut analyser la variable globale `errno` pour diagnostiquer le problème, les erreurs possibles étant les mêmes que pour l'appel système `write()` que nous verrons plus loin.

On peut également fermer tous les flux ouverts par un processus avec la fonction `fcloseall()` qui est une extension Gnu déclarée dans `<stdio.h>` :

```
| int fcloseall (void);
```

Normalement, tous les flux sont fermés à la fin d'un processus, mais dans certains cas – arrêt abrupt à cause d'un signal par exemple – les buffers de sortie peuvent ne pas être écrits effectivement. Il est alors possible d'appeler `fcloseall()` dans le gestionnaire de signal concerné avant d'invoquer `abort()`.

Présentation des buffers associés aux flux

Il est temps d'étudier plus précisément les buffers associés aux flux, car ils sont souvent source de confusions.

Il existe, lors d'une écriture dans un flux, trois niveaux de buffers susceptibles de différer l'écriture. Tout d'abord, le flux est lui-même l'association d'une zone tampon et d'un descripteur de fichier. Il est possible de paramétrer le comportement de ce buffer au moyen de plusieurs fonctions que nous verrons un peu plus loin. On peut aussi forcer l'écriture du contenu d'un buffer en utilisant la fonction `fflush()`, déclarée ainsi :

```
| int fflush (FILE * flux);
```

Avec cette fonction, la bibliothèque C demande au noyau d'écrire le contenu du buffer associé au flux indiqué. Elle renvoie 0 si elle réussit et EOF en cas d'erreur. Les erreurs sont celles qui peuvent se produire en invoquant l'appel système `write()`.

La fonction `fflush()` n'a d'effet que sur les flux utilisés en écriture. Il est totalement illusoire de tenter d'invoquer `fflush(stdin)`, par exemple. Cet appel, malheureusement fréquemment utilisé par les débutants, n'a aucun effet et peut même déclencher une erreur sur certains systèmes (pas avec la Glibc toutefois).

En Python, la méthode `flush()` de l'objet de type `File` joue le même rôle.

Lorsqu'on appelle `fflush()`, la bibliothèque C invoque alors l'appel système `write()` sur les données qui n'étaient pas encore transmises. Ceci se produit également lorsqu'on ferme le flux ou lorsque le buffer est plein (ou encore en fin de ligne dans certains cas). Nous reviendrons sur ces détails.

Lorsqu'un processus se termine, nous sommes donc assurés que le noyau a reçu toutes les données que nous désirions écrire dans le fichier. Les fonctions `fflush()` et `fclose()` éliminent donc tout risque d'ambiguïté si deux processus tentent d'accéder simultanément au même fichier, puisque le noyau s'interpose entre eux pour assurer la cohérence des données écrites d'un côté et lues de l'autre.

Toutefois, un deuxième niveau de buffer intervient à ce moment. Le noyau en effet implémente un mécanisme de mémoire cache pour limiter les accès aux disques. Ce mécanisme varie en fonction des systèmes de fichier utilisés (et des attributs des descripteurs de fichiers). En règle générale, le noyau diffère les écritures le plus longtemps possible. Ceci permet qu'une éventuelle modification ultérieure du même bloc de données n'ait lieu que dans la mémoire centrale, en évitant toute la surcharge due à une séquence lecture-modification-écriture sur le disque.

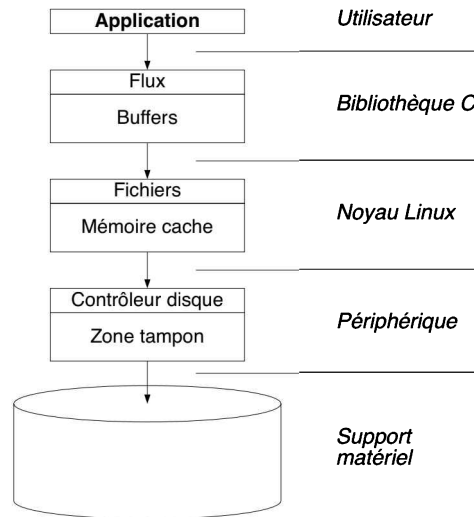
Pour s'assurer que les données sont réellement envoyées sur le disque, le noyau offre un appel système `sync()`.

```
| int sync (void);
```

En Python, `os.sync()` est disponible depuis la version 3.3 du langage. Cet appel système transmet au contrôleur du disque les blocs de données modifiés depuis leur dernière écriture réelle. Sur d'autres systèmes Unix, l'appel système `sync()` garantit uniquement que le noyau va commencer à mettre à niveau ses buffers, mais revient tout de suite. Sous Linux, l'appel reste bloquant tant que tous les blocs n'ont pas été transmis sur le disque. Il existe un utilitaire `/bin/sync` qui sert uniquement à invoquer l'appel système `sync()`. Sur les anciens Unix, on utilisait classiquement la séquence « `sync ; sync ; sync` » dans les scripts d'arrêt de la machine pour être à peu près sûr que tous les blocs en attente soient écrits sur le disque. Sous Linux, un seul appel à `/bin/sync` suffit.

Le noyau nous assure donc que lorsqu'un système de fichiers est démonté ou lorsque `sync()` revient, tous les blocs en attente auront été transmis au disque. Malheureusement, certains contrôleurs de disques (*SCSI*, *SATA*...) disposent de buffers internes très grands (des centaines de Mo), et rien ne garantit que les données soient immédiatement écrites physiquement. Ce point doit être pris en considération lors de la conception de systèmes informatiques basés sur des machines Linux avec des données critiques (gestion répartie, supervision de systèmes industriels ou scientifiques). On pourra alors utiliser une alimentation secourue ou des ordinateurs portables sur batterie pour garantir un certain laps de temps pour la sauvegarde physique des données en cas de défaillance secteur.

Figure 21-1
Mécanismes successifs
des buffers en écriture



La fonction `fflush()` que nous avons vue plus haut peut également être employée avec un argument `NULL`. Dans ce cas, elle vide les buffers de tous les flux en attente d'écriture.

Ouvertures particulières de flux

La fonction `fopen()` dispose de deux variantes permettant d'ouvrir un flux de deux manières légèrement différentes. La première fonction est `fdopen()`, ou `os.fdopen()` en Python, dont le prototype est :

```
| FILE * fdopen (int descripteur, const char * mode);
```

Cette fonction permet de disposer d'un flux construit autour d'un descripteur de fichier déjà obtenu auparavant. Ce descripteur doit avoir été fourni précédemment par l'un des appels système suivants :

- `open()`, `creat()`, ouvrant un fichier disque ;
- `pipe()`, qui crée un tube de communication entre processus ;
- `socket()`, permettant d'établir une liaison réseau ;
- `dup()`, `dup2()`, qui servent à dupliquer un descripteur existant.

Nous détaillerons ces fonctions dans les chapitres à venir. Ce qu'il faut retenir pour le moment c'est la possibilité de créer un flux à partir de toutes les sources de communication offertes par le noyau Linux.

Le mode indiqué en second argument doit être compatible avec les possibilités offertes par le descripteur existant. Plus particulièrement, si le mode réclamé nécessite des accès en écriture, ceux-ci doivent être possibles sur le descripteur fourni. Il faut remarquer également que les modes `w` ou `w+` ne permettent pas dans ce cas de ramener à zéro la taille du fichier associé, car celui-ci est déjà ouvert par le descripteur. Rappelons aussi que ce descripteur n'est pas nécessairement associé à un fichier, et que la modification de taille d'une socket réseau par exemple n'aurait pas de sens.

En cas d'échec, `fdopen()` renvoie `NULL`, sinon elle transmet un pointeur sur le flux désiré.

Cette fonction est donc principalement utile pour accéder sous forme de flux à des sources de données qu'on ne peut pas obtenir par un `fopen()` classique, comme les tubes ou les sockets. Nous en verrons plusieurs exemples dans le chapitre 23.

La seconde fonction dérivée de `fopen()` est `freopen()`, dont le prototype est le suivant :

```
FILE * freopen (const char * fichier, const char * mode,  
                FILE * flux);
```

Cette fonction commence par fermer le flux indiqué en dernier argument, en ignorant toute erreur susceptible de se produire. Ensuite, elle ouvre le fichier demandé, avec le mode précisé en second argument, en utilisant le même flux que le précédent. Un pointeur sur ce dernier est renvoyé, ou `NULL` en cas d'erreur.

Étant donné que `freopen()` ne vérifie pas les erreurs susceptibles de se produire en fermant le flux original, il est indispensable d'appeler `fflush()` – et de surveiller sa valeur de retour – si le flux original a servi au préalable à écrire des données.

L'intérêt principal de cette fonction est de pouvoir rediriger les flux standards `stdin`, `stdout` et `stderr` depuis ou vers des fichiers, au sein même du programme. Tous les affichages sur `stderr` par exemple pourront ainsi être envoyés vers un fichier de débogage, sans redirection au niveau du shell. Il est aussi possible de rediriger `stderr` vers `/dev/null` pour supprimer tous les messages de diagnostic par exemple, bien que d'autres méthodes comme `syslog()` soient largement préférables. Voici un exemple de programme où on redirige la sortie standard du processus.

```
exemple-freopen.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    printf("Cette ligne est envoyee sur la sortie normale\n");
    if (freopen("essai.freopen", "w", stdout) == NULL) {
        perror("freopen");
        exit(EXIT_FAILURE);
    }
    printf("Cette ligne doit se trouver dans le fichier\n");
    return EXIT_SUCCESS;
}
```

La première écriture sur `stdout` se trouve normalement affichée sur la sortie standard, la seconde est redirigée vers le fichier désiré.

```
$ ./exemple-freopen
Cette ligne est envoyee sur la sortie normale
$ ls -l essai.freopen
-rw-rw-r-- 1 cpb cpb 44 4 août 10:35 essai.freopen
$ cat essai.freopen
Cette ligne doit se trouver dans le fichier
$ rm essai.freopen
$
```

On notera que, les descripteurs de fichiers correspondant aux flux hérités au cours d'un appel `fork()`, la redirection est toujours valable pour le processus fils.

Avec la bibliothèque Glibc, les flux `stdin`, `stdout` et `stderr` sont des variables globales. Il serait donc tout à fait possible d'écrire :

```
fclose(stdout);
if ((stdout = fopen("essai.freopen", "w")) == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

Toutefois ce ne serait pas portable car de nombreuses implémentations de la bibliothèque C standard définissent `stdin`, `stdout` et `stderr` sous forme de macros. Il est donc indispensable d'utiliser `freopen()` dans ce cas.

Lectures et écritures dans un flux

Nous avons vu comment ouvrir, refermer les flux, et vider les buffers associés. Il est maintenant nécessaire d'étudier les fonctions servant à écrire effectivement ou à lire des données.

L'essentiel des fonctions d'entrée-sortie sur un flux a déjà été étudié dans le chapitre 20. Nous y avons vu successivement `fprintf()`, `vfprintf()`, `fputc()`, `fputs()` pour les écritures, `fgetc()`, `fgets()`, `fscanf()` et `vfscanf()` pour les lectures, ainsi que `fungetc()` pour rejeter un caractère dans un flux, et les fonctions équivalentes pour caractères larges.

Nous allons ici nous intéresser aux fonctions dites d'entrée-sortie binaires. Celles-ci permettent de lire ou d'écrire le contenu intégral d'un bloc mémoire, sans se soucier de son interprétation. La fonction d'écriture est `fwrite()`, dont le prototype est le suivant :

```
int fwrite (const void * bloc,
            size_t taille_elements,
            size_t nb_elements,
            FILE *flux);
```

Elle permet d'écrire dans le flux indiqué un certain nombre d'éléments consécutifs, dont on indique la taille et l'adresse de départ. Pour sauvegarder le contenu d'une table d'entiers par exemple, on pourra utiliser :

```
int table[NB_ENTIERS];
[...]
fwrite(table, sizeof(int), NB_ENTIERS, fichier);
```

Cette fonction renvoie le nombre d'éléments correctement écrits. Si cette valeur diffère de celle qui est transmise en troisième argument lors de l'appel, une erreur s'est produite, qui doit être diagnostiquée à l'aide de la variable globale `errno`. Généralement, une telle erreur sera critique et correspondra à un problème de disque saturé ou de liaison perdue avec un système de fichiers *NFS* distant. Toutefois, il peut arriver que l'erreur soit bénigne, si le flux a été ouvert par la fonction `fdopen()` autour d'une socket de connexion réseau ou d'un tube de communication. Dans ces deux cas en effet, les écritures peuvent être bloquantes tant que le récepteur n'est pas disponible, et un signal peut interrompre l'appel système `write()` sous-jacent. À cette occasion, il sera possible de réitérer l'appel de la fonction, avec les éléments non écrits correctement.

La fonction symétrique `fread()` permet de lire le contenu d'un flux et de l'inscrire dans un bloc de mémoire. Son prototype est :

```
int fread (void * bloc,
           size_t taille_elements,
           size_t nb_elements,
           FILE * flux);
```

Les arguments de `fread()` sont identiques à ceux de `fwrite()`. À ce propos, on notera que ces prototypes sont une source fréquente d'erreurs à cause de la position du pointeur `FILE *` en dernier argument, contrairement aux fonctions `fprintf()` et `fscanf()` qui le placent en premier. Le problème est que l'inversion entre le pointeur sur le flux et celui sur le bloc peut être ignorée par le compilateur si certains avertissements sont désactivés. De toute manière, il est difficile de se souvenir sans erreur des positions respectives de la taille des éléments et de leur nombre. Aussi, on s'imposera comme règle avant chaque utilisation de `fread()` ou de `fwrite()` de jeter un coup d'œil rapide sur leur page de manuel commune `fread(3)`.

Comme pour `fwrite()`, la valeur de retour de `fread()` correspond au nombre d'éléments correctement lus. Toutefois, à l'inverse de `fwrite()`, le nombre effectivement lu peut être inférieur à celui qui est réclamé, sans qu'une erreur critique ne se soit produite, si on atteint la fin du fichier par exemple.

Ces fonctions sont très utiles pour sauvegarder des tables de données, des structures, à condition qu'on ne les réutilise que sur la même machine. Les données écrites sont en effet une reproduction directe de la représentation des informations en mémoire. Cette représentation peut varier non seulement entre deux systèmes différents, par exemple en fonction de l'ordre des octets pour stocker des entiers, mais aussi sur la même machine en fonction des options utilisées par le compilateur. En voici un exemple :

```
exemple-enum.c :
#include <stdio.h>
#include <stdlib.h>

typedef enum {
    un, deux, trois
} enum_t;

int main (void)
{
    fprintf(stdout, "sizeof(enum_t) = %d\n", sizeof(enum_t));
    return EXIT_SUCCESS;
}
```

En fonction des options de compilation de gcc, la taille des données de type énuméré varie.

```
$ cc exemple-enum.c -o exemple-enum -Wall
$ ./exemple-enum
sizeof(enum_t) = 4
$ cc exemple-enum.c -o exemple-enum -Wall -fshort-enums
$ ./exemple-enum
sizeof(enum_t) = 1
$
```

Il faut donc être très prudent avec l'emploi des fonctions `fread()` et `fwrite()`, et ne les considérer que comme des moyens de sauvegarder et de récupérer des données sur une seule et même machine, sans pérennité dans le temps.

Il est conseillé dans toute application importante de prévoir des fonctionnalités d'exportation et d'importation des données moins rapides que les accès binaires directs, employant des fichiers plus volumineux mais transférables entre plusieurs systèmes hôtes ou entre diverses versions de la même application. Pour cela le plus simple est d'employer une représentation textuelle, en utilisant les fonctions `fprintf()` et `fscanf()` pour écrire et relire les données. On peut également trouver des bibliothèques libres qui permettent de sauver et restaurer des structures complexes, au format XML par exemple.

Nous allons quand même présenter un exemple d'utilisation de `fread()` et de `fwrite()`, sauvegardant le contenu d'une table de structures représentant des points dans l'espace. La table est initialisée avec les points situés aux sommets d'un cube centré sur l'origine. Nous sauvegardons la table, la rechargeons, et affichons les coordonnées pour vérifier le fonctionnement.

```
exemple-fwrite.c :
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    double x;
    double y;
    double z;
} point_t;

int main (void)
{
    point_t * table;
    int      n;
    int      i;
    FILE *   fp;

    n = 8;
    table = calloc(n, sizeof (point_t));
    if (table == NULL) {
        perror("calloc");
        exit(EXIT_FAILURE);
    }

    // Initialisation
    table[0].x = -1.0; table[0].y = -1.0; table[0].z = -1.0;
    table[1].x = 1.0;  table[1].y = -1.0; table[1].z = -1.0;
    table[2].x = -1.0; table[2].y = 1.0;  table[2].z = -1.0;
```



```
table[3].x = 1.0; table[3].y = 1.0; table[3].z = -1.0;
table[4].x = -1.0; table[4].y = -1.0; table[4].z = 1.0;
table[5].x = 1.0; table[5].y = -1.0; table[5].z = 1.0;
table[6].x = -1.0; table[6].y = 1.0; table[6].z = 1.0;
table[7].x = 1.0; table[7].y = 1.0; table[7].z = 1.0;

// Sauvegarde
if ((fp = fopen("essai.fread", "w")) == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
// Écriture du nombre de points, suivi de la table
if ((fwrite(&n, sizeof(int), 1, fp) != 1)
    || (fwrite(table, sizeof(point_t), 8, fp) != 8)) {
    perror("fwrite");
    exit(EXIT_FAILURE);
}
fclose(fp);
free(table);
table = NULL;
n = 0;

// Récupération
if ((fp = fopen("essai.fread", "r")) == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
if (fread(&n, sizeof(int), 1, fp) != 1) {
    perror("fread");
    exit(EXIT_FAILURE);
}
if ((table = calloc(n, sizeof(point_t))) == NULL) {
    perror("calloc");
    exit(EXIT_FAILURE);
}
if (fread(table, sizeof(point_t), n, fp) != 8) {
    perror("fread");
    exit(EXIT_FAILURE);
}
fclose(fp);

// Affichage
for (i = 0; i < n; i++)
    fprintf(stdout, "point[%d] : % f, % f, % f\n",
            i, table[i].x, table[i].y, table[i].z);

return EXIT_SUCCESS;
}
```

Comme on peut s'y attendre, l'exécution donne le résultat suivant.

```
$ ./exemple-fwrite
point[0] : -1.000000, -1.000000, -1.000000
point[1] : 1.000000, -1.000000, -1.000000
point[2] : -1.000000, 1.000000, -1.000000
point[3] : 1.000000, 1.000000, -1.000000
point[4] : -1.000000, -1.000000, 1.000000
point[5] : 1.000000, -1.000000, 1.000000
point[6] : -1.000000, 1.000000, 1.000000
point[7] : 1.000000, 1.000000, 1.000000
$
```

On remarquera au passage l'emploi d'un caractère d'espacement entre le '%' et le 'f' du format de `fprintf()`, ce qui permet de conserver l'alignement des données en affichant un espace avant les valeurs positives.

Il existe deux anciennes fonctions, `getw()` et `putw()`, qu'on peut considérer comme obsolètes, permettant de lire ou d'écrire un entier. Leurs prototypes sont :

```
| int getw (FILE * flux);
```

et :

```
| int putw (int entier, FILE * flux);
```

Ces fonctions renvoient EOF en cas d'échec. Sinon, elles transmettent respectivement la valeur lue et 0. Le gros défaut avec `getw()` est qu'il est impossible de distinguer une erreur de la lecture effective de la valeur EOF (qui vaut généralement -1). Il est donc préférable de remplacer ces deux fonctions par `fread()` et `fwrite()`.

Positionnement dans un flux

Il est rare dans une application un tant soit peu complexe qu'on ait uniquement besoin de lire les données d'un fichier séquentiellement, du début à la fin, sans jamais revenir en arrière ou sauter des portions d'informations. Il est donc naturel que la bibliothèque C mette à notre disposition des fonctions permettant de se déplacer librement dans un fichier avant de lire son contenu ou d'y écrire des données.

Ceci est géré en fait directement au niveau du descripteur de fichier, par le noyau, en mémorisant la position à laquelle se fera le prochain accès dans le fichier. Cette position est mise à jour après chaque lecture ou écriture.

Il existe trois types de fonctions pour consulter ou indiquer la position dans le fichier : le couple `ftell()` / `fseek()`, qui oblige l'indicateur de position à être de type `long int`, le couple `ftello()` / `fseeko()`, qui fonctionne de manière similaire mais sans cette restriction de type, et enfin `fgetpos()` / `fsetpos()`, qui sont encore plus portables.

Si la plupart des flux obtenus par `fopen()` depuis un fichier disque ne posent aucun problème de positionnement, ce n'est toutefois pas le cas de tous les flux possibles. Un certain nombre de sources de données sont fondamentalement séquentielles, et il n'est pas possible de se déplacer en leur sein. Tel est par exemple le cas d'un tube de communication entre processus. On ne peut y avancer qu'en lisant les données, et on ne peut en aucun cas reculer la position de lecture. Le même phénomène se produit avec les sockets de liaison réseau ou les fichiers spéciaux d'accès aux périphériques. Avec de tels flux, toute tentative de consultation ou de modification de la position courante échouera.

Positionnement classique

Les fonctions les plus simples d'accès aux positions sont certainement `ftell()` et `fseek()`, dont les prototypes sont :

```
| long ftell (FILE * flux);
```

et :

```
| int fseek (FILE * flux, long position, int depart);
```

En Python, les objets de type `File` ont des méthodes `fseek()` et `ftell()` correspondantes.

La fonction `ftell()` transmet la position courante dans le flux, mesurée en octets depuis le début du fichier. Si le positionnement n'est pas possible sur ce type de flux, `ftell()` renvoie `-1`.

La fonction `fseek()` permet de se déplacer dans le fichier. La position est indiquée en octets, depuis le point de départ fourni en troisième argument. Celui-ci peut prendre les valeurs suivantes.

Constante	Valeur	Action
<code>SEEK_SET</code>	0	On mesure la position depuis le début du fichier.
<code>SEEK_CUR</code>	1	Le déplacement est indiqué à partir de la position courante dans le fichier.
<code>SEEK_END</code>	2	La position est mesurée par rapport à la fin du fichier.

Nous avons exceptionnellement indiqué les valeurs des constantes symboliques définies dans `<stdio.h>`. En effet, ces constantes ne sont apparues que relativement tard, et de nombreuses applications Unix contiennent ces valeurs codées en dur dans leurs fichiers source. On peut également rencontrer les constantes obsolètes `L_SET`, `L_INCR`, `L_XTND`, qui sont des équivalentes BSD de `SEEK_SET`, `SEEK_CUR` et `SEEK_END`, définies par souci de compatibilité dans `<sys/file.h>`.

La fonction `fseek()` renvoie 0 si elle réussit, et -1 en cas d'échec.

La fonction `rewind()` permet de ramener la position courante au début du flux. Son prototype est :

```
| void rewind (FILE * fp);
```

On pourrait la définir en utilisant `fseek(fichier, 0, SEEK_SET)`.

Lorsque `fseek()` ou `rewind()` sont invoquées, le contenu éventuel du buffer de sortie associé au flux est écrit dans le fichier avant le déplacement. Il existait d'ailleurs sur de nombreux systèmes Unix une restriction à l'utilisation d'un flux en lecture et écriture. Sur de tels systèmes, une lecture ne pouvait suivre une opération d'écriture que si on avait invoqué `fflush()`, `fseek()`, `fseeko()`, `fsetpos()` ou `rewind()` entre les deux opérations. De même, avant une écriture suivant une lecture, il fallait obligatoirement invoquer `fseek()`, `fseeko()`, `fsetpos()` ou `rewind()`. Même lorsque l'écriture devait avoir lieu exactement à la position courante résultant de la dernière lecture, il fallait employer `fseek(fichier, 0, SEEK_CUR)`. Ces limitations n'ont pas cours sous Linux et ne sont probablement plus nécessaires sur les Unix actuels.

Si le fichier contient des caractères larges, il ne faut jamais essayer de se positionner directement à un emplacement calculé, les seules valeurs acceptées en troisième argument de `fseek()` sont celles qui ont été renvoyées par un précédent appel à `ftell()` ou zéro.

Notre premier exemple va employer `ftell()` pour repérer l'emplacement des caractères de retour à la ligne `'\n'`, et afficher les longueurs successives des lignes. Nous analyserons le contenu de l'entrée standard afin de voir directement certains cas d'échec.

```
| exemple-ftell.c :  
| #include <stdio.h>  
| #include <stdlib.h>  
  
| int main (void)  
| {  
|     long derniere;  
|     long position;  
|     int caractere;  
  
|     position = ftell(stdin);
```

```
if (position == -1) {
    perror("ftell");
    exit(EXIT_FAILURE);
}
derniere = position;
while ((caractere = getchar()) != EOF) {
    if (caractere == '\n') {
        position = ftell(stdin);
        if (position == -1) {
            perror("ftell");
            exit(EXIT_FAILURE);
        }
        fprintf(stdout, "%ld ", position - derniere - 1);
        derniere = position;
    }
}
fprintf(stdout, "\n");
return EXIT_SUCCESS;
}
```

Nous allons essayer de l'exécuter successivement à partir d'un fichier, depuis un tube créé par le pipe '|' du shell et depuis un fichier spécial de périphérique.

```
$ ./exemple-ftell < exemple-ftell.c
18 19 0 15 1 15 15 16 1 25 22 18 21 2 21 41 26 27 24 20 23 4 52 23 3
2 23 21 1
$ cat exemple-ftell.c | ./exemple-ftell
ftell: Illegal seek
$ ./exemple-ftell < /dev/tty
ftell: Illegal seek
$
```

Nous voyons que les flux obtenus à partir d'un tube ou d'un fichier spécial de périphérique ne permettent pas le positionnement.

Positionnement compatible Unix 98

Le défaut des fonctions `fseek()` et `ftell()` est de restreindre la taille d'un fichier à celle d'un long. Sous Linux 32 bits, un long `int` est implémenté à l'aide 4 octets. Ce qui signifie que la taille d'un fichier est limitée à $2^{31} - 1$ octets (un bit est réservé pour le signe) correspondant à 2 Go. Cette dimension est assez grande, mais des fichiers de 2 Go apparaissent de plus en plus pour stocker des vidéos, des bases de données, etc.

Pour passer outre la limitation de `fseek()` et de `ftell()`, les spécifications *Unix 98* ont introduit deux nouvelles fonctions, `fseeko()` et `ftello()`, utilisant un type de

données spécifique, `off_t`. Ce type est d'ores-et-déjà étendu à 8 octets sur les processeurs 64 bits. Sur les processeurs 32 bits avec la Glibc, `off_t` mesure encore 4 octets, mais il peut être étendu en utilisant un mécanisme nommé LFS (*Large File Support*).

```
taille-off-t.c :  
// #define _FILE_OFFSET_BITS 64  
#include <stdio.h>  
#include <stdlib.h>  
  
int main (void)  
{  
    printf("sizeof(off_t) = %d\n", sizeof(off_t));  
    return EXIT_SUCCESS;  
}
```

Sur un processeur 32 bits, sans définir la constante `_FILE_OFFSET_BITS` (en commentaire), on obtient :

```
$ ./taille-off-t  
sizeof(off_t) = 4  
$
```

Si la constante est définie avec la valeur 64, on obtient :

```
$ ./taille-off-t  
sizeof(off_t) = 8  
$
```

Sur un processeur 64 bits, même sans cette constante, on obtient :

```
$ ./taille-off-t  
sizeof(off_t) = 8  
$
```

Les prototypes de ces nouvelles fonctions sont :

```
off_t ftello (FILE * flux);  
int fseeko (FILE * flux, off_t position, int depart);
```

Leur fonctionnement est exactement le même que `ftell()` et `fseek()`, au type `off_t` près.

Nous allons créer un programme de démonstration qui servira à retourner intégralement le contenu d'un fichier. Celui-ci va uniquement utiliser des primitives `fseeko()`, `ftello()`, `fgetc()` et `fputc()`. Rappelons que nous en avons déjà construit une version bien plus efficace à l'aide de `mmap()` dans le chapitre consacré à la gestion de l'espace mémoire d'un processus.

```
exemple-fseeko.c :
// #define _FILE_OFFSET_BITS 64
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char * argv[])
{
    int    i;
    FILE * fp;
    int    caractere;
    int    echange;

    off_t debut;
    off_t fin;

    if (argc < 2) {
        fprintf(stderr, "syntaxe : %s fichier...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    for (i = 1; i < argc ; i++) {
        if ((fp = fopen(argv[i], "r+")) == NULL) {
            fprintf(stderr, "%s inaccessible\n", argv[i]);
            continue;
        }
        if (fseeko(fp, 0, SEEK_END) != 0) {
            fprintf(stderr, "%s non positionnable\n", argv[i]);
            fclose(fp);
            continue;
        }
        fin = ftello(fp) - 1;
        debut = 0;
        while (fin > debut) {
            if (fseeko(fp, fin, SEEK_SET) != 0)
                break;
            caractere = fgetc(fp);
            if (fseeko(fp, debut, SEEK_SET) != 0)
                break;
            echange = fgetc(fp);
            if (fseeko(fp, debut, SEEK_SET) != 0)
                break;
            fputc(caractere, fp);
        }
    }
}
```

```

        if (fseeko(fp, fin, SEEK_SET) != 0)
            break;
        fputc(echange, fp);
        fin --;
        debut ++;
    }
    fclose(fp);
}
return EXIT_SUCCESS;
}

```

Nous utilisons deux pointeurs qui se rapprochent l'un de l'autre à chaque itération pour éviter d'avoir à s'interroger sur le point d'arrêt au milieu du fichier en fonction de la parité de la dimension du fichier. Nous exécutons le programme en lui demandant de retourner son propre fichier source (on reconnaît les mots-clés `return`, `fclose`, `int` etc., à l'envers).

```

$ ./exemple-fseeko exemple-fseeko.c
$ cat exemple-fseeko.c
}
;SSECCUS_TIXE nruter
}
;)pf(esolcf
}
[...]
;i    tni
{
)[vgra * rahc ,cgra tni( niam tni
[...]
$ ./exemple-fseeko exemple-fseeko.c
$ cat exemple-fseeko.c
[...]
int main (int argc, char * argv[])
{
    int    i;
    [...]
}
    fclose(fp);
}
    return EXIT_SUCCESS;
}
$

```


Problèmes de portabilité

Les fonctions que nous avons vues plus haut se comportent parfaitement bien sur un système Gnu /Linux et sur l'essentiel des systèmes Unix en général. Malgré tout, certains problèmes peuvent se poser, en particulier sur des architectures qui distinguent le stockage des données dans des fichiers binaires ou dans des fichiers de texte. Ces derniers sont parfois représentés, sur le disque, par des tables de pointeurs vers des chaînes de caractères. Le positionnement dans un tel fichier est donc repéré à la fois par le numéro de chaîne et par l'emplacement du caractère courant dans celle-ci.

Dans un tel cas, on ne peut plus considérer un fichier comme une succession linéaire de caractères ou d'octets, mais bien comme une entité dont la topologie peut s'étendre sur deux dimensions ou plus. L'utilisation du type `long int` avec `fseek()` et `ftell()`, ou du type `off_t` avec `fseeko()` et `ftello()`, n'est plus suffisante.

Pour assurer un maximum de portabilité à un programme, on se tournera vers les fonctions `fgetpos()` et `fsetpos()` :

```
| int fgetpos (FILE * flux, fpos_t * position);
```

et :

```
| int fsetpos (FILE * flux, fpos_t * position);
```

Elles permettent de lire la position courante ou de la déplacer, en utilisant comme stockage un pointeur sur un objet de type `fpos_t`. Ce dernier est un type opaque, susceptible d'évoluer suivant les systèmes, les versions de bibliothèque, ou même les options de compilation.

Il n'est donc pas possible de se livrer à des calculs arithmétiques sur les déplacements mesurés par ces fonctions. La portabilité d'un programme sera assurée si on ne transmet à `fsetpos()` que des pointeurs sur des valeurs ayant été obtenues précédemment avec `fgetpos()`. Ces deux fonctions renvoient zéro si elles réussissent, une valeur non nulle sinon, et remplissent alors la variable globale `errno`.

Elles présentent malgré tout un certain nombre d'inconvénients, comme l'impossibilité de sauter directement à la fin du fichier, et nécessitent en général de mémoriser un nombre important de positions (le début du fichier, de chaque section, sous-section, enregistrement...). C'est le prix à payer pour assurer une portabilité optimale, principalement en ce qui concerne des fichiers de texte.

Paramétrage des buffers associés à un flux

Nous avons signalé rapidement qu'un flux est une association d'un descripteur de fichier et d'un buffer de sortie, mais finalement nous n'avons pas étudié en détail ce mécanisme. Pourtant, la bibliothèque standard C offre plusieurs possibilités de paramétrage des buffers, en fonction des opérations qu'on désire effectuer sur le flux.

Type de buffers

Il existe trois types de buffers associés à un flux.

- **Buffer de bloc** : le flux dispose d'un tampon qui est rempli intégralement par les données avant qu'on invoque véritablement l'appel système `write()` pour faire l'écriture. Un gain de temps important est alors assuré puisqu'on réduit considérablement le nombre d'appels système à réaliser. Ce type de buffer est normalement utilisé pour tous les fichiers résidant sur le disque.
- **Buffer de ligne** : les données sont conservées dans le buffer jusqu'à ce que ce dernier soit plein, ou jusqu'à ce qu'on envoie un caractère de saut de ligne `'\n'`. Ce type de buffer est utilisé sur les flux qui sont connectés à un terminal (généralement `stdout`).
- **Pas de buffer** : toutes les données sont immédiatement transmises sans délai. L'appel système `write()` est invoqué à chaque écriture. Cette configuration est celle par défaut du flux `stderr`, ce qui permet aux messages écrits sur cette sortie interactive d'apparaître immédiatement à l'utilisateur.

Il est bien évident que le buffer de ligne ne présente d'intérêt que si le flux est utilisé pour transmettre du texte. Dans le cas de données binaires, le saut de ligne `'\n'` n'a pas plus de signification que tout autre caractère, et peut survenir à tout moment.

Il est toujours possible de forcer l'écriture immédiate du contenu du buffer en employant la fonction `fflush()` que nous avons vue plus haut. De même, lorsqu'on effectue une lecture sur un flux (par exemple `stdin`), tous les buffers de lignes des flux actuellement ouverts sont écrits. C'est important, par exemple pour que le message d'accueil suivant soit correctement affiché lors de la saisie, même sans retour à la ligne :

```
fprintf (stdout, "Veuillez entrer votre nom : ");  
fgets (chaine, LG_CHAINE, stdin);
```

Dans ce cas, le message est écrit dans le buffer associé à `stdout`, puis, lorsque la lecture est invoquée, ce buffer est effectivement affiché, ce qui permet d'avoir un curseur placé à la suite du message pour faire la saisie.

Voici un petit programme d'exemple destiné à montrer que les données écrites sur `stdout` sont affichées :

- à la détection d'un saut de ligne ;
- sur une demande explicite `fflush()` ;
- lors d'une tentative de lecture d'un flux d'entrée.

Les données de `stderr` sont, elles, affichées immédiatement.

```
exemple-buffers.c :
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    char chaine[10];

    fprintf(stdout, "1 stdout : ligne + \\n\\n");
    fprintf(stdout, "2 stdout : ligne seule");
    fprintf(stderr, "\\n3 stderr : avant fflush(stdout)\\n");
    fflush(stdout);
    fprintf(stderr, "\\n4 stderr : apres fflush (stdout)\\n");
    fprintf(stdout, "5 stdout : ligne seule ");
    fprintf(stderr, "\\n6 stderr : avant fgets(stdin)\\n");
    fgets(chaine, 10, stdin);
    fprintf(stderr, "\\n7 stderr : apres fgets(stdin)\\n");

    return EXIT_SUCCESS;
}
```

L'exécution donne le résultat suivant.

```
$ ./exemple-buffers
1 stdout : ligne + \n

3 stderr : avant fflush(stdout)
2 stdout : ligne seule
4 stderr : apres fflush (stdout)

6 stderr : avant fgets(stdin)
5 stdout : ligne seule
    (Entrée)
7 stderr : apres fgets(stdin)
$
```

Nous voyons bien que la ligne 1 est affichée immédiatement car elle se termine par un retour à la ligne. Mais la ligne 2 reste dans le buffer. La ligne 3 sur `stderr` apparaît tout de suite. Lorsqu'on invoque `fflush()`, la ligne 2 est effectivement affichée. La fonction `fflush()` ne revient que lorsque le buffer a réellement été vidé.

Lorsque la ligne 5 est écrite, elle reste dans le buffer. La ligne 6 est affichée immédiatement puisqu'elle arrive sur `stderr`. En demandant une lecture sur `stdin`, les buffers sont vidés et la ligne 5 est alors affichée. Nous appuyons sur la touche « Entrée » pour terminer la saisie.

Le comportement est exactement identique avec le script Python suivant, qui fait usage des arguments `file` (pour préciser le flux de sortie) et `end` (pour éliminer le retour-chariot) de la fonction `print()`.

```
exemple-buffers.py :
#!/usr/bin/python
from __future__ import print_function
import sys

print("1 stdout : ligne + \n")
print("2 stdout : ligne seule", end='')
print("\n3 stderr : avant flush()", file=sys.stderr)
sys.stdout.flush()
print("\n4 stderr : apres flush ()", file=sys.stderr)
print("5 stdout : ligne seule ", end='')
print("\n6 stderr : avant sys.stdin.read()", file=sys.stderr)
sys.stdin.readline()
print("\n7 stderr : apres sys.stdin.read(stdin)", file=sys.stderr)
```

Il ne faut pas confondre la notion de buffer de ligne, qui est interne aux flux de données, et le mode de contrôle du terminal. Lorsqu'on doit taper sur la touche « Entrée » pour valider une ligne de saisie, c'est le terminal qui gère cette ligne, et non le buffer de `stdin`. Si on veut pouvoir lire les caractères « au vol », sans attendre la touche « Entrée », il faut se pencher sur les modes de contrôle du terminal, comme nous le ferons au chapitre 32.

Modification du type et de la taille du buffer

Lorsqu'un flux utilise un buffer, la mémoire nécessaire pour celui-ci est allouée lors de la première tentative d'écriture. La taille du buffer est définie par la constante symbolique `BUFSIZ`, qu'on trouve dans `<stdio.h>`. Avec la Glibc, cette constante correspond à 8 Ko. Toutefois, si l'allocation échoue, la bibliothèque essaye d'obtenir un buffer de 4 Ko, puis de 2 Ko, et ainsi de suite jusqu'à la limite de 128 octets, où le mécanisme de la mémoire tampon n'a plus d'intérêt.

Lors de l'ouverture d'un flux, la bibliothèque prévoit un buffer de type bloc, sauf si le flux est connecté à un terminal, dans ce cas, le buffer est de type ligne. Le flux `stderr` représente une exception puisqu'il n'a jamais de buffer. En Python, le troisième argument de `open()` permet de préciser directement le type de buffer désiré.

Nous pouvons désirer, pour de multiples raisons, modifier le type ou la contenance du buffer associé à un flux. Ceci est possible avec plusieurs fonctions, qui ont grossièrement le même effet. La fonction la plus complète est `setvbuf()`, déclarée ainsi :

```
| int setvbuf (FILE * flux, char * buffer, int mode, size_t taille);
```

Le premier argument est le flux sur lequel on veut agir. Le second est un pointeur sur un buffer qu'on fournit. Si ce pointeur est `NULL`, la fonction allouera elle-même une zone tampon de la taille précisée en quatrième argument. Nous préciserons les précautions à prendre lors de l'emploi d'un buffer personnalisé.

Le troisième argument correspond au type de buffer désiré. Cette valeur peut être l'une des constantes symboliques suivantes.

Nom	Signification
<code>_IOFBF</code>	(<i>IO Full Buffered</i>) indique qu'on désire un buffer de bloc.
<code>_IOLBF</code>	(<i>IO Line Buffered</i>) pour réclamer un buffer de ligne.
<code>_IONBF</code>	(<i>IO No Buffered</i>) si on ne veut aucun buffer.

Dans ce dernier cas, les second et quatrième arguments sont ignorés.

Lorsqu'on fournit un buffer personnalisé, il doit pouvoir contenir au moins la taille indiquée en dernier argument. Ce buffer sera utilisé par le flux de manière opaque, il ne faut pas tenter d'y accéder. Il est très important de vérifier que le buffer reste bien disponible tant que le flux est ouvert.

En Python, à l'ouverture d'un flux, la valeur 0 en troisième argument indique que l'on ne veut aucun buffer, la valeur 1 réclame un buffer de ligne et une valeur supérieure à 1 demande un buffer de bloc de cette taille.

La fermeture du flux ne libère que les buffers qui ont été alloués par la bibliothèque C elle-même. Ceci inclut les buffers créés par défaut et ceux qui sont alloués lors de l'invocation de `setvbuf()` avec un second argument `NULL`.

Il y a un risque important de bogue, difficile à retrouver, lorsqu'un flux persiste à utiliser la zone mémoire qui lui a été affectée alors que celle-ci a déjà été libérée. Même la fonction `fclose()` est dangereuse si le buffer n'est plus valide. Voici un exemple de code erroné :

```
| #define TAILLE_BUFFER  
  
| int main (void)  
| {  
|     char buffer[TAILLE_BUFFER];  
|     FILE * fp = NULL;
```

```
    fp = fopen(...)  
    setvbuf(fp, buffer, _IOFBF, TAILLE_BUFFER);  
    fwrite(...)  
    [...]  
    return EXIT_SUCCESS;  
}
```

Ce code est faux car la fermeture automatique des flux ouverts se produit après le retour de la fonction `main()`, et donc après la libération du buffer alloué automatiquement dans la pile. Cette zone n'étant plus valide, la fonction de libération va accéder à une portion de mémoire interdite et déclencher un signal `SIGSEGV` après le retour de `main()`. La pile n'étant pas toujours gérée de la même manière suivant les systèmes d'exploitation et les compilateurs, l'erreur peut apparaître de manière totalement intempestive lors d'un portage d'application. Il faut donc être très prudent avec les buffers alloués explicitement. Il vaut mieux, autant que possible, laisser la bibliothèque C gérer l'allocation et la libération, en lui passant un pointeur `NULL`. Si ce n'est pas possible, il est préférable d'utiliser un buffer alloué dynamiquement et de s'assurer que la libération a eu lieu après la fermeture du flux.

La constante `BUFSIZ` représente une valeur qui est normalement adéquate pour tout type de buffer. Toutefois, il vaut peut-être mieux employer une valeur encore plus adaptée au fichier. Pour cela, il faut interroger le noyau en utilisant l'appel système `stat()`. Ce dernier, qui sera détaillé dans le chapitre 29, remplit une structure de type `struct stat`, dont le membre `st_blksize` contient la taille de bloc optimale pour les entrées-sorties sur le système de fichiers utilisé. Il suffit donc de choisir une taille de buffer égale ou multiple de cette valeur :

```
FILE * ouvre_fichier (const char * nom)  
{  
    FILE * fp = NULL;  
    struct stat etat;  
    int    taille_buffer = BUFSIZ;  
  
    if ((fp = fopen(nom, "w+")) == NULL)  
        return NULL;  
    if (stat(nom, & etat) == 0)  
        taille_buffer = etat.st_blksize;  
    setvbuf(fp, NULL, _IOFBF, taille_buffer);  
    return fp;  
}
```

La fonction `setvbuf()` renvoie 0 si elle réussit. Sinon, le buffer précédent n'est pas modifié.

La fonction `setbuf()` permet uniquement de fournir un nouveau buffer, sans modifier son type, ou de supprimer toute mémoire tampon :

```
| void setbuf (FILE * flux, char * buffer);
```

Si le second argument est `NULL`, le flux n'a plus de buffer. Sinon, il faut fournir un pointeur sur une zone mémoire de taille `BUFSIZ` au minimum.

Il existe deux fonctions obsolètes `setbuffer()` et `setlinebuf()`, qu'on peut parfois rencontrer, et qui sont un héritage de BSD :

```
| void setbuffer (FILE * flux, char * buffer, size_t taille);
```

et :

```
| void setlinebuf (FILE * flux);
```

Elles peuvent toutes les deux être implémentées ainsi :

```
| void setbuffer (FILE * flux, char * buffer, size_t taille)
| {
|     if (buffer == NULL)
|         setvbuf(flux, NULL, _IONBF, 0);
|     else
|         setvbuf(flux, buffer, _IOFBF, taille);
| }
```

```
| void setlinebuf (FILE * flux)
| {
|     setvbuf(flux, NULL, _IOLBF, BUFSIZ);
| }
```

On peut s'interroger sur la nécessité d'utiliser ces fonctions puisque la bibliothèque Glibc attribue apparemment des buffers adéquats dans toutes les situations. Voici donc quelques cas où ces fonctions se révèlent utiles.

- Une application dont le seul rôle est de filtrer des lignes de texte, à la manière de `grep` par exemple, améliore ses performances en forçant un buffer de ligne sur `stdout`, même si ce flux n'est pas connecté à un terminal. En effet, cette sortie peut être redirigée par un tube du shell vers une autre application qui finira par faire l'affichage sur le terminal. La cohérence de l'ensemble sera mieux assurée si tous les composants du tube traitent des lignes de texte en une seule fois.
- Un programme recevant des données en temps réel, sur une socket réseau par exemple, pour les traiter et les renvoyer sur sa sortie standard pourra forcer la sup-

pression du buffer sur stdout, pour laisser les informations ressortir au même rythme qu'il les a reçues.

- Un processus peut employer une socket réseau pour envoyer des messages à afficher sur la console d'un administrateur. Un buffer de type ligne installé sur cette socket rendra la communication plus efficace, en évitant notamment de laisser des lignes à moitié affichées en cas de ralentissement du trafic sur le réseau.
- Enfin, nous l'avons vu, pour améliorer les performances en écriture sur un fichier disque, il est possible d'interroger le noyau pour connaître la taille de bloc optimale et de configurer un buffer binaire en conséquence.

État d'un flux

Toute opération sur un flux est susceptible de poser des problèmes, et il est important de bien vérifier les conditions de retour de chaque lecture ou écriture. La difficulté ici ne se situe pas tellement au niveau de la programmation ou de l'implémentation, mais bien plus au niveau de la conception du logiciel. L'attitude à adopter en cas de détection d'un problème sur un fichier doit être définie d'une manière homogène pour toutes les entrées-sorties de l'application. Le couple drastique `perror()` / `exit()` ne peut guère être employé que dans des petits programmes, du niveau des exemples que nous fournissons ici. L'utilisateur attend d'une application qui s'exécute dans un environnement graphique une attitude un peu plus conviviale qu'un simple arrêt abrupt à la première difficulté, d'autant que, la plupart du temps, la sortie d'erreur standard de ces applications est redirigée par le gestionnaire de fenêtres vers le fichier `.xsession-errors`, et n'est donc pas visible immédiatement. L'attitude la plus simple est souvent de proposer à l'utilisateur de recommencer la sauvegarde ou la lecture après avoir modifié le nom du fichier ou libéré de la place sur le disque.

Les conditions d'erreur en lecture sont indiquées par un retour `NULL` pour `fgets()`, par exemple, ou par un nombre d'éléments lus inférieur à celui qui est demandé avec `fread()`.

Dans ces deux cas, il n'est pas possible de distinguer immédiatement une fin de fichier normale d'une erreur plus grave (système de fichiers corrompu, liaison NFS interrompue, support amovible extrait par erreur...). Pour cela, il faut appeler l'une des fonctions `feof()` ou `ferror()` déclarées ainsi :

```
int feof (FILE * flux);  
int ferror (FILE * flux);
```

La première renvoie une valeur non nulle si la fin du fichier a été atteinte, et la seconde adopte la même attitude si une autre erreur s'est produite. Dans ce cas, la variable globale `errno` peut être utilisée pour le diagnostic.

Il faut bien réaliser que ces deux fonctions n'ont de signification qu'après l'échec d'une lecture. Le code suivant est donc invalide :

```
void copie_flux_texte (FILE * flux_entree, FILE * flux_sortie)
{
    char    chaine[TAILLE_MAXI];

    while (! feof(flux_entree)) {
        fgets(chaine, TAILLE_MAXI, flux_entree);
        fputs(chaine, flux_sortie);
    }
}
```

En effet, la fin de fichier n'est détectée que lorsque la lecture a échoué. Comme elle n'a pas modifié la chaîne, qui contient la dernière ligne du fichier, celle-ci est écrite à nouveau une seconde fois. De plus, ce programme ne teste justement pas les conditions d'erreur.

En voici une version exacte, mais guère conviviale :

```
void copie_flux_texte (FILE * flux_entree, FILE * flux_sortie)
{
    char chaine[TAILLE_MAXI];

    while (fgets(chaine, TAILLE_MAXI, flux_entree) != NULL) {
        if (fputs(chaine, flux_sortie) == EOF) {
            perror("fputs");
            exit(EXIT_FAILURE);
        }
    }
    if (ferror(flux_entree)) {
        perror("fgets");
        exit(EXIT_FAILURE);
    }
}
```

En fait, la solution la meilleure consisterait probablement à renvoyer une valeur nulle en cas de réussite, et -1 en cas d'échec. La routine appelante pourrait alors vérifier avec `ferror()` le flux ayant posé un problème, afficher un message dans une boîte de dialogue, et proposer de recommencer après avoir modifié les noms des fichiers.

On peut effacer volontairement les indicateurs d'erreur et de fin de fichier associés à un flux. Cela se fait automatiquement lorsqu'on invoque une fonction de positionnement comme `fseek()`, `fsetpos()` ou `rewind()`, mais aussi à l'aide de la routine `clearerr()` :

```
void clearerr (FILE * flux);
```

Nous avons bien indiqué qu'un flux est construit, par `fopen()`, autour d'un descripteur de fichier bas niveau. Celui-ci est représenté par un `int` ayant une signification pour le noyau. Il est possible d'obtenir le numéro de descripteur associé à un flux en utilisant la fonction `fileno()` :

```
| int fileno (FILE * flux);
```

Cette fonction renvoie le numéro du descripteur, ou `-1` en cas d'échec (si le flux mentionné n'est pas valide par exemple). En Python, la méthode `fileno()` d'un objet de type `File` renvoie le descripteur sous-jacent.

Nous verrons plus tard que la fonction `fcntl()` nous permet de manipuler des paramètres importants des descripteurs de fichiers, comme la lecture non bloquante ou les verrouillages, alors que ces opérations ne sont pas possibles directement avec les flux.

Conclusion

Nous avons examiné dans ce chapitre l'essentiel des fonctionnalités concernant la manipulation des fichiers sous forme de flux.

La fonction `fileno()` nous transmet donc le numéro du descripteur de fichier associé à un flux, mais dans certains cas nous désirerons travailler directement avec ces descripteurs en employant des primitives de bas niveau, des appels système, que nous allons étudier dans le prochain chapitre.

Descripteurs de fichiers

Nous analyserons dans ce chapitre les fonctions traitant directement les descripteurs de fichiers, tant du point de vue de la lecture ou écriture que pour les mécanismes plus complexes de contrôle des accès (verrouillage, lecture non bloquante...). Nous nous retrouvons donc à un niveau plus bas que dans le chapitre précédent ; ici nous serons plus proches du noyau.

Ouverture et fermeture d'un descripteur de fichier

Un descripteur est un entier compris entre 0 et la valeur de la constante `OPEN_MAX` qui est définie dans `<limits.h>` (1024 sous Linux). Les descripteurs 0, 1 et 2 sont réservés respectivement pour l'entrée et la sortie standard, ainsi que pour la sortie d'erreur. Ces valeurs sont employées directement dans un si grand nombre d'applications qu'elles sont probablement immuables, mais on peut toutefois les remplacer à profit par les constantes symboliques `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO`, qui sont définies dans `<unistd.h>`.

Il est possible d'obtenir des descripteurs à partir d'autres éléments que des fichiers. Les appels système `pipe()` ou `socket()` permettent d'avoir les descripteurs d'un tube de communication ou d'une liaison réseau. Nous reviendrons sur ces types de descripteurs dans les prochains chapitres.

Pour l'instant, nous allons nous intéresser au moyen d'obtenir un descripteur sur un fichier. Il existe pour cela deux appels système, `open()` et `creat()`, le premier a un prototype avec un argument optionnel :

```
int open(const char * nom_fichier, int attributs,
        ... /* mode_t mode */);
int creat (const char * nom_fichier, mode_t mode);
```

L'appel système `creat()`, de moins en moins utilisé, est en fait équivalent à :

```
open (nom_fichier, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

La fonction Python `os.open()` est identique à celle du C.

Elle prend en premier argument le nom d'un fichier à ouvrir. Le principe est le même qu'avec `fopen()` ; si cette chaîne commence par un `'/'` elle est considérée comme un chemin débutant à la racine du système de fichiers, sinon elle est prise en compte à partir du répertoire actuel.

Le second argument est une combinaison de plusieurs éléments assemblés par un *OU* binaire. Tout d'abord, il faut impérativement utiliser l'une des trois constantes suivantes.

Nom	Signification
O_RDONLY os.O_RDONLY	Ouvrir le fichier en lecture seule.
O_WRONLY os.O_WRONLY	Ouvrir le fichier en écriture seule.
O_RDWR os.O_RDWR	Ouvrir le fichier en lecture et écriture.

Il est important de bien réaliser qu'il s'agit de trois constantes indépendantes et que le mode lecture-écriture n'est pas une association du mode lecture seule et du mode écriture seule. La constante symbolique `O_RDWR` n'est pas un *OU* binaire entre les deux autres.

Ensuite on peut utiliser les constantes suivantes, qui permettent de préciser le mode d'ouverture.

Nom	Signification
O_CREAT os.O_CREAT	Créer le fichier s'il n'existe pas. Ceci fonctionne même avec une ouverture <code>O_RDONLY</code> , bien que le fichier reste désespérément vide. Si l'argument <code>O_CREAT</code> n'est pas fourni, l'appel système échoue quand le fichier n'existe pas.
O_EXCL os.O_EXCL	Cette constante doit être employée conjointement à <code>O_CREAT</code> . L'ouverture échouera si le fichier existe déjà. Ceci nous permet de garantir que nous venons de créer le fichier. L'appel système étant atomique, nous sommes également assurés de ne pas entrer en conflit avec un processus concurrent tentant la même opération.
O_TRUNC os.O_TRUNC	Si le fichier existe déjà, sa taille sera ramenée à zéro. Cette option ne doit normalement être utilisée qu'en ouverture <code>O_RDWR</code> ou <code>O_WRONLY</code> .

Enfin, les constantes suivantes sont utilisées pour paramétrer le mode de fonctionnement du fichier lors des lectures ou écritures.

Nom	Signification
O_APPEND os.O_APPEND	Il s'agit d'un mode d'ajout. Toutes les écritures auront lieu automatiquement en fin de fichier. Ce mode d'écriture peut aussi être modifié après l'ouverture du fichier, en utilisant l'appel système <code>fcntl()</code> . Il ne faut pas confondre le mode d'écriture en fin de fichier et le mode d'ouverture lui-même. <code>O_APPEND</code> peut très bien être associé à <code>O_TRUNC</code> par exemple, même si cela paraît étonnant au premier abord. C'est le moyen de créer des fichiers de journalisation qu'on réinitialise à chaque démarrage du programme. L'avantage de ce mode d'écriture est que le déplacement en fin de fichier est lié de manière atomique à l'écriture, ce qui est indispensable quand plusieurs processus doivent écrire dans le même fichier (justement dans le cas d'une journalisation).
O_NOCTTY os.O_NOCTTY	Si le descripteur ouvert est un terminal, il ne faut pas le prendre comme terminal de contrôle du processus, même si ce dernier n'en a pas à ce moment-là.
O_NONBLOCK os.O_NONBLOCK	Cet attribut indique que les accès aux descripteurs seront non bloquants. En fait, cette option n'est jamais intéressante avec les fichiers disque, aussi son emploi avec <code>open()</code> est-il très rare. On le réserve aux files ou à certains fichiers spéciaux correspondant à des périphériques. Traditionnellement, on se sert plutôt de l'appel système <code>fcntl()</code> pour configurer l'option de non-blocage après l'ouverture des descripteurs où il peut servir (sockets, tubes...). Les seuls cas où <code>O_NONBLOCK</code> est indispensable avec <code>open()</code> sont l'ouverture d'un fichier spécial correspondant à un port série et l'ouverture des deux extrémités d'un tube nommé dans le même processus. Nous décrivons ces deux situations dans les chapitres 25 et 32.
O_SYNC os.O_SYNC	Les écritures sur le descripteur auront lieu de manière synchronisée. Cela signifie que le noyau garantit que l'appel système <code>write()</code> ne reviendra pas avant que les données aient été transmises au périphérique. Rappelons que, dans le cas de disques SCSI par exemple, les contrôleurs peuvent encore garder les données en mémoire tampon pendant une durée certaine avant leur écriture physique. Nous reparlerons de cette option en étudiant l'appel <code>write()</code> .

En fait, on utilise couramment `O_CREAT` et `O_TRUNC`, plus rarement `O_APPEND` et `O_EXCL`. Pour travailler sur les ports série, on emploie souvent `O_NONBLOCK` (ou `O_NDELAY` qui est un alias obsolète), mais les autres constantes sont nettement moins sollicitées.

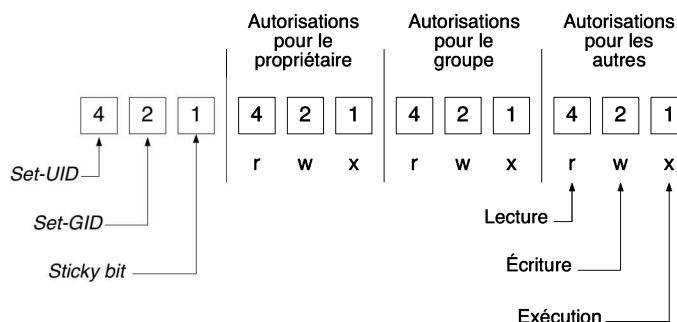
Le troisième argument de l'appel `open()` ne sert que lors d'une création de fichier. Il faut donc que l'attribut `O_CREAT` ait été indiqué. Cette valeur, de type `mode_t`, sert à signaler les autorisations d'accès au fichier nouvellement créé. On peut la fournir directement en mentionnant la valeur numérique. Celle-ci n'est lisible que dans une représentation octale, et doit donc être préfixée par un '0' en langage C pour être comprise comme telle par le compilateur. Il est toutefois préférable de cumuler, par l'intermédiaire d'un OU binaire, les constantes suivantes.

Constante	Valeur octale	Signification
S_ISUID	04000	Activation du bit <i>Set-UID</i> . Le programme s'exécutera avec l' <i>UID effectif</i> de son propriétaire.
S_ISGID	02000	Activation du bit <i>Set-GID</i> . Le programme s'exécutera avec le <i>GID effectif</i> de son groupe.
S_ISVTX	01000	Activation du bit « <i>Sticky</i> ». N'a apparemment plus d'utilité pour les fichiers réguliers de nos jours.
S_IRUSR	00400	Autorisation de lecture pour le propriétaire du fichier.
S_IWUSR	00200	Autorisation d'écriture pour le propriétaire du fichier.
S_IXUSR	00100	Autorisation d'exécution pour le propriétaire du fichier.
S_IRWXU	00700	Lecture + Écriture + Exécution pour le propriétaire du fichier.
S_IRGRP	00040	Autorisation de lecture pour le groupe du fichier.
S_IWGRP	00020	Autorisation d'écriture pour le groupe du fichier.
S_IXGRP	00010	Autorisation d'exécution pour le groupe du fichier.
S_IRWXG	00070	Lecture + Écriture + Exécution pour le groupe du fichier.
S_IROTH	00004	Autorisation de lecture pour les autres utilisateurs.
S_IWOTH	00002	Autorisation d'écriture pour les autres utilisateurs.
S_IXOTH	00001	Autorisation d'exécution pour les autres utilisateurs.
S_IRWXO	00007	Lecture + Écriture + Exécution pour les autres utilisateurs.

L'ensemble d'autorisations qu'on utilise le plus fréquemment est « S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH » (0644), qui permet de donner les droits de lecture à tous et les droits d'écriture seulement au propriétaire. Un programme peut parfois créer un fichier exécutable (disons un script shell, pas obligatoirement un exécutable binaire !). Dans ce cas, il utilisera probablement les permissions « S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH » (0755).

Figure 22-1

Valeur octale des autorisations d'accès



Il faut être très prudent avec les autorisations qu'on accorde au groupe du fichier. En effet, les systèmes Unix peuvent adopter deux attitudes différentes lors de la création d'un fichier :

- certains donnent au nouveau fichier le groupe effectif du processus qui le crée ;
- d'autres utilisent le groupe du répertoire dans lequel le fichier est placé. Ceci permet d'assurer la cohérence de larges arborescences.

Ces deux mécanismes étant autorisés par SUSv4, il est important de vérifier, après la création d'un fichier, que son groupe est bien celui qui est attendu, si on a employé `S_IWGRP` par exemple.

Linux adopte l'attitude a priori la plus sage, qui consiste à utiliser le groupe effectif du processus créateur, à moins que le bit *Set-GID* ne soit positionné sur le répertoire d'accueil. Dans ce cas, c'est le groupe de ce dernier qui est choisi.

Le mode ainsi transmis est toutefois filtré à travers le `umask` du processus. Cette valeur, à laquelle nous verrons comment accéder dans un prochain chapitre, est retirée du mode indiqué. Ainsi, si le `umask` du processus vaut 0002, un mode 0666 sera automatiquement converti en 0664.

Il est important de fournir un argument mode lorsqu'on utilise l'option `O_CREAT` de `open()`, sinon les autorisations d'accès sont totalement imprévisibles (et généralement désastreuses).

Lorsqu'on a fini d'utiliser un descripteur, on le referme à l'aide de l'appel système `close()`, ou `os.close()` en Python :

```
| int close (int fd);
```

Comme nous l'avons remarqué avec `fclose()`, la valeur de retour de `close()` est la dernière chance de détecter une erreur qui s'est produite durant une écriture différée dans le fichier. Si `close()` ne renvoie pas 0, le contenu du fichier est probablement inexact, et il est important de prévenir l'utilisateur, afin de recommencer la sauvegarde des données par exemple.

Un aspect déroutant des entrées-sorties de bas niveau, par rapport à la bibliothèque `<stdio.h>`, est que les prototypes et les constantes utilisés sont répartis dans une multitude de fichiers d'en-tête qui sont évidemment susceptibles de changer suivant les versions d'Unix. Pour les fonctions que nous avons étudiées, il faut inclure les fichiers suivants.

Fichier	Utilité
<code><fcntl.h></code>	Contient les prototypes de <code>open()</code> et de <code>creat()</code> , ainsi que les constantes <code>O_XXX</code> .
<code><sys/stat.h></code>	Contient les constantes de mode <code>S_IXXX</code> .

Fichier	Utilité
<sys/types.h>	Pas obligatoire sous Linux, ce fichier peut être nécessaire sous d'autres versions d'Unix pour obtenir la définition de <code>mode_t</code> .
<unistd.h>	Contient la déclaration de <code>close()</code> . Cette fonction n'est en effet pas limitée aux fichiers, mais sert pour tous les descripteurs Unix.

Il est donc conseillé d'inclure systématiquement ces quatre fichiers en début de programme pour pouvoir utiliser les descripteurs avec le maximum de portabilité.

L'exemple suivant présente plusieurs tentatives d'ouverture de fichiers. Nous affichons à chaque fois les arguments employés et le résultat.

```
exemple-open.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

void ouverture_fichier (char * nom, char * type, int attribut, mode_t
mode)
{
    int fd;
    fprintf(stderr, "%s (%s) : ", nom, type);
    fd = open(nom, attribut, mode);
    if (fd < 0) {
        perror("");
    } else {
        fprintf(stderr, "Ok\n");
        close(fd);
    }
}

int main (void)
{
    ouverture_fichier("/etc/passwd", "O_RDONLY", O_RDONLY, 0);
    ouverture_fichier("/etc/passwd", "O_RDWR", O_RDWR, 0);
    ouverture_fichier("essai.open", "O_RDONLY", O_RDONLY, 0);
    ouverture_fichier("essai.open", "O_RDWR", O_RDWR, 0);
    ouverture_fichier("essai.open", "O_RDONLY|O_CREAT, 0640",
        O_RDONLY | O_CREAT,
        S_IRUSR | S_IWUSR | S_IRGRP);
    ouverture_fichier("essai.open", "O_RDWR|O_CREAT|O_EXCL, 0640",
        O_RDWR | O_CREAT | O_EXCL,
        S_IRUSR | S_IWUSR | S_IRGRP);
    return EXIT_SUCCESS;
}
```


Ou une implémentation en Python :

```
exemple-open.py :
#!/usr/bin/python
from __future__ import print_function
import os

def open_file(name, label, mode):
    print("os.open({}, {}) :".format(name, label), end='')
    try:
        fd = os.open(name, mode, 0640)
        os.close(fd)
        print("Ok")
    except OSError:
        print("Error")

open_file("/etc/passwd", "O_RDONLY",
          os.O_RDONLY)
open_file("/etc/passwd", "O_RDWR",
          os.O_RDWR)
open_file("essai.fopen", "O_RDONLY",
          os.O_RDONLY)
open_file("essai.fopen", "O_RDONLY | O_CREAT",
          os.O_RDONLY|os.O_CREAT)
open_file("essai.fopen", "O_RDWR | O_CREAT | O_EXCL",
          os.O_RDWR | os.O_CREAT | os.O_EXCL)
```

Lors de l'exécution, les tentatives d'ouverture d'un fichier système dans `/etc/` ne fonctionnent évidemment qu'en lecture seule (ne pas exécuter ce programme avec les droits root !). En ce qui concerne le fichier `essai.open`, il n'est pas possible de l'ouvrir s'il n'existe pas, tant qu'on ne précise pas l'option `O_CREAT`. Par contre, dans ce cas, l'ouverture échoue si le fichier existe et qu'on a demandé l'exclusivité avec `O_EXCL`.

```
$ ./exemple-open
/etc/passwd (O_RDONLY) : Ok
/etc/passwd (O_RDWR) : Permission denied
essai.open (O_RDONLY) : No such file or directory
essai.open (O_RDWR) : No such file or directory
essai.open (O_RDONLY|O_CREAT, 0640) : Ok
essai.open (O_RDWR|O_CREAT|O_EXCL, 0640) : File exists
$ ls -l essai.open
-rw-r----- 1 cpb cpb 0 5 août 14:25 essai.open
$ rm essai.open
$
```

Nous vérifions que les droits accordés sont bien ceux qu'on a demandés. En revanche, l'exemple très simple qui suit montre l'influence de l'attribut `umask` du processus créant le fichier.

```
exemple-open-2.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main (void)
{
    int fd;
    if ((fd=open("essai.open", O_RDWR|O_CREAT|O_EXCL, 0777)) < 0)
        perror("open");
    else
        close(fd);
    return EXIT_SUCCESS;
}
```

Ou :

```
exemple-open-2.py :
#!/usr/bin/python
from __future__ import print_function
import os

try:
    fd = os.open("essai.open", os.O_RDWR | os.O_CREAT | os.O_EXCL, 0777)
    os.close(fd)
except OSError:
    print("Error")
```

Nous demandons la création d'un fichier avec toutes les autorisations possibles.

```
$ umask
0002
$ ./exemple-open-2
$ ls -l essai.open
-rwxrwxr-x 1 cpb cpb 0 5 août 14:28 essai.open
$ rm essai.open
$
```

Lors de l'exécution du programme, le contenu de notre attribut `umask` est extrait des autorisations demandées, ce qui supprime le droit d'écriture pour tout le monde.

Nous avons indiqué, dans le paragraphe concernant l'ouverture d'un flux, que la bibliothèque Glibc ajoutait une extension Gnu à la fonction `fopen()`, en permettant de demander une ouverture exclusivement si le fichier n'existe pas. Ce mécanisme peut être indispensable pour s'assurer que deux processus concurrents ne risquent pas d'écrire simultanément dans le même fichier. Cette option n'étant généralement pas disponible sur d'autres environnements que la Glibc, on peut être tenté de l'implémenter naïvement ainsi :

```
FILE * fopen_exclusif (const char * nom_fichier, const char * mode)
{
    FILE * fp;
    if ((fp = fopen(nom_fichier, "r")) != NULL) {
        fclose(fp);
        errno = EEXIST;
        fp = NULL;
    } else {
        fp = fopen(nom_fichier, mode);
    }
    return fp;
}
```

Cette routine ne fonctionne pas car le processus peut fort bien être interrompu entre la première tentative d'ouverture, qui sert à vérifier l'existence, et l'ouverture effective du fichier. Le noyau peut alors commuter vers une autre tâche concurrente qui crée également le même fichier. Les deux processus auront l'impression d'accéder exclusivement au fichier alors que ce ne sera pas le cas.

Pour éviter ce problème, il faut s'arranger pour que la vérification d'existence et l'ouverture même soient atomiquement liées. Ceci est garanti par l'appel système `open()` avec l'attribut `O_EXCL`. On peut alors utiliser `fdopen()` pour obtenir un flux construit autour du descripteur ainsi ouvert. Le programme suivant implémente correctement un `fopen()` exclusif.

```
exemple-open-3.c :
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
FILE * fopen_exclusif (const char * nom_fichier, const char * mode_flux)
{
    int lecture    = 0;
    int ecriture   = 0;
    int ajout      = 0;
    int creation    = 0;
    int troncature  = 0;
    int flags      = 0;
    int i;
    int fd;
    FILE * fp;

    for (i = 0; i < strlen(mode_flux); i ++) {
        switch (mode_flux[i]) {
            case 'a' :
                ecriture = lecture = ajout = 1;
                break;
            case 'r' :
                lecture = 1;
                break;
            case 'w' :
                ecriture = creation = troncature = 1;
                break;
            case '+' :
                ecriture = lecture = 1;
                break;
            default :
                // soyons tolerants... on ne dit rien
                break;
        }
    }
    if (lecture & ecriture)
        flags = O_RDWR;
    else if (lecture)
        flags = O_RDONLY;
    else if (ecriture)
        flags = O_WRONLY;
    else {
        errno = EINVAL;
        return (NULL);
    }
    if (creation)
        flags |= O_CREAT;
    if (troncature)
        flags |= O_TRUNC;
    flags |= O_EXCL;
    fd = open(nom_fichier, flags, 0644);
    if (fd < 0)
        return NULL;
}
```

```
    fp = fdopen(fd, mode_flux);
    return fp;
}

void ouverture (const char * nom, const char * mode, int exclusif)
{
    FILE * fp;
    fprintf(stderr, "Ouverture %sde %s, mode %s : ",
        (exclusif ? "exclusive " : ""), nom, mode);
    if (exclusif)
        fp = fopen_exclusif(nom, mode);
    else
        fp = fopen(nom, mode);
    if (fp == NULL)
        perror("");
    else {
        fprintf(stderr, "Ok\n");
        fclose(fp);
    }
}

int main (void)
{
    ouverture("essai.open_3", "w+", 1);
    ouverture("essai.open_3", "w+", 1);
    ouverture("essai.open_3", "w+", 0);
    return EXIT_SUCCESS;
}
```

Vérifions que les ouvertures réussissent quand le fichier n'existe pas, et qu'elles échouent sinon :

```
$ ./exemple-open-3
Ouverture exclusive de essai.open_3, mode w+ : Ok
Ouverture exclusive de essai.open_3, mode w+ : File exists
Ouverture de essai.open_3, mode w+ : Ok
$ rm essai.open_3
$
```

Nous avons vu les principales méthodes permettant d'obtenir un descripteur de fichier. Nous examinerons ultérieurement la notion de duplication d'un descripteur, mais pour le moment nous allons nous intéresser aux primitives permettant d'en lire le contenu ou d'y écrire des données.

Lecture ou écriture sur un descripteur de fichier

Contrairement au foisonnement de fonctions qui sont mises à notre disposition par la bibliothèque C pour lire ou écrire sur un flux de données, le nombre d'appels système manipulant les descripteurs est particulièrement concis. Il existe en tout six appels système pour lire ou écrire des données, dont quatre sont rarement employés et sont en fait des dérivés des deux principaux. Toutefois, à cause de notre proximité avec le noyau lors de l'utilisation de ces primitives de bas niveau, il est important de bien comprendre l'ensemble des phénomènes entrant en jeu.

Primitives de lecture

La routine de lecture la plus courante s'appelle... `read()`. Son prototype est déclaré ainsi dans `<unistd.h>` :

```
| ssize_t read (int descripteur, void * bloc, size_t taille);
```

Cette fonction lit dans le descripteur le nombre d'octets réclamés en troisième argument et les place dans le buffer fourni en deuxième argument. Si le descripteur permet le positionnement (par exemple un fichier disque), la lecture a lieu à l'emplacement indiqué par son indicateur de position, que nous étudierons dans une prochaine section. Ensuite, cet indicateur est augmenté du nombre d'octets lus. Si, au contraire, le descripteur ne permet pas le positionnement (port de communication série par exemple), la lecture a lieu à la position courante du descripteur. Dans ce cas, l'indicateur de positionnement n'est pas mis à jour.

En Python, `os.read(descripteur, taille)` renvoie une chaîne de la taille maximale indiquée.

L'appel système `read()` renvoie le nombre d'octets lus. Si cette valeur correspond à la taille demandée, tout s'est bien passé. Si cette valeur est inférieure à la taille attendue mais qu'elle est positive, l'appel système n'a pu lire qu'une partie des données voulues :

- pour un descripteur correspondant à un fichier ordinaire, on a probablement atteint la fin du fichier ;
- pour un tube de communication, le correspondant a fermé son extrémité du tube ;
- pour une socket, le protocole réseau utilise certainement des paquets de données de taille inférieure à celle qui est réclamée.

Dans le dernier cas, la situation est normale et se répétera probablement à chaque lecture. Par contre, dans le cas d'un tube ou d'un fichier, il est presque certain que

nous sommes arrivés à la fin des données lisibles (fin du fichier ou fermeture du tube). Pour s'en assurer, la lecture suivante devrait renvoyer 0.

Si on indique une taille nulle en troisième argument, `read()` n'a aucun effet et renvoie 0.

En cas de véritable erreur, `read()` renvoie -1. Le type `ssize_t` de sa valeur de retour correspond à un `size_t` signé. La valeur maximale que peut contenir ce type de donnée est indiquée par la constante symbolique `SSIZE_MAX` définie dans `<limits.h>`. Avec la Glibc sur un processeur 32 bits, elle vaut 2 147 483 647.

Il faut donc prendre trois cas en considération dans la valeur de retour de `read()` :

- Valeur de retour strictement positive : la lecture s'est bien passée, mais nous ne disposons que du nombre d'octets indiqué par la valeur de retour de la fonction. Si ce nombre est inférieur à la taille réclamée, ce n'est une erreur que si le contexte de l'application exige une lecture correspondant exactement à la dimension voulue.
- Valeur de retour nulle : fin de fichier ou de communication, mais pas d'erreur rencontrée jusque-là.
- Valeur de retour inférieure à zéro : une erreur s'est produite, il faut analyser la variable globale `errno`. Si cette dernière contient la valeur `EINTR`, il y a simplement eu un signal qui a interrompu l'appel système `read()` avant qu'il ait eu le temps de lire quoi que ce soit. Dans ce cas on peut recommencer sereinement la lecture.

Dans le cas d'un descripteur correspondant à un tube, à une socket, ou à un fichier spécial de périphérique pour lequel on a demandé des lectures non bloquantes, `read()` peut également renvoyer -1, et placer `EAGAIN` dans la variable `errno` simplement si aucune donnée n'est disponible.

Dans un programme travaillant avec des liaisons réseau ou des tubes de communication – cas où `read()` est un appel système lent –, il est ainsi fréquent d'en encadrer toutes les invocations ainsi :

```
while ((nb_octets_lus = read(fd, buffer, taille_voulue)) == -1)
    if (errno != EINTR)
        break;
if (nb_octets_lus > 0)
    ...
```

Cette boucle permet de recommencer la lecture tant que l'appel système est interrompu par un signal. Le problème des lectures non bloquantes est plus complexe, car on ne peut se contenter de faire une boucle `while()`, comme dans le cas de `EINTR`, au risque de voir notre programme boucler en consommant inutilement des cycles du processeur. Pour éviter cela, il existe plusieurs méthodes fondées sur les appels système `select()` et `poll()`, que nous verrons dans le chapitre 25.

Notons que si la lecture est bloquante et si un signal interrompt `read()` alors qu'il a déjà lu quelques octets, il renverra le nombre lu, sans signaler d'erreur. Les applications faisant un large usage de signaux et de tubes de communication (ou de sockets réseau) sont souvent obligées d'implémenter un mécanisme de mémoire tampon autour de l'appel système `read()` lorsqu'il faut lire des enregistrements constitués d'un nombre précis d'octets.

Un processus qui tente de lire depuis son terminal de contrôle alors qu'il se trouve en arrière-plan reçoit un signal `SIGTTIN`. Ce signal, par défaut, arrête le processus mais sans le tuer. Si toutefois `SIGTTIN` est ignoré, l'appel système `read()` échoue avec l'erreur `EIO`. On peut imaginer un programme demandant un certain nombre de confirmations à l'utilisateur en fonctionnement interactif, mais désirant ignorer volontairement `SIGTTIN`, lorsqu'il est lancé en arrière-plan, pour continuer à s'exécuter comme si de rien n'était. Il devra alors utiliser des valeurs par défaut pour les saisies attendues quand `read()` déclenche l'erreur `EIO`.

La seconde fonction de lecture que nous allons étudier est `readv()`, qui permet de regrouper plusieurs lectures dans un seul appel. L'intérêt de cette routine est de répartir sur plusieurs zones de données le coût d'un appel système. Cette fonction est déclarée ainsi dans `<sys/uio.h>` :

```
ssize_t readv (int descripteur,  
               const struct iovec * vecteurs,  
               int nombre);
```

Cette fonction lit séquentiellement les données provenant du descripteur indiqué en premier argument, et remplit les zones mémoire correspondant au nombre de vecteurs mentionné en dernier argument. Un tableau de vecteurs est transmis en second argument. Les vecteurs de type `struct iovec` contiennent les membres suivants.

Type	Nom	Utilisation
<code>void *</code>	<code>iov_base</code>	Un pointeur sur la zone mémoire de ce vecteur.
<code>size_t</code>	<code>iov_len</code>	La longueur de cette zone mémoire.

La valeur renvoyée est le nombre total d'octets lus (et pas le nombre de vecteurs remplis). Suivant les systèmes, le type de la valeur de retour de cet appel système peut être `int` ou `ssize_t`, aussi ne faut-il pas s'étonner d'avoir un avertissement – sans conséquence – du compilateur. Les conditions d'erreur sont les mêmes que pour `read()`. On peut l'utiliser par exemple pour grouper la lecture binaire de plusieurs variables :

```
int    numero;  
double x, y, z;
```



```
struct iovec vecteur[4];

vecteur[0].iov_base = & numero;
vecteur[0].iov_len = sizeof(int);
vecteur[1].iov_base = & x;
vecteur[1].iov_len = sizeof(double);
vecteur[2].iov_base = & y;
vecteur[2].iov_len = sizeof(double);
vecteur[3].iov_base = & z;
vecteur[3].iov_len = sizeof(double);

nb_lus = readv(fd, vecteur, 4);
if (nb_lus != sizeof(int) + 3 * sizeof(double))
    return -1;
```

On pourrait effectuer le même travail avec une structure regroupant les diverses variables, mais le compilateur insère, pour aligner les champs, des octets supplémentaires susceptibles de nous compliquer la lecture. Malgré tout, cet exemple n'est certainement pas le meilleur car les lectures groupées avec `readv()` deviennent surtout performantes lorsqu'il y a un nombre important de vecteurs de petites tailles.

Il existe une autre fonction de lecture nommée `pread()`, assez peu utilisée. Elle a été implémentée sous Linux sous forme d'appel système à partir du noyau 2.2. Elle est déclarée dans `<unistd.h>` :

```
ssize_t pread (int descripteur, void * bloc,
               size_t taille, off_t position);
```

Pour qu'elle soit effectivement déclarée dans `<unistd.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur 500, avant d'inclure le fichier d'en-tête. Cette fonction sert à implémenter les mécanismes d'entrée-sortie asynchrones que nous verrons dans le chapitre 25.

Le comportement de cette routine ainsi que sa valeur de retour sont identiques à ceux de `read()`, sauf que les données ne sont pas lues directement à la position courante dans le descripteur mais à celle qui est indiquée en quatrième argument. Cette position est mesurée en octets depuis le début du fichier. De plus, `pread()` ne modifie pas la position courante du descripteur, celle-ci restant inchangée au retour de l'appel système. Bien entendu cette fonction échoue si le descripteur ne permet pas le positionnement (par exemple un tube). Il est important de remarquer que cette fonction, malgré son nom, n'a aucun rapport avec `popen()` et `pclose()` que nous avons analysées dans le chapitre 3.

Primitives d'écriture

Les trois appels système d'écriture que nous allons examiner représentent le contrepoint des primitives de lecture. Nous trouvons `write()`, `writew()` et `pwrite()`, dont les prototypes sont :

```
ssize_t write(int descripteur, const void * bloc,  
              size_t taille);  
ssize_t writew (int descripteur, const struct iovec * vecteur,  
                int nombre);  
ssize_t pwrite (int descripteur, const void * bloc,  
                size_t taille, off_t position);
```

La fonction `write()` écrit le contenu du bloc indiqué en deuxième argument dans le descripteur fourni en premier argument. Elle renvoie le nombre d'octets effectivement écrits ou -1 en cas d'erreur. Si la taille du bloc indiquée est nulle, `write()` transmet simplement 0, sans autre effet.

Lorsque le descripteur autorise le positionnement, et s'il n'a pas l'attribut `O_APPEND`, l'écriture prend place à la position courante de celui-ci. Sinon, l'écriture a lieu à la fin du fichier.

En Python, on utilise `os.write(descripteur, chaine)` pour obtenir un résultat similaire.

Pour bien analyser les problèmes de déplacement au sein du fichier et de concurrence des processus, nous devons observer le mécanisme interne des entrées-sorties. Ces concepts datent des premières versions d'Unix et sont restés à peu près constants au cours des évolutions de ce système.

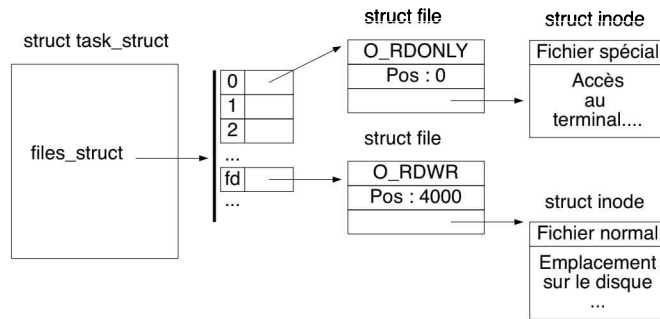
Dans la mémoire du noyau, chaque processus est représenté par une structure `task_struct`, naturellement inaccessible depuis l'espace utilisateur. Dans un champ de cette structure, il existe une table des descripteurs ouverts nommée `files_struct`. Cette table comprend pour chaque descripteur divers attributs (comme celui de fermeture sur exécution, que nous verrons plus loin) et un pointeur sur une structure `file`.

La structure `file` comporte, entre autres, le mode d'utilisation du descripteur (lecture, écriture, ajout...) ainsi que la position courante. Elle dispose indirectement d'un pointeur sur une structure `inode` qui rassemble toutes les informations nécessaires à la localisation réelle du fichier sur le disque ou à l'accès aux données si le descripteur correspond à une socket ou à un tube.

La structure `file` contient des pointeurs sur des fonctions qui, à la manière des méthodes de classe C++, implémentent les primitives d'entrée-sortie (`open`, `read`, `write`, `lseek`, `mmap`...) correspondant au type de fichier employé.

Un descripteur est donc entièrement décrit par trois niveaux de détails, qui font partie de l'implémentation traditionnelle d'Unix. Le processus dispose d'une table des descripteurs attribuant des numéros à chaque descripteur ouvert. Ceux-ci ont une correspondance dans la table des fichiers contenant notamment le mode d'accès et la position. Les fichiers possèdent à leur tour un correspondant dans la table des i-nœuds (*inode* ou *index node*) du système.

Figure 22-2
Indirections des descripteurs de fichiers



Nous devons revenir sur ce mécanisme lorsque nous étudierons les possibilités de partage de fichier. Pour l'instant, nous retiendrons que la position dans un descripteur appartient à la structure `file`, alors que la longueur du fichier est contenue dans la structure `inode`.

Lorsqu'on écrit des données dans un descripteur, le noyau emploie le pointeur de fonction `write()` de la structure `file` associée pour transmettre les informations. Ensuite, il augmente la position courante de cette structure du nombre d'octets écrits. Si cette position dépasse la taille du fichier mémorisée dans l'i-nœud, celle-ci est mise à jour.

Lorsque le mode d'écriture du descripteur correspond à un ajout en fin de fichier, la position d'écriture prend d'abord la valeur de la taille de l'i-nœud associé avant de faire l'écriture. L'écriture sur un descripteur en mode `O_APPEND` est donc atomiquement constituée d'un déplacement de la position courante en fin de fichier, suivi de l'envoi effectif des données. Ceci est très important si deux processus ou plus essaient d'écrire simultanément en fin du même fichier. Ils ne risquent pas d'être interrompus entre le moment du positionnement en fin de fichier et l'écriture proprement dite, comme cela pourrait être le cas avec une implémentation naïve :

```
lseek (fd, 0, SEEK_END);
write (fd, bloc, taille);
```

L'utilisation de `O_APPEND` est donc parfaitement adaptée à la construction de fichiers de journalisation mémorisant les événements survenus dans plusieurs processus concurrents.

Ceci est également vrai avec les flux ouverts en mode « a » ou « a+ », qui permettent d'employer la fonction `fprintf()` pour écrire plus facilement des messages comprenant la valeur de certaines variables. En revanche, il faut savoir que deux écritures successives peuvent être séparées dans le fichier par des données provenant d'un autre processus.

Finalement, l'appel système `write()` peut également s'assurer que les données sont immédiatement transmises au disque si le mode d'utilisation du descripteur contient l'option `O_SYNC`. Dans ce cas, le noyau demande le stockage immédiat des informations, y compris celles qui correspondent à la structure même du fichier (l'i-nœud). Bien entendu cette option diminue considérablement les performances du programme. Elle ne doit être employée que dans des cas particulièrement rares (système d'enregistrement embarqué de type « boîte noire » par exemple). Les différents processus qui accèdent à un fichier ont de toute manière une vision cohérente de celui-ci, et il n'est pas nécessaire d'utiliser `O_SYNC` pour d'autres besoins que la gestion des cas d'arrêts critiques ou de l'extraction inopinée d'un périphérique amovible.

Dans le même ordre d'idées, l'appel système `fsync()` permet de demander la mise à jour des informations sur le disque de manière ponctuelle. Il est déclaré ainsi dans `<unistd.h>` :

```
| int fsync (int fd);
```

En Python : `os.fsync(fd)`.

On lui transmet simplement le descripteur du fichier à réécrire, et il renvoie 0 s'il réussit ou -1 s'il échoue. En ce cas, `errno` peut indiquer une erreur d'entrée-sortie grave avec le code `EIO`. Ceci peut arriver en cas de retrait inattendu d'un support amovible comme une clé USB. L'appel système `sync()`, qui ne prend pas d'argument, sert à synchroniser l'ensemble des données en attente d'écriture différée. Nous avons déjà évoqué cet appel système en détaillant le fonctionnement des buffers associés aux flux.

Lorsque `write()` réussit, il transmet le nombre d'octets écrits. Si une erreur s'est produite, `write()` renvoie -1, et on peut analyser `errno`.

Il est très important de vérifier le code de retour de chaque écriture. Quel que soit le disque dur utilisé, une seule chose est à peu près sûre, c'est qu'un jour ou l'autre il sera saturé. Cela peut se produire de manière tout à fait accidentelle. Voici une anecdote qui m'est arrivée il y a quelques semaines, qui illustre bien ce cas de figure. Après une modification de configuration de l'environnement graphique, celui-ci tentait de jouer des fichiers sonores pour la plupart des événements (ouverture d'une fenêtre, mise en icône...). Malheureusement, pour une question de droit d'accès au périphérique sonore, le gestionnaire audio ne pouvait arriver à jouer ses échantillons et affichait un message sur `stderr`. Ce message était, comme d'habitude, redirigé vers le fichier `.xsession-errors`, auquel évidemment personne ne fait attention, d'autant qu'il

n'apparaît pas dans un `ls -l`. Au bout de quelques jours de travail sans déconnexion, l'ensemble des messages d'erreur produits à chaque événement du gestionnaire de fenêtres représentait un fichier `.xsession-errors` de près d'un Go ! La partition correspondant au répertoire `/home` étant déjà très chargée, elle s'est trouvée saturée.

Le traitement de texte que j'utilisais à ce moment-là m'a indiqué que, le disque étant plein, il était obligé de désactiver les sauvegardes régulières automatiques. Cet avertissement précieux, attirant mon attention, a prouvé ainsi qu'aucune vérification du code de retour d'une écriture ne doit être négligée, y compris dans les fonctionnalités annexes comme les sauvegardes automatiques.

Une application bien conçue doit être prête à résister aux erreurs les plus farfelues de l'utilisateur :

- tentative de sauvegarde dans un répertoire correspondant à un CD-Rom ;
- extraction inopinée d'une clé USB en cours d'écriture ;
- administrateur système débutant ayant effacé par mégarde le nœud spécial `/dev/null` qui sera recréé automatiquement en tant que fichier normal par la première redirection exécutée par `root`, et qui remplira peu à peu la partition racine du système de fichiers.

Ceci sans compter tous les problèmes qui peuvent se poser avec un système de fichiers monté par NFS ou Samba, au gré des caprices du réseau, de l'alimentation électrique et de l'administrateur système distant.

La robustesse d'un programme dépendra donc de sa capacité à détecter au plus tôt les erreurs et à diagnostiquer correctement les problèmes pour proposer à l'utilisateur de remédier au défaut avant de recommencer la sauvegarde. La situation de la détection d'erreur au cours d'un `write()` est beaucoup plus cruciale que pendant un `read()`, car l'application est alors en possession de données non sauvegardées, qui peuvent représenter plusieurs heures de travail et qu'il faut absolument arriver à enregistrer. Il est hors de question que tout se termine tragiquement avec un simple message d'erreur.

Les situations d'erreur susceptibles d'être repérées lors d'un appel `write()` varient en fonction du type de descripteur utilisé. Nous pouvons toutefois résumer quelques scénarios classiques à prendre en considération.

- Le système de fichiers correspondant au descripteur ouvert est saturé. L'appel `write()` renvoie le nombre d'octets qu'il a écrits. S'il n'en a écrit aucun, il renvoie `-1` et `errno` contient l'erreur `ENOSPC`.
- Le fichier représenté par le descripteur a dépassé la limite maximale autorisée pour l'utilisateur. Nous verrons un exemple plus bas. L'appel `write()` renvoie le nombre d'octets écrits. Si aucun caractère n'est écrit, le processus reçoit le signal `SIGXFSZ`. Si ce signal est intercepté ou ignoré, `write()` renvoie `-1` et `errno` contient `EFBIG`.
- Une erreur physique s'est produite sur le disque ou l'utilisateur a inconsidérément extrait la clé de sauvegarde avant la fin du transfert. L'appel système `write()` échoue donc avec une erreur `EIO`.

- Le descripteur correspond à un tube ou à une socket connectée. Le processus lecteur a fermé l'autre extrémité du tube ou la connexion réseau est rompue. Le processus reçoit alors un signal SIGPIPE. S'il ignore ou intercepte ce signal, `write()` renvoie -1 et `errno` contient EPIPE.

Tout comme nous l'avions observé avec `read()`, il existe des situations d'échec de `write()` moins tragiques que les précédentes. Dans ce cas, on peut recommencer la tentative.

- L'écriture se fait dans un descripteur de type socket ou tube, qu'on a basculé en mode non bloquant. Si le descripteur est plein, l'appel système `write()` échoue et déclenche l'erreur EAGAIN en attendant qu'un processus lise les données déjà enregistrées.
- Durant l'écriture, un signal a été reçu alors qu'aucune donnée n'avait été écrite. Dans ce cas, `write()` renvoie -1 et place EINTR dans `errno`.
- On tente d'écrire dans une portion de fichier sur laquelle un autre processus vient de placer un verrouillage strict, comme nous le verrons plus loin. L'écriture échoue alors avec une erreur EAGAIN.

On remarquera qu'en cas d'échec de `write()` avec une erreur EAGAIN, il est probablement inutile de réessayer immédiatement l'écriture. Il vaut mieux laisser un peu de temps au processus lecteur pour vider le tube plein, ou à celui qui a verrouillé le fichier pour écrire ses données. On évitera donc de faire une boucle du type :

```
while ((nb_ecrits = write(fd, buffer, taille)) == -1)
    if ((errno != EINTR) && (errno != EAGAIN))
        break;
```

Cette boucle consomme inutilement du temps processeur. Il est préférable que le processus appelant se mette quelques instants en sommeil, en cas d'erreur EAGAIN, avant de recommencer sa tentative. Le code suivant est déjà préférable.

```
while ((nb_ecrits = write(fd, buffer, taille)) == -1) {
    if (errno == EINTR)
        continue;
    if (errno != EAGAIN)
        break;
    sleep(1);
}
```

Une autre solution encore plus performante peut être construite autour de l'appel système `select()`, que nous étudierons dans le chapitre 25.

Théoriquement, `write()` ne peut pas renvoyer une valeur nulle, sauf si on lui a demandé explicitement d'écrire 0 octet. Si l'appel système a pu écrire quelques caractères avant qu'une erreur se produise, il renvoie ce nombre d'octets, sinon il renvoie -1. Si un programme est susceptible de recevoir des signaux tout en employant des appels

système `write()` pouvant bloquer (sockets, tubes...), il faut construire une boucle permettant d'envoyer toutes les données, éventuellement en plusieurs fois.

On peut utiliser par exemple un code du genre :

```
ssize_t mon_write (int fd, const void * buffer, size_t taille)
{
    const void *   debut    = buffer;
    size_t         restant  = taille;
    ssize_t        ecrits   = 0;

    while (restant > 0) {
        while ((ecrits = write(fd, debut, restant)) == -1) {
            if (errno == EINTR)
                continue;
            if (errno != EAGAIN)
                return -1;
            sleep(1);
        }
        restant -= ecrits;
        debut += ecrits;
    }
    return taille;
}
```

Ceci, rappelons-le, ne concerne que des écritures se faisant dans des descripteurs susceptibles de bloquer (sockets, tubes, files...) alors que le processus risque de recevoir des signaux utilisés par l'application.

L'exemple suivant va mettre en relief le comportement de `write()` lors d'une tentative de dépassement de la taille maximale autorisée pour un fichier. Nous allons d'abord réduire la limite à une valeur plus faible et tenter des écritures successives. Nous restreignons la limite `FSIZE` à une valeur qui n'est pas un multiple de la taille du buffer écrit, afin d'obtenir en premier lieu un nombre d'octets écrits inférieur à celui qui est attendu. À la tentative suivante, `write()` échouera en déclenchant d'ailleurs le signal `SIGXFSZ`.

```
exemple-write.c :
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/resource.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```

#define TAILLE_BLOC 1024
#define DEPASSEMENT 767

void gestionnaire (int numero)
{
    fprintf(stderr, "Signal %d recu : %s\n", numero,
              strsignal(numero));
}

int main (void)
{
    struct rlimit  limite;
    int           fd;
    char bloc[TAILLE_BLOC];
    int           nb_ecrits;

    signal(SIGXFSZ, gestionnaire);
    if (getrlimit(RLIMIT_FSIZE, & limite) != 0) {
        perror("getrlimit");
        exit(EXIT_FAILURE);
    }
    limite.rlim_cur = 3 * TAILLE_BLOC + DEPASSEMENT;
    if (setrlimit(RLIMIT_FSIZE, & limite) != 0) {
        perror("setrlimit");
        exit(EXIT_FAILURE);
    }
    fd = open("essai.write", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    memset(bloc, 1, TAILLE_BLOC);
    do {
        nb_ecrits = write(fd, bloc, TAILLE_BLOC);
        if (nb_ecrits != TAILLE_BLOC) {
            fprintf(stderr, "nb_ecrits = %d\n", nb_ecrits);
            if (errno != 0) {
                fprintf(stderr, "errno = %d : ", errno);
                perror("");
            }
        }
    } while (nb_ecrits != -1);
    close(fd);
    return EXIT_SUCCESS;
}

```

L'exécution suivante montre bien les deux appels `write()` qui échouent : le premier n'écrit que 767 octets au lieu des 1 024 attendus. Le second appel déclenche `SIGXFSZ` et renvoie -1, et `errno` est correctement remplie.


```
$ ./exemple-write
nb_ecrits = 767
Signal 25 reçu : File size limit exceeded
nb_ecrits = -1
errno = 27 : File too large
$
```

Dans l'exemple en Python ci-après, on observe le même comportement :

```
exemple-write.py :
#!/usr/bin/python
from __future__ import print_function
import resource
import os

TAILLE_BLOC = 1024
DEPASSEMENT = 767
limit = 3 * TAILLE_BLOC + DEPASSEMENT
resource.setrlimit(resource.RLIMIT_FSIZE, (limit, limit))

fd = os.open("essai.write",
             os.O_WRONLY | os.O_CREAT | os.O_TRUNC, 0644)
s = "1" * TAILLE_BLOC

while True:
    n = os.write(fd, s)
    print(n, "bytes written")
```

Contrairement au programme précédent, ce script affiche les trois premières écritures qui réussissent normalement. En outre, on ne gère pas l'exception d'erreur, une fois la taille limite dépassée.

```
$ ./exemple-write.py
1024 bytes written
1024 bytes written
1024 bytes written
767 bytes written
Traceback (most recent call last):
  File "./exemple-write.py", line 18, in <module>
    n = os.write(fd, s)
OSError: [Errno 27] File too large
```

Les deux autres fonctions permettant d'écrire dans un descripteur sont `writew()` et `pwrite()`. L'appel système `writew()` est symétrique à `readv()` ; il permet d'écrire une

succession de valeurs en un seul appel. Son prototype est défini dans `<sys/uio.h>`. Les conditions d'échec sont les mêmes que celles de `write()`.

Pour que `pwrite()` soit déclaré dans `<unistd.h>`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur 500 avant l'inclusion du fichier d'en-tête. Cet appel système fonctionne comme `write()` mais en effectuant l'écriture à la position indiquée en dernier argument et sans modifier la position courante du descripteur. En plus des conditions d'échec identiques à celles de `write()` s'ajoutent celles de l'appel système de positionnement que nous allons voir à présent. Comme son homologue `pread()`, cet appel système sert surtout à implémenter les entrées-sorties asynchrones.

Positionnement dans un descripteur de fichier

Il n'existe qu'un seul appel système, nommé `lseek()`, permettant de consulter ou de modifier la position courante dans un descripteur de fichier. Son prototype est déclaré dans `<unistd.h>` :

```
| off_t lseek (int descripteur, off_t position, int debut);
```

Nous avons parlé du type `off_t` et de son implémentation sous forme d'entiers 32 ou 64 bits dans le chapitre précédent.

En Python, `os.lseek(descriptor, pos, debut)` se comporte de manière identique.

Cette fonction permet de déplacer la position courante dans le descripteur à la nouvelle valeur indiquée en second argument. Le point de départ, fourni en troisième argument, peut prendre comme avec `fseek()` l'une des valeurs suivantes : `SEEK_SET`, `SEEK_CUR` ou `SEEK_END`. Cet appel système renvoie la nouvelle position, mesurée en octets depuis le début du fichier, ou `-1` en cas d'erreur. Pour connaître la position courante, il suffit donc d'utiliser `lseek(fd, 0, SEEK_CUR)`.

Nous avons déjà indiqué que le positionnement dans un descripteur est mémorisé dans la table des fichiers et non dans la table des descripteurs. Si un processus ouvre un descripteur avant d'invoquer `fork()`, il partagera avec son fils la structure `file` de la table des fichiers. Le positionnement sera donc commun aux deux processus, tel que l'indique l'exemple ci-dessous, dans lequel nous avons supprimé toutes les vérifications d'erreur en retour de `lseek()`, afin de simplifier le listing.

```
| exemple-lseek.c :  
| #include <fcntl.h>  
| #include <stdio.h>  
| #include <stdlib.h>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>

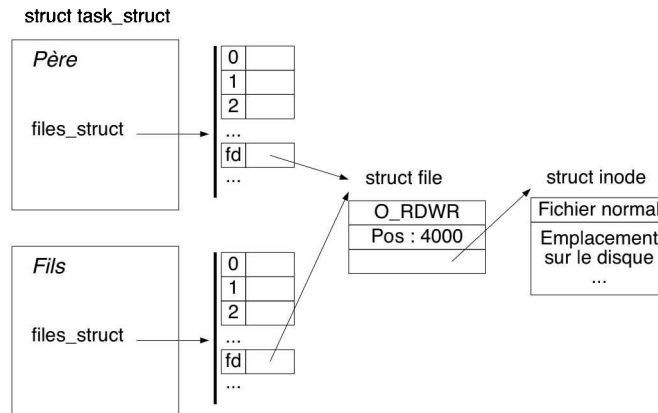
int main (void)
{
    int fd;
    pid_t pid_fils;
    off_t position;

    fd = open("essai.lseek", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    // On écrit quelques octets
    if (write(fd, "ABCDEFGHJIJ", 10) != 10) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    // Puis on sépare les processus
    if ((pid_fils = fork()) < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (pid_fils != 0) {
        // Processus père
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Pere : position = %ld\n", position);
        sleep(1);
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Pere : position = %ld\n", position);
        lseek(fd, 5, SEEK_SET);
        fprintf(stderr, "Pere : déplacement en position 5\n");
        waitpid(pid_fils, NULL, 0);
    } else {
        // Processus fils
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Fils : position = %ld\n", position);
        lseek(fd, 2, SEEK_SET);
        fprintf(stderr, "Fils : déplacement en position 2\n");
        sleep(2);
        position = lseek(fd, 0, SEEK_CUR);
        fprintf(stderr, "Fils : position = %ld\n", position);
    }
    close(fd);
    return EXIT_SUCCESS;
}
```

Dans cet exemple le processus père et le fils modifient alternativement la position d'un descripteur. Nous voyons que les déplacements effectués dans un processus sont immédiatement répercutés dans l'autre :

```
$ ./exemple-1seek
Pere : position = 10
Fils : position = 10
Fils : déplacement en position 2
Pere : position = 2
Pere : déplacement en position 5
Fils : position = 5
$
```

Figure 22-3
Partage de fichier entre
processus père et fils



Il est important de remarquer que l'utilisation de `lseek()` n'implique aucune entrée-sortie sur le système de fichiers correspondant au descripteur. Il ne s'agit que de la consultation ou de la modification d'un champ de la structure `file`, mais pas d'un accès réel au fichier. L'emploi de `lseek()` n'est donc pas exigeant en termes de performances (mis à part le coût d'un appel système) ; il ne risque pas de bloquer, mais ne fournit pas non plus de réelle information sur l'état du fichier correspondant. Les erreurs devront être détectées dans les appels système `read()`, `write()` ou `close()` suivants.

Manipulation et duplication de descripteurs

Nous avons observé qu'en cas d'utilisation de `fork()`, la table des descripteurs correspondant au processus père est copiée dans l'environnement du processus fils, mais que la structure `file` est commune aux deux processus.

Un processus peut aussi employer les appels système `dup()` ou `dup2()` pour obtenir une seconde copie d'un descripteur ouvert, pointant sur la même structure file que l'original. L'intérêt principal de ce mécanisme est de pouvoir modifier les descripteurs d'entrée et de sortie standard, en utilisant des sockets ou des tubes par exemple. Les prototypes de ces appels système sont déclarés dans `<unistd.h>` :

```
int dup(int descripteur);  
int dup2 (int descripteur, int nouveau);
```

La fonction `dup()` ou `os.dup()` en Python, permet d'obtenir une copie du descripteur fourni en argument. Cet appel système garantit que le numéro renvoyé sera le premier disponible dans la table des descripteurs du processus. Nous savons par ailleurs que par tradition les numéros de descripteur de `stdin`, `stdout` et `stderr` sont les trois premiers de cette table. Ainsi nous pouvons rediriger la sortie standard, par exemple dans un fichier, en utilisant le code suivant :

```
fd = open(fichier, ...)  
...  
close(STDOUT_FILENO);  
dup(fd);
```

Le premier numéro libre sera celui du descripteur de `stdout` une fois que celui-ci sera refermé. L'appel `dup()` permettra donc de le réaffecter. Nous avons déjà vu ce genre de comportement avec `freopen()` et les flux, mais l'avantage de `dup()` est de permettre la redirection vers des sockets, des tubes, des files, bref tous les éléments utilisables par le noyau sous forme de descripteurs.

Nous allons dans l'exemple suivant nous contenter d'un `dup()` sur un descripteur de fichier, qu'on met en place sur `stdout`, avant d'invoquer `ls`. L'affichage de ce dernier programme est donc redirigé vers le fichier désiré.

```
exemple-dup.c :  
#include <fcntl.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main (void)  
{  
    int fd;  
    fd = open("essai.dup", O_RDWR | O_CREAT | O_TRUNC, 0644);  
    if (fd < 0) {  
        perror("open");  
        exit(EXIT_FAILURE);  
    }  
}
```

```

close(STDOUT_FILENO);
if (dup (fd) < 0) {
    perror("dup");
    exit(EXIT_FAILURE);
}
close(fd);
execlp("ls", "ls", NULL);
perror("execlp");
exit(EXIT_FAILURE);
}

```

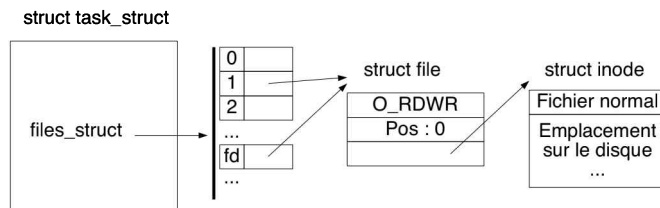
Nous voyons que la redirection a fonctionné, c'est la méthode utilisée par le shell pour exécuter la commande « `ls > essai.dup` ».

```

$ ./exemple-dup
$ cat essai.dup
essai.dup
exemple-dup
exemple-dup.c
exemple-fcntl
exemple-fcntl.c
exemple-fcntl-2.c
[...]
exemple-write
exemple-write.c
Makefile
$

```

Figure 22-4
Duplication d'un descripteur
vers la sortie standard



Le défaut de ce procédé réside dans le risque qu'un signal interrompe le processus entre la fermeture du descripteur à rediriger et la duplication du descripteur cible. Si le gestionnaire de ce signal utilise un appel système `open()`, `creat()`, `pipe()` ou `socket()` par exemple, il va occuper la place qu'on réservait pour la redirection. Aussi le noyau met-il à notre disposition un appel système `dup2()` qui effectue la redirection complète de manière atomique.

L'invocation de

```
| dup2 (fd, ancien);
```

ou `os.dup2(fd, ancien)` en Python, permet de fermer le descripteur ancien s'il est ouvert, puis de dupliquer `fd` en lui associant une nouvelle entrée à la position `ancien` dans la table des descripteurs. Cette méthode est donc plus fiable en ce qui concerne le risque d'interruption par un signal, mais elle permet aussi de rediriger à coup sûr le descripteur voulu, sans présumer de la numérotation effective des descripteurs de `stdin`, `stdout` et `stderr`.

Les programmes offrant des services réseau peuvent choisir d'utiliser leur propre système de connexion, construisant une socket et restant à l'écoute des demandes des clients, ou d'employer les services du démon `xinetd`, souvent surnommé le *super-serveur réseau*. Celui-ci gère automatiquement la mise en place d'un serveur sur un port précisé dans le fichier de configuration `/etc/xinetd.conf`. Lorsqu'un client établit une connexion, `xinetd` se duplique en utilisant `fork()` afin de relancer l'écoute en attente d'un autre client. Le processus fils redirige, avec l'appel système `dup2()`, son entrée et sa sortie standard vers la socket obtenue, avant de faire appel à `exec()` pour lancer l'application prévue. Celle-ci peut alors travailler directement `stdin` et `stdout` sans se soucier des détails de la programmation réseau.

Les appels système `dup()` comme `dup2()`, renvoient le nouveau descripteur obtenu, ou `-1` en cas d'erreur. Il existe une différence entre la copie du descripteur et l'original. La table des descripteurs contient en effet un attribut supplémentaire qui est remis à zéro lors de la duplication : l'attribut *close-on-exec*. Lorsqu'un processus invoque un appel système de la famille `exec()` pour lancer un autre programme, les descripteurs pour lesquels cet attribut est validé sont automatiquement fermés. L'attribut *close-on-exec* est remis à zéro de façon automatique lors d'une duplication, ce qui nous arrange puisqu'on utilise généralement `dup()` ou `dup2()` pour transmettre un fichier ouvert à un processus qu'on veut exécuter.

La modification de l'attribut *close-on-exec* peut se faire, entre autres, à l'aide de l'appel système `fcntl()` qui permet de consulter ou de paramétrer plusieurs aspects d'un descripteur. Cette fonction est déclarée dans `<fcntl.h>` ainsi :

```
| int fcntl (int descripteur, int commande, ...);
```

Les points de suspension finals indiquent que des arguments supplémentaires peuvent être ajoutés, en fonction de la commande invoquée. En Python, cette fonction est disponible dans le module `fcntl` :

```
| fcntl.fcntl(descripteur, commande[, argument])
```

Les commandes possibles sont variées, nous allons en observer quelques-unes.

Duplication de descripteur

Avec la commande `F_DUPFD`, `fcntl()` permet de dupliquer un descripteur à la manière de `dup()` ou de `dup2()`. Cette commande prend en troisième argument un numéro. Elle duplique le descripteur transmis en premier argument et lui attribue le premier emplacement libre de la table des descripteurs qui soit supérieur ou égal au numéro passé en troisième argument.

Ainsi :

```
| fcntl (fd, F_DUPFD, 0);
```

est équivalent à :

```
| dup (fd);
```

car il recherche le plus petit descripteur libre. De même :

```
| close (ancien);  
| fcntl (fd, F_DUPFD, ancien);
```

est équivalent à :

```
| dup2 (fd, ancien);
```

sauf que `dup2()` renvoie `EBADF` si `ancien` n'est pas dans les valeurs correctes pour un descripteur, alors que `fcntl()` renvoie `EINVAL`.

Accès aux attributs du descripteur

Les commandes `F_GETFD` et `F_SETFD` permettent de consulter ou de modifier les attributs du descripteur de fichier. De manière portable il n'existe qu'un seul attribut, *close-on-exec*, qu'on représente par la constante `FD_CLOEXEC`. Cet attribut est effacé par défaut lors de l'ouverture d'un descripteur.

On peut activer l'attribut *close-on-exec* d'un descripteur en utilisant :

```
| etat = fcntl (fd, F_GETFD);  
| etat |= FD_CLOEXEC;  
| fcntl (fd, F_SETFD, etat);
```


Le programme ci-dessous ouvre un descripteur de fichier puis, en fonction de son argument en ligne de commande, bascule l'attribut *close-on-exec* du descripteur. Ensuite il va exécuter un second programme qui vérifie si le descripteur dont le numéro lui est fourni sur sa ligne de commande est valide pour lui aussi. Le premier programme est :

```
exemple-fcntl.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main (int argc, char * argv[])
{
    char    numero[5];
    int fd;
    int etat;
    if (argc != 2)
        || ((strcasecmp(argv[1], "ferme") != 0)
        && (strcasecmp(argv[1], "laisse") != 0)) {
        fprintf(stderr, "syntaxe : %s [ferme|laisse]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fd = open("essai.fcntl", O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    write(fd, "AZERTYUIOP", 10);
    if ((etat = fcntl(fd, F_GETFD)) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
    if (strcasecmp(argv[1], "ferme") == 0)
        etat |= FD_CLOEXEC;
    else
        etat &= ~FD_CLOEXEC;
    if (fcntl(fd, F_SETFD, etat) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
    snprintf(numero, 5, "%d", fd);
    execlp("./verifie-descripteur", "./verifie-descripteur",
           numero, NULL);
    perror("execlp");
    exit(EXIT_FAILURE);
}
```

Notre second programme est le suivant :

```
verifie-descripteur.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main (int argc, char * argv[])
{
    int i;
    int fd;
    struct stat status;

    for (i = 1; i < argc; i++) {
        fprintf(stdout, "%s :", argv[i]);
        if (sscanf(argv[i], "%d", & fd) != 1) {
            fprintf(stdout, "invalide\n");
            continue;
        }
        if (fstat(fd, & status) != 0) {
            fprintf(stdout, "inaccessible\n");
            continue;
        }
        fprintf (stdout, "Ok (taille = %ld)\n", status.st_size);
    }
    return 0;
}
```

Lorsqu'on exécute le premier programme avec l'argument « ferme », celui-ci active l'attribut *close-on-exec* du descripteur, ce qui déclenchera la fermeture automatique avant d'invoquer notre second programme, qui indiquera que le descripteur est inaccessible :

```
$ ./exemple-fcntl ferme
3 :inaccessible
$
```

En revanche, si on l'invoque avec l'argument « laisse », le descripteur restera valide dans le nouveau programme, qui nous affichera la longueur du fichier correspondant (que nous venons d'initialiser avec une chaîne de caractères :

```
$ ./exemple-fcntl laisse
3 :Ok (taille = 10)
$
```

Attributs du fichier

Les attributs auxquels on peut accéder avec les commandes `F_GETFL` et `F_SETFL` sont ceux qui ont été indiqués lors de l'ouverture du fichier avec `open()`. Ces attributs appartiennent à la structure `file` de la table des fichiers et sont donc communs aux différents descripteurs qui pointent sur elle et qui sont obtenus à travers des appels `dup()` ou `fork()`.

Pour consulter le mode d'ouverture d'un fichier, il faut passer la valeur renvoyée à travers le masque `O_ACCMODE`, qui permet d'isoler les bits correspondant à `O_RDWR`, `O_RDONLY`, `O_WRONLY`. Le programme suivant permet d'examiner ces modes pour les trois descripteurs d'entrées-sorties standards.

```
exemple-fcntl-2.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    int etat;
    etat = fcntl(STDIN_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stdin : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    etat = fcntl(STDOUT_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stdout : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    etat = fcntl(STDERR_FILENO, F_GETFL) & O_ACCMODE;
    fprintf(stderr, "stderr : %s\n",
        (etat == O_RDWR) ? "R/W" : (etat == O_RDONLY) ? "R" : "W");
    return EXIT_SUCCESS;
}
```

Il est amusant de voir que le shell configure différemment les descripteurs des flux d'entrée-sortie standard en fonction du fichier sous-jacent.

```
$ ./exemple-fcntl-2
stdin : R/W
stdout : R/W
stderr : R/W
$ ./exemple-fcntl-2 < essai.fcntl
stdin : R
stdout : R/W
stderr : R/W
```

```
$ ./exemple-fcntl-2 > essai.fcntl
stdin : R/W
stdout : W
stderr : R/W
$
```

On obtient les mêmes résultats avec le script Python suivant :

```
exemple-fcntl-2.py :
#!/usr/bin/python
from __future__ import print_function
import fcntl
import os
import sys

def access_test(fd, name):
    etat = fcntl.fcntl(fd, fcntl.F_GETFL)
    etat = etat & (os.O_RDWR | os.O_RDONLY | os.O_WRONLY)
    if etat == os.O_RDWR:
        print(name, ": R/W")
    else:
        if etat == os.O_RDONLY:
            print(name, ": R")
        else:
            print(name, ": W")

access_test(sys.stdin.fileno(), "stdin")
access_test(sys.stdout.fileno(), "stdout")
access_test(sys.stderr.fileno(), "stderr")
```

La commande `F_SETFL` ne permet de modifier que les autres éléments du mode d'ouverture : `O_APPEND` et `O_NONBLOCK`.

Il est recommandé d'utiliser d'abord la commande `F_GETFL` afin d'obtenir l'état complet, puis d'y ajouter ou d'en extraire les constantes désirées avant d'invoquer la commande `F_SETFL`, contrairement à ce qui se fait trop souvent.

L'intérêt de cette commande de modification concerne essentiellement les descripteurs qu'on obtient autrement qu'avec l'appel système `open()`. Nous y reviendrons donc plus en détail dans le chapitre concernant les communications entre processus.

Quatre autres commandes seront étudiées avec les mécanismes d'entrées-sorties asynchrones puisqu'elles servent à configurer le ou les processus qui sont avertis par un signal lorsque des données sont prêtes à être lues :

- les commandes `F_GETOWN` et `F_SETOWN` indiquent les processus concernés ;
- les commandes `F_GETSIG` et `F_SETSIG` précisent le signal à utiliser.

Verrouillage d'un descripteur

On peut verrouiller l'accès à un fichier pour en assurer l'exclusivité de deux manières : avec la fonction `flock()` et avec les commandes `F_GETLK`, `F_SETLK` ou `F_SETLKW` de `fcntl()`. Ces deux méthodes sont distinctes et n'ont pas de répercussions l'une sur l'autre. La commande `flock()` est un héritage BSD qu'il vaut mieux éviter de nos jours.

Il est possible, avec `fcntl()`, de verrouiller une partie d'un fichier afin de garantir qu'un seul processus à la fois pourra modifier cette portion. Ce verrouillage peut être *coopératif*, ce qui signifie que les processus doivent vérifier eux-mêmes l'existence d'un verrou et s'abstenir de faire des modifications s'ils en trouvent un. C'est le seul comportement véritablement défini par Posix. Le défaut de ce mécanisme est l'impossibilité de se prémunir des modifications sauvages effectuées par un processus ne se pliant pas à l'autorité du verrouillage. Pour cela, le noyau Linux implémente comme de nombreux autres systèmes Unix un verrouillage *strict*. La portion de fichier ainsi bloquée est totalement immunisée contre les modifications par d'autres processus, même s'ils sont exécutés par root. La distinction entre verrouillage coopératif et verrouillage strict se fait au niveau du fichier lui-même, aussi étudierons-nous d'abord le principe du verrou coopératif, puis nous verrons comment le transformer en verrou strict.

Les verrous sont représentés par des structures `flock` qui sont définies dans `<fcntl.h>`, avec les cinq membres suivants.

Nom	Type	Utilisation
<code>l_type</code>	<code>short int</code>	Ce membre indique le type de verrouillage. Il peut s'agir de <code>F_RDLCK</code> pour un verrou en lecture, <code>F_WRLCK</code> pour un verrou en écriture, ou <code>F_UNLCK</code> pour supprimer le verrou.
<code>l_whence</code>	<code>short int</code>	On signale ainsi le point de départ de la mesure annonçant le début du verrouillage. C'est l'équivalent du troisième argument de <code>lseek()</code> , qui peut prendre les valeurs <code>SEEK_SET</code> , <code>SEEK_CUR</code> ou <code>SEEK_END</code> .
<code>l_start</code>	<code>off_t</code>	Ce champ précise le début de la portion verrouillée du fichier.
<code>l_len</code>	<code>off_t</code>	Longueur de la partie à verrouiller dans le fichier, mesurée en octets.
<code>l_pid</code>	<code>pid_t</code>	Ce membre est rempli automatiquement par le système pour indiquer le processus détenteur d'un verrou.

Le type de verrou, indiqué dans le premier membre de cette structure, a la signification suivante.

- `F_RDLCK` : le processus demande un accès en lecture sur la portion concernée du fichier, en s'assurant ainsi qu'aucun autre processus ne viendra modifier la partie qu'il lit. Plusieurs processus peuvent disposer simultanément d'un verrouillage en lecture sur la même portion de fichier.

- `F_WRLCK` : le processus veut modifier une partie du fichier. Il s'assure ainsi qu'aucun autre processus ne risque d'écrire au même endroit mais également qu'aucun ne tentera de verrouiller en lecture la portion concernée.

Le comportement peut donc être résumé ainsi :

- si une zone d'un fichier n'a aucun verrou, un processus pourra en placer un en lecture ou en écriture ;
- si une zone est verrouillée en lecture, un autre verrou en lecture sera accepté, mais pas un verrou en écriture ;
- si une zone dispose d'un verrou en écriture, aucun autre verrouillage ne sera accepté.

Lorsqu'on parle de deux verrouillages sur la même zone, il suffit en fait que les deux zones verrouillées aient une intersection non vide. Le noyau vérifie en effet les superpositions des portions demandées.

Si on indique une longueur `l_len` en nulle, cela signifie « jusqu'à la fin du fichier ». Bien entendu le point de départ peut être placé n'importe où, éventuellement au début si on veut verrouiller tout le fichier. Le verrouillage peut s'étendre au-delà de la fin du fichier si on désire y inscrire de nouvelles données.

Pour placer un verrou sur une portion d'un fichier, on peut employer les commandes `F_SETLK` ou `F_SETLKW` de `fcntl()`. Cette dernière commande est bloquante (*w* signifie *wait*). L'appel système `fcntl()` reste bloqué dans ce cas si un verrou est déjà présent, jusqu'à ce qu'il soit retiré. Cette fonction est toutefois interruptible par un signal, dans ce cas elle échoue et renvoie `EINTR`. La commande `F_SETLK` ne reste pas bloquée mais peut renvoyer `EACCES` ou `EAGAIN` suivant le type de verrouillage déjà présent. Voici donc deux méthodes de verrouillage en écriture de l'ensemble du fichier.

```
struct flocklock;
char          chaine[2];

lock.l_type = F_WRLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;

while (fcntl(fd, F_SETLK, &lock) < 0) {
    fprintf(stdout, "Fichier verrouille, reessayer ? ");
    fgets(chaine, 2, stdin);
    if (toupper(chaine[0]) == 'O')
        continue;
    return -1;
}
```

```
// Ici l'accès est autorisé,  
// on peut faire les modifications,  
// puis libérer le verrou  
fcntl(fd, F_UNLCK, & lock);  
return 0;
```

Voici à présent l'attente bloquante :

```
struct flock lock;  
char  chaine[2];  
  
lock.l_type = F_WRLCK;  
lock.l_whence = SEEK_SET;  
lock.l_start = 0;  
lock.l_len = 0;  
  
while (fcntl(fd, F_SETLKW, & lock) < 0)  
    if (errno != EINTR)  
        return -1;  
// Ici l'accès est autorisé,  
// on peut faire les modifications,  
// puis libérer le verrou  
fcntl(fd, F_UNLCK, & lock);  
return 0;
```

On peut aussi demander l'état du verrouillage sur un fichier en utilisant la commande `F_GETLK`, le troisième argument étant un pointeur sur une structure `flock`, comme avec `F_SETLK`. Cette structure sera modifiée au cours de l'appel pour représenter le verrou actuellement actif qui bloque l'accès à la portion voulue. Si aucun verrou n'est présent, le membre `l_type` est alors rempli avec la valeur `F_UNLCK`. Cette commande ne doit être utilisée qu'avec précaution, car l'état du fichier peut très bien être modifié entre le retour de l'appel système `fcntl()` et l'instruction suivante. Il ne faut s'en servir qu'à titre indicatif, notamment pour connaître le PID du processus tenant le fichier, comme dans cette attente non bloquante :

```
struct flock actuel;  
...  
  
while (fcntl(fd, F_SETLK, & lock) < 0) {  
    // Copier le verrou voulu dans la structure servant  
    // pour l'interrogation/  
    memcpy(& actuel, & lock, sizeof (struct flock));  
    // Interroger le noyau sur le verrouillage  
    if (fcntl(fd, F_GETLK, & actuel) < 0)  
        continue;
```

```

    if (actuel.l_type == F_UNLCK)
        // Le verrou a été supprimé entre temps
        continue;
    printf("Fichier verrouille par processus %d, reessayer ?",
           actuel.l_pid);
    fgets(chaine, 2, stdin);
    if (toupper(chaine[0]) == 'O')
        continue;
    return -1;
}
...

```

Il existe des situations où le verrouillage d'un descripteur conduit à un interblocage de deux processus. Supposons en effet que chaque processus a verrouillé une partie d'un fichier et réclame chacun un second verrou sur la partie tenue par l'autre. On arrive à une situation de blocage « à mort » que le noyau doit détecter et essayer d'éviter. Ceci peut se produire notamment quand plusieurs copies d'un même processus tentent simultanément d'ajouter des données à la fin d'un fichier et de mettre à jour une table des matières située au début. Dans un tel cas, le noyau fait échouer la tentative de verrouillage avec l'erreur EDEADLK (*dead lock*), comme nous allons le voir avec l'exemple suivant.

```

exemple-fcntl-3.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (void)
{
    int fd;
    int pid;
    struct flock lock;

    // Création d'un fichier avec quelques données
    if ((fd=open("essai.fcntl",O_RDWR|O_CREAT|O_TRUNC,0644)) < 0){
        perror("open");
        exit(EXIT_FAILURE);
    }
    write(fd, "ABCDEFGHJKLMNOPQRSTUVWXYZ", 26);
    if ((pid = fork()) == 0) {
        fprintf(stderr, "Fils : Verrou en Lecture de 0-1-2\n");
        lock.l_type = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start = 0;
        lock.l_len = 3;
    }
}

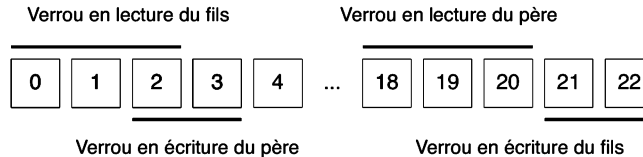
```



```
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror("FILS");
    else
        fprintf(stderr, "FILS : Ok\n");
    sleep(1);
    fprintf(stderr, "FILS : Verrou en Ecriture de 20-21-22\n");
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 20;
    lock.l_len = 3;
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror("FILS");
    else
        fprintf(stderr, "FILS : Ok\n");
    sleep(2);
} else {
    fprintf(stderr, "PERE : Verrou en Lecture de 18-19-20\n");
    lock.l_type = F_RDLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 18;
    lock.l_len = 3;
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror("PERE");
    else
        fprintf(stderr, "PERE : Ok\n");
    sleep(2);
    fprintf(stderr, "PERE : Verrou en Ecriture de 2-3\n");
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 2;
    lock.l_len = 2;
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror("PERE");
    else
        fprintf(stderr, "PERE : Ok\n");
    fprintf(stderr, "PERE : Liberation du verrou 18-19-20\n");
    lock.l_type = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 18;
    lock.l_len = 3;
    if (fcntl(fd, F_SETLKW, & lock) < 0)
        perror("PERE");
    else
        fprintf(stderr, "PERE : Ok\n");
    waitpid(pid, NULL, 0);
}
return EXIT_SUCCESS;
}
```

Nous remarquons que les zones verrouillées par les deux processus ne coïncident pas tout à fait, elles ont simplement des intersections communes.

Figure 22-5
Blocage entre père et fils



L'exécution montre bien que le noyau détecte un risque de blocage complet et fait échouer un appel système `fcntl()` :

```
$ ./exemple-fcntl-3
PERE : Verrou en Lecture de 18-19-20
PERE : Ok
FILS : Verrou en Lecture de 0-1-2
FILS : Ok
FILS : Verrou en Ecriture de 20-21-22
PERE : Verrou en Ecriture de 2-3
PERE: Resource deadlock avoided
PERE : Liberation du verrou 18-19-20
PERE : Ok
FILS : Ok
$
```

Encore une fois, nous nous sommes contentés d'utiliser des sommeils `sleep()` pour synchroniser les différentes phases des processus, ce qui ne fonctionne véritablement que sur un système peu chargé, mais permet de conserver des exemples assez simples. La détection des situations de blocage complet est assez performante puisqu'elle marche également quand de multiples processus tiennent chacun un maillon d'une chaîne en attendant la libération du suivant.

Ces verrouillages sont de type *coopératif*, ce qui signifie que chaque processus désireux de modifier un fichier doit se discipliner et utiliser les procédures d'accès adéquates. Aucune protection n'est assurée contre un processus qui outrepassse les verrouillages et modifie le fichier de manière anarchique. Pour éviter cela, le noyau implémente un mécanisme de verrouillage strict. Il suffit simplement de modifier le mode de protection du fichier et tous les verrouillages vus précédemment seront automatiquement renforcés par le noyau.

Un fichier est marqué comme verrouillable de manière stricte en modifiant les bits d'autorisation pour positionner le bit *Set-GID* tout en effaçant la permission d'exécu-

tion pour le groupe. Cette combinaison n'a pas de sens par ailleurs, aussi a-t-elle été choisie comme marque de verrouillage strict. On peut fixer les bits voulus ainsi :

```
$ chmod g-x fichier
$ chmod g+s fichier
```

ou à la création du fichier :

```
fd = open(fichier, O_RDWR | O_CREAT | O_EXCL, 02644);
```

Attention, certains systèmes de fichiers, par exemple *msdos* ou *vfat*, ne permettent pas de fixer les attributs *Set-GID* des fichiers. Les verrous y seront donc toujours coopératifs.

Lorsqu'un verrouillage est ainsi placé sur une portion de fichier, toutes les tentatives de modification de son contenu échoueront. Dans l'exemple suivant nous créons un fichier que nous verrouillons entièrement, puis nous attendons que l'utilisateur appuie sur « *Entrée* » pour le libérer.

```
exemple-fcntl-4.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    char   chaine[80];
    int    fd;
    struct flock flock;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_fichier\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    fd = open(argv[1], O_RDWR | O_CREAT | O_EXCL, 02644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    write(fd, "ABCDEFGHJIJ", 10);

    flock.l_type = F_WRLCK;
    flock.l_start = 0;
    flock.l_whence = SEEK_SET;
    flock.l_len = 10;
```

```

    if (fcntl(fd, F_SETLK, & flock) < 0) {
        perror("fcntl");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Verrou OK\n");
    fgets(chaine, 80, stdin);
    close(fd);
    return EXIT_SUCCESS;
}

```

Le blocage strict peut parfois être dangereux, principalement avec des répertoires exportés en NFS, car même root ne peut plus modifier le contenu d'un fichier verrouillé par un utilisateur. Aussi, le noyau gère-t-il une validation partition par partition des verrouillages stricts. En d'autres termes, il faut que la partition contenant le système de fichiers considéré ait été montée avec l'option *mand*, qui autorise les *mandatory locks*, c'est-à-dire les verrous stricts. Cette option n'est généralement pas validée par défaut, aussi root doit-il modifier le fichier `/etc/fstab` pour ajouter l'option *mand* et remonter la partition (ce qui ne nécessite pas de redémarrer la machine pour autant).

```

$ su
Mot de passe :
# vi /etc/fstab
    [ modification de /etc/fstab...]
# cat /etc/fstab
[...]
/dev/sda6    /home    ext3    defaults,mand    1 2
# mount -o remount /home
# mount
[...]
/dev/sda6 on /home type ext3 (rw,mand)
# exit
$

```

Bien sûr, chacun devra adapter ces commandes en fonction de la partition sur laquelle il travaille.

```

$ rm -fessai.*
$ ./exemple-fcntl-4 essai.lock
Verrou OK

```

Nous exécutons alors sur un autre terminal les tentatives d'écriture suivantes.

```
$ ls -l essai.lock
-rw-r-Sr-- 1 cpb cpb 10 7 août 10:54 essai.lock
$ cat essai.lock
(Contrôle-C car nous restons bloqués)
$ cat exemple-fcntl-4.c > essai.lock
bash: essai.lock: Ressource temporairement non disponible
$ su
Mot de passe :
# cat exemple-fcntl-4.c > essai.lock
bash: essai.lock: Ressource temporairement non disponible
# exit
$
```

Les appels système de lecture restent bloqués, tandis que les appels système d'écriture échouent avec l'erreur EAGAIN. Notons toutefois que le fichier n'est pas pour autant protégé contre une suppression par `rm`, car c'est l'écriture dans le répertoire qui est concernée lors d'un effacement et non une écriture dans le fichier. Ceci peut poser des problèmes avec les applications qui modifient des fichiers en commençant par en effectuer une copie (par exemple dans `/tmp`) avant de supprimer l'original et de le réécrire.

Autre méthode de verrouillage

Il existe un autre appel système permettant le verrouillage coopératif de fichiers. Héritée de l'univers BSD, la fonction `flock()` n'est pas normalisée par SUSv4 et est de moins en moins utilisée. Elle permet de verrouiller uniquement un fichier complet par l'intermédiaire d'un descripteur. Le prototype est déclaré dans `<sys/file.h>` :

```
| int flock (int descripteur, int commande);
```

En Python, c'est encore dans le module `fcntl` que cette fonction est implémentée : `fcntl.flock(descripteur, commande)`.

Les commandes possibles sont les suivantes :

- `LOCK_SH` : pour placer un verrou partagé. Plusieurs processus peuvent disposer simultanément d'un verrou partagé sur le même fichier. C'est l'équivalent des verrous `F_RDLCK` de `fcntl()`, qui permettent de s'assurer que le fichier ne sera pas modifié pendant qu'on veut le lire ;
- `LOCK_EX` : pour placer un verrou exclusif. Il ne peut y avoir qu'un seul verrouillage exclusif à la fois. Un processus l'utilise lorsqu'il veut écrire dans le descripteur ;
- `LOCK_UN` : pour supprimer un verrouillage précédemment installé.

On peut également ajouter avec un *OU* binaire la constante `LOCK_NB` pour empêcher la demande de verrouillage d'être bloquante. Si un verrou incompatible est déjà présent, l'appel système `flock()` échouera en renvoyant `-1` et en plaçant `EWOULDBLOCK` dans `errno`. Lorsqu'il réussit, l'appel `flock()` renvoie `0`.

Le verrouillage `flock()` n'est pas compatible avec celui qui est fourni par `fcntl()`, et les verrous `flock()` ne sont jamais renforcés de manière stricte par le noyau. De plus, il ne permet pas de bloquer uniquement certaines parties d'un fichier. Aussi évitera-t-on au maximum de l'employer.

Conclusion

Nous avons étudié dans ce chapitre l'essentiel des fonctionnalités concernant la manipulation des fichiers, sous forme de descripteurs de bas niveau.

Nous reviendrons sur certaines fonctionnalités concernant les descripteurs dans le prochain chapitre, traitant des communications entre processus, et dans le chapitre 25, traitant notamment des entrées-sorties asynchrones.

Communications classiques entre processus

Dès qu'une application dépasse un certain degré de complexité pour ce qui concerne les fonctionnalités indépendantes, on peut être tenté de la scinder en plusieurs entités distinctes, sous forme de processus par exemple.

Prenons le cas d'une base de données offrant des possibilités de consultation par l'intermédiaire de connexions TCP/IP. On peut diviser cette application en plusieurs tâches indépendantes. Le noyau principal s'occupe de superviser la base de données elle-même, en gérant notamment les problèmes d'accès simultanés. Un second module assure l'écoute des demandes de connexion et leurs initialisations. Enfin, on peut imaginer disposer d'une multitude de copies d'un dernier module, chargé du déroulement complet de la liaison avec le client, y compris l'interface de dialogue.

Pour construire ce genre de système, plusieurs options se présentent.

- Un seul processus s'occupe de tous les travaux. On conserve en mémoire une copie des données nécessaires au suivi de la connexion pour chaque client. Le processus bascule d'une fonction à l'autre au gré des requêtes grâce à l'appel système `select()` que nous étudierons dans le chapitre 25.
- On utilise un système à base de threads, l'accès aux informations globales de la base de données devant être strictement régi par des mutex. Les données propres à chaque connexion sont conservées dans des variables locales de la routine centrale du thread de communication.

- On scinde l'application en plusieurs processus, le noyau principal restant à l'écoute des requêtes de ses fils. Chaque module de communication est représenté par un processus indépendant doté de ses données propres, dialogue avec le client sur une liaison réseau et avec le noyau central par l'intermédiaire de l'une des différentes méthodes que nous allons étudier dans ce chapitre.

Finalement, chacune de ces méthodes a des avantages et des défauts.

- Le processus unique est plus facilement portable sur d'autres systèmes qu'Unix mais, en contrepartie, l'écriture et la maintenance de cette application sont plus compliquées car des fonctionnalités sans rapport entre elles sont regroupées dans le même logiciel.
- La combinaison de plusieurs threads offre une grande souplesse et une bonne portabilité, mais l'indépendance des modules n'est qu'illusoire. Lors d'une évolution du logiciel initial, l'accès à des données globales peut engendrer subitement des bogues difficiles à découvrir.
- La division en plusieurs processus permet d'avoir des modules vraiment indépendants, devant simplement se plier à une interface bien définie. Toutefois, le système est dépendant de l'architecture Unix, et la création d'un nouveau processus pour chaque connexion peut parfois être pénalisante.

Dans la première partie de ce chapitre, nous allons examiner le moyen de communication le plus simple pour deux processus issus de la même application (père et fils, ou frères) : les tubes (en anglais : *pipes*).

Il y a également des cas où l'ensemble applicatif repose sur plusieurs logiciels totalement indépendants. Ces programmes doivent disposer d'un autre moyen de communication puisque les tubes ne leur sont plus adaptés. Linux offre alors le concept de tubes nommés (*named pipes*), qui sont conçus pour cette situation, et que nous observerons en seconde partie.

Nous nous limitons pour le moment aux communications entre deux processus résidant dans le même système. Lorsqu'on veut faire dialoguer des logiciels se trouvant sur des stations différentes, il faut employer des méthodes que nous examinerons dans les chapitres traitant de la programmation réseau (mais qui ne diffèrent par ailleurs pas beaucoup des principes étudiés ici).

Les tubes

Un tube de communication est un tuyau dans lequel un processus écrit des données qu'un autre processus peut lire. Le tube est créé par l'appel système `pipe()`, déclaré dans `<unistd.h>` :

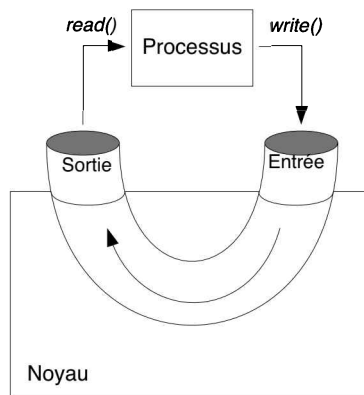
```
| int pipe (int descripteur [2]);
```


En Python, la fonction `os.pipe()` renvoie un couple de deux descripteurs (`fd_read`, `fd_write`).

Lorsqu'elle réussit, cette fonction crée un nouveau tube au sein du noyau et remplit le tableau passé en argument avec les descripteurs des deux extrémités. Étant donné que le langage C passe les arguments du type tableau par référence, la routine `pipe()` reçoit un pointeur sur la table et peut donc écrire dans les deux emplacements réservés. Les descripteurs correspondent respectivement à la sortie et à l'entrée du tube.

La situation est résumée sur la figure 23-1.

Figure 23-1
Tube de communication



Le tube est entièrement sous le contrôle du noyau. Il réside en mémoire (et pas sur le disque), et le processus reçoit les deux descripteurs correspondant à l'entrée et à la sortie du tube. Le descripteur d'indice 0 est la sortie du tube, il est ouvert en lecture seule. Le descripteur 1 est l'entrée, ouverte en écriture seule.

Nous observons en effet que les tubes sont des systèmes de communication unidirectionnels. Si on désire obtenir une communication complète entre deux processus, il faut créer deux tubes et les employer dans des sens opposés. Dans notre premier exemple, nous allons simplement créer un tube, écrire des données dedans, lire son contenu et vérifier que les informations sont identiques.

```
exemple-pipe-1.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    int tube[2];
    unsigned char buffer[256];
    int i;
```

```
fprintf(stdout, "Creation du tube\n");
if (pipe(tube) != 0) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
for (i = 0; i < 256; i++)
    buffer[i] = i;
fprintf(stdout, "Ecriture dans le tube\n");
if (write(tube[1], buffer, 256) != 256) {
    perror("write");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "Lecture depuis le tube\n");
if (read(tube[0], buffer, 256) != 256) {
    perror("read");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "Verification...");
for (i = 0; i < 256; i++)
    if (buffer[i] != i) {
        fprintf(stdout, "Erreur : i=%d buffer[i]=%d\n",
            i, buffer[i]);
        exit(EXIT_FAILURE);
    }
fprintf(stdout, "Ok\n");
return EXIT_SUCCESS;
}
```

Nous vérifions son fonctionnement.

```
$ ./exemple-pipe-1
Creation du tube
Ecriture dans le tube
Lecture depuis le tube
Verification...Ok
$
```

Le script suivant, plus concis, fait le même travail en Python :

```
exemple-pipe-1.py :
#!/usr/bin/python
from __future__ import print_function
import os

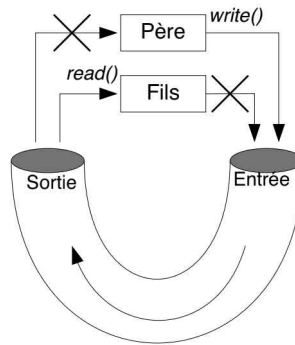
(fd_r, fd_w) = os.pipe()
w = "Hello Python's world"
```

```
n = os.write(fd_w, w)
r = os.read(fd_r, 4096)
if r == w:
    print("Ok")
```

Les descripteurs d'un tube, au même titre que tous les descripteurs de fichiers sont automatiquement fermés à la terminaison du processus.

Utiliser un tube pour transférer des données au sein du même processus ne présente aucun intérêt. Aussi nous allons utiliser ce mécanisme pour faire communiquer deux processus (ou plus). Pour cela, nous devons invoquer l'appel système `fork()` après avoir créé le tube. Si celui-ci doit aller du processus père vers le fils, le père ferme son descripteur de sortie de tube, et le fils son descripteur d'entrée. Nous expliquerons plus bas pourquoi la fermeture des extrémités inutilisées est importante. La figure 23-2 présente cet état de fait.

Figure 23-2
Tube du père vers le fils



Notre second exemple permet de tester ceci.

```
exemple-pipe-2.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (void)
{
    int tube[2];
    unsigned char buffer[256];
    int i;

    fprintf(stdout, "Creation tube\n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}
```

```

switch (fork()) {
    case -1 :
        perror("fork()");
        exit(EXIT_FAILURE);
        break;
    case 0 :
        fprintf(stdout, "Fils : Fermeture entree\n");
        close(tube[1]);
        fprintf(stdout, "Fils : Lecture tube\n");
        if (read(tube[0], buffer, 256) != 256) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        fprintf(stdout, "Fils : Verification \n");
        for (i = 0; i < 256; i++)
            if (buffer[i] != i) {
                fprintf(stdout, "Fils : Erreur\n");
                exit(EXIT_FAILURE);
            }
        fprintf(stdout, "Fils : Ok\n");
        break;
    default :
        fprintf(stdout, "Pere : Fermeture sortie\n");
        close(tube[0]);
        for (i = 0; i < 256; i++)
            buffer[i] = i;
        fprintf(stdout, "Pere : Ecriture dans tube\n");
        if (write(tube[1], buffer, 256) != 256) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        wait(NULL);
        break;
}
return EXIT_SUCCESS;
}

```

L'exécution confirme bien le fonctionnement du tube allant du processus père vers son fils.

```

$ ./exemple-pipe-2
Creation tube
Pere : Fermeture sortie
Pere : Ecriture dans tube
Fils : Fermeture entree
Fils : Lecture tube
Fils : Verification
Fils : Ok
$

```

Une implémentation en Python, assez proche :

```
exemple-pipe-2.py :
#!/usr/bin/python
from __future__ import print_function
import os

(fd_r, fd_w) = os.pipe()

if os.fork() == 0:
    print("Child process: closing the pipe input")
    os.close(fd_w)
    print("Child process: reading from the pipe")
    r = os.read(fd_r, 4096)
    print("Child process got:", r)
else:
    print("Parent process: closing the pipe output")
    os.close(fd_r)
    print("Parent process: writing into the pipe")
    os.write(fd_w, "Hello from your parent!")
    os.wait()
```

Ce script donne le résultat suivant :

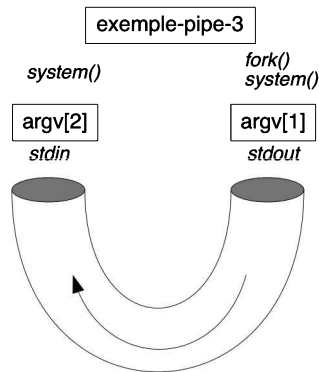
```
$ ./exemple-pipe-2.py
Parent process: closing the pipe output
Parent process: writing into the pipe
Child process: closing the pipe input
Child process: reading from the pipe
Child process got: Hello from your parent!
```

Nous remarquons que ce système est semblable au principe du *pipe* des shells, qui permet grâce au caractère « | » de diriger la sortie standard d'un processus vers l'entrée standard d'un autre. Pour illustrer ce principe, nous allons créer un programme qui prend deux commandes en arguments et qui les exécute en redirigeant la sortie standard de la première vers un tube connecté à l'entrée standard de la seconde.

Pour lancer les commandes nous utilisons `fork()` pour dissocier deux processus, le fils exécutant la première commande, et le père la seconde. Pour éviter d'avoir à analyser les chaînes de caractères pour séparer les commandes de leurs arguments, nous faisons appel à la fonction `system()`. En fait, il aurait été plus élégant d'employer `execvp()`, mais le traitement des chaînes de commande aurait été plus compliqué.

Le principe de notre programme est illustré par la figure 23-3.

Figure 23-3
Création d'un tube entre
deux commandes



Nous utilisons l'appel système `dup2()` que nous avons décrit dans le chapitre 22 pour remplacer les flux `stdin` et `stdout` des processus par les extrémités du tube.

```
exemple-pipe-3.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    int tube[2];

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s commande_1 commande_2\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    switch (fork()) {
        case -1 :
            perror("fork()");
            exit(EXIT_FAILURE);
            break;
        case 0 :
            close(tube[0]);
```

```

        dup2(tube[1], STDOUT_FILENO);
        system(argv[1]);
        break;
    default :
        close(tube[1]);
        dup2(tube[0], STDIN_FILENO);
        system(argv[2]);
        break;
    }
    return EXIT_SUCCESS;
}

```

Pour vérifier notre programme, nous allons lui faire exécuter l'équivalent de la commande shell « `ls -l /dev | grep sda1` » :

```

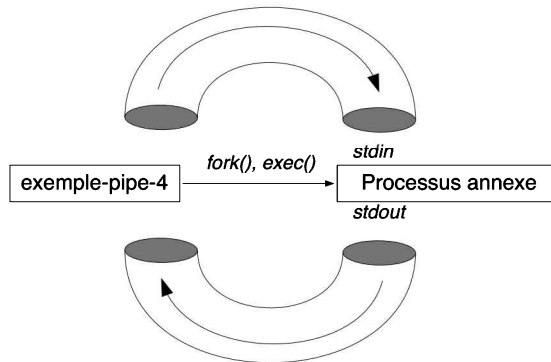
$ ./exemple-pipe-3 "ls -l /dev" "grep sda1"
brw-rw----1 root disk      8,   1 6 août 23:09 sda1
$

```

Nous voyons ainsi un mécanisme de plus employé par les interpréteurs de commande pour implémenter toutes les fonctionnalités qu'ils offrent. Nous allons améliorer encore notre programme en implémentant une possibilité rarement proposée par les shells. Notre processus va rediriger à la fois l'entrée et la sortie standard d'un programme qu'il exécute. Ceci permet d'utiliser une autre application comme une sous-routine du programme. Le principe est un peu semblable à celui de `popen()`, mais cette fonction ne pouvait rediriger que l'entrée ou la sortie du processus appelé, alors que nous allons traiter les deux simultanément. Pour garder un exemple assez simple, nous invoquerons l'utilitaire `wc` qui peut compter le nombre de caractères, de mots ou de lignes dans un fichier de texte. Nous allons lui transmettre le contenu complet d'un fichier sur son entrée standard et lire sa réponse sur sa sortie standard. La figure 23-4 illustre cet exemple.

Figure 23-4

Utilisation d'un processus annexe



```
exemple-pipe-4.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int invoque_processus (const char * commande, int fd[2])
{
    int tube_1[2];
    int tube_2[2];
    if ((pipe(tube_1) != 0) || (pipe(tube_2) != 0))
        return -1;
    switch (fork()) {
        case -1 : // Erreur
            close(tube_1[0]); close(tube_1[1]);
            close(tube_2[0]); close(tube_2[1]);
            return -1;
        case 0 :
            close(tube_1[1]);
            close(tube_2[0]);
            dup2(tube_1[0], STDIN_FILENO);
            dup2(tube_2[1], STDOUT_FILENO);
            system(commande);
            exit(EXIT_SUCCESS);
        default :
            close(tube_1[0]);
            close(tube_2[1]);
            fd[0] = tube_2[0];
            fd[1] = tube_1[1];
    }
    return 0;
}

int main (int argc, char * argv[])
{
    int tube[2];
    FILE * fichier;
    char * contenu;
    struct stat status;
    char c;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (stat(argv[1], & status) != 0) {
        perror("stat");
        exit(EXIT_FAILURE);
    }
    if ((contenu = malloc(status.st_size)) == NULL) {
```



```
    perror("malloc");
    exit(EXIT_FAILURE);
}
if ((fichier = fopen(argv[1], "r")) == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
if (fread(contenu, 1, status.st_size, fichier) != status.st_size){
    perror("fread");
    exit(EXIT_FAILURE);
}
fclose(fichier);

if (invoke_processus("wc -w", tube) != 0) {
    perror("invoke_processus");
    exit(EXIT_FAILURE);
}
write(tube[1], contenu, status.st_size);
close(tube[1]);

fprintf(stdout, "Nombre de mots : ");
while (read(tube[0], & c, 1) == 1)
    fputc(c, stdout);
close(tube[0]);

if (invoke_processus("wc -l", tube) != 0) {
    perror("invoke_processus");
    exit(EXIT_FAILURE);
}
write(tube[1], contenu, status.st_size);
close(tube[1]);

fprintf(stdout, "Nombre de lignes : ");
while (read(tube[0], & c, 1) == 1)
    fputc(c, stdout);
close(tube[0]);

free(contenu);
return EXIT_SUCCESS;
}
```

Notre programme appelle successivement « wc -w » pour avoir le nombre de mots, et « wc -l » pour le nombre de lignes.

```
$ ./exemple-pipe-4 exemple-pipe-4.c
Nombre de mots : 211
Nombre de lignes : 90
$
```

On peut noter que nous avons pris soin, lorsque nous avons fini d'écrire toutes nos données dans le tube, de fermer cette extrémité. À ce moment, en effet, le noyau voit qu'il n'y a plus de processus disposant d'un descripteur sur l'entrée du tube puisque nous avons fermé également la copie de ce descripteur dans le processus fils. Dès qu'un processus tentera de lire à nouveau dans le tube, comme le fait l'utilitaire `wc`, le noyau lui renverra une erreur EOF (*End of File* – fin de fichier). Il est donc important de bien refermer l'extrémité du tube qu'on n'utilise pas immédiatement après avoir appelé `fork()`.

Symétriquement, au moment d'une tentative d'écriture dans un tube dont tous les descripteurs de sortie ont été fermés, le processus appelant `write()` reçoit le signal `SIGPIPE`. Nous pouvons vérifier que le programme suivant est bien tué par un signal `SIGPIPE` :

```
exemple-pipe-5.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (void)
{
    int    tube[2];
    char * buffer = "AZERTYUIOP";

    fprintf(stdout, "Creation tube\n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Fermeture sortie\n");
    close(tube[0]);
    fprintf(stdout, "Ecriture dans tube\n");
    if (write(tube[1], buffer, strlen(buffer)) != strlen(buffer)) {
        perror("write");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Fin du programme\n");
    return EXIT_SUCCESS;
}
```

Notre premier indice est l'absence de message en fin de programme.

```
$ ./exemple-pipe-5
Creation tube
Fermeture sortie
Ecriture dans tube
$
```

Nous pouvons alors vérifier – immédiatement après la fin de notre programme – le contenu de la variable spéciale `$?` du shell :

```
$ echo $?  
141  
$
```

Lorsqu'un processus se termine normalement, cette variable contient la valeur renvoyée (par `return` de `main()` par exemple). Lorsque le processus est tué par un signal `N`, la variable `$?` contient la valeur `128+N`. Nous pouvons en déduire que notre programme a été tué par le signal 13 (`141-128`). Pour connaître le nom du signal, faisons appel à la commande `kill` et son option `-l` (*list*) :

```
$ kill -l 13  
PIPE  
$
```

Certains shells affichent directement le message *Broken pipe* quand un processus est tué par `SIGPIPE`. La plupart des shells actuels comme Bash n'affichent plus ce message, jugé difficilement compréhensible. Si nous ignorons le signal `SIGPIPE` ou si nous le capturons, l'appel système `write()` échoue avec l'erreur `EPIPE`. Le programme `exemple-pipe-6` est une copie de `exemple-pipe-5` avec la ligne suivante ajoutée en début de fonction `main()` :

```
signal(SIGPIPE, SIG_IGN);
```

L'exécution montre alors que l'échec se produit à présent dans l'appel `write()`.

```
$ ./exemple-pipe-6  
Creation tube  
Fermeture sortie  
Ecriture dans tube  
write: Broken pipe  
$
```

Nous avons constaté dans nos premiers exemples qu'il est tout à fait possible d'écrire dans un tube sans en lire immédiatement le contenu – à condition qu'un descripteur de sa sortie reste ouvert. Cela signifie donc que le noyau associe une mémoire tampon à chaque tube. Jusqu'au noyau 2.6.10, la capacité de cette mémoire tampon était de 4 Ko (4096), depuis elle est passée à 64 Ko (65536). Nous pouvons le vérifier

en écrivant dans un tube un caractère à la fois, en regardant au bout de combien d'octets l'écriture devient bloquante.

```
exemple-pipe-7.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    int tube[2];
    char c = 'c';
    int i;

    fprintf(stdout, "Creation tube\n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Ecriture dans tube\n");
    for (i = 0; ; i++) {
        fprintf(stdout, "%d octets écrits\n", i);
        if (write(tube[1], &c, 1) != 1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
    }
    return EXIT_SUCCESS;
}
```

Lorsque le buffer est plein, l'écriture reste bloquée, et nous interrompons le processus en appuyant sur *Contrôle-C* :

```
$ uname -r
2.6.33.6-147.2.4.fc13.i686
$ ./exemple-pipe-7
Creation tube
Ecriture dans tube
0 octets écrits
1 octets écrits
2 octets écrits
3 octets écrits
4 octets écrits
[...]
65533 octets écrits
65534 octets écrits
```

```
65535 octets écrits
65536 octets écrits
  (Contrôle-C après blocage)
$
```

La constante `PIPE_BUF` définie dans `<sys/limits.h>` correspond à la taille maximale d'un bloc de données qui puisse être écrit de manière atomique, elle correspond sous Linux à la taille d'une page mémoire, 4 096 octets sur les architectures x86. Lorsque plusieurs processus partagent un même descripteur sur l'entrée du tube, leurs écritures respectives ne seront pas entremêlées si elles ne dépassent pas `PIPE_BUF` octets à la fois. Le noyau garantit dans ce cas l'atomicité du transfert des données vers le buffer – quitte à bloquer l'écriture avant le transfert s'il n'y a pas assez de place dans la zone tampon.

On peut s'interroger sur les informations renvoyées lorsqu'on invoque l'appel système `stat()` sur un descripteur de fichier. Nous pouvons en faire l'expérience.

```
exemple-pipe-8.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main (void)
{
    int tube[2];
    struct stat status;

    fprintf(stdout, "Creation tube\n");
    if (pipe(tube) != 0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    if (fstat(tube[0], & status) != 0) {
        perror("fstat");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Status : ");
    if (S_ISFIFO(status.st_mode))
        fprintf(stdout, "FIFO\n");
    else
        fprintf(stdout, "? \n");
    return EXIT_SUCCESS;
}
```

Finalement le programme est un peu biaisé car nous devinons déjà ce que nous pouvons attendre.

```
$ ./exemple-pipe-8
Creation tube
Status : FIFO
$
```

Le type d'un descripteur de fichier appartenant à un tube est donc FIFO. Les autres membres de la structure `stat` sont vides ou sans signification – hormis les champs `st_uid` et `st_gid` qui représentent l'identité du processus ayant créé le tube.

Les tubes obtenus par l'appel système `pipe()` représentent donc un moyen de communication simple mais très efficace entre des processus différents. Un problème se pose pourtant, car il faut nécessairement que les interlocuteurs aient un ancêtre commun, le processus qui a créé le tube¹. Il n'est pas possible de lancer des programmes indépendants – par exemple un serveur et des clients – et qu'ils établissent un dialogue.

Les tubes nommés

Pour permettre ce genre de communications, le concept de tube a été étendu pour disposer d'un nom dans le système de fichiers, donnant naissance au terme peu esthétique de « tube nommé » (*named pipe*).

Un tube nommé est donc simplement un nœud dans le système de fichiers. Lorsqu'on l'ouvre pour la première fois, le noyau crée un tube de communication en mémoire. Chaque écriture et chaque lecture auront donc lieu dans ce tube, avec les mêmes principes que ceux que nous avons étudiés à la section précédente.

Ce moyen de communication disposant d'une représentation dans le système de fichiers, des processus indépendants peuvent l'employer pour dialoguer, sans qu'ils soient obligés d'être tous lancés par la même application. Les processus peuvent même appartenir à des utilisateurs différents.

Par ailleurs, il est fréquent que plusieurs processus clients écrivent dans le même tube nommé, afin qu'un processus serveur lise les requêtes. Nous écrirons un tel programme plus bas. Ceci est possible aussi avec des tubes simples, mais c'est de plus en plus rare car on préfère dans ce cas créer un canal de communication pour chaque client.

Le nœud du système de fichiers représentant un tube nommé est du type `Fifo` (*first in first out*), dont nous reparlerons dans le chapitre 29. La création d'un tel nœud peut se faire avec la fonction `mkfifo()`, déclarée dans `<sys/stat.h>` :

```
| int mkfifo (const char * nom, mode_t mode);
```

1. Bien entendu, tous les processus ont un ancêtre commun, `init` de PID 1, et la plupart d'entre eux descendent du même shell. Ce qui nous gêne ici c'est que l'ancêtre doit appartenir à la même application.

En Python, il s'agit de `os.mkfifo()`. Cette fonction renvoie 0 si la création a réussi et -1 en cas d'échec, par exemple si le nœud existait déjà. Le mode indiqué en second argument est identique à celui qui est employé dans l'appel système `open()`. En fait, cette fonction de bibliothèque invoque directement l'appel système `mknod()` ainsi :

```
int mkfifo (const char * nom, mode_t mode)
{
    dev_t dev = 0;
    return mknod(nom, mode | S_IFIFO, dev);
}
```

Le troisième argument de `mknod()` est ignoré quand le nœud n'est pas un fichier spécial de périphérique.

On peut aussi employer l'utilitaire `/usr/bin/mkfifo`, qui sert de frontal à cette fonction, avec une option `-m` permettant d'indiquer le mode désiré. Une fois le nœud créé, on peut l'ouvrir avec `open()` – avec les restrictions dues au mode d'accès –, écrire dedans avec `write()`, y lire avec `read()`, et le refermer avec `close()`. La suppression d'un tube nommé se fait avec l'appel système `unlink()`.

Lorsqu'on ouvre un tube nommé en lecture seule, l'appel `open()` reste bloqué jusqu'à ce que le tube soit ouvert en écriture par un autre processus. Parallèlement, une ouverture en écriture seule est bloquante jusqu'à ce que le tube soit ouvert en lecture par un autre processus.

L'ouverture en lecture et écriture n'est pas portable, car même si la plupart des systèmes l'acceptent, SUSv4 précise que ce comportement est indéfini. Pour pouvoir ouvrir simultanément les deux extrémités dans le même processus, on emploie l'attribut `O_NONBLOCK` lors de l'appel système `open()` afin de permettre une ouverture non bloquante. Dans ce cas, une ouverture en lecture seule n'attendra pas qu'un autre processus ouvre le tube en écriture, et inversement. Nous nous trouvons alors dans la même situation que lorsque le correspondant a fermé son extrémité du tube.

Une lecture depuis un tube non ouvert du côté écriture renverra l'erreur `EOF`, et une écriture dans un tube dont la sortie est fermée déclenchera `SIGPIPE`. Il faut savoir que l'attribut `O_NONBLOCK` concerne aussi les lectures ou écritures ultérieures. Les appels système `read()` et `write()` deviennent alors non bloquants, comme nous le verrons dans le chapitre 25.

Une ouverture non bloquante, en lecture seule, renverra toujours 0, alors qu'une ouverture non bloquante en écriture seule déclenchera l'erreur `ENXIO`.

Il est théoriquement possible d'employer `fopen()` pour ouvrir une Fifo. Toutefois, je préfère utiliser systématiquement `open()` renvoyant un descripteur, suivi éventuellement d'un `fdopen()` me fournissant un flux à partir du descripteur obtenu, ceci pour plusieurs raisons.

- Lors d'un `fopen()`, la bibliothèque C invoque `open()`, mais nous ne savons pas toujours avec quels arguments. Nous pouvons examiner les sources de la Glibc mais, en cas de portage sur un autre système, nous n'avons pas nécessairement accès aux sources de la bibliothèque C.
- La fonction `fopen()` ne permet pas, contrairement à `open()`, de demander une ouverture non bloquante, ce qui est souvent indispensable, notamment lorsqu'on veut ouvrir les deux extrémités d'un tube nommé dans le même processus.
- L'appel système `open()` n'autorise pas la création d'un fichier si l'attribut `O_CREAT` n'est pas présent. Au contraire, `fopen()` en mode « w » risque de créer un fichier normal si le nœud de la Fifo n'a pas encore été créé par un autre processus. Non seulement les communications ne fonctionneront pas, mais si nous détruisons le fichier en fin de programme avec `unlink()`, nous avons peu de chances de nous rendre compte du problème sans passer par une session de débogage assez pénible.

Dans le programme suivant nous allons utiliser plusieurs tubes nommés pour faire dialoguer un processus serveur avec plusieurs clients. Le serveur crée un nœud dans le système de fichier et y lit les requêtes des clients. Ce tube dispose d'un nom connu par tous les processus. Pour répondre à la requête d'un client, le serveur doit pouvoir écrire dans un autre tube nommé, spécifique au client. Les requêtes doivent avoir une taille inférieure à la constante `PIPE_BUF`, afin d'être sûr qu'elles ne seront pas mélangées dans le tube d'interrogation. Comme il faut bien donner un travail à faire au serveur vis-à-vis des clients, nous allons simplement lui faire renvoyer une anagramme de la chaîne de caractères transmise dans la requête.

Le principe retenu pour faire fonctionner l'ensemble est le suivant.

- Le serveur crée un nœud dans le système de fichiers, nommé « `anagramme.fifo` ». Il ouvre ensuite ce tube en lecture et en écriture, puis le rouvre sous forme de flux.
- Un client essaye d'ouvrir en écriture seule le tube du serveur. S'il n'existe pas, le processus client se termine. Sinon, le client crée un tube personnel, nommé « `anagramme.<pid>` », afin d'être unique dans notre application.
- Le client envoie au serveur une requête constituée du nom du tube pour la réponse, suivi d'un retour chariot et de la chaîne de caractères dont on désire un anagramme, suivie d'un retour chariot. Le serveur peut alors lire grâce à `fgets()` ces éléments et répondre dans le tube du client.
- Le client ouvre son tube en lecture seule, lit la réponse, l'affiche et se termine, après avoir supprimé avec `unlink()` son tube de réponse.
- Si la chaîne de caractères reçue vaut `FIN`, le serveur se termine également en supprimant le tube d'interrogation.

Plusieurs clients peuvent travailler simultanément avec le serveur, car le noyau nous assure que toute requête dont la taille est inférieure à `PIPE_BUF` sera traitée de

manière atomique. Si le buffer ne dispose pas d'assez de place pour stocker les données, l'appel `write()` attendra qu'il se libère, mais la copie dans le buffer sera accomplie en une seule fois, sans qu'une autre écriture ne puisse interférer.

Nous avons ouvert, dans le serveur, le tube d'interrogation en lecture et écriture. Ceci nous évite de rester bloqués durant l'ouverture en attendant qu'un autre processus soit prêt à écrire, mais on y trouve aussi un second avantage. Si nous ouvrons le tube en lecture seule, à chaque fois que le client se termine, la lecture dans le serveur avec `fgets()` échoue car le noyau déclenche une erreur EOF. En demandant un tube en lecture et écriture, nous évitons cette situation, car il reste toujours au moins un descripteur ouvert sur la sortie. Nous supprimons ainsi un cas d'échec possible.

Ce programme doit juste être considéré comme un exemple simpliste pour démontrer les possibilités des tubes nommés ; il lui manque un grand nombre de vérifications des conditions d'erreur.

```
exemple-serveur.c :
#define _GNU_SOURCE /* Pour strfry() */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

static char * nom_noeud = "anagramme.fifo";

static int repondre (const char * nom_fifo, char * chaine)
{
    FILE * reponse;
    int fd;
    char * anagramme;

    if ((fd = open(nom_fifo, O_WRONLY)) >= 0) {
        reponse = fdopen(fd, "w");
        anagramme = strdup(chaine);
        strfry(anagramme);
        fprintf(reponse, "%s\n", anagramme);
        fclose(reponse);
        free(anagramme);
    }
    if ((strcasecmp(chaine, "FIN") == 0)
        || (strcasecmp(nom_fifo, "FIN") == 0))
        return 1;
    return 0;
}
```

```

int main (void)
{
    FILE * fichier;
    int    fd;
    char   nom_fifo [128];
    char   chaine [128];

    if (mkfifo(nom_noeud, 0644) != 0) {
        fprintf(stderr, "Impossible de creer le noeud Fifo\n");
        exit(EXIT_FAILURE);
    }
    fd = open(nom_noeud, O_RDWR);
    fichier = fdopen(fd, "r");
    while (1) {
        fgets(nom_fifo, 128, fichier);
        if (nom_fifo[strlen(nom_fifo) - 1] == '\n')
            nom_fifo[strlen(nom_fifo) - 1] = '\0';
        fgets(chaine, 128, fichier);
        if (chaine[strlen(chaine) - 1] == '\n')
            chaine[strlen(chaine) - 1] = '\0';

        if (repondre(nom_fifo, chaine) != 0)
            break;
    }
    unlink(nom_noeud);
    return EXIT_SUCCESS;
}

```

Le processus client est construit ainsi :

```

exemple-client.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main (void)
{
    FILE * question;
    FILE * reponse;
    int    fd;
    char   nom_fifo[128];
    char   chaine[128];

```

```

fprintf(stdout, "Chaine a traiter : ");
if (fgets(chaine, 128, stdin) == NULL)
    exit(EXIT_SUCCESS);
sprintf(nom_fifo, "anagramme.%ld", (long)getpid());
if (mkfifo(nom_fifo, 0644) != 0) {
    fprintf(stderr, "Impossible de creer la fifo\n");
    exit(EXIT_FAILURE);
}

if ((fd = open("anagramme.fifo", O_WRONLY)) < 0) {
    fprintf(stderr, "Impossible d'ouvrir la fifo du serveur\n");
    exit(EXIT_FAILURE);
}
question = fdopen(fd, "w");
fprintf(question, "%s\n%s", nom_fifo, chaine);
fclose(question);

fd = open(nom_fifo, O_RDONLY);
reponse = fdopen(fd, "r");

if (fgets(chaine, 128, reponse) != NULL)
    fprintf(stdout, "Reponse = %s\n", chaine);
else
    perror("fgets");
fclose(reponse);
unlink(nom_fifo);
return EXIT_SUCCESS;
}

```

Le processus serveur est lancé en arrière-plan, le shell nous indique quand il se termine avec la ligne `Done exemple-serveur`.

```

$ ./exemple-serveur &
[1] 4950
$ ls -l *.fifo
prw-r--r-- 1 cpb cpb 0 7 août 22:51 anagramme.fifo
$ ./exemple-client
Chaine a traiter : Azertyuiop
Reponse = ytAizepour

$ ./exemple-client
Chaine a traiter : Linux
Reponse = Lnixu

```

```
$ ./exemple-client
Chaine a traiter : fin
Reponse = inf

[1]+  Done                  ./exemple-serveur
$ ls -l *.fifo
ls: impossible d'accéder à *.fifo: Aucun fichier ou dossier de ce type
$
```

On peut remarquer que le type F'ifo est indiqué par le caractère p (*pipe*) dans l'affichage de la commande « `ls -l` ». J'encourage vivement le lecteur à expérimenter différents cas de figure dans les ouvertures des tubes nommés, afin de bien saisir les points de blocage et les moments où `fgets()` échoue car le correspondant a refermé son extrémité du tube. Pour bien suivre l'état des différents processus concernés, on peut regarder régulièrement le contenu du pseudo-répertoire `/proc/<pid>/fd/`, qui contient des liens symboliques décrivant les divers descripteurs employés par un programme.

Conclusion

Nous avons observé ici une méthode permettant de transférer des données d'un processus à un autre. Dans le cas du tube simple, on est limité à des descendants du processus de démarrage de l'application, mais le tube nommé permet d'étendre ce mécanisme en utilisant un nom dans le système de fichiers.

En combinant les tubes et les signaux, on peut très bien parvenir à une bonne communication entre les processus. Toutefois, d'autres mécanismes sont également disponibles, comme nous le verrons dans le prochain chapitre. Nous nous intéresserons aux possibilités de multiplexage d'entrées-sorties dans le chapitre 25, qui sont très utiles dans les communications avec des tubes.

Le principe de fonctionnement des tubes est important car il introduit un concept qu'on développera très largement dans les chapitres 26 et 27 avec les communications réseau.

Communications avec les IPC

Ce qu'on nomme généralement IPC recouvre trois mécanismes de communication entre processus (IPC, *Inter Process Communication*) qui n'ont été que tardivement intégrés dans les standards comme Unix 98, puis par Posix et finalement par SUSv4.

Les IPC proposent trois mécanismes de communication : les files de messages, la mémoire partagée, et les sémaphores – nécessaires pour organiser l'accès aux données en mémoire partagée. Toutefois, il existe deux API différentes pour utiliser ces mécanismes. La première, apparue en 1983 dans la première version d'Unix Système V est parfois un peu compliquée à utiliser. En outre, beaucoup de programmeurs rechignaient à employer cette API – dite « *IPC Système V* », car elle n'est pas fondée sur le concept des descripteurs de fichiers, contrairement aux tubes, aux tubes nommés ou même aux sockets. Une seconde implémentation s'est généralisée au début des années 2000, et fut normalisée par Posix puis SUSv4. De nos jours, cette seconde API, nommée *IPC Posix*, est utilisée fréquemment dans les applications modernes.

On notera dès à présent que les mécanismes IPC peuvent être autorisés ou non lors de la compilation du noyau Linux. Ainsi, si les appels système décrits dans ce chapitre échouent toujours avec l'erreur ENOSYS, il faudra recompiler le noyau pour pouvoir les utiliser. En outre, les IPC Posix ne furent intégralement disponibles sous Linux qu'à partir du noyau 2.6.10.

Nous allons commencer dans ce chapitre par l'étude des IPC Posix, car ils sont performants et faciles à utiliser dans de nouvelles applications. Nous examinerons ensuite rapidement les IPC Système V, car des applications anciennes peuvent continuer à reposer sur cette interface.

Communications avec les IPC Posix

Files de messages

Les files de messages Posix permettent à des processus distincts d'envoyer des données ou d'en recevoir de manière asynchrone. Les données ne sont pas émises à destination d'un processus particulier, mais dans une file de message dans laquelle le correspondant viendra lire des données quand il le désirera.

Ce mécanisme est assez proche des tubes nommés que nous avons étudiés dans le chapitre précédent, toutefois il existe une différence importante : dans un tube nommé, on transmet des octets sous forme de flux, sans que la quantité de données écrite ou lue ne soit prise en considération. Par exemple, un processus peut écrire 500 octets dans un tube, puis 200 octets un peu plus tard. Si un autre processus vient lire 600 octets dans le tube, il les obtiendra en une seule fois, et il restera 100 octets en attente dans le tube.

Avec une file de message, la situation est différente : on écrit ou on lit un message complet (éventuellement de longueur variable). Supposons à nouveau qu'un processus envoie un message de 500 octets, puis un second message de 200 octets. Un autre processus qui vient lire un message dans le tube recevra uniquement les 500 premiers octets. Même s'il a indiqué qu'il était prêt à en recevoir plus. On dit que les frontières des messages sont préservées.

Les fonctions Posix d'accès aux files de messages sont les suivantes, décrites dans `<mqqueue.h>`, et nécessitant la bibliothèque *realtime* à l'édition des liens :

```
mqd_t mq_open (const char * nom, int flags, mode_t mode,
               struct mq_attr* attr);
```

Ces fonctions ne sont pas directement accessibles en Python. Une implémentation spécifique existe avec la classe `Queue` (devenue `queue` dans Python 3), mais elle est plutôt conçue pour des communications entre threads qu'entre processus. Le module `multiprocessing` propose aussi une classe de files de messages – entre processus – mais s'éloigne un peu des appels système C décrits ici. Il existe un module `posix_ipc` (hors de la bibliothèque standard) qui implémente néanmoins des fonctionnalités des files de messages Posix.

Le descripteur d'une file de message (*message queue*) est représenté par le type `mqd_t`. Le nom passé en premier argument est une chaîne de caractères choisie arbitrairement, qui doit commencer obligatoirement par un slash '/' et ne pas comporter d'autres caractères slash. L'argument `flags` correspond à l'argument identique de `open()` (avec `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`, etc...). Les deux derniers arguments ne sont utiles que s'il y a création d'une nouvelle file. L'argument `mode` correspond aux droits d'accès (pour le

propriétaire, le groupe, etc.) pour la file, et la structure `mq_attr` représente ses paramètres (un pointeur `NULL` permettant de prendre les paramètres par défaut du système).

Si deux processus appellent simultanément ou successivement `mq_open()` avec le même argument `nom`, ils accéderont à la même file de message. En cas d'erreur, `mq_open()` renvoie la valeur (`mqd_t`) `-1` et nous trouverons le code d'erreur dans `errno`.

Lorsqu'on n'a plus besoin d'une file, on peut appeler (implicite à la terminaison d'un processus) :

```
| int mq_close (mqd_t mqd);
```

Même après fermeture, une file reste persistante, à la disposition des autres processus. La destruction d'une file de message s'effectue avec

```
| int mq_unlink (const char * nom);
```

Si `mq_unlink()` réussit, elle renvoie zéro et le nom de la file est immédiatement supprimé (elle n'est plus accessible par `mq_open()`). Toutefois, les processus qui ont déjà ouvert cette file peuvent continuer à l'utiliser, son contenu ne sera détruit qu'après le dernier `mq_close()`.

Pour envoyer des données, on emploie :

```
| int mq_send (mqd_t mq, const char * message,  
              size_t lg_message, unsigned int priorite);
```

La priorité du message se trouve dans l'intervalle suivant : `[0, sysconf(_SC_MQ_PRIO_MAX)-1]`, soit `[0, 32767]` avec Linux 2.6. Les messages sont dépilés dans l'ordre décroissant de priorité (messages de priorités zéro en dernier). Si la file est déjà pleine, `mq_send()` reste bloquant jusqu'à ce que la place nécessaire soit libérée. Il existe une version temporisée qui échoue si le message n'a pas pu être mis en file (file pleine) avant un instant donné :

```
| int mq_timedsend (mqd_t mq, const char * message,  
                  size_t lg_message, unsigned int priorite,  
                  const struct timespec * limite);
```

Cette fonction renvoie `-1` et place `ETIMEDOUT` dans `errno` si on a échoué à envoyer le message avant la limite, qui est une heure absolue et non pas une durée.

Voyons un exemple de programme qui envoie dans une file un message passé en argument :

```
| exemple-mq-send.c :  
| #include <fcntl.h>
```

```

#include <mqueue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[])
{
    mqd_t mq;
    int priorite;

    if (argc != 4) {
        fprintf(stderr, "Syntaxe : %s file priorite message\n",
            argv[0]);
        exit(EXIT_FAILURE);
    }
    mq=mq_open(argv[1],O_WRONLY|O_CREAT,0644,NULL);
    if (mq == (mqd_t) -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    if (sscanf(argv[2], "%d", & priorite) != 1) {
        fprintf(stderr, "Priorite invalide : %s\n", argv[2]);
        exit(EXIT_FAILURE);
    }
    if (mq_send(mq, argv[3], strlen(argv[3]), priorite) != 0) {
        perror("mq_send");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}

```

Envoyons quelques messages dans notre file :

```

$ ./exemple-mq-send azerty 10 "Premier message"
azerty: Invalid argument
$ ./exemple-mq-send /azerty 10 "Premier message"
$ ./exemple-mq-send /azerty 20 "Deuxieme message"
$ ./exemple-mq-send /azerty 10 "Troisieme message"
$

```

La réception des données se fait avec :

```

ssize_t mq_receive      (mqd_t mq, char * buffer, size_t lg_max,
                        unsigned int * prio);
ssize_t mq_timedreceive (mqd_t mq, char * buffer, size_t lg_max,
                        unsigned int * prio, const struct timespec * limite);

```


Ces routines renvoient le nombre d'octets reçus, ou -1 si une erreur se produit. Attention, ces fonctions doivent passer un buffer suffisamment grand pour contenir le plus grand message que la file puisse transporter. Cette dimension (8 192 par défaut sous Linux) est accessible – ainsi que les autres attributs – avec :

```
mqd_t mq_getattr (mqd_t mq, struct mq_attr * attr);
mqd_t mq_setattr (mqd_t mq, const struct mq_attr * attr,
                  struct mq_attr * precedente);
```

Les attributs des files de messages sont stockés dans la structure `mq_attr` avec les champs suivants :

Champ	Type	Signification
<code>mq_flags</code>	<code>long</code>	Mode d'accès (0 : normal, <code>O_NONBLOCK</code> : non bloquant).
<code>mq_maxmsg</code>	<code>long</code>	Nombre maximal de messages dans la file.
<code>mq_msgsize</code>	<code>long</code>	Taille maximale d'un message.
<code>mq_curmsgs</code>	<code>long</code>	Nombre de messages actuellement présents.

Voici un exemple de programme de lecture des messages dans la file :

```
exemple-mq-receive.c :
#include <fcntl.h>
#include <queue.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (int argc, char * argv[])
{
    int    n;
    mqd_t mq;
    struct mq_attr attr;
    char * buffer = NULL;
    unsigned int priorite;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s file\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((mq = mq_open(argv[1], O_RDONLY)) == (mqd_t) -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    if (mq_getattr(mq, & attr) != 0) {
        perror("mq_getattr");
    }
```

```
    exit(EXIT_FAILURE);
}
if ((buffer = malloc(attr.mq_msgsize)) == NULL) {
    perror("malloc");
    exit(EXIT_FAILURE);
}
if ((n=mq_receive(mq,buffer,attr.mq_msgsize,& priorite)) < 0){
    perror("mq_receive");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "[%d] %s\n", priorite, buffer);
return EXIT_SUCCESS;
}
```

Utilisons-le, et vérifions que les messages précédemment envoyés sont bien extraits en fonction de leurs priorités :

```
$ ./exemple-mq-receive /azerty
[20] Deuxieme message
$ ./exemple-mq-receive /azerty
[10] Premier message
$ ./exemple-mq-receive /azerty
[10] Troisieme message
$
```

Si nous relançons le programme une quatrième fois, il reste bloqué en attente de message. Enfin, un processus peut demander à être notifié dès l'arrivée d'un message dans une file vide avec :

```
int mq_notify (mqd_t mq, const struct sigevent * notification);
```

La structure `sigevent` sera étudiée dans le prochain chapitre, elle permet de configurer le type de notification désirée : l'arrivée d'un signal – que nous choisissons – ou le démarrage d'un thread du processus.

Mémoire partagée

Les files de messages sont des mécanismes performants de par leur aspect asynchrone. Un processus peut envoyer des requêtes, qui seront extraites et traitées plus tard par un autre processus. En outre, plusieurs processus peuvent écrire dans la même file sans qu'il y ait le moindre risque de collision.

Toutefois, le débit d'une file de message est assez limité. Cela est dû au fait que deux appels système sont nécessaires pour communiquer (`mq_send()` et `mq_receive()`) avec la charge qu'ils imposent au processeur et à la MMU.

Pour communiquer de gros volumes de données entre applications, on préfère généralement se tourner vers un mécanisme beaucoup plus rapide : la mémoire partagée. Chaque processus partageant cette mémoire peut y lire ou y écrire sans appel système particulier, juste comme un accès mémoire ordinaire. En revanche, un certain nombre de précautions devront être prises pour organiser correctement les accès concurrents.

Le principe de la mémoire partagée consiste à ouvrir un segment de mémoire avec l'appel système `shm_open()` et à le projeter dans l'espace mémoire du processus avec `mmap()`.

```
| int shm_open (const char *nom, int flags, mode_t mode);
```

Cette fonction n'est pas directement accessible en Python (comme pour les files de messages, une implémentation existe dans le module indépendant `posix_ipc`), mais nous en verrons un contournement plus loin.

Le principe est le même qu'avec `mq_open()`, on utilise un nom sous forme de chaîne de caractères (comportant un seul slash en tête de nom) pour identifier le segment, et on indique le type d'ouverture `O_RDONLY`, `O_RDWR`, `O_WRONLY`, avec éventuellement l'attribut `O_CREAT` pour créer le segment s'il n'existe pas (auquel cas le troisième argument indique les autorisations d'accès pour le propriétaire, le groupe, etc.).

Le descripteur de segment renvoyé est du même type que les descripteurs de fichiers usuels. Lorsque nous n'aurons plus besoin d'accéder à la mémoire partagée, on pourra invoquer `close(descripteur)`.

Un second appel système spécifique permet de détruire un segment de mémoire partagée. Cela s'effectue au moyen de `shm_unlink()`, le segment ne sera plus accessible par son nom, mais les processus l'ayant déjà projeté pourront continuer à l'utiliser :

```
| int shm_unlink (const char *nom);
```

Lors de la création d'un segment – par exemple, avec `shm_open(nom, O_RDWR|O_CREAT, 0600)` – nous ne précisons pas sa taille. Pour ce faire, nous allons utiliser le descripteur qui est renvoyé, et appeler `ftruncate()`. Cet appel système sert généralement à tronquer la taille d'un fichier à une longueur donnée.

```
| int ftruncate (int fd, off_t longueur);
```

Toutefois, si le fichier est plus petit que cette longueur, il est étendu (avec des zéros) jusqu'à la limite précisée. Nous allons utiliser ainsi `ftruncate()` pour dimensionner

initialement le segment. Si nous rappelons à nouveau `ftruncate()` par la suite, l'appel système n'aura aucun effet, car le segment aura déjà la bonne longueur.

Une fois le segment dimensionné correctement, nous allons le projeter en mémoire avec `mmap()`, que nous avons déjà rencontré dans le chapitre 15.

Voici un exemple de programme qui projette une zone de mémoire de la taille d'un entier, et l'utiliser comme compteur, incrémenté toutes les secondes.

```
exemple-shm.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char * argv[])
{
    int    fd;
    int * compteur;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s nom_segment\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = shm_open(argv[1], O_RDWR | O_CREAT, 0600)) == -1) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    if (ftruncate(fd, sizeof(int)) != 0) {
        perror("ftruncate");
        exit(EXIT_FAILURE);
    }
    compteur = mmap(NULL, sizeof(int), PROT_READ|PROT_WRITE,
                    MAP_SHARED, fd, 0);
    if (compteur == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    while (1) {
        (* compteur) ++;
        fprintf(stdout, "compteur = %d\n", (* compteur));
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Lorsque nous lançons un exemplaire de ce programme, il compte toutes les secondes :

```
$ ./exemple-shm /abcdefg
compteur = 1
compteur = 2
compteur = 3
compteur = 4
  (Contrôle-C)
$
```

Relançons-le après l'avoir arrêté, le comptage continue :

```
$ ./exemple-shm /abcdefg
compteur = 5

compteur = 6
compteur = 7
compteur = 8
  (Contrôle-C)
$
```

La mémoire est persistante (jusqu'au *reboot* du système).

Lançons à présent deux exemplaires sur deux terminaux :

```
$ ./exemple-shm /abcdefg
compteur = 9
compteur = 10

compteur = 12

compteur = 14

compteur = 16

compteur = 18
  (Contrôle-C)

$ ./exemple-shm /abcdefg
compteur = 11

compteur = 13

compteur = 15

compteur = 17

compteur = 19
compteur = 20
  (Contrôle-C)
$
```

Les deux comptages sont entremêlés, preuve que les deux processus indépendants accèdent à la même variable.

Sous Linux, l'implémentation de la mémoire partagée consiste à projeter avec `mmap()` un fichier se trouvant dans un système de fichiers particulier nommé `tmpfs`. Il s'agit d'une arborescence créée en mémoire RAM et dimensionnée selon les besoins. Suivant les distributions, ce système de fichiers sera monté à différents endroits de l'arborescence (`/dev/shm`, `/run/shm`, `/var/shm`, etc.).

```
$ mount | grep tmpfs
none on /sys/fs/cgroup type tmpfs (rw)
udev on /dev type devtmpfs (rw,mode=0755)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
none on /run/lock type tmpfs (rw,noexec,nosuid,nodev,size=5242880)
none on /run/shm type tmpfs (rw,nosuid,nodev)
none on /run/user type tmpfs (rw,noexec,nosuid,nodev,size=104857600,mode=0755)
```

Nous voyons sur cette distribution Ubuntu, par exemple, qu'un répertoire `/run/shm` existe. Vérifions son contenu :

```
$ ls -l /run/shm/
total 580
-rw----- 1 cpb   cpb       4 mars 8 11:54 abcdefg
[...]
```

Nous retrouvons bien notre zone de mémoire partagée précédente. Il suffirait donc, en Python, de l'ouvrir avec `os.open()` et de la projeter dans notre mémoire avec `mmap.mmap()` pour obtenir le même résultat que précédemment.

Pour obtenir l'entier contenu dans la zone projetée, le plus facile est d'utiliser la fonction `struct.unpack()` qui permet de préciser un format (ici, « *i* » pour *integer* 32 bits) et une zone mémoire d'où extraire un tuple de valeurs (une unique valeur dans notre cas).

Pour écrire dans la zone, la fonction `struct.pack_into()` réalise l'opération inverse. Le programme ci-après attend en argument un fichier de mémoire partagée avec son chemin complet.

```
exemple-shm.py :
#!/usr/bin/python
from __future__ import print_function
import mmap
```

```
import os
import struct
import sys
import time

if len(sys.argv) < 2:
    print("usage:", sys.argv[0], "shm-file")
    sys.exit(1)
try:
    stat = os.stat(sys.argv[1])
    size = stat.st_size
    fd = os.open(sys.argv[1], os.O_RDWR)
    mapping = mmap.mmap(fd, size, mmap.MAP_SHARED, mmap.PROT_READ |
mmap.PROT_WRITE)
except:
    print("Wrong file :", sys.argv[1])
    sys.exit(1)

while True:
    (counter,) = struct.unpack("i", mapping[:])
    counter = counter + 1
    struct.pack_into("i", mapping, 0, counter)
    print(counter)
    time.sleep(1)
```

L'exécution est comparable au programme précédent en C.

```
$ ./exemple-shm.py /run/shm/abcdefg
21
22
23
24
[...]
```

Notez bien qu'il n'y a ici aucun mécanisme évitant les erreurs d'accès concurrents. Ceci est dû au fait que l'incrémentement du compteur n'est pas une opération atomique, même en C. Supposons que l'opération `(* compteur)++` se déroule en trois étapes : lecture de la valeur `(* compteur)` depuis la mémoire et stockage dans un registre du processeur, incrémentation du registre, réécriture de ce registre en mémoire. Deux processus qui exécuteraient cette opération simultanément pourraient aboutir à :

- le compteur est initialisé avec une valeur, 10 par exemple ;
- le processus 1 lit dans le registre la valeur 10 ;
- le processus 1 incrémente le registre qui passe à 11 ;

- sur un autre processeur (ou un autre cœur) un processus 2 lit la valeur 10 ;
- le processus 2 incrémente le registre qui passe à 11 ;
- le processus 1 réécrit la valeur 11 depuis son registre vers le compteur ;
- le processus 2 réécrit la valeur 11 depuis son registre vers le compteur.

Deux processus ont incrémenté le compteur, pourtant celui-ci n'a augmenté que d'une unité !

Pour éviter ce genre de souci il est nécessaire, à chaque fois que l'on utilise un partage de mémoire, de synchroniser les accès. Dans les applications multithreads, nous avons vu que le système mettait à notre disposition des mutex. Pour les communications multiprocessus, on utilise plutôt les sémaphores.

Sémaphores

Les sémaphores Posix sont parfois appelés sémaphores temps réel, car ils ont été décrits dans l'extension 1B de Posix qui décrivait toutes les fonctionnalités permettant un fonctionnement temps réel. Ils permettent de synchroniser des tâches – threads ou processus – et d'organiser l'accès aux ressources partagées.

Un sémaphore est dans sa forme la plus simple un drapeau qui peut être levé ou baissé. Il sert à contrôler l'accès à une ressource critique grâce à deux opérations :

- Avant l'accès, un processus attend que le drapeau soit levé, puis il le baisse.
- Après avoir utilisé la ressource protégée, le processus relève le drapeau, et le noyau réveille les autres processus bloqués dans l'opération précédente.

Le test qui intervient dans la première de ces opérations est atomiquement lié à la modification qui le suit. Cela garantit qu'en aucun cas deux processus ne verront simultanément le drapeau baissé et se l'attribueront.

Ces deux opérations sont généralement notées $P()$ et $V()$ en abréviation des traductions des termes *tester* et *incrémenter* en hollandais, langue natale de l'inventeur des sémaphores, Edsger Dijkstra.

Pour étendre la portée des sémaphores, on leur adjoint un compteur, qui représente le nombre d'accès simultanés autorisés pour les ressources qu'ils protègent. Le plus souvent, ce compteur est initialisé à 1, ce qui conduit à un sémaphore à exclusion mutuelle, équivalent à un mutex. Toutefois, on peut imaginer un système de calcul parallèle, où l'on désire limiter le nombre de threads actifs simultanément à 8 pour des raisons de consommation CPU. Un sémaphore est donc initialisé avec la valeur 8. À chaque entrée dans la boucle de calcul, on insère une opération $P()$ sur le sémaphore, qui va décrémenter le compteur. Lorsque le compteur atteindra zéro, le drapeau du sémaphore sera baissé, et les opérations $P()$ suivantes resteront bloquées. Symétriquement,

lorsqu'un thread sort de la boucle de calcul, il appelle `V()` qui incrémente le compteur, et permet donc à un nouveau thread d'accéder à la zone de calcul.

Les sémaphores Posix sont des variables de type `sem_t`, déclarés dans `<semaphore.h>`. Comme tous les éléments de l'API Posix.1b, ils sont implémentés dans la bibliothèque `rt` (*real-time*), et nécessitent donc l'utilisation de l'option `-lrt` de `gcc` au moment de l'édition des liens, en outre avec les versions récentes de la bibliothèque C, il faut également utiliser l'option `-pthread`.

Il existe deux types de sémaphores Posix : les sémaphores *anonymes* et les sémaphores *nommés*. Un sémaphore anonyme est placé dans une zone de mémoire accessible par les tâches qui vont y accéder (threads ou processus). Dans le cas d'un partage entre processus, on le place sur un segment de mémoire partagée obtenu avec l'appel `shm_open()`. Le sémaphore anonyme est initialisé avec `sem_init()`, puis supprimé après usage avec `sem_destroy()` :

```
int sem_init (sem_t * sem, int partage, unsigned int valeur);  
int sem_destroy (sem_t * sem);
```

Le second argument de `sem_init()` est un booléen indiquant si le sémaphore va être partagé entre différents processus ou non, le troisième paramètre est la valeur initiale du compteur.

Il existe plusieurs implémentations de sémaphores en Python, dont celle proposée par les modules `threading` ou `multiprocessing`, mais il n'y a que le module indépendant `posix_ipc` qui implémente les sémaphores nommés traités dans ce chapitre.

Un sémaphore *nommé* dispose d'un nom qui permet à n'importe quel processus d'y accéder sans passer par la projection d'un segment de mémoire partagée. Le sémaphore est créé et ouvert par `sem_open()`, et détruit par `sem_unlink()`. La fonction `sem_open()` accepte deux prototypes. Le premier est utilisé lorsqu'il y a création du sémaphore nommé (quand l'attribut `O_CREAT` est présent dans le second paramètre), auquel cas il faut préciser également les permissions d'accès pour les autres utilisateurs, et la valeur initiale du compteur. Le second prototype est utilisé lorsqu'on accède à un sémaphore nommé déjà créé :

```
sem_t * sem_open (const char * nom, int flags,  
                 mode_t permissions, unsigned int valeur);  
sem_t * sem_open (const char * nom, int flags);  
int      sem_unlink (const char * nom);
```

Comme toujours, `sem_unlink()` efface le nom du sémaphore qui ne peut donc plus être ouvert avec `sem_open()`, mais les processus qui l'avaient déjà ouvert peuvent continuer à l'utiliser normalement.

L'utilisation des sémaphores proprement dite est très simple, on appelle `sem_wait()` pour prendre le sémaphore – opération P() – ou l'une de ses variantes `sem_trywait()` et `sem_timedwait()` respectivement non bloquante et temporisée.

```
int sem_wait    (sem_t * sem);
int sem_trywait (sem_t * sem);
int sem_timedwait (sem_t * sem, const struct timespec * limite);
```

Attention, la limite est une heure absolue, pas un délai.

Pour rendre le sémaphore – opération V() – on utilise la fonction `sem_post()` qui incrémente le compteur du sémaphore, et dans le cas où le compteur devient positif à ce moment, réveille une autre tâche en attente :

```
int sem_post (sem_t * sem);
```

Voici un petit exemple de sémaphore nommé où plusieurs processus vont prendre le même sémaphore pendant quelques secondes avant de le libérer. On commence par essayer une ouverture simple et si elle échoue, on passe à une tentative de création du sémaphore :

```
exemple-semaphore.c :
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char * argv[])
{
    int i;
    sem_t * sem;

    if (argc != 2) {
        fprintf(stderr, "usage: %s nom_semaphore\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    sem = sem_open(argv[1], O_RDWR);
    if (sem == SEM_FAILED) {
        if (errno != ENOENT) {
            perror(argv[1]);
            exit(EXIT_FAILURE);
        }
        sem = sem_open(argv[1], O_RDWR | O_CREAT, 0666, 1);
```

```

    if (sem == SEM_FAILED) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "[%d] Creation de %s\n", getpid(), argv[1]);
}
for (i = 0; i < 3; i++) {
    fprintf(stderr, "[%d] en attente...\n", getpid());
    sem_wait(sem);
    fprintf(stderr, "    [%d] tient le semaphore\n", getpid());
    sleep(4);
    fprintf(stderr, "    [%d] lache le semaphore\n", getpid());
    sem_post(sem);
    sleep(2);
}
return EXIT_SUCCESS;
}

```

Voici un exemple de deux exécutions en parallèle sur le même terminal, le préfixe entre crochets indique le numéro de processus et nous permet de distinguer les traces.

```

$ ./exemple-semaphore /azerty & ./exemple-semaphore /azerty
[1] 8617
[8617] Creation de /azerty
[8617] en attente...
    [8617] tient le semaphore
[8618] en attente...
    [8617] lache le semaphore
    [8618] tient le semaphore
[8617] en attente...
    [8618] lache le semaphore
    [8617] tient le semaphore
[8618] en attente...
    [8617] lache le semaphore
    [8618] tient le semaphore
[8617] en attente...
    [8618] lache le semaphore
    [8618] tient le semaphore
    [8618] lache le semaphore
[1]+ Done ./exemple-semaphore /azerty
$

```

Administration des ressources existantes

Les mécanismes IPC Posix que nous avons observés sont très efficaces et plutôt faciles à utiliser dans une application. Toutefois, un problème se pose pour l'administration du système : notre application précédente a créé un sémaphore nommé, mais n'a pas appelé `sem_unlink()`. Le sémaphore restera donc persistant après la fin de l'exécution du processus, jusqu'au *reboot* du système, ce qui peut poser des problèmes d'encombrement mémoire et de déni de service.

Linux propose une solution pour que l'administrateur puisse accéder aux ressources IPC créées : une représentation dans un système de fichiers virtuel nommé `tmpfs`. Ce système de fichiers est habituellement monté sur `/dev/shm` :

```
$ mount
/dev/sda5 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw)
[...]
$
```

Dans le répertoire `/dev/shm` nous allons trouver :

```
$ ls -l /dev/shm/
total 20
-rw----- 1 cpb cpb          4 23 août 09:00 abcdefg
-r----- 1 gdm gdm 67108904 18 août 12:34 pulse-shm-3000747417
-rw-rw-r-- 1 cpb cpb        16 23 août 10:32 sem.azerty
$
```

Nous remarquons la présence de `abcdefg`, notre segment de mémoire partagée, créé avec `exemple-shm` et `sem.azerty`, le sémaphore `azerty` que nous venons d'utiliser.

Il est possible de supprimer un sémaphore ou une zone de mémoire partagée avec `rm`, de la renommer avec `mv`, d'en faire une copie avec `cp`, et même d'accéder au contenu d'une zone mémoire avec `cat` par exemple ou `read()/write()`.

En ce qui concerne les files de messages, le système de fichiers virtuel est nommé `mqueue`, mais n'est pas toujours monté par défaut. On emploie alors :

```
$ su
Mot de passe :
```

```
# mkdir /dev/mqueues
# mount none /dev/mqueues/ -t mqueue
# exit
exit
$ ls -l /dev/mqueues/
total 0
-rw-r--r-- 1 cpb cpb 80 25 août 11:16 azerty
$
```

La file de message que nous avons créée en début de chapitre avec `exemple-mq-send` est bien présente dans ce répertoire.

L'accès aux ressources IPC par le biais d'un système de fichiers virtuel peut être très utile, autant pour des tâches d'administration, que pour l'automatisation de certaines opérations via des scripts shell (sauvegarde/restauration d'une image d'un segment de mémoire, etc.). Nous encourageons le lecteur à expérimenter ces possibilités en utilisant les exemples précédents, et en manipulant le contenu des systèmes de fichiers virtuels.

Les mécanismes IPC Système V

Les IPC Système V recouvrent les mêmes fonctionnalités que précédemment, avec une interface de programmation différente. On utilise une notion de clé, que chaque processus demande et qui permet ensuite l'accès à la ressource partagée désirée.

Obtention d'une clé

Chaque ressource (file de message, segment mémoire, sémaphores) est identifiée par un numéro, commun aux processus désirant l'utiliser. Pour partager ce numéro d'identification, un processus peut demander au système de créer la ressource de manière privée, puis transmettre directement le numéro – par l'intermédiaire d'un fichier par exemple – aux autres processus avec lesquels il veut communiquer. Ce schéma est souvent utilisé si un processus ne désire communiquer qu'avec ses descendants, car il crée alors la ressource IPC privée avant d'invoquer `fork()`. Il est aussi possible de se servir, lors de la demande de création de ressource, d'une clé qui permettra au système d'identifier l'IPC désiré. Cette clé peut être commune à plusieurs processus, qui se mettent d'accord pour employer une valeur figée, un peu à la manière des numéros de ports lors des connexions réseau. Il faut alors documenter proprement l'application pour bien indiquer les clés qu'elle utilise.

Une dernière possibilité consiste à demander au système de créer lui-même une clé, fondée sur des références communes pour tous les processus. La clé est constituée en employant un nom de fichier et un identificateur de projet. De cette manière, tous les processus d'un ensemble donné pourront choisir de créer leur clé commune en utilisant le chemin d'accès du fichier exécutable de l'application principale, ainsi qu'un numéro de version par exemple.

Une clé est fournie par le système sous forme d'un objet de type `key_t`, défini dans `<sys/type.h>`. La constante symbolique `IPC_PRIVATE`, définie dans `<sys/ipc.h>` représente une clé privée, demandant sans condition la création d'une nouvelle ressource IPC, comme dans le premier schéma que nous avons imaginé. Pour créer une nouvelle clé à partir d'un nom de fichier et d'un identificateur de projet, on emploie la fonction `ftok()` (*file to key*), déclarée ainsi dans `<sys/ipc.h>` :

```
| key_t ftok (char * nom_fichier, char projet);
```

La clé créée emploie une partie du numéro d'i-nœud du fichier indiqué, le numéro mineur du périphérique sur lequel il se trouve et la valeur transmise en second argument pour créer une clé sur 32 bits. La fonction `ftok()` ne garantit pas réellement l'unicité de la clé, car plusieurs liens matériels sur le même fichier renvoient le même numéro d'i-nœud. De plus, la restriction au numéro mineur de périphérique ainsi que l'utilisation seulement des 16 bits de poids faibles de l'i-nœud rendent possible – quoique très improbable – l'existence de fichiers différents renvoyant la même clé.

Une fois qu'on a obtenu une clé ou qu'on a choisi d'utiliser une ressource privée avec `IPC_PRIVATE`, on demande l'accès à l'IPC proprement dite.

Ouverture de l'IPC

L'obtention de la ressource IPC se fait à l'aide de l'une des trois commandes `msgget()`, `shmget()` et `semget()`. Les détails d'appel seront précisés plus bas, mais ces fonctions demandent au système de créer éventuellement la ressource si elle n'existe pas, puis de renvoyer un numéro d'identification. Si la ressource existe déjà et si le processus appelant n'a pas les autorisations nécessaires pour y accéder, les routines échouent en renvoyant -1.

À partir de l'identifiant ainsi obtenu, il sera possible respectivement :

- d'envoyer et de recevoir des messages dans une file, à l'aide des fonctions `msgsnd()`, `rtmsget()` ;
- d'attacher puis de détacher un segment de mémoire partagée dans l'espace d'adressage du processus avec `shmat()` ou `shmdt()` ;
- de lever de manière bloquante ou non un sémaphore, puis de le relâcher avec la fonction commune `semop()`.

Il faut bien comprendre que l'emploi de `IPC_PRIVATE` dans `msgget()`, `shmget()` ou `semget()` n'empêche pas l'accès à la ressource par un autre processus, mais garantit uniquement qu'une nouvelle ressource sera créée. En effet, l'identifiant renvoyé par la routine d'ouverture n'aura rien d'exceptionnel et un autre processus pourra très bien l'employer – à condition bien sûr d'avoir les autorisations d'accès nécessaires.

Contrôle et paramétrage

Les IPC proposent quelques options de paramétrage spécifiques au type de communication, ou générales. Pour cela, il existe trois fonctions, `msgctl()`, `shmctl()` et `semctl()`, qui permettent de consulter des attributs regroupés dans des structures `msqid_ds`¹, `shmid_ds` et `semid_ds`. Dans tous les cas, ces structures permettent l'accès à un objet de type `struct ipc_perm`, défini ainsi dans `<sys/ipc.h>` :

Nom	Type	Signification
<code>__key</code>	<code>key_t</code>	Clé associée à la ressource IPC.
<code>__seq</code>	<code>unsigned short</code>	Numéro de séquence, utilisé de manière interne par le système, à ne pas toucher.
<code>mode</code>	<code>unsigned short</code>	Autorisations d'accès à la ressource, comme pour les permissions des fichiers.
<code>uid</code>	<code>uid_t</code>	UID effectif de la ressource IPC.
<code>gid</code>	<code>gid_t</code>	GID effectif de la ressource IPC.
<code>cuid</code>	<code>uid_t</code>	UID du créateur de la ressource.
<code>cgid</code>	<code>gid_t</code>	GID du créateur de la ressource.

Ces informations permettent bien entendu de contrôler l'accès à la ressource IPC. Les modifications de mode ne peuvent être réalisées que par le propriétaire, le créateur de la ressource ou par un processus ayant la capacité `CAP_IPC_OWNER`. Les fonctions de contrôle permettent également de détruire une ressource IPC.

Files de messages

Les files de messages sont des listes chaînées gérées par le noyau pour contenir des données organisées sous forme d'un type suivi d'un bloc de message proprement dit. Cette représentation complique un peu la manipulation des messages, mais permet – grâce au type transmis – de les hiérarchiser par priorité ou d'obtenir un multiplexage en distinguant plusieurs processus destinataires différents lisant la même file de messages.

1. Attention, il s'agit bien de `msqid_ds` – message queue identifier data structure – et non de `msgid_ds` comme on pourrait s'y attendre.

Le noyau gère un maximum de MSGMNI files indépendantes chacune pouvant comporter des messages de tailles inférieures à MSGMAX, soit 8 192 octets. Pour accéder à une file existante ou en créer une nouvelle, on appelle `msgget()`, déclarée ainsi dans `<sys/msg.h>` :

```
int msgget (key_t key, int attribut);
```

Cette routine renvoie l'identificateur de la file de messages demandée, ou -1 en cas d'erreur. Le premier argument doit comprendre une clé caractérisant la file désirée à l'aide de la fonction `ftok()` ou `IPC_PRIVATE`.

Le second argument est une composition binaire des constantes suivantes :

Nom	Signification
IPC_CREAT	Créer une nouvelle file s'il n'y en a aucune présentement associée à la clé transmise en premier argument.
IPC_EXCL	Toujours créer une nouvelle file. La fonction <code>msgget()</code> échouera si une file existe déjà avec la clé indiquée.

Dans cet argument s'insèrent également les permissions d'accès à la file créée, avec le même format que le dernier argument de `open()`.

Attention à ne pas oublier d'introduire ces autorisations d'accès lors de la création, sinon la file aura les permissions 000, ce qui la rend pour le moins difficile à utiliser !

Une fois que la file est créée, on peut – sous réserve d'avoir les autorisations adéquates – y écrire des messages. Un message est une zone de mémoire contiguë contenant un entier `long` représentant le type du message, suivi des données proprement dites. Le type du message, qui doit être supérieur à zéro, est simplement une description interne à l'application, qui n'a pas de signification pour le noyau. On l'emploiera pour filtrer les messages à l'arrivée.

La méthode la plus simple est donc de définir une structure regroupant le type du message et les données qu'on veut envoyer :

```
typedef struct {  
    // Type pour msgsnd() et msgrcv()  
    long    type;  
    // Données de l'application  
    char    identifiant[25];  
    double  x;  
    double  y;  
};
```



```
double vitesse;  
time_t estimation_arrivee;  
} mon_message_t;
```

La structure envoyée ne peut naturellement pas comprendre de pointeurs puisqu'ils n'auraient aucune signification dans l'espace d'adressage du processus récepteur. La transmission d'une chaîne de caractères doit autant que possible se faire en employant une zone allouée automatiquement dans la structure, comme le champ identifiant de notre exemple précédent. La transmission de données allouées dynamiquement est assez compliquée puisqu'elle nécessite la réservation d'un espace de la dimension d'un `long int` avant la zone véritablement utilisée.

L'envoi d'un message se fait par l'intermédiaire de l'appel système `msgsnd()`, déclaré ainsi :

```
int msgsnd (int file, const void * message, int taille,  
            int attributs);
```

Le numéro de file indiqué doit avoir été fourni préalablement par `msgget()`, le second argument pointe sur le message tel que nous venons de le décrire, et le troisième argument indique la longueur utile du message sans compter son type. En dernière position, le seul attribut qu'on puisse transmettre éventuellement est `IPC_NOWAIT`, pour que l'appel système ne soit pas bloquant. Sinon, s'il n'y a pas assez de place dans la file pour stocker le message, `msgsnd()` reste en attente.

La valeur de retour de `msgsnd()` est zéro si tout s'est bien passé, et -1 sinon, `errno` contenant un code d'erreur décrivant le problème. Le message est copié dans la file, aussi il est possible d'écraser les données originales dès le retour de `msgsnd()`.

La lecture dans une file de messages se fait en invoquant l'appel système `msgrcv()`, déclaré ainsi :

```
int msgrcv (int file, void * message, int taille,  
            long type, int attributs);
```

Les premiers arguments ont la même signification que dans `msgsnd()`, le troisième indiquant la taille maximale de la zone de données du message à lire, qui doit donc être disponible en seconde position. On emploie généralement `MSGMAX`, qui correspond à la taille maximale d'un message sur le système, ou une dimension fixée si tous les messages sont constitués autour de la même structure.

Attention, la taille ne prend pas en compte le type du message, il s'agit uniquement de la zone utile du message.

Le type indiqué en quatrième position permet de sélectionner les messages qu'on désire recevoir. Le comportement de `msgrcv()` varie en fonction de cette valeur.

- Un type nul indique qu'on veut recevoir le prochain message disponible dans la file. C'est le comportement habituel d'une file de messages, où on traite les données dans l'ordre d'arrivée.
- Un type positif permet de réclamer le premier message dudit type disponible dans la file. Cette méthode donne la possibilité de multiplexer plusieurs processus en écriture et plusieurs en lecture sur la même file. Chacun d'eux utilise un identifiant unique – par exemple, son PID –, et la file permet à n'importe quel processus d'envoyer un message vers un destinataire précis.
- Un type négatif sert à réclamer le premier message disponible ayant le plus petit type inférieur ou égal à la valeur absolue de ce quatrième argument. Il est ainsi possible d'introduire des priorités entre les messages. Le message de type 1 sera délivré avant tous les autres, même s'ils sont en attente depuis plus longtemps.

Le dernier argument peut contenir un *OU* binaire avec les constantes suivantes :

Nom	Signification
IPC_NOWAIT	Ne pas rester en attente si aucun message du type réclamé n'est disponible, mais au contraire échouer avec l'erreur <code>ENOMSG</code> .
MSG_EXCEPT	Réclame un message de n'importe quel type, sauf celui qui est indiqué en quatrième argument, qui doit être nécessairement strictement positif.
MSG_NOERROR	Si le message extrait est trop long, il sera tronqué sans que l'erreur <code>E2BIG</code> se produise, contrairement au comportement par défaut.

Lorsqu'elle réussit, cette fonction renvoie le nombre d'octets du message – non compris son type – et -1 lorsqu'elle échoue. Dans ce cas, `errno` contient le code d'erreur.

Finalement, le contrôle et le paramétrage d'une file de messages se font à l'aide de la fonction `msgctl()`. Celle-ci est déclarée ainsi :

```
| int msgctl (int file, int commande, struct msqid_ds * attributs);
```

Il y a trois commandes possibles, qu'on passe en second argument :

- `IPC_STAT` : pour obtenir les paramètres concernant la file de messages et les stocker dans la structure `msqid_ds` passée en dernière position.
- `IPC_SET` : pour configurer certains paramètres en utilisant la structure passée en troisième argument. Il faut que le processus appelant soit le propriétaire ou le créateur de la file de messages, ou qu'il ait la capacité `CAP_SYS_ADMIN`.
- `IPC_RMID` : pour supprimer la file de messages. Tous les processus en attente de lecture ou d'écriture sur la file seront réveillés. Les opérations ultérieures d'accès à

cette file échoueront. Ce manque de fiabilité est l'un des arguments employés par les détracteurs des IPC Système V.

La structure `msgid_ds` est définie dans `<sys/msg.h>`. Ses membres susceptibles de nous intéresser sont :

Nom	Type	Signification
<code>msg_perm</code>	<code>struct ipc_perm</code>	Autorisations d'accès à la file de messages.
<code>msg_stime</code>	<code>time_t</code>	Heure du dernier <code>msgsnd()</code> sur la file.
<code>msg_rtime</code>	<code>time_t</code>	Heure du dernier <code>msgrcv()</code> sur la file.
<code>msg_ctime</code>	<code>time_t</code>	Heure du dernier paramétrage de la file.
<code>msg_qnum</code>	<code>unsigned short</code>	Nombre de messages actuellement présents dans la file.
<code>msg_qbytes</code>	<code>unsigned short</code>	Taille maximale en octets de la file.
<code>msg_lspid</code>	<code>pid_t</code>	PID du processus ayant effectué le dernier <code>msgsnd()</code> .
<code>msg_lrpid</code>	<code>pid_t</code>	PID du processus ayant effectué le dernier <code>msgrcv()</code> .

Nous allons construire trois petits programmes servant d'interface en ligne de commande pour `msgsnd()`, `msgrcv()` et `msgctl()` avec la commande `IPC_RMID`. Le premier prend en arguments le nom d'un fichier servant pour créer la clé IPC, une valeur indiquant le type du message, et une chaîne de caractères composant le corps du message émis.

Des programmes d'illustration de `msgsnd()`, `msgrcv()`, et `msgctl()` sont présents dans les sources des exemples de ce livre. Je ne les détaille pas ici, car l'utilisation des IPC Système V est de moins en moins répandue de nos jours.

Étant donné que les files d'attente, comme les autres ressources IPC par ailleurs, sont persistantes jusqu'à leur suppression par `msgctl()` ou par l'arrêt du système, il existe deux utilitaires d'administration, `/usr/bin/ipcs` et `/usr/bin/ipcrm`, permettant de les manipuler. La première commande affiche la liste de toutes les ressources IPC en cours d'utilisation sur le système, avec des informations sur le propriétaire, l'espace occupé, etc. Le second utilitaire permet de supprimer une ressource – à condition d'en avoir le droit – en indiquant le type de ressource et son identifiant, fourni par `ipcs`.

Mémoire partagée

Le système de la mémoire partagée offert par les IPC Système V est aussi efficace que celui des IPC Posix, mais ne permet pas d'accéder au contenu du segment comme s'il s'agissait d'un fichier.

- Une fonction `shmget()` permet à partir d'une clé `key_t` d'obtenir l'identifiant d'un segment de mémoire partagée existant ou d'en créer un au besoin.
- L'appel système `shmat()` permet d'attacher le segment dans l'espace d'adressage du processus.
- La fonction `shmdt()` sert à détacher le segment si on ne l'utilise plus.
- Enfin, l'appel système `shmctl()` permet de paramétrer ou de supprimer un segment partagé.

Les prototypes de ces routines sont déclarés dans `<sys/shm.h>` ainsi :

```
int    shmget (key_t key, int taille, int attributs);
char * shmat(int identifiant, char * adresse, int attributs);
int    shmdt(char * adresse);
int    shmctl (int identifiant, int commande,
              struct shmid_ds * attributs);
```

L'appel système `shmget()` fonctionne comme `msgget()`, en employant la clé transmise en premier argument pour rechercher ou créer un bloc de mémoire partagée. Les attributs indiqués en dernière position comportent les 9 bits de poids faibles de l'autorisation d'accès, et éventuellement les constantes `IPC_CREAT` et `IPC_EXCL` qui ont les mêmes significations qu'avec les files de messages. Les erreurs renvoyées par cette fonction sont équivalentes à celles de `msgget()`.

Le second argument de cette routine est la taille du segment désiré, en octets. Cette taille sert lors de la création d'une nouvelle zone de mémoire partagée. La valeur indiquée est arrondie au multiple supérieur de la taille des pages mémoire sur le système (4 Ko sur un PC). Si la taille demandée lors de la création est inférieure à la valeur `SHMMIN` ou supérieure à `SHMMAX`, une erreur se produit. Pour accéder à une zone mémoire déjà existante, il faut demander une valeur inférieure ou égale à la taille effective du segment. On emploie généralement zéro dans ce cas, car le système ne réduit pas la taille de la projection d'un segment existant.

Une fois obtenu l'identifiant d'un segment partagé, on doit l'attacher dans l'espace mémoire du processus à l'aide de la fonction `shmat()`. On indique en second argument l'adresse désirée pour l'attachement. Si cette adresse est nulle, le noyau recherche un emplacement libre dans l'espace d'adressage du processus, y réalise la projection, et l'appel système `shmat()` renvoie l'adresse du premier octet de la zone partagée. C'est bien entendu le mécanisme qu'on utilisera toujours. Le fait de mentionner explicitement une adresse d'attachement ne se justifie que dans des cas exceptionnels (par exemple des émulateurs ou des débogueurs) ne nous concernant pas ici. Mentionnons quand même que l'adresse transmise dans ce cas doit être alignée sur une frontière de page, ou alors il faut passer l'attribut `SHM_RND` dans le dernier argument pour demander au noyau d'arrondir l'adresse indiquée à la limite de page inférieure.

L'attachement peut être réalisé en lecture seule si l'attribut `SHM_RDONLY` est passé en troisième argument de `shmat()`, sinon la projection est réalisée en lecture et écriture.

La fonction `shmctl()` permet, à la manière de `msgctl()`, d'agir sur un segment partagé. La commande employée en seconde position peut être :

- `IPC_STAT` : pour remplir la structure `shmid_ds` que nous allons détailler ci-après.
- `IPC_SET` : pour modifier l'appartenance ou les autorisations d'accès au segment.
- `IPC_RMID` : pour supprimer le segment. Ce dernier est alors marqué comme « prêt pour la suppression », mais ne sera effectivement détruit qu'une fois qu'il aura été détaché par le dernier processus qui l'utilise. Cela signifie aussi que tant qu'un processus conserve le segment attaché, il est toujours possible de le lier à nouveau avec `shmat()`, même s'il a été marqué pour la destruction.
- `SHM_LOCK` : permet de verrouiller le segment en mémoire pour s'assurer qu'il ne sera pas envoyé sur le périphérique de *swap*. Nous avons déjà étudié ce mécanisme dans le chapitre 15 avec l'appel système `mlock()`. Cette opération réduisant la mémoire vive disponible pour les autres processus, elle est privilégiée et nécessite un UID nul ou la capacité `CAP_IPC_LOCK`.
- `SHM_UNLOCK` : permet symétriquement de déverrouiller une page de la mémoire, autorisant à nouveau son transfert en mémoire secondaire.

La structure `shmid_ds` contenant les paramètres associés au segment de mémoire partagée comprend notamment les membres suivants :

Nom	Type	Signification
<code>shm_perm</code>	<code>struct ipc_perm</code>	Autorisation d'accès au segment de mémoire.
<code>shm_segsz</code>	<code>size_t</code>	Taille en octets du segment.
<code>shm_atime</code>	<code>time_t</code>	Heure du dernier attachement.
<code>shm_dtime</code>	<code>time_t</code>	Heure du dernier détachement.
<code>shm_ctime</code>	<code>time_t</code>	Heure de la dernière modification des autorisations.
<code>shm_cpid</code>	<code>pid_t</code>	PID du processus créateur du segment.
<code>shm_lpid</code>	<code>pid_t</code>	PID du processus ayant réalisé la dernière intervention.
<code>shm_nattch</code>	<code>unsigned short</code>	Nombre actuel d'attachements en mémoire.

Sémaphores

Le principe des sémaphores Système V est quasiment le même que celui des sémaphores Posix, mais les fonctions sont totalement différentes. Ici, nous manipulons des ensembles de sémaphores. Il est possible de demander en une fois des opérations `P()` ou `V()` indépendantes sur chaque sémaphore d'un ensemble. Ces opérations sont

liées atomiquement, ce qui signifie que le noyau les réalisera toutes ou n'en exécutera aucune. Il peut aussi rester bloqué longtemps en attente d'une ressource.

Finalement, étant donné qu'un processus peut se terminer à tout moment – notamment à cause d'un signal –, il est important de relâcher automatiquement les sémaphores qu'il maintenait. Pour cela, il existe un mécanisme d'annulation programmable pour chaque action. Au moment de la fin du processus, le noyau effectue l'opération inverse de celle qui a été réalisée. Mais cela complique à nouveau l'utilisation des sémaphores.

Les routines servant à manipuler les sémaphores sont `semget()`, qui accomplit une tâche comparable à `msgget()` ou à `shmget()`, `semop()` qui regroupe les opérations `P()` et `V()`, et `semctl()`, qui permet, entre autres, de configurer ou de supprimer un ensemble de sémaphores. Leurs prototypes sont déclarés dans `<sys/sem.h>` ainsi :

```
int semget (key_t key, int nombre, int attributs);
int semop(int identifiant, struct sembuf * operation,
          unsigned nombre);
int semctl (int identifiant, int numero,
            int commande, union semun attributs);
```

L'appel système `semget()` fonctionne comme ses confrères `msgget()` et `shmget()`, avec simplement en second argument le nombre de sémaphores dans l'ensemble. Cette valeur n'est prise en compte que lors de la création de la ressource, pas au moment de l'accès à un ensemble existant. Le troisième argument peut contenir comme d'habitude `IPC_CREAT`, `IPC_EXCL` et les autorisations d'accès.

La routine `semop()` sert à la fois pour les opérations `P()` et `V()` sur de multiples sémaphores appartenant au jeu indiqué en premier argument. Chaque opération est décrite par une structure `sembuf`, définie dans `<sys/sem.h>` ainsi :

Nom	Type	Signification
<code>sem_num</code>	<code>short int</code>	Numéro du sémaphore concerné dans l'ensemble. La numérotation débute à zéro.
<code>sem_op</code>	<code>short int</code>	Valeur numérique correspondant à l'opération à réaliser.
<code>sem_flg</code>	<code>short int</code>	Attributs pour l'opération.

L'opération effectuée est déterminée ainsi.

- Lorsque le champ `sem_op` d'une structure `sembuf` est strictement positif, le noyau incrémente le compteur interne associé au sémaphore de la valeur indiquée et réveille les processus en attente.
- Lorsque le champ `sem_op` est strictement négatif, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit supérieur à `sem_op`, puis il décrémente le compteur de cette valeur avant de continuer l'exécution du processus.

- Lorsque le champ `sem_op` est nul, le noyau endort le processus jusqu'à ce que le compteur associé au sémaphore soit nul, puis il continue l'exécution du programme. Cette fonctionnalité permet de synchroniser les processus.

Il existe deux options possibles pour le membre `sem_flg` :

- `IPC_NOWAIT` : l'opération ne sera pas bloquante, même si le champ `sem_op` est négatif ou nul, mais l'appel système indiquera l'erreur `EAGAIN` dans `errno` si l'opération n'est pas réalisable.
- `SEM_UNDO` : pour être sûr que le sémaphore retrouvera un état correct, même en cas d'arrêt intempestif du programme, le noyau va mémoriser l'opération inverse de celle qui a été réalisée et l'effectuera automatiquement à la fin du processus. Nous allons préciser ce mécanisme plus loin.

En fait, la routine `semop()` prend en second argument une table de structures `sembuf`. Le nombre d'éléments dans cette table est indiqué en dernière position. Le noyau garantit que les opérations seront atomiquement liées, ce qui signifie qu'elles seront toutes réalisées ou qu'aucune ne le sera. Bien entendu, il suffit qu'une seule opération avec `sem_op` négatif ou nul échoue avec l'attribut `IPC_NOWAIT` pour que toutes les modifications soient annulées.

La plupart des applications ne manipulant qu'un seul sémaphore à la fois, on redéfinit souvent des opérations `P()` et `V()` semblables au `sem_wait()` et `sem_post()` que nous avons étudiées précédemment :

```
int P (int identifiant)
{
    struct sembuf buffer;
    buffer.sem_num = 0;
    buffer.sem_op = -1;
    buffer.sem_flg = IPC_UNDO;
    return semop(identifiant, & buffer, 1);
}
```

```
int V (int identifiant)
{
    struct sembuf buffer;
    buffer.sem_num = 0;
    buffer.sem_op = 1;
    buffer.sem_flg = IPC_UNDO;
    return semop(identifiant, & buffer, 1);
}
```

L'option `SEM_UNDO` employée lors d'une opération permet au processus de s'assurer qu'en cas de terminaison imprévue alors qu'il bloque un sémaphore, le noyau en restituera l'état initial. Cela est réalisé en utilisant un compteur par sémaphore et par pro-

cessus qui a demandé un accès à l'ensemble. Ce mécanisme est donc – légèrement – coûteux en mémoire. Le noyau modifie l'état de ce compteur à chaque opération sur le processus en y inscrivant l'opération inverse. Lorsque le processus se termine, le noyau ajoute le compteur d'annulation à celui du sémaphore. Si le processus a bien libéré le sémaphore, le compteur d'annulation est nul, et rien ne se passe. Si, en revanche, le processus s'est terminé après avoir effectué `P()`, mais sans avoir réalisé `V()`, le compteur d'annulation vaut 1 et le noyau libère ainsi automatiquement le sémaphore.

Une question peut se poser dans le cas inverse, si le noyau doit décrémenter le compteur du sémaphore lors de la fin d'un processus : doit-il attendre que le compteur du sémaphore soit supérieur à celui d'annulation, au risque de bloquer indéfiniment ? La réponse est loin d'être évidente. L'implémentation actuelle sous Linux consiste à diminuer immédiatement le compteur, mais à limiter ce dernier à zéro. D'autres systèmes peuvent préférer bloquer indéfiniment – à la manière d'un processus zombie qui attend la lecture de son code de retour pour disparaître entièrement – pour garantir l'annulation de n'importe quelle opération.

La fonction `semctl()` permet de consulter ou de modifier le paramétrage d'un jeu de sémaphore, mais également de fixer l'état du compteur. Cette routine utilise traditionnellement en dernier argument une union définie ainsi :

```
union semun {  
    int          valeur;  
    struct semid_ds * buffer;  
    unsigned short int * table;  
}
```

En fait, cette union n'est pas définie dans les fichiers d'en-tête système, elle doit être déclarée manuellement dans le programme utilisateur. En réalité, le prototype de `semctl()`, vu par le compilateur, est en substance le suivant :

```
int semctl (int identifiant, int numero, int commande, ...);
```

Les points d'élision en fin de liste indiquent la présence éventuelle d'un argument supplémentaire, dont le type n'est pas mentionné. On peut donc transmettre n'importe quel type de donnée, c'est la commande indiquée en troisième position qui déterminera la conversion. Pour garder une certaine homogénéité aux appels `semctl()`, on préfère généralement regrouper les diverses possibilités dans une union, qui permet quand même une vérification minimale.

En fonction de la commande, le numéro indiqué en seconde position et l'union en dernier argument auront donc des rôles différents :

Commande	Signification
IPC_STAT	Remplir le membre <code>buffer</code> de l'union <code>semun</code> avec le paramétrage de l'ensemble de sémaphores. Le second argument de <code>semctl()</code> est ignoré.
IPC_SET	Utiliser le membre <code>buffer</code> de l'union <code>semun</code> pour paramétrer les autorisations d'accès sur l'ensemble de sémaphores. Le second argument de <code>semctl()</code> est ignoré.
IPC_RMID	Supprimer l'ensemble de sémaphores. Tous les processus en attente sont réveillés et peuvent recevoir l'erreur <code>EIDRM</code> . Les second et quatrième arguments sont ignorés.
GETALL	Recopier la valeur de tous les sémaphores de l'ensemble dans le membre <code>table</code> de la structure <code>semun</code> . Cette table doit être correctement dimensionnée avec un <code>unsigned short</code> par sémaphore. Le second argument est ignoré.
SETALL	Fixer les compteurs de tous les sémaphores de l'ensemble avec les valeurs contenues dans le membre <code>table</code> de la structure <code>semun</code> . Les processus en attente sur un sémaphore sont réveillés si son compteur augmente. La table doit être correctement dimensionnée avec un <code>unsigned short</code> par sémaphore. Le second argument est ignoré.
GETVAL	Lire la valeur du sémaphore dont le numéro est indiqué dans le second argument de <code>semctl()</code> . Cette valeur est renvoyée, tandis que le quatrième argument est ignoré.
SETVAL	Fixer la valeur du sémaphore dont le numéro est indiqué dans le second argument en employant le contenu du membre <code>valeur</code> de l'union <code>semun</code> . Les processus en attente sont réveillés au besoin.
GETNCNT	Renvoyer le nombre de processus en attente d'augmentation du compteur du sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.
GETZCNT	Renvoyer le nombre de processus en attente d'annulation du compteur du sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.
GETPID	Renvoyer le PID du processus ayant réalisé la dernière opération sur le sémaphore dont le numéro est indiqué en second argument. Le quatrième argument est ignoré.

La structure `semid_ds` qui représente le paramétrage d'un jeu de sémaphores contient notamment les membres suivants :

Nom	Type	Signification
<code>sem_perm</code>	<code>struct ipc_perm</code>	Autorisations d'accès à l'ensemble de sémaphores.
<code>sem_otime</code>	<code>time_t</code>	Heure de la dernière opération <code>semop()</code> .
<code>sem_ctime</code>	<code>time_t</code>	Heure de la dernière modification de <code>sem_perm</code> .
<code>sem_nsems</code>	<code>unsigned short</code>	Nombre de sémaphores dans l'ensemble.

Lorsqu'un ensemble de sémaphores est créé, les compteurs sont initialement vides. Aucun processus ne peut donc se les attribuer. Il faut donc leur donner une valeur

initiale à l'aide de la commande SETALL. En général, on vérifie auparavant si le jeu existe déjà ainsi :

```
if ((sem = semget(cle, nb_sem, 0)) == -1) && (errno == ENOENT)) {
    // L'ensemble n'existe pas
    if ((sem=semget(cle, nb_sem,IPC_CREAT|IPC_EXCL|0600))== -1) {
        // Pas assez de mémoire par exemple
        perror("semget");
        exit(EXIT_FAILURE);
    }
    // Maintenant qu'il est créé, initialiser les compteurs
    for (i = 0; i < nb_sem; i ++){
        table_sem[i]= 1;
        semun.table = table_sem;
        if (semctl(sem, 0, SETALL, semun) < 0)
            perror("semctl");
    }
    // L'ensemble peut à présent être utilisé
```

Un dernier mot pour préciser l'utilisation de l'opération `semop()` avec une valeur `sem_op` nulle. Cela sert, nous l'avons dit, à synchroniser des processus. En fait, on emploie surtout cette opération pour exécuter `N` processus, puis pour attendre qu'ils aient tous atteint un point particulier de leur déroulement avant de les laisser continuer. Pour cette technique de rendez-vous, on utilise un sémaphore qu'on initialise à la valeur `N`. Chaque processus exécute sa première partie, puis arrivé au point de rendez-vous dans son code, il invoque `P()` pour décrémenter la valeur du compteur, suivi d'une opération d'attente avec `sem_op` nulle. Lorsque tous les processus seront arrivés à leur point de rencontre respectifs, le compteur aura été diminué de `N`, et atteindra donc zéro. Ils pourront alors continuer leur exécution, notamment en appelant `V()` pour restaurer le compteur du sémaphore. En résumé on a :

```
// Création d'un sémaphore
if ((sem = semget(cle, 1, IPC_CREAT|IPC_EXCL|0600)) == -1) {
    perror("semget");
    exit(EXIT_FAILURE);
}
// Initialisation
table_sem[0]= N;
semun.table = table_sem;
if (semctl(sem, 0, SETALL, semun) < 0)
    perror("semctl");
}

// Départ des processus
... fork() ...
```

```

// Point de rendez-vous
// P()
sembuf.sem_num = 0;
sembuf.sem_op = -1;
sembuf.sem_flg = 0;
semop(sem, & sembuf, 1);
// Attente
sembuf.sem_op = 0;
semop(sem, & sembuf, 1);
// Rendez-vous Ok
// V()
sembuf.sem_op = 1;
semop(sem, & sembuf, 1);

// Suite

```

Ce procédé fonctionne, car le noyau réveille tous les processus en attente quand le compteur arrive à zéro, avant que les opérations `V()` suivantes ne modifient à nouveau le sémaphore.

Nous allons observer un petit exemple de compteur placé sur un segment partagé, dont l'accès est correctement protégé par un sémaphore :

```

exemple-shmat.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/types.h>

typedef union semun {
    int          val;
    struct semid_ds * buffer;
    unsigned short int * table;
} semun_t;

int main (int argc, char * argv[])
{
    key_t          key;
    int            sem;
    int            shm;
    struct sembuf sembuf;
    semun_t        u_semun;
    int *          compteur = NULL;
    unsigned short table [1] {1};
}

```

```
if (argc != 2) {
    fprintf(stderr, "Syntaxe : %s fichier_cle \n", argv[0]);
    exit(EXIT_FAILURE);
}
if ((key = ftok(argv[1], 0)) == -1) {
    perror("ftok");
    exit(EXIT_FAILURE);
}
if ((shm = shmget(key, sizeof(int), IPC_CREAT | 0600)) == -1) {
    perror("shmget");
    exit(EXIT_FAILURE);
}
if ((compteur = shmat(shm, NULL, 0)) == (void *)-1) {
    perror("shmat");
    exit(EXIT_FAILURE);
}
if ((sem = semget(key, 1, 0)) == -1) {
    if ((sem = semget(key, 1, IPC_CREAT | IPC_EXCL | 0600)) == -1) {
        perror("semget");
        exit(EXIT_FAILURE);
    }
    u_semun.table = table;
    if (semctl(sem, 0, SETALL, u_semun) < 0)
        perror("semctl");
}
sembuf.sem_num = 0;
sembuf.sem_flg = SEM_UNDO;
while (1) {
    // V()
    sembuf.sem_op = -1;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    // Incrémentation du compteur
    (* compteur) ++;
    fprintf(stdout, "%d\n", *compteur);
    // P()
    sembuf.sem_op = 1;
    if (semop(sem, & sembuf, 1) < 0) {
        perror("semop");
        exit(EXIT_FAILURE);
    }
    sleep(1);
}
return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution sur deux terminaux :

```
$ ./exemple-shmat ./exemple-shmat.c
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
(Contrôle-C)
$
```

Conclusion

Nous l'avons déjà dit, les IPC Système V font souvent office de parents pauvres dans les fonctionnalités standard disponibles sur les machines Unix. Reconnaissons que leur interface de programmation est plutôt contraignante, et qu'ils n'entrent aucunement dans le schéma Unix classique utilisant des descripteurs de fichiers.

Cela a été corrigé avec les IPC Posix, que je conseille d'utiliser systématiquement pour les nouvelles applications. Notons que les zones de mémoires partagées, correctement protégées par des sémaphores, représentent un mécanisme extrêmement rapide et puissant pour la communication entre applications indépendantes.

L'implémentation de ces mécanismes en Python est assez spécifique en ce qui concerne la bibliothèque standard, mais on peut trouver une assez bonne compatibilité avec les fonctions du langage C grâce au module `posix_ipc`.

Entrées-sorties avancées

Nous allons nous intéresser dans ce chapitre à plusieurs mécanismes différents permettant d'améliorer le fonctionnement des entrées-sorties. Nous allons d'abord observer comment permettre aux appels système `read()` et `write()` de ne plus être bloquants, même s'ils ne peuvent pas accomplir leurs tâches immédiatement. Nous étudierons ensuite le multiplexage des entrées-sorties. Cette technique, très utilisée dans les communications entre processus et dans la programmation réseau, permet d'attendre simultanément sur plusieurs canaux l'arrivée de données ou la libération d'un descripteur en écriture.

Enfin, nous examinerons une technique qui peut être très performante dans certaines circonstances, reposant sur des entrées-sorties asynchrones par rapport au déroulement du programme. Bien entendu, un certain nombre de précautions devront être prises avec ce procédé.

Entrées-sorties non bloquantes

Nous avons déjà vu dans le chapitre 23 comment employer l'attribut `O_NONBLOCK` lors de l'ouverture d'un descripteur, afin d'éviter de rester bloqué, même si ce descripteur ne permet pas d'effectuer les opérations demandées. Ceci est très utile avec les tubes de communication et devient même indispensable avec certaines liaisons série.

Cependant nous n'avons pas pu obtenir du système qu'il nous laisse effectuer des appels système `read()` ou `write()` non bloquants sur des descripteurs obtenus autrement qu'avec `open()`, comme un tube ou des entrées-sorties standard. Ceci est

malgré tout nécessaire dans certains cas, pour vérifier si des données sont arrivées depuis un descripteur correspondant par exemple au clavier, tout en continuant de mettre à jour régulièrement des informations à l'écran. Notons que cet exemple n'est peut-être pas très judicieux car le clavier, comme nous l'avons déjà remarqué dans le chapitre 20, ne permet pas dans sa configuration par défaut de capturer des caractères au vol sans appuyer sur la touche « *Entrée* ». Nous verrons dans le chapitre 32 comment résoudre ce problème.

Quoi qu'il en soit, nous aimerions disposer de la faculté de lire les données sur un descripteur, sans que cela nous bloque si rien n'est disponible, ou d'écrire dans un tube avec une fonction qui nous signale simplement une erreur si le tube est plein. Ceci est possible en modifiant le comportement du descripteur grâce à l'appel système `fcntl()` que nous avons déjà rencontré dans le chapitre 22. Rappelons que son prototype est défini dans `<fcntl.h>` ainsi :

```
| int fcntl (int fd, int commande, ...);
```

Ici la commande employée est celle qui modifie l'attribut du fichier `F_SETFL`, et nous insérons dans le troisième argument l'attribut `O_NONBLOCK`. Dans notre premier exemple, nous allons reprendre le principe d'un programme que nous avons utilisé dans le chapitre 23, en remplissant entièrement un tube. En revanche, nous basculons le descripteur de l'entrée du tube en écriture non bloquante. Ainsi, les appels `write()` après la saturation du buffer (de 65536 octets depuis Linux 2.6.10) échoueront sans rester bloqués. Lorsqu'une tentative d'écriture échoue, on endort le processus pendant une seconde pour éviter de consommer inutilement du temps CPU.

```
| exemple-nonblock-1.c :  
| #include <fcntl.h>  
| #include <stdio.h>  
| #include <stdlib.h>  
| #include <unistd.h>  
  
| int main (void)  
| {  
|     int tube[2];  
|     char c = 'c';  
|     int i;  
  
|     if (pipe(tube) != 0) {  
|         perror("pipe");  
|         exit(EXIT_FAILURE);  
|     }  
|     fcntl(tube[1], F_SETFL, O_NONBLOCK);  
|     i = 0;
```



```

while (1) {
    if (write(tube[1], & c, 1) != 1) {
        perror("write");
        sleep(1);
    } else
        i ++;
    fprintf(stdout, "%d octets écrits\n", i);
}
return EXIT_SUCCESS;
}

```

L'exécution illustre bien le comportement attendu. On arrête le programme en interrompant sa boucle avec *Contrôle-C*.

```

$ ./exemple-nonblock-1
1 octets écrits
2 octets écrits
3 octets écrits
[...]
65534 octets écrits
65535 octets écrits
65536 octets écrits
write: Resource temporarily unavailable
65536 octets écrits
write: Resource temporarily unavailable
(Contrôle-C)
$

```

Le message « *Resource temporarily unavailable* » correspond à l'erreur EAGAIN, qui indique que l'écriture est momentanément impossible en attendant que le buffer soit vidé par la sortie du tube.

Le script suivant se comporte de la même manière :

```

exemple-nonblock-1.py :
#!/usr/bin/python
from __future__ import print_function
import os
import fcntl

(fd_r, fd_w) = os.pipe()
n = 0
fcntl.fcntl(fd_w, fcntl.F_SETFL, os.O_NONBLOCK)
while True:
    try:
        os.write(fd_w, 'c')

```

```
        n = n +1
        print("Written bytes:", n)
except Exception as e:
    print(str(e))
```

Dans ces programmes, nous avons pris la liberté d'écrire en une seule opération l'ensemble des attributs du descripteur en y indiquant uniquement `O_NONBLOCK`. Le nombre d'attributs étant très restreint (`O_NONBLOCK`, `O_APPEND`, `O_SYNC`), cela peut être acceptable pour des petits exemples tels que présentés ici. Toutefois, dans une application plus conséquente et portable, il serait préférable de lire d'abord la configuration du descripteur avec la commande `F_GETFL` de `fcntl()`, avant de la modifier en ajoutant l'attribut `O_NONBLOCK` puis de la réécrire avec `F_SETFL`.

Dans notre second exemple, nous allons examiner la lecture non bloquante. Un processus va se scinder en deux, le processus fils assurant une écriture toutes les 700 millisecondes dans un tube. Le père essaye de lire toutes les 100 millisecondes, aussi certaines lectures réussissent-elles quand des données sont disponibles, tandis que d'autres échouent.

```
exemple-nonblock-2.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    int tube[2];
    char c;

    if (pipe(tube) !=0) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    switch (fork()) {
        case -1 :
            perror("fork");
            exit(EXIT_FAILURE);
        case 0 : // Fils : écriture
            close(tube[0]);
            while (1) {
                write(tube[1], & c, 1);
                usleep(700000);
            }
    }
```

```
default : // Père : lecture
close(tube[1]);
fcntl(tube[0], F_SETFL, O_NONBLOCK);
while (1) {
    if (read(tube[0], & c, 1) == 1)
        printf("Ok\n");
    else
        printf("Non\n");
    usleep(100000);
}
return EXIT_SUCCESS;
}
```

Le programme assure bien une tentative régulière de lecture sans se laisser perturber par ses échecs ni par ses succès.

```
$ ./exemple-nonblock-2
Non
Ok
Non
Non
Non
Non
Non
Ok
Non
Non
Non
Non
Non
Non
Ok
Non
(Contrôle-C)
$
```

Le même comportement est obtenu avec le script Python suivant :

```
exemple-nonblock-2.py :
#!/usr/bin/python
from __future__ import print_function
import os
import fcntl
import time
```

```

(fd_r, fd_w) = os.pipe()

if os.fork() == 0:
    while True:
        os.write(fd_w, 'c')
        time.sleep(0.7)
else:
    fcntl.fcntl(fd_r, fcntl.F_SETFL, os.O_NONBLOCK)
    while True:
        try:
            r = os.read(fd_r, 1)
            print("Ok")
        except:
            print("Error")
            time.sleep(0.1)

```

Nous n'avons pas affiché de message d'erreur complet, mais la fonction `read()` renvoie aussi le code `EAGAIN` dans `errno` quand une lecture ne donne rien sur un descripteur non bloquant. Le comportement des opérations `read()` et `write()` non bloquantes est le suivant.

Appel	Situation du descripteur	Résultat
read()	Aucune donnée disponible.	retour : zéro errno = EAGAIN.
	Moins de données disponibles que le nombre demandé.	retour : quantité lue.
	Autant ou plus de données disponibles que le nombre demandé.	retour : quantité demandée.
write()	Pas de place.	retour : zéro errno = EAGAIN.
	Pas assez de place pour toute la quantité à écrire.	retour : quantité écrite.
	Suffisamment de place pour toute l'écriture.	retour : quantité demandée.

À cela s'ajoutent bien entendu les valeurs de retour inférieures à zéro (normalement -1) qui correspondent à des erreurs d'entrée-sortie de bas niveau. Dans nos programmes précédents, nous écrivions un seul caractère à la fois, mais en réalité on doit généralement écrire quelque chose comme :

```

int retour;
retour = read(fd, buffer, taille);
if (retour == -1) {
    perror("read");
}

```

```
    exit(EXIT_FAILURE);  
}  
if (retour!= 0)  
    traite_donnees_dans_buffer(buffer, taille);
```

Il faut noter que le fait d'ouvrir un descripteur avec l'attribut `O_NONBLOCK` est parfois indispensable, alors que nous voudrions par la suite que les lectures et écritures soient bloquantes. C'est le cas par exemple lors de l'ouverture des deux extrémités d'un tube nommé dans le même processus. Dans cette situation, on utilise alors un arrangement comme celui-ci :

```
fd = open(nom_du_fichier, O_RDONLY|O_NONBLOCK);  
if (fd >= 0) {  
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) & (~O_NONBLOCK));  
    ...  
}
```

Avec les lectures et écritures non bloquantes, nous pouvons déjà écrire des applications avec un comportement assez dynamique, dont le déroulement continue imperperturbablement, que des données arrivent ou non. Voici par exemple un squelette de jeu dans lequel on lit le clavier de manière non bloquante :

```
int main (void)  
{  
    char touche;  
    fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);  
    while (1) {  
        if (calcul_des_nouveaux_evenements() == FIN_PARTIE)  
            break;  
        if (read(STDIN_FILENO, & touche, 1) == 1)  
            calcul_deplacement_joueur(touche);  
        affichage_nouvelle_situation();  
        usleep(UN_25EME_DE_SECONDE);  
    }  
    affichage_du_score();  
    return EXIT_SUCCESS;  
}
```

De même, les écritures non bloquantes peuvent être très utiles lorsque l'émission de données vers un autre processus est une fonctionnalité annexe ne devant en aucun cas freiner le cours de l'application principale. Ceci concerne par exemple les sorties de journalisation d'un système temps réel – sauf bien entendu lorsque cette journalisation est considérée comme un mécanisme de boîte noire dont l'importance est cruciale pour le suivi de l'application.

Dans le cas d'un processus serveur ayant des connexions par tube – ou socket réseau – avec de multiples clients, on pourrait être tenté d'écrire une boucle principale comme :

```
while (1) {  
    for (i = 0; i < nombre_de_clients; i ++)  
        if (read(tube_depuis_client[i], & requete,  
                sizeof(requete_t)) == sizeof(requete_t))  
            repondre_a_la_requete(i, & requete);  
}
```

Toutefois ce code serait très mauvais, car il accomplirait la plupart du temps des boucles vides, consommant inutilement et exagérément du temps processeur. La seule situation où on pourrait tolérer ce genre de comportement serait dans des portions courtes de logiciel temps réel, où on attend simultanément des messages sur plusieurs canaux de communication, et quand la réponse doit être fournie dans un délai ne tolérant pas le risque que le processus soit endormi temporairement. Hormis ce cas très particulier, on se tournera plutôt vers le mécanisme d'attente passive permettant un multiplexage des entrées.

Multiplexage d'entrées-sorties

Attente d'événements – Multiplexage d'entrées

On a souvent besoin, principalement dans les applications de serveur réseau, de surveiller l'arrivée de données en provenance de multiples sources. Mais étant donné que toutes les entrées-sorties ont lieu sous le contrôle du noyau, il est généralement inutile d'effectuer des boucles d'attente active comme nous l'avons vu à la section précédente. L'appel système `select()` – apparu en 1982 dans BSD 4.2 – et l'appel `poll()` – intégré à Système V R 3 en 1986 – permettent de dire en substance au noyau : « Voici la liste des descripteurs qui m'intéressent, préviens-moi s'il se passe quelque chose, en attendant je fais un petit somme. » L'application relâche entièrement le processeur, au bénéfice des autres programmes. Quand des données arrivent, quel que soit le type de descripteur, elles passent par le noyau, qui se souvient alors qu'un processus est en attente et peut le réveiller en lui indiquant que des informations sont prêtes à être lues. Une nouvelle API nommée `epoll` a vu le jour par la suite permettant un contrôle encore plus fin des cas de réveil du processus.

Ces appels système sont essentiels dans de nombreuses situations. `Select()` et `poll()` sont décrits dans SUSv4, la famille des fonctions `epoll` est spécifique à Linux. L'appel `select()` est le plus employé même si sa syntaxe est plus compliquée, aussi le décrirons-nous en premier. Il est déclaré dans `<sys/types.h>` ainsi :

```
int select (int nb_descripteurs,  
            fd_set * ensemble_a_lire,  
            fd_set * ensemble_a_ecrire,  
            fd_set * ensemble_exceptions,  
            struct timeval * delai_maxi);
```

Il prend en arguments trois pointeurs sur des ensembles de descripteurs (un pointeur NULL correspondant à un ensemble ignoré) :

- Le premier ensemble est surveillé par le noyau en attente de données à lire. Dès que des informations sont disponibles, le processus est réveillé. Nous étudierons d'abord ce principe.
- Les descripteurs du second ensemble correspondent à des sorties du processus. On désire ici qu'un de ces descripteurs accepte de recevoir des données. On attend par exemple qu'un buffer se vide en libérant de la place ou qu'un processus lecteur ait ouvert un tube nommé.
- Le troisième ensemble est rarement utilisé car il contient des descripteurs sur lesquels on attend l'arrivée de conditions exceptionnelles. Ceci correspond généralement à l'arrivée de données urgentes hors bande sur une socket réseau TCP.

L'appel système `select()` permet aussi de configurer un délai d'attente maximal. Lorsque celui-ci est écoulé, le noyau termine l'appel avec un code de retour nul. Lorsque le pointeur sur la structure `timeval` est NULL, l'appel système reste bloqué indéfiniment en attente d'une condition favorable sur un descripteur. Si, au contraire, la valeur du délai vaut 0 (dans les champs `tv_sec` et `tv_usec` de la structure `timeval`), l'appel revient immédiatement sans bloquer.

Le premier argument de `select()` est finalement le plus compliqué. Il s'agit du numéro du plus grand descripteur de fichier contenu dans les ensembles surveillés, augmenté de 1. Ceci sert au noyau pour dimensionner un masque de bits lui indiquant en interne quels descripteurs surveiller. Pour positionner cette valeur, on peut employer par exemple :

```
int plus_grand_descripteur = -1;  
for (i = 0; i < nombre_de_descripteurs; i ++)  
    if (descripteur[i] > plus_grand_descripteur)  
        plus_grand_descripteur = descripteur[i];  
select(plus_grand_descripteur + 1, ...);
```

Sachant que le noyau fournit les descripteurs de fichiers dans l'ordre, en commençant par 0, 1 et 2, qui sont attribués aux flux standard d'entrée, de sortie et d'erreur, on peut aussi avoir recours à quelques astuces, dont la plus courante est d'employer la constante symbolique `FD_SETSIZE`, qui correspond à la taille maximale d'un ensemble de descripteurs.

Ces ensembles sont du type opaque `fd_set`. On les manipule par le biais des macros suivantes :

```
FD_ZERO(fd_set * ensemble);
FD_SET (int fd, fd_set * ensemble);
FD_CLR (int fd, fd_set * ensemble);
FD_ISSET (int fd, fd_set * ensemble);
```

La macro `FD_ZERO()` permet d'initialiser un ensemble vide. Il faut toujours l'employer au début de l'utilisation d'un ensemble. La macro `FD_SET()` ajoute un descripteur dans un ensemble, tandis que `FD_CLR()` en supprime un. Enfin, `FD_ISSET()` permet de vérifier si un descripteur est présent ou non dans un ensemble.

En effet, au retour de `select()`, l'appel système renvoie le nombre de descripteurs se trouvant dans les conditions attendues et modifie les ensembles passés en arguments, pour n'y laisser que les descripteurs concernés. Si l'appel système `select()` est interrompu par un signal, il renvoie -1 et configure `EINTR` dans `errno`.

Naturellement, on utilise `select()` uniquement sur des descripteurs correspondant à des tubes, des FIFO, des fichiers spéciaux de périphériques ou des sockets réseau. Si toutefois on transmet un descripteur de fichier régulier, `select()` considère que des données sont disponibles tant qu'on n'est pas arrivé à la fin du fichier.

En oubliant pour le moment les descripteurs en attente d'écriture et de conditions exceptionnelles, nous pouvons considérer le multiplexage de plusieurs lectures :

```
int attente_reception (int descripteurs[],
                      int nb_descripteurs,
                      int delai_maxi)
{
    struct timeval attente;
    fd_set         ensemble;
    int            plus_grand = -1;
    int            i;
    int            retour;

    attente.tv_sec = delai_maxi;
    attente.tv_usec = 0;
    // Initialisation de l'ensemble
    FD_ZERO(& ensemble);
    for (i=0; i < nb_descripteurs; i++) {
        if (descripteur[i] > FD_SETSIZE) {
            fprintf(stderr, "Descripteur trop grand \n");
            return -1;
        }
        FD_SET(descripteur[i], & ensemble);
        if (descripteur[i] > plus_grand)
```



```
        plus_grand = descripteur[i];
    }
    // Attente
    do {
        retour = select(plus_grand + 1,
                        & ensemble, NULL, NULL,
                        & attente);
    } while ((retour == -1) && (errno == EINTR));
    if (retour < 0) {
        perror("select");
        return -1;
    }
    if (retour == 0) {
        fprintf(stderr, "delai dépasse \n");
        return -1;
    }
    // Examen des descripteurs prêts
    for (i = 0; i < nb_descripteurs ; i++)
        if (FD_ISSET(descripteur[i], & ensemble))
            lecture_descripteur(descripteur[i]);
    return 0;
}
```

Nous contrôlons que les descripteurs ont bien une valeur inférieure à `FD_SETSIZE`. C'est une attitude vraiment paranoïaque, ayant rarement cours dans les applications courantes, le risque de dépasser cette valeur étant infime (sauf si on ouvre à répétition un descripteur en oubliant de le refermer).

En Python, la configuration de `select.select()` est un peu plus simple puisque l'on passe directement les trois listes (lecture, écriture, exceptions) en argument, suivies éventuellement d'un `timeout` sous forme d'un nombre de secondes avec décimales.

Dans le programme suivant, nous créons dix fils et dix tubes de communication avec leur processus père. Ce dernier va surveiller les arrivées avec `select()`. Les dix fils enverront régulièrement un caractère à leur père, chacun avec une fréquence différente variant entre une fois par seconde et une fois toutes les dix secondes. Il manque de nombreuses vérifications d'erreur, qui auraient alourdi inutilement le listing.

```
exemple-select.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

#define NB_FILS 10
```

```
int main (void)
{
    int tube[NB_FILS][2];
    int i;
    int max = -1;
    char c;
    fd_set ensemble;

    for (i = 0; i < NB_FILS; i ++) {
        if (pipe (tube[i]) < 0) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
        if (fork() == 0) {
            // Fils
            close(tube[i][0]);
            c = '0' + i;
            while(1) {
                sleep(i + 1);
                write(tube[i][1], & c, 1);
            }
        }
        close(tube[i][1]);
    }
    // Père
    while (1) {
        FD_ZERO(& ensemble);
        for (i = 0; i < NB_FILS; i ++) {
            FD_SET(tube[i][0], & ensemble);
            if (tube[i][0] > max)
                max = tube[i][0];
        }
        if (select(max, & ensemble, NULL, NULL, NULL) <= 0) {
            perror("select");
            break;
        }
        for (i = 0; i < NB_FILS; i ++) {
            if (FD_ISSET(tube[i][0], & ensemble)) {
                read(tube[i][0], & c, 1);
                fprintf(stderr, "%d ", i);
            }
        }
    }
    return EXIT_SUCCESS;
}
```

Lors de son exécution, ce programme adopte bien le comportement dynamique qu'on attend, tout en évitant de faire des boucles actives consommatrices inutiles de ressources processeur.

```
$ ./exemple-select
0 1 0 2 0 3 1 0 4 0 5 2 1 0 6 0 3 7 1 0 2 8 0 4 9 1 0 0 3 5 2 1 0
(Contrôle-C)
$
```

Le comportement est identique avec le script Python suivant. On remarque que les listes de descripteurs fournies en argument à `select()` ne sont pas modifiées, mais que ce dernier renvoie un triplet de trois nouvelles listes contenant les descripteurs pour lesquels des actions sont possibles :

```
exemple-select.py :
#!/usr/bin/python
from __future__ import print_function
import os
import fcntl
import select
import sys
import time

read_list = []

for i in range (0, 9):
    (fd_r, fd_w) = os.pipe()
    if os.fork() != 0:
        os.close(fd_r)
        while True:
            time.sleep(i+1)
            os.write(fd_w, chr(48+i))
    else:
        read_list.append(fd_r)

while True:
    (r, w, x) = select.select(read_list, [], [])
    for fd in r:
        c = os.read(fd, 4096)
        print(c, end=' ', file=sys.stderr)
```

Lorsqu'un appel système `select()` se termine avant que le délai maximal soit écoulé, soit parce qu'un descripteur est prêt, soit parce qu'un signal l'a arrêté, le noyau Linux modifie le contenu de la structure `timeval` passée en dernier argument afin qu'elle contienne la durée restante non écoulée. Ce comportement est bien commode dans certains cas, notamment lorsqu'on implémente un bouclage pour ignorer les interruptions dues aux signaux :

```
do {
    retour = select (FD_SETSIZE, & ensemble, NULL,
                    NULL, & attente);
} while ((retour == -1) && (errno == EINTR));
```

Toutefois, il faut savoir que ce comportement n'est pas portable. La plupart des autres Unix de la famille Système V ne modifient pas ce délai. Le même problème peut se poser pour porter sous Linux une application qui considère que cette variable n'est pas modifiée et la réutilise directement. Pour pallier ce problème, le noyau Linux offre la possibilité de modifier la personnalité du processus grâce à l'appel système `personality()`. Ce dernier permet de demander au noyau d'adopter, avec ce processus, une attitude différente dans certains appels système, ainsi que de numérotter autrement les signaux. Nous ne détaillerons pas cette fonction car elle est très spécifique et peu recommandée. Il vaut mieux corriger l'application défectueuse que de demander au noyau d'émuler les bogues des autres systèmes.

Pour revenir au problème de la modification du délai, l'attitude la plus prudente consiste à ne pas faire de supposition concernant l'état de la variable et à la remplir à nouveau avant chaque appel, en calibrant la nouvelle attente avec `gettimeofday()`.

Une variante de `select()`, nommée `pselect()` est disponible, mais moins utilisée :

```
int pselect (int nb_descripteurs,
            fd_set * ensemble_a_lire,
            fd_set * ensemble_a_ecrire,
            fd_set * ensemble_exceptions,
            struct timeval * delai_maxi,
            const sigset_t * signaux);
```

Elle prend un argument supplémentaire par rapport à `select()` : c'est le masque des signaux qui sera utilisé durant l'attente. Ceci permet d'attendre de manière fiable et simultanée l'occurrence de signaux (que l'on bloque en dehors de l'appel système pour ne pas risquer d'en perdre) et de situations sur des descripteurs.

Il existe un autre appel de multiplexage, `poll()`, issu de l'univers Système V, déclaré dans `<poll.h>` :

```
int poll (struct pollfd * pollfd,
          unsigned int nb_structures,
          int delai_maxi);
```

Les descripteurs à surveiller sont indiqués dans une table de structures `pollfd`, contenant les membres suivants.

Nom	Type	Signification
fd	int	Descripteur de fichier à surveiller.
events	short int	Liste des événements qui nous intéressent concernant ce descripteur.
revents	short int	Ensemble des événements survenus, au retour de l'appel système.

Le noyau examine donc les événements attendus sur chaque descripteur et modifie le membre `revents` avant de revenir de l'appel système. Les événements qui peuvent survenir sont les suivants.

Nom	Signification
POLLIN	Données disponibles pour la lecture.
POLLOUT	Descripteur prêt à recevoir des données.
POLLPRI	Données urgentes disponibles (informations hors bande sur connexion TCP).
POLLERR	Erreur survenue sur le descripteur (uniquement en réponse dans <code>revents</code>).
POLLHUP	Déconnexion d'un correspondant (uniquement en réponse dans <code>revents</code>).
POLLNVAL	Descripteur invalide (uniquement en réponse dans <code>revents</code>).

Comme `select()`, `poll()` renvoie normalement le nombre de structures pour lesquelles il s'est passé quelque chose (éventuellement une erreur) et une valeur nulle si le délai est dépassé. S'il est interrompu par un signal, `poll()` renvoie -1 et positionne `errno` avec `EINTR`.

L'avantage de `poll()` par rapport à `select()` c'est qu'il n'y a pas de limite absolue au nombre de descripteurs surveillés simultanément. En revanche, cette fonction est nettement moins répandue que `select()` dans les Unix de la famille BSD. Les programmeurs préfèrent donc généralement employer `select()`.

Enfin, les applications spécifiques à Linux et ne visant pas la portabilité sur d'autres systèmes compatibles SUSv4 peuvent employer l'API `epoll`. Celle-ci ajoute la notion de notification par fronts ou par niveaux. Prenons l'exemple d'un descripteur sur lequel on veut lire des données. Dans la première configuration, la notification est réalisée quand le descripteur passe de l'état « rien à lire » à l'état « des données sont disponibles ». Dans la seconde configuration, la notification est réalisée dès lors que des données sont lisibles, même si elles étaient déjà présentes avant l'appel. C'est le comportement usuel de `select()`.

```
#include <sys/epoll.h>
int epoll_create(int unused);
```

Cet appel système crée un descripteur spécifique pour les appels `epoll`. L'argument n'est plus utilisé mais doit être supérieur à zéro. Par convention, on indique générale-

ment le nombre de descripteurs de fichiers que l'on supervisera avec ce descripteur `epoll`. Il existe une variante `epoll_create1()` acceptant le flag `EPOLL_CLOEXEC` et précisant que le descripteur obtenu doit être automatiquement fermé lors d'un appel système `execve()`.

```
int epoll_ctl (int epoll_fd, int operation, int fd,
               struct epoll_event *event);
```

Cet appel permet d'inscrire un descripteur de fichier `fd` (résultat d'un `open()`, `socket()`, etc.) dans le descripteur `epoll_fd` obtenu précédemment. Les événements à superviser sont décrits dans une structure `epoll_event` contenant les champs suivants :

Nom	Type	Signification
<code>events</code>	<code>uint32_t</code>	Événement attendu (masque de bits, voir ci-après).
<code>data</code>	<code>union epoll_data</code>	Champ pour l'utilisateur (union avec champs <code>fd</code> , <code>ptr</code> , <code>u32</code> ou <code>u64</code>).

Le champ `events` de la structure `epoll_event` peut contenir un ou plusieurs des éléments suivants :

Nom	Signification
<code>EPOLLIN</code>	Attente de données à lire.
<code>EPOLLOUT</code>	Attente de place pour écrire.
<code>EPOLLRDHUP</code>	Attente d'une fermeture de connexion réseau.
<code>EPOLLPRI</code>	Attente de données urgentes à lire.
<code>EPOLLERR</code>	Attente d'une condition d'erreur. Cet événement est implicitement présent dans tout appel à <code>epoll_wait()</code> .
<code>EPOLLHUP</code>	Attente d'une fin de communication. Cet événement est également présent implicitement dans toute invocation de <code>epoll_wait()</code> .
<code>EPOLLONESHOT</code>	Ne détecter qu'une seule occurrence des événements présents et « oublier » le descripteur de fichier ensuite.

Lors de l'appel à `epoll_ctl()`, on précise le type d'opération à réaliser :

```
int epoll_wait (int epoll_fd,
                 struct epoll_event *ret_events,
                 int nb_events, int timeout);
int epoll_pwait(int epoll_fd,
                 struct epoll_event *events,
                 int nb_events, int timeout,
                 const sigset_t *sigmask);
```

Nom	Signification
<code>EPOLL_CTL_ADD</code>	Ajouter le descripteur <code>fd</code> dans le descripteur <code>epoll_fd</code> pour les événements indiqués dans <code>event.events</code> .
<code>EPOLL_CTL_DEL</code>	Retirer le descripteur <code>fd</code> du descripteur <code>epoll_fd</code> . Le champ <code>event</code> est ignoré (donc généralement <code>NULL</code>).
<code>EPOLL_CTL_MOD</code>	Modifier la liste des événements pour lesquels le descripteur <code>fd</code> est présent dans <code>epoll_fd</code> .

Ces appels système se mettent en attente sur le descripteur `epoll_fd` jusqu'à ce qu'un événement attendu se produise sur l'un des descripteurs de fichiers qu'il contient, ou jusqu'à ce que le `timeout` en millisecondes soit atteint (un `timeout` de `-1` correspond à une attente infinie).

Au retour, l'appel remplira au plus `nb_events` des structures `epoll_event` contenues dans la table `ret_events` (que nous devons donc allouer avant l'appel). Chacune contiendra dans son membre `events` l'événement effectivement survenu, et dans son membre `data` la donnée associée fournie par l'utilisateur (on utilise généralement le champ `fd` de l'union `epoll_data`).

L'appel `epoll_pwait()` permet en outre de préciser une liste de signaux que l'on souhaite attendre. En cela, il est très proche de `pselect()`.

Voici un exemple d'utilisation de l'API `epoll`.

```
exemple-epoll.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/epoll.h>

#define NB_FILS 10

int main (void)
{
    int tube[NB_FILS][2];
    int i;
    int n;
    char c;
    int efd;
    struct epoll_event evt;
    struct epoll_event * revts = NULL;

    for (i = 0; i < NB_FILS; i ++) {
        if (pipe (tube[i]) < 0) {
            perror("pipe");
            exit(EXIT_FAILURE);
        }
    }
```

```

        if (fork() == 0) {
            // Fils
            close(tube[i][0]);
            c = '0' + i;
            while(1) {
                sleep(i + 1);
                write(tube[i][1], & c, 1);
            }
        }
        close(tube[i][1]);
    }

    // Père
    while (1) {
        efd = epoll_create(NB_FILS);
        if (efd < 0) {
            perror("epoll_create");
            exit(EXIT_FAILURE);
        }

        for (i = 0; i < NB_FILS; i++) {
            evt.events = EPOLLIN;
            evt.data.fd = tube[i][0];
            if (epoll_ctl(efd, EPOLL_CTL_ADD, evt.data.fd, & evt) != 0 {
                perror("epoll_ctl");
                exit(EXIT_FAILURE);
            }
        }
        revts = calloc(NB_FILS, sizeof(struct epoll_event));
        if (revts == NULL) {
            perror("calloc");
            exit(EXIT_FAILURE);
        }
        n = epoll_wait(efd, revts, NB_FILS, -1);
        if (n < 0)
            break;
        for (i = 0; i < n; i++) {
            read(revts[i].data.fd, & c, 1);
            fprintf(stderr, "%c ", c);
        }
    }
    return EXIT_SUCCESS;
}

```

Distribution de données – Multiplexage de sorties

Le multiplexage de données en sortie est plus rare, car un processus qui veut envoyer des données à un correspondant préfère souvent rester bloqué quelque temps mais

être sûr que ses informations sont émises. Il est toutefois possible d'avoir à écrire un volume important de données sur plusieurs descripteurs simultanément. On peut alors implémenter un système de mémoire tampon, avec lequel on écrit avec des `write()` non bloquants sur les descripteurs qui sont prêts.

On peut aussi cumuler un multiplexage d'entrées et de sorties dans le même appel `select()`. Supposons qu'on reçoive de manière continue des données provenant d'un tube source et qu'on doive les distribuer autant que possible vers des tubes cibles. On pourrait utiliser un buffer de sortie pour chaque cible. Une implémentation pourrait être :

```
int distribution (int source, int cibles [], int nb_cibles)
{
    fd_set    ensemble_lecture;
    fd_set    ensemble_ecriture;
    int       i;
    char **    buffer_cible = NULL;
    int *      contenu_buffer = NULL;
    char       buffer_source[LG_BUFFER];
    int        contenu_source;
    int        nb_ecrits;

    // Allouer un tableau de buffers (1 pour chaque source)
    // avec un indicateur du contenu pour chaque buffer
    if ((contenu_buffer = calloc(nb_cibles, sizeof(int))) == NULL)
        return -1;
    if ((buffer_cible=calloc(nb_cibles,sizeof(char *)))==NULL){
        free(contenu_buffer);
        return -1;
    }
    // Allouer les buffers proprement dits
    for (i = 0; i < nb_cibles; i++) {
        if ((buffer_cible[i] = malloc(LG_BUFFER)) == NULL) {
            while (--i >= 0)
                free(buffer_cible[i]);
            free(buffer_cible);
            free(contenu_buffer);
            return -1;
        }
        contenu_buffer[i] = 0;
    }
    while (1) {
        FD_ZERO(& ensemble_lecture);
        FD_SET(source, & ensemble_lecture);
        FD_ZERO(& ensemble_ecriture);
        for (i = 0; i < nb_cibles; i++)
            if (contenu_buffer[i] > 0)
                // Il y a des données à écrire sur cette cible
                FD_SET(cibles[i], & ensemble_ecriture);
```

```

// Attendre l'arrivée de données ou la libération
// d'une cible vers laquelle il reste des données
// à écrire
while (select(FD_SETSIZE, & ensemble_lecture,
              & ensemble_ecriture, NULL,
              NULL) < 0)
    if (errno != EINTR)
        perror("select");
if (FD_ISSET(source, & ensemble_lecture)) {
    // Il y a des données à lire
    contenu_source=read(source,buffer_source,LG_BUFFER);
    if (contenu_source > 0)
        // On les transmet dans les buffers des cibles
        for (i = 0; i < nb_cibles; i++)
            if (contenu_source
                +contenu_buffer[i] < LG_BUFFER) {
                // Il y a assez de place sur cette cible
                memcpy(
                    &(buffer_cible[i][contenu_buffer[i]]),
                    buffer_source,
                    contenu_source);
                contenu_buffer[i] += contenu_source;
            } else {
                // La cible ne lit pas assez vite,
                // copions ce que l'on peut.
                memcpy (
                    &(buffer_cible[i][contenu_buffer[i]]),
                    buffer_source,
                    LG_BUFFER - contenu_buffer[i]);
                contenu_buffer[i] = LG_BUFFER;
            }
        }
    for (i = 0; i < nb_cibles; i++) {
        if (FD_ISSET(buffer_cible[i], & ensemble_ecriture)) {
            // Il y a de la place libérée sur la cible
            nb_ecrits = write(cibles[i],
                             buffer_cible[i],
                             contenu_buffer[i]);
            if (nb_ecrits > 0) {
                memmove(buffer_cible[i],
                        & (buffer_cible[i][nb_ecrits]),
                        contenu_buffer[i] - nb_ecrits);
                contenu_buffer[i] -= nb_ecrits;
            }
        }
    }
}
}
}
}

```

On imagine bien la puissance de l'appel système `select()`, puisqu'il permet de centraliser totalement la surveillance de multiples canaux de communication, l'application étant prête à réagir immédiatement lorsque des données arrivent ou dès qu'un descripteur de sortie est libre.

Ce mécanisme est particulièrement précieux dès qu'une application doit se comporter en serveur vis-à-vis de multiples clients, mais il est parfois insuffisant. Une application fonctionnant sous X-Window par exemple doit invoquer lors de son initialisation une routine de boucle principale fournie par les bibliothèques graphiques, généralement `XtAppMainLoop()`, qui ne lui rend pas la main mais invoquera les fonctions *callback* associées aux composants graphiques lors de leurs activations. Un tel programme ne peut donc pas rester bloqué sur `select()`.

Une solution possible est d'employer un `fork()` avant l'initialisation graphique, pour laisser le processus fils s'occuper de tout le dialogue avec les clients alors que le processus père ne gère que l'interface graphique. La communication entre les processus est réduite au minimum, et s'établit au moyen de deux tubes et de signaux.

Nous pouvons également utiliser le mécanisme des entrées-sorties asynchrones offert par l'appel système `fcntl()`.

Entrées-sorties asynchrones

Asynchronisme utilisant `fcntl()`

L'appel système `fcntl()`, que nous avons déjà rencontré dans le chapitre 22, permet d'implémenter un mécanisme d'entrées-sorties asynchrones assez intéressant car il offre un multiplexage des entrées ou des sorties comme avec `select()`, tout en laissant le processus libre d'exécuter du code au lieu de le bloquer en attente.

Ce système est plutôt réservé aux tubes de communication ou aux sockets. Le principe consiste à associer au descripteur de fichier le PID du processus, et de laisser le noyau nous prévenir lorsque des données seront disponibles en lecture ou lorsqu'il y aura à nouveau de la place pour écrire dans le descripteur. Dès que l'une de ces conditions se présente, le noyau nous envoie le signal `SIGIO`.

Nous avons donc une sorte d'équivalent de `select()`, puisque nous serons prévenus lorsqu'une condition d'entrée-sortie sera réalisée, tout en conservant la liberté d'exécuter le reste du programme.

Les commandes employées avec `fcntl()` sont les suivantes :

- `F_SETOWN` : on indique le PID du processus devant recevoir le signal. Un PID négatif correspond à un groupe de processus.

- Normalement, on utilise `fcntl(fd, F_SETOWN, getpid())`.
- `F_GETOWN` : l'appel `fcntl(fd, F_GETOWN)` renvoie le PID du processus ou du groupe récepteur.
- `F_SETSIG` : ceci permet de configurer le numéro du signal à employer pour prévenir des modifications de conditions. Par défaut ou en cas de valeur nulle, le signal `SIGIO` est envoyé. Toutefois, nous préférons utiliser un signal temps réel, car des informations supplémentaires seront disponibles. Nous pouvons employer un signal différent pour chaque descripteur, quoique cela ne soit pas indispensable avec les signaux temps réel.
- On appellera donc `fcntl (fd, F_SETSIG, SIGRTMIN + 5)`.
- `F_GETSIG` : avec `fcntl(fd, F_GETSIG)`, on peut connaître le signal employé.

Le gestionnaire de signal reçoit, s'il s'agit d'un signal temps réel, une structure `siginfo` dont le champ `si_code` contient la valeur `SI_SIGIO`, et le membre `si_fd` le descripteur du fichier concerné. Si on utilise le signal `SIGIO` d'origine, le gestionnaire ne dispose pas de cette structure `siginfo`, il devra donc employer `select()` avec un délai nul pour déterminer quel descripteur est devenu disponible.

Pour activer le comportement asynchrone, il faut faire appel à la commande `F_SETFL` de `fcntl()` et activer l'attribut `O_ASYNC` ainsi :

```
| fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_ASYNC);
```

Notons qu'à la différence de `select()` le noyau nous prévient ici lorsque les conditions de lecture ou d'écriture sont modifiées. Si des données sont disponibles en lecture avant l'activation de `O_ASYNC` ou s'il y a déjà de la place pour l'écriture, nous n'en serons pas prévenus. Une solution consiste à s'envoyer systématiquement un signal juste après le basculement en mode asynchrone et à vérifier au sein du gestionnaire l'état du descripteur, à l'aide d'un appel `select()` avec un délai nul.

La portabilité de cette méthode n'est pas totale. Certains systèmes n'implémentent l'attribut `O_ASYNC`, aussi Il existe également un autre mécanisme : l'asynchronisme introduit par la norme `Posix.1b` et défini de nos jours par `SUSv4`.

Asynchronisme compatible `Posix.1b`

Jusqu'à présent nous avons réussi à optimiser la communication sur plusieurs canaux simultanément grâce au multiplexage de `select()`, et à laisser le programme se dérouler normalement en réitérant de temps à autre les tentatives d'entrée-sortie.

Toutefois ces mécanismes ne sont pas suffisants dans le cas où on désire vraiment recevoir ou émettre des données de manière suffisamment fiable, avec un mode opératoire totalement asynchrone par rapport au reste du programme. En effet, lorsque

`select()` – ou un signal programmé par `fcntl()` – nous indique que des données sont disponibles en lecture, tout ce que nous savons c’est que le descripteur est prêt à nous délivrer un octet. Si nous désirons en lire plusieurs, l’appel `read()` peut être bloquant. Si on bascule en lecture non bloquante, il faut de surcroît gérer un buffer interne pour recevoir assez d’informations avant de les traiter.

Le fonctionnement des écritures n’est pas plus sûr : `select()` nous précise qu’on peut écrire au moins un octet. Maintenant, si on veut transmettre une trame complète de données, il faut s’attendre à ce que `write()` bloque indéfiniment. L’écriture non bloquante nécessiterait d’établir un tampon de sortie pour s’assurer que l’ensemble de données sera écrit.

Heureusement, on peut employer des procédures d’entrées-sorties totalement asynchrones nous épargnant la gestion d’un buffer. On programme une opération de lecture ou d’écriture, le noyau la démarre, et lorsqu’elle est terminée le processus est averti par exemple par l’arrivée d’un signal. Durant le temps de l’opération d’entrée-sortie, le programme est libre de faire ce que bon lui semble, utiliser le processeur, faire des appels système, dormir...

Une expérience instructive consiste à lancer la commande :

```
| $ find / -name introuvable
```

tout en surveillant l’activité du processeur, par l’intermédiaire de `top` par exemple. Cette commande va parcourir toute l’arborescence du système de fichiers à la recherche des nœuds ayant le nom « introuvable ». On observe que le disque est fortement mis à contribution, une activité incessante et prolongée s’y déroulant. En revanche on remarque que le processeur lui-même reste dans un état calme, sa charge étant très faible. Les procédures d’entrée-sortie utilisent donc très peu de ressources de calcul de la machine.

Dans une application temps réel, il peut donc être très intéressant de déléguer une part du travail d’enregistrement par exemple, en le laissant s’exécuter automatiquement tandis que l’application peut continuer à répondre aux événements survenant entre-temps.

Les mécanismes d’entrées-sorties asynchrones sont disponibles si la constante `_POSIX_ASYNCIO` est définie dans `<unistd.h>`, ce qui est le cas depuis Linux 2.2.

Jusqu’au noyau Linux 2.4 inclus, ces mécanismes étaient en fait des fonctions de la bibliothèque C qui sont implémentées au moyen des threads Posix. L’édition des liens devait se faire avec la bibliothèque `librt.so` (*real time*) et la bibliothèque `libpthread.so` au moyen des options `-lrt -lpthread` en ligne de commande.

Depuis le noyau 2.6, il s'agit véritablement de mécanismes asynchrones implémentés par le noyau, seule l'option `-lrt` reste nécessaire.

Le principe des entrées-sorties asynchrones conformes à la norme SUSv4 n'est guère plus compliqué que celui des entrées-sorties classiques : on prépare un bloc constitué par une structure `aiocb` (*Asynchronous I/O Control Block*), contenant en substance le buffer, le descripteur de fichier et le type d'opération désirée. Ce bloc est transmis au noyau qui programme l'entrée-sortie, puis prévient le processus par un signal lorsque l'opération est terminée. La structure `aiocb` comprend les membres suivants.

Nom	Type	Signification
<code>aio_fildes</code>	<code>int</code>	Descripteur du fichier concerné par l'opération d'entrée-sortie.
<code>aio_offset</code>	<code>off_t</code>	Emplacement au sein du fichier où commence l'opération.
<code>aio_buf</code>	<code>void *</code>	Buffer pour les données à écrire ou à lire.
<code>aio_nbytes</code>	<code>size_t</code>	Nombre d'octets à transférer.
<code>aio_reqprio</code>	<code>int</code>	Priorité de l'opération.
<code>aio_sigevent</code>	<code>struct sigevent</code>	Description du mécanisme de signalisation une fois le transfert terminé.
<code>aio_lio_opcode</code>	<code>int</code>	Code opératoire décrivant le transfert (uniquement dans certains cas).

À ma connaissance, ces fonctions n'existent pas dans la bibliothèque Python standard. Des implémentations sont parfois proposées par des modules externes.

Pour programmer une lecture ou une écriture, on emploie les fonctions `aio_read()` ou `aio_write()`, déclarées ainsi dans `<aio.h>` :

```
int aio_read(struct aiocb * aiocb);  
int aio_write (struct aiocb * aiocb);
```

La structure `aiocb` contient donc le numéro du descripteur de fichier, l'adresse du buffer pour les données et la taille désirée, mais également le décalage où l'opération doit avoir lieu dans le fichier. Ce décalage est mesuré en octets, comme avec `lseek()`, depuis le début du fichier. En effet, la position courante dans le fichier n'est jamais significative avec les entrées-sorties asynchrones. L'emplacement de lecture ou d'écriture doit toujours être indiqué. Nous reviendrons sur ce point ultérieurement.

Le membre `aio_reqprio` dispose d'une valeur numérique indiquant la valeur qui doit être soustraite de la priorité du processus pour exécuter l'opération d'entrée-sortie. Ceci n'a d'intérêt que si on déclenche de nombreuses opérations simultanées. Plus cette valeur est élevée, moins l'opération sera prioritaire par rapport à ses consœurs. La priorité de l'opération n'est que rarement utilisée, aussi la remplit-on généralement avec

une valeur nulle. Nous en verrons toutefois un exemple d'utilisation plus tard. Avec les appels `aio_read()` et `aio_write()`, on n'emploie pas non plus le membre `aio_lio_opcode` puisque le système sait toujours quelle opération doit avoir lieu.

Pour indiquer que le transfert asynchrone est terminé – avec succès ou non –, le système peut nous envoyer un signal ou démarrer un thread sur une fonction spéciale. Pour configurer ce comportement, on utilise la structure `sigevent` du champ `aio_sigevent`, définie dans `<signal.h>` ainsi.

Nom	Type	Signification
<code>sigev_notify</code>	<code>int</code>	Type de notification désirée pour indiquer la fin d'une opération asynchrone.
<code>sigev_signo</code>	<code>int</code>	Numéro du signal à employer pour la notification.
<code>sigev_value</code>	<code>sigval_t</code>	Valeur à transmettre au gestionnaire de signal ou au thread.
<code>sigev_notify_function</code>	<code>void (* f) (sigval_t)</code>	Fonction à déclencher dans un nouveau thread.
<code>sigev_notify_attributes</code>	<code>pthread_attr_t</code>	Attribut du nouveau thread.

Attention, les membres `sigev_notify_function` et `sigev_notify_attributes` sont en réalité des macros qui donnent accès aux champs d'une union assez complexe. On évitera donc de nommer ainsi des variables.

Le type `sigval_t` est un autre nom de union `sigval`, que nous avons rencontrée dans le chapitre 13, et qui peut prendre les formes suivantes.

Nom	Type
<code>sival_int</code>	<code>int</code>
<code>sival_ptr</code>	<code>Void *</code>

Le membre `sigev_notify` contient l'une des constantes symboliques suivantes :

- `SIGEV_NONE` : aucune notification n'est demandée. Le processus pourra toutefois s'assurer de la fin d'une opération en employant des routines que nous décrirons plus bas ;
- `SIGEV_SIGNAL` : le système enverra au processus le signal mentionné dans le champ `sigev_signo` pour indiquer que l'opération est terminée. S'il s'agit d'un signal temps réel Posix.1b, le gestionnaire recevra dans son argument `siginfo_t` des informations supplémentaires, dont la valeur du membre `sigev_value`. Le

champ `si_code` de la structure `siginfo_t` est rempli avec le code `SI_ASYNCIO`, comme nous l'avons déjà évoqué dans le chapitre 13 ;

- `SIGEV_THREAD` : la bibliothèque C démarrera un nouveau thread, qui exécutera la fonction sur laquelle le champ `sigev_notify_function` représente un pointeur. Cette routine recevra en argument le contenu du membre `sigev_value`. Le thread créé reçoit les attributs décrits par le champ `sigev_notify_attributes`. Il s'agit des attributs au sens des Pthreads, comme nous les avons vus dans le chapitre 6 (détachable, joignable, etc.).

Avec `SIGEV_THREAD` comme avec `SIGEV_SIGNAL`, on remplit généralement le membre `sival_ptr` du champ `sigev_value` avec un pointeur sur la structure `aio_cb` elle-même, afin que le nouveau thread ou le gestionnaire aient accès à l'opération réalisée. Naturellement, on ne peut pas réemployer la même structure avant que l'opération soit terminée.

Pour savoir si une opération est terminée ou non, on utilise la fonction `aio_error()` :

```
| int aio_error (const struct aio_cb * aio_cb);
```

Cette routine renvoie l'erreur `EINPROGRESS` si l'opération décrite par la structure `aio_cb` n'est pas terminée. Sinon elle transmet éventuellement un indicateur d'erreur. Une fois qu'une opération est finie, et uniquement à ce moment-là, on peut appeler la fonction `aio_return()` pour avoir le compte rendu de l'entrée-sortie :

```
| ssize_t aio_return (const struct aio_cb * aio_cb);
```

Cette fonction renvoie tout simplement la valeur de retour des appels système `read()` ou `write()` sous-jacents. Cette fonction ne doit être appelée qu'une seule fois car Posix.1b autorise une implémentation où elle servirait à libérer des données internes.

Même dans le gestionnaire de signal servant à la notification, il faut donc employer toujours la séquence :

```
| if (aio_error(aio_cb) == EINPROGRESS)
|     return;
| if ((retour = aio_return(aio_cb)) != aio_cb -> aio_nbytes)
|     // Traitement d'erreur
|     [...]
| else
|     // Réussite
|     [...]
```

On peut très bien éviter la notification et vérifier plus tard explicitement si l'opération s'est bien terminée. Dans le programme suivant nous allons utiliser trois lectures asynchrones, employant les trois possibilités de notification. La lecture demandée est

la même à chaque fois, on réclame les 256 premiers octets du fichier dont le nom est passé en argument.

```
exemple-aio-read.c :
#include <aio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/stat.h>

#define SIGNAL_IO (SIGRTMIN + 3)

void gestionnaire (int signum, siginfo_t * info, void * vide)
{
    struct aiocb * cb;
    ssize_t      nb_octets;
    if (info->si_code == SI_ASYNCIO) {
        cb = info->si_value.sival_ptr;
        if (aio_error(cb) == EINPROGRESS)
            return;
        nb_octets = aio_return(cb);
        fprintf(stdout, "Lecture 1 : %d octets lus \n", nb_octets);
    }
}

void thread (sigval_t valeur)
{
    struct aiocb * cb;
    ssize_t      nb_octets;
    cb = valeur.sival_ptr;
    if (aio_error(cb) == EINPROGRESS)
        return;
    nb_octets = aio_return(cb);
    fprintf(stdout, "Lecture 2 : %d octets lus \n", nb_octets);
}

int main (int argc, char * argv[])
{
    int          fd;
    struct aiocb  cb[3];
    char         buffer[256][3];
    struct sigaction action;
    int          nb_octets;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier\n", argv[0]);
    }
}
```

```

    exit(EXIT_FAILURE);
}
if ((fd = open(argv[1], O_RDONLY)) < 0) {
    perror("open");
    exit(EXIT_FAILURE);
}
action.sa_sigaction = gestionnaire;
action.sa_flags      = SA_SIGINFO;
sigemptyset(& action.sa_mask);
if (sigaction(SIGNAL_IO, & action, NULL) < 0) {
    perror("sigaction");
    exit(EXIT_FAILURE);
}

// Lecture 0 : Pas de notification
cb[0].aio_fildes  = fd;
cb[0].aio_offset  = 0;
cb[0].aio_buf     = buffer[0];
cb[0].aio_nbytes  = 256;
cb[0].aio_reqprio = 0;
cb[0].aio_sigevent.sigev_notify = SIGEV_NONE;

// Lecture 1 : Notification par signal
cb[1].aio_fildes  = fd;
cb[1].aio_offset  = 0;
cb[1].aio_buf     = buffer [1];
cb[1].aio_nbytes  = 256;
cb[1].aio_reqprio = 0;
cb[1].aio_sigevent.sigev_notify = SIGEV_SIGNAL;
cb[1].aio_sigevent.sigev_signo = SIGNAL_IO;
cb[1].aio_sigevent.sigev_value.sival_ptr = & cb[1];

// Lecture 2 : Notification par thread
cb[2].aio_fildes  = fd;
cb[2].aio_offset  = 0;
cb[2].aio_buf     = buffer [2];
cb[2].aio_nbytes  = 256;
cb[2].aio_reqprio = 0;
cb[2].aio_sigevent.sigev_notify = SIGEV_THREAD;
cb[2].aio_sigevent.sigev_notify_function = thread;
cb[2].aio_sigevent.sigev_notify_attributes = NULL;
cb[2].aio_sigevent.sigev_value.sival_ptr = & cb[2];

// Lancement des lectures
if ((aio_read(& cb[0]) < 0)
    || (aio_read(& cb[1]) < 0)
    || (aio_read(& cb[2]) < 0)) {
    perror("aio_read");
    exit(EXIT_FAILURE);
}

```

```

    fprintf(stdout, "Lectures lancees\n");
    while ((aio_error(& cb[0]) == EINPROGRESS)
        || (aio_error(& cb[1]) == EINPROGRESS)
        || (aio_error(& cb[2]) == EINPROGRESS))
        sleep(1);
    nb_octets = aio_return(& cb[0]);
    fprintf(stdout, "Lecture 0 : %d octets lus \n", nb_octets);
    return EXIT_SUCCESS;
}

```

La vérification (`aio_error(cb) == EINPROGRESS`) est indispensable dans le gestionnaire de signal, car `SIGRTMIN+3` peut provenir d'une autre source. Au sein du thread ce contrôle est inutile car on ne doit normalement pas appeler cette routine directement. Je l'ai laissée car c'est une bonne habitude – paranoïaque – pour s'assurer de la fin d'un transfert avant d'appeler `aio_return()`.

En attendant que toutes les lectures soient terminées, le programme s'endort par période d'une seconde – ou moins quand le signal arrive – pour éviter de consommer inutilement de ressources CPU. L'exécution se déroule comme prévu

```

$ ./exemple-aio-read ./exemple-aio-read
Lectures lancees
Lecture 1 : 256 octets lus
Lecture 2 : 256 octets lus
Lecture 0 : 256 octets lus
$ ls -l Makefile
-rw-r--r--  1 ccb      ccb      242 Aug 2 14:02 Makefile
$ ./exemple_aio_read Makefile
Lectures lancées
Lecture 1 : 242 octets lus
Lecture 2 : 242 octets lus
Lecture 0 : 242 octets lus
$

```

Étant donné qu'une lecture ou une écriture asynchrone modifie la position du pointeur dans le fichier, il faut considérer que cette valeur peut changer à tout moment tant que l'opération n'est pas terminée. Et à ce moment encore la position restera indéterminée tant qu'elle n'aura pas été revalidée avec `lseek()`.

Cela signifie qu'il faut absolument éviter d'employer de lecture ou d'écriture synchrones habituelles pendant que des opérations asynchrones ont lieu. Il faudrait en effet, dans le cours normal du processus, lier atomiquement le déplacement du pointeur avec `lseek()` et l'appel système `read()` ou `write()` qui suit. Ceci ne peut se faire qu'à l'aide des appels système `pread()` et `pwrite()`, que nous avons examinés dans le

chapitre 22 et qui sont employés par la bibliothèque C pour implémenter `aio_read()` et `aio_write()`.

Lorsque plusieurs opérations simultanées doivent être accomplies sur le même fichier ou sur des fichiers différents, il est possible de programmer un ensemble d'entrées-sorties avec `lio_listio()` :

```
int lio_listio (int mode, struct aiocb * liste_aiocb[],
               int nb_aiocb, struct sigevent * notification);
```

Le premier argument est le mode de fonctionnement de `lio_listio()`. Il peut prendre l'une des deux valeurs suivantes :

- `LIO_NOWAIT` : la fonction lance toutes les opérations décrites dans les arguments suivants de manière asynchrone et se termine. Une fois que toutes les opérations auront été réalisées, le processus recevra une notification décrite dans le dernier argument de `lio_listio()`. Naturellement, les notifications individuelles sont également reçues au fur et à mesure de l'accomplissement des travaux ;
- `LIO_WAIT` : la fonction attend pour se terminer que toutes les opérations soient finies. Ce mécanisme est surtout utilisé avec une seule opération à la fois, pour réaliser une lecture ou une écriture normale, synchrone, alors que des opérations asynchrones ont lieu sur le même fichier. Le système préserve en effet l'atomicité du positionnement du pointeur et de l'entrée-sortie sur le fichier.

Le second argument est un tableau de pointeurs sur des structures `aiocb`. Il y a donc un niveau d'indirection supplémentaire. L'avantage c'est qu'un pointeur `NULL` dans ce tableau est ignoré. On peut donc préparer une table avec de nombreuses opérations et remplir le tableau de pointeurs en ignorant facilement des opérations qu'on ne souhaite pas effectuer immédiatement. Le troisième argument est le nombre d'opérations dans le tableau.

Dans chaque structure `aiocb`, il faut à présent remplir le champ `aio_lio_opcode` avec l'une des valeurs suivantes :

- `LIO_READ` : on veut faire une lecture ;
- `LIO_WRITE` : on veut une écriture ;
- `LIO_NOP` : pour ignorer l'opération.

Pour utiliser `lio_listio()` au lieu du lancement successif des trois lectures asynchrones de l'exemple précédent, on ajoute dans les variables de `main()` la structure `sigevent` et la table

```
struct sigevent lio_sigev;
struct aiocb * lio [3];
```

puis on lance les lectures ainsi :

```
// Lancement des lectures
lio[0] = & cb[0];
lio[1] = & cb[1];
lio[2] = & cb[2];
lio_sigev.sigev_notify = SIGEV_NONE;
if (lio_listio(LIO_NOWAIT, lio, 3, & lio_sigev) < 0) {
    perror("lio_listio");
    exit(EXIT_FAILURE);
}
```

Le déroulement du programme est identique à celui de `exemple-aiio-read`.

```
$ ./exemple-lio-listio exemple-lio-listio
Lectures lancees
Lecture 1 : 256 octets lus
Lecture 2 : 256 octets lus
Lecture 0 : 256 octets lus
$
```

Dans notre programme nous avons laissé le processus dans de courtes périodes de sommeil entre lesquelles nous avons examiné l'état des opérations en cours. Il existe une fonction plus adaptée à cette attente, nommée `aiio_suspend()` :

```
int aiio_suspend (const struct aiocb *liste_aiocb[], int nb_aiocb,
                  const struct timespec * delai_maxi);
```

Cette routine prend en argument un tableau de pointeurs sur des structures `aiocb`, comme le faisait `lio_listio()`, et attend que l'une au moins des opérations du tableau se termine. Elle rend alors la main au processus. On peut ensuite examiner, avec `aiio_error()`, quelle opération s'est achevée et vérifier son code de retour avec `aiio_return()`. L'opération terminée peut être supprimée de la liste d'attente en remplaçant son pointeur par `NULL`.

On peut ainsi éviter les notifications par l'intermédiaire d'un gestionnaire de signal impliquant un changement de contexte du processus. Le programme ci-après emploie ce principe et n'utilise ni gestionnaire de signal ni thread supplémentaire.

Si `aiio_suspend()` est interrompue par un signal, elle échoue avec l'erreur `EINTR`. Ce cas peut être tout à fait normal s'il s'agit du signal notifiant la fin d'une opération. Le dernier argument est un délai d'attente maximal. Si rien ne s'est produit durant ce

temps, la fonction échoue avec l'erreur EAGAIN. Dans le programme suivant, nous n'employons pas de délai, aussi ce pointeur est-il NULL.

```
exemple-aio-suspend.c :
#include <aio.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

#define NB_OP 10

int main (int argc, char * argv[])
{
    int          fd;
    int          i;
    struct aiocb  cb[NB_OP];
    char         buffer[256][NB_OP];

    struct sigevent lio_sigev;
    struct aiocb *  lio[NB_OP];

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s fichier\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = open(argv[1], O_RDONLY)) < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    for (i = 0; i < NB_OP; i++) {
        cb[i].aio_fildes  = fd;
        cb[i].aio_offset  = 0;
        cb[i].aio_buf     = buffer[i];
        cb[i].aio_nbytes  = 256;
        cb[i].aio_reqprio = i;
        cb[i].aio_lio_opcode = LIO_READ;
        cb[i].aio_sigevent.sigev_notify = SIGEV_NONE;
        lio[i] = & cb[i];
    }

    lio_sigev.sigev_notify = SIGEV_NONE;
    if (lio_listio(LIO_NOWAIT, lio, NB_OP, & lio_sigev) < 0) {
        perror("lio_listio");
        exit(EXIT_FAILURE);
    }
    fprintf(stdout, "Lectures lancees\n");
}
```

```
while (1) {
    for (i = 0; i < NB_OP; i++)
        if (lio[i] != NULL)
            break;
    if (i == NB_OP)
        // Toutes les opérations sont finies
        break;
    if (aio_suspend(lio, NB_OP, NULL) == 0) {
        for (i = 0; i < NB_OP; i++)
            if (lio[i] != NULL)
                if (aio_error(lio[i]) != EINPROGRESS) {
                    fprintf(stdout, "Lecture %d : %d octets \n",
                        i, aio_return(lio[i]));
                    lio[i] = NULL;
                }
            }
        }
    }
    return EXIT_SUCCESS;
}
```

Nous lançons dix lectures simultanées ; nous obtenons :

```
$ ./exemple-aio-suspend exemple-aio-suspend
Lectures lancees
Lecture 0 : 256 octets
Lecture 1 : 256 octets
Lecture 2 : 256 octets
Lecture 3 : 256 octets
Lecture 4 : 256 octets
Lecture 5 : 256 octets
Lecture 6 : 256 octets
Lecture 7 : 256 octets
Lecture 8 : 256 octets
Lecture 9 : 256 octets
$
```

Mentionnons enfin l'existence d'une fonction `aio_cancel()` permettant théoriquement d'annuler une opération qui n'a pas encore eu lieu.

```
| int aio_cancel (int fd, struct aiocb * aiocb);
```

Cette routine tente d'annuler l'opération indiquée en second argument sur le descripteur de fichier fourni en première position. Si le pointeur `aiocb` est `NULL`, cette fonction tente d'annuler toutes les opérations ayant lieu sur le descripteur indiqué.

La routine `aio_cancel()` ne donnant aucune garantie de réussite et ne permettant pas de savoir si l'opération a réellement été annulée, elle n'a quasiment aucune utilité.

Écritures synchronisées

Les mécanismes d'entrée-sortie avancés, tels que ceux qui sont décrits par la norme SUSv4, introduisent un concept d'écritures synchronisées, qui ne doivent pas être confondues avec les écritures synchrones ou asynchrones. En fait, il est tout à fait possible d'employer des écritures asynchrones synchronisées.

La notion d'écriture synchronisée fait référence au transfert effectif des données vers le disque. Nous avons déjà observé dans le chapitre 21 que des informations écrites dans un flux traversaient trois niveaux de buffers successifs. Ici, nous travaillons directement avec le descripteur de fichier et nous ignorons donc le premier buffer associé au flux¹. Nous ne pouvons pas contrôler non plus la zone tampon intégrée dans le lecteur de disque, mais nous allons nous intéresser à la mémoire cache gérée par le noyau.

Il peut être utile dans certaines applications temps réel, dans des systèmes de gestion de bases de données ou dans des logiciels d'enregistrement de type « boîte noire », de pouvoir passer outre la mémoire cache du noyau et s'assurer que les données écrites par un appel système `write()` ont bien été transmises au contrôleur de périphérique, à défaut du support physique réel. Ceci peut être réalisé avec plusieurs degrés de précision.

Tout d'abord rappelons que l'appel système `fsync()`, décrit dans le chapitre 22, sert à synchroniser le contenu du fichier sur lequel on lui passe un descripteur. Cette routine attend que toutes les données écrites soient effectivement transmises au contrôleur de disque, puis elle revient en renvoyant zéro si tout s'est bien passé, ou -1 sinon. Dans le cas d'un échec de synchronisation, l'erreur `EIO` est renvoyée dans `errno`.

Il existe également un appel système `sync()` qui ne prend pas d'argument et renvoie toujours zéro. Ici, toutes les écritures en attente dans la mémoire cache du noyau sont réalisées avant le retour. Comme nous l'avons précisé dans le chapitre 22, il existe un utilitaire `/bin/sync` qui invoque cet appel système. Sur les premières versions de Linux, les écritures pouvaient rester en suspens indéfiniment tant que la mémoire cache n'était pas pleine et qu'on n'invoquait pas `sync()`. Pour cela, l'utilitaire `/bin/sync` était appelé régulièrement par une commande de la table `crond`. À présent, ce n'est plus nécessaire, un thread du noyau se chargeant de cette tâche.

1. Les méthodes d'écritures synchronisées étudiées ici sont appliquées la plupart du temps directement aux descripteurs pour des raisons d'efficacité. Si toutefois l'utilisation d'un flux est indispensable, on pourra se tourner vers `fflush()` ou `setvbuf()`, que nous avons rencontrées dans le chapitre 21.

L'utilisation de `fsync()` peut parfois être suffisante dans une application, car elle permet de créer des points où on connaît l'état du fichier. Le vidage de la mémoire tampon est quand même assez coûteux, d'autant qu'il faut mettre à jour non seulement les données proprement dites, mais également des informations de contrôle qui ne sont pas nécessairement indispensables, comme la date de modification de l'i-nœud. Pour cela, Posix.1b a introduit l'appel système `fdatasync()`, disponible si la constante `_POSIX_SYNCHRONIZED_IO` est définie dans `<unistd.h>` :

```
| int fdatasync (int descripteur);
```

Cette routine – et `os.fdatasync(descripteur)` en Python – se comporte comme `fsync()` au niveau de l'application, mais elle n'écrit réellement que les données indispensables, en laissant les autres dans la mémoire cache du noyau. En cas d'arrêt brutal du système, l'état du disque est tel que les informations pourront être récupérées, éventuellement après le passage d'un utilitaire de réparation comme `fsck`.

Si toutes les écritures dans un fichier doivent être considérées de la même manière, il est agaçant de devoir invoquer `fsync()` ou `fdatasync()` après chaque `write()`. Pour éviter cette manipulation, nous pouvons configurer directement le comportement de toutes les écritures grâce à l'appel système `open()`. L'attribut `O_SYNC` ajouté lors de l'ouverture d'un descripteur signifie que toutes les écritures seront synchronisées, comme si on invoquait `fsync()` immédiatement après.

Notons que le noyau Linux n'autorise pas la modification de l'attribut `O_SYNC` d'un fichier après son ouverture, contrairement à d'autres systèmes qui le permettent avec `fcntl()`. Ceci peut d'ailleurs poser des problèmes lorsque le descripteur a été hérité du processus père.

Il existe deux autres constantes différentes acceptées lors de l'ouverture d'un descripteur.

Nom	Signification
<code>O_DSYNC</code>	Pour ne synchroniser automatiquement que les données écrites dans le descripteur, sans se soucier des informations relatives à l'i-nœud. C'est l'équivalent d'une <code>fdatasync()</code> après chaque <code>write()</code> .
<code>O_RSYNC</code>	Lors d'un <code>read()</code> , le noyau doit mettre à jour l'heure de dernière lecture de l'i-nœud. Avec cette constante, cette mise à jour sera synchronisée. Lorsque <code>read()</code> se termine, l'i-nœud a été mis à jour. Ce mécanisme est rarement utile. Sous Linux il n'est pas implémenté différemment de <code>O_SYNC</code> .

Comme nous l'avons déjà indiqué, les écritures synchronisées peuvent également se faire de manière asynchrone. Lorsque la notification sera envoyée au processus, les données écrites auront été entièrement transférées sous la houlette du contrôleur de périphérique. Cela se fait en utilisant `O_SYNC` lors de l'ouverture du descripteur.

Pour obtenir l'équivalent asynchrone des fonctions `fsync()` et `fdatasync()`, c'est-à-dire la garantie ponctuelle de vidage de la mémoire cache, on emploie la fonction `aio_fsync()`, déclarée ainsi :

```
int aio_fsync (int mode, struct aiocb * aiocb);
```

Cette routine déclenche `fsync()` – de manière asynchrone – sur le descripteur `aiocb.aio_fildes` si son premier argument vaut `O_SYNC`, ou simplement `fdatasync()` s'il s'agit de `O_DSYNC`. Lorsque la synchronisation est terminée, la notification inscrite dans `aiocb.aio_sigevent` est déclenchée. Il faut bien comprendre que la routine `aio_fsync()` n'attend pas la fin du vidage de la mémoire cache. Si on désire obtenir ce comportement, il faut appeler directement `fsync()`.

Les écritures synchronisées sont évidemment très coûteuses en temps d'exécution. Le programme suivant va en faire la démonstration : il crée un fichier dans lequel il écrit cinq cent mille blocs (valeur à ajuster en fonction des performances de votre système) de 256 octets. Suivant la valeur du second argument sur la ligne de commande, les écritures seront synchronisées ou pas.

```
exemple-osync.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    int fd;
    char buffer[256];
    int i;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s fichier sync \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((argv[2][0] == 'o') || (argv[2][0] == 'O')) {
        fprintf(stdout, "Ecritures synchronisees \n");
        if ((fd = open(argv[1], O_RDWR|O_CREAT|O_SYNC, 0644)) < 0) {
            perror("open");
            exit(EXIT_FAILURE);
        }
    }
    else {
        fprintf(stdout, "Ecritures non-synchronisees \n");
        if ((fd = open(argv[1], O_RDWR|O_CREAT, 0644)) < 0) {
            perror("open");
            exit(1);
        }
    }
}
```

```
    }  
  }  
  for (i = 0; i < 500000; i ++)  
    if (write(fd, buffer, 256) < 0) {  
      perror("write");  
      exit(EXIT_FAILURE);  
    }  
  fsync(fd);  
  close(fd);  
  return EXIT_SUCCESS;  
}
```

L'appel `fsync()` final nous permet d'être sûrs que toutes les écritures ont eu lieu au moment de la fin du programme. Sinon des transferts continueraient à se produire alors que nous serions déjà revenus au shell, faussant ainsi les résultats.

Pour avoir des statistiques d'exécution assez précises, on pourrait utiliser la commande `time` du shell et invoquer le programme plusieurs fois afin d'obtenir des valeurs moyennes. Les différences sont telles qu'il n'y a même pas besoin d'utiliser une surveillance si précise. Il suffit d'encadrer l'appel au programme par des commandes `date` pour connaître sa durée :

```
$ sync  
$ date ; ./exemple-osync essai.osync N ; date  
ven. août 13 13:59:06 CEST 2010  
Ecritures non-synchronisees  
ven. août 13 13:59:14 CEST 2010  
$ rm -f essai.osync  
$ date ; ./exemple-osync essai.osync 0 ; date  
ven. août 13 12:10:59 CEST 2010  
Ecritures synchronisees  
ven. août 13 13:58:58 CEST 2010  
$ sync  
$
```

Huit secondes dans un cas, contre près de deux heures dans l'autre !

La différence est aussi importante car nous avons demandé de nombreuses écritures successives de blocs de petite taille. L'i-nœud du fichier doit donc être mis à jour pour chaque écriture (taille du fichier, heure de dernière modification...). Lors d'écritures non synchronisées, cet i-nœud reste en mémoire entre chaque modification. Il n'est écrit sur le disque qu'à une ou deux reprises – à cause du thread noyau effectuant la synchronisation par exemple. De même, chaque bloc disque (1 Ko) est touché successivement par quatre écritures de 256 octets, et on gagne largement à le garder en mémoire le plus longtemps possible.

On restreindra donc l'utilisation des écritures synchronisées aux applications qui en ont réellement besoin, avec des contraintes importantes en tolérance de panne. Dans la plupart des cas, un simple appel à `fsync()` en des points-clés du logiciel suffira pour les besoins de fiabilité, tout en conservant un bon temps de réponse à l'application. Rappelons que l'écriture synchronisée garantit uniquement que les données sont parvenues au contrôleur de disque, mais pas qu'elles sont effectivement écrites sur le support physique. Si ce point devient critique – comme dans un système d'enregistrement embarqué –, il faudra choisir avec soin le matériel utilisé afin de minimiser la latence des écritures effectives.

Conclusion

Nous avons examiné ici plusieurs méthodes permettant d'améliorer les entrées-sorties d'un processus, dans le but de rendre les écritures plus fiables (synchronisées) ou de rendre le fil d'exécution principal du programme indépendant des événements survenant sur les descripteurs (multiplexage et entrées-sorties asynchrones).

Les mécanismes de multiplexage sont bien entendu applicables sur les tubes de communication, mais ils sont le plus fréquemment utilisés avec les sockets, qui sont une extension de ces tubes à l'échelle d'un réseau. Les prochains chapitres vont développer les concepts et les principes mis en œuvre dans ce type de logiciel.

26

Programmation réseau

Nous allons essayer dans ce chapitre de mettre en place les bases de la programmation réseau sous Linux, principalement en ce qui concerne la détermination des adresses et des numéros de ports, ainsi que la manipulation de l'ensemble de ces données.

L'essentiel du travail dans la programmation réseau revient en effet à déterminer comment joindre le correspondant. La communication elle-même ne diffère pas beaucoup des méthodes observées dans le chapitre 23 avec les tubes. Ceci sera abordé dans le chapitre suivant, par l'intermédiaire de l'interface proposée par les sockets BSD.

Réseaux et couches de communication

Le but de notre étude est de permettre la mise au point d'applications pouvant recevoir ou envoyer des informations, en dialoguant avec des correspondants se trouvant n'importe où dans le monde, à partir du moment où une connectivité réseau a été établie.

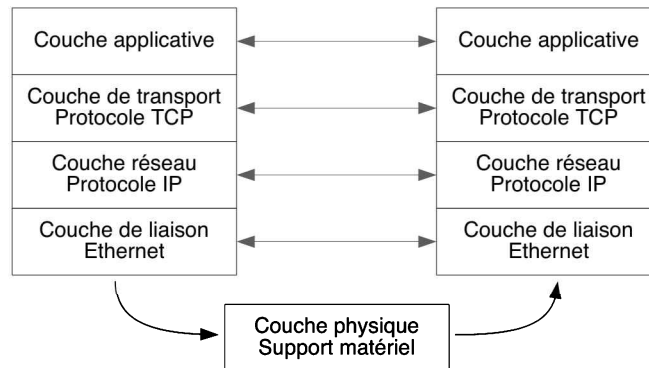
On représente communément les fonctionnalités réseau par une série de couches successives de communication. Ce modèle est intéressant car il permet de bien distinguer la manière dont les différents protocoles sont liés. Chaque couche ne peut dialoguer directement qu'avec la couche supérieure ou inférieure, seule la couche physique peut mettre en relation les différentes stations. Un exemple de stratification réseau est présenté dans la figure 26-1.

Lorsqu'une application désire envoyer des informations à une autre application se déroulant sur un ordinateur distant, elle prépare un paquet de données qu'elle transmet à la couche de transport (TCP sur ce schéma). Celle-ci encadre les données avec ses

propres informations – en l’occurrence des champs servant à s’assurer de l’intégrité du message transmis – puis passe le paquet à la couche réseau, IP. Cette dernière encadre à nouveau le paquet par des informations permettant le routage dans le réseau et passe le relais à la couche de liaison, Ethernet. À ce niveau, les dernières informations ajoutées permettent d’identifier la carte réseau de l’ordinateur cible. Le niveau matériel assure la transmission électrique des données. À l’arrivée, le processus inverse se déroule, chaque couche supprime les éléments qui lui étaient propres et donne le paquet restant à la couche supérieure. Finalement, la couche applicative reçoit les données qui lui étaient destinées. On peut donc considérer que chaque couche dialogue virtuellement avec la couche correspondante sur l’ordinateur cible, bien qu’elle n’ait de véritable contact qu’avec ses couches supérieure et inférieure.

Les noms des différentes couches sont dérivés d’un document de 1984 nommé modèle OSI (*Open Systems Interconnection*), qui sert à représenter les communications réseau avec sept niveaux successifs. Malheureusement les protocoles les plus répandus, TCP/IP et UDP/IP, ne sont pas fondés sur ce modèle et n’emploient que cinq niveaux, comme on le voit sur la figure. Les termes ont été conservés par habitude, mais ils ne conviennent pas tout à fait.

Figure 26-1
Exemple de couches de communication



Le support physique permettant de relier des stations peut prendre des formes diverses. Les plus communes sont les interfaces Ethernet avec une liaison en paires torsadées (prise RJ45) et les liaisons Wifi. Entre deux stations données peuvent se trouver de nombreux éléments, comme des concentrateurs (*hub* ou *switch*), des passerelles qui permettent l’interconnexion de réseaux différents, ou des routeurs qui servent à orienter les données entre plusieurs sous-réseaux. Sur un réseau, les machines sont identifiées de manière unique. Les contrôleurs Ethernet par exemple – qui composent la couche de liaison – comportent un identificateur numérique sur 48 bits, appelé adresse MAC (*Medium Access Control*), dont l’unicité est assurée par le fabricant du contrôleur. Cette valeur peut être examinée à l’aide de l’utilitaire `/sbin/ifconfig` par exemple ¹:

```
$ /sbin/ifconfig eth0
eth0  Lien encap:Ethernet HWaddr 00:50:04:8C:82:5E
      inet adr:172.16.1.51 Bcast:172.16.1.255 Masque:255.255.255.0
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      Paquets Reçus:117 erreurs:0 jetés:0 débordements:0 trames:0
      Paquets transmis:66 erreurs:0 jetés:0 débordements:0 carrier:0
      collisions:0 lg file transmission:100
      Interruption:3 Adresse de base:0x200
$
```

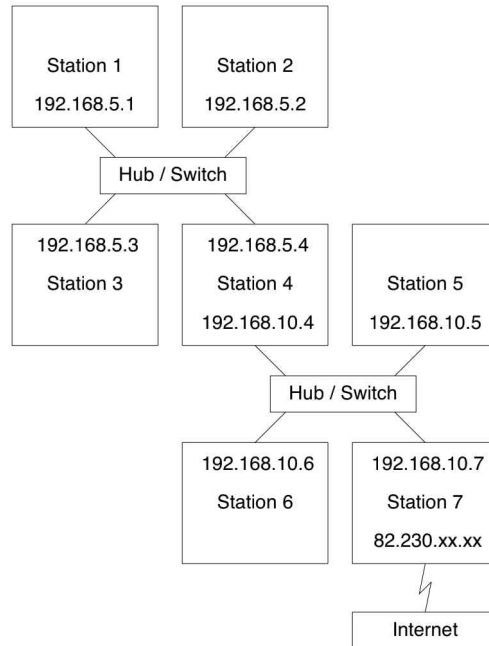
L'identificateur MAC (00:50:04:8C:82:5E en l'occurrence) est indiqué dans la rubrique *Hardware Address*. Une machine est capable, grâce à son adresse, de reconnaître si un bloc de données lui est destiné et de le transmettre aux protocoles de dialogue se trouvant au-dessus. On trouvera au besoin plus de renseignements dans [Ferrero 1993] *Les réseaux Ethernet*.

Plusieurs protocoles peuvent être employés pour transmettre des informations au-dessus de cette couche de liaison, mais l'essentiel des communications au niveau applicatif se fait en employant IP (*Internet Protocol*). Le protocole IP permet d'envoyer un paquet de données à destination d'un hôte particulier, en l'identifiant à l'aide d'une adresse sur 4 octets. Celle-ci est presque toujours représentée avec la notation dite « pointée », c'est-à-dire en écrivant les valeurs décimales des octets séparées par des points. Dans l'exemple précédent, `/sbin/ifconfig` affichait l'adresse IP de l'interface `eth0` (172.16.1.51) avec le titre *Internet Address*. Le protocole actuel IP version 4 sera remplacé peu à peu par IP version 6 (il n'existe pas de version 5) où les adresses sont représentés par 16 octets séparés par des deux-points.

Au niveau du protocole IP, les données peuvent être routées. La communication n'est plus limitée aux machines se trouvant sur le même réseau matériel². Au contraire, il existe des passerelles permettant de transférer les paquets d'un réseau vers un autre. La figure 26-2 montre un exemple de réseaux reliés entre eux. Un premier sous-réseau (192.168.5.x) est connecté à un second sous-réseau (192.168.10.x) par l'intermédiaire d'une passerelle (la station 4). Chaque machine peut dialoguer avec toutes les autres, par le biais de la couche IP. En outre, une machine spécifique (la station 7) assure le rôle de passerelle vers Internet (via un fournisseur d'accès et une ligne ADSL, par exemple).

1. L'utilitaire `ifconfig`, simple et bien connu de tous les utilisateurs de Linux, est désormais obsolète et remplacé peu à peu par la commande `ip`, plus générale mais à mon avis plus compliquée à mémoriser.
2. La mise en correspondance entre l'adresse IP et l'adresse MAC se fait par l'intermédiaire d'un protocole nommé ARP (*Address Resolution Protocol*) sortant du cadre de notre propos.

Figure 26-2
Exemple de passerelles entre réseaux



Par exemple le noyau de la station 1 sait, grâce à ses tables de routage configurées avec l'utilitaire `route`, que pour atteindre une machine ne se trouvant pas directement sur son sous-réseau, par exemple la station 5, il doit demander le relais à la station 4. Les paquets transmis à la couche de liaison seront donc dirigés vers l'adresse MAC de cette machine 4. Toutefois, l'adresse IP du destinataire sera celle de la station 5. La couche de liaison s'occupe uniquement de l'adresse Ethernet et pas de l'adresse IP.

La station 4 dispose de deux cartes Ethernet. Son noyau est configuré grâce à `iptables` pour laisser passer les paquets d'un réseau à l'autre. Lorsqu'un bloc de données arrive sur une carte réseau à destination d'un autre sous-réseau, celle-ci assure le transfert d'une interface à l'autre.

Quant à la station 4, elle offre un accès à Internet pour les deux sous-réseaux précédents. Elle sert aussi de passerelle, mais comme les machines des réseaux 192.168.x.x ne sont pas connues directement sur Internet, la configuration est légèrement plus compliquée car il faut employer un mécanisme de traduction d'adresse (NAT pour *Network Address Translation*). On trouvera des renseignements sur toutes ces notions de routage dans [Kirch 2001] *L'administration réseau sous Linux*, et dans le document *Linux-Networking-HOWTO*.

Ce qu'on retiendra ici, c'est que le protocole IPv4 est capable d'envoyer un paquet de données à destination d'un hôte précis, dont l'adresse est indiquée par 4 octets, en franchissant les éléments de routage.

Les adresses IP sont organisées en sous-réseaux, afin de simplifier les routages. Chaque sous-réseau possède une adresse et chaque machine a elle-même une adresse au sein du sous-réseau. L'adresse IP complète est obtenue en faisant suivre l'adresse du sous-réseau par celle de la station. Ainsi, si on a un sous-réseau 192.1.1, sa station numéro 2 aura l'adresse 192.1.1.2. Cette organisation permet à un routeur de savoir sur quel brin Ethernet se trouve un hôte sous sa responsabilité.

Les sous-réseaux définis par le NIC (*Network Information Center*), qui gère les adresses sur Internet, sont répartis en plusieurs catégories :

- Les 127 sous-réseaux de classe A ont des adresses allant de 1. à 127.. Chacun d'eux peut adresser ses stations sur 24 bits, ce qui leur permet de regrouper plus d'un million de machines.
- Les adresses des sous-réseaux de classe B vont de 128.0. à 191.255.. Il y en a plus de 16 000, chacun pouvant contenir des machines avec des adresses sur 16 bits. Il y a un peu plus de 65 000 stations adressables par réseau.
- Les sous-réseaux de classe C s'étendent de 192.0.0. à 223.255.255.. Chacun de ces deux millions de sous-réseaux peut contenir 254 hôtes, car les adresses .0 et .255 ne sont pas utilisables.
- La classe D s'étend de 224.0.0.0 à 239.255.255.255. Il ne s'agit pas d'adresses de machines, mais uniquement d'adresses de diffusion *multicast*. Nous détaillerons ce concept dans le prochain chapitre.

Il existe d'autres classes pour les adresses supérieures à 240., mais il s'agit uniquement d'utilisations expérimentales. On notera l'existence également de réseaux privés : 10. pour la classe A, 172.16. pour la classe B et 192.168. pour la classe C sont des réseaux qui ne sont jamais routés vers Internet. Ils sont utilisés pour la plupart des réseaux domestiques ou d'entreprise en amont de la passerelle (*box*) donnant accès à Internet via un mécanisme NAT.

L'utilisation directe du protocole IP est plutôt rare au niveau d'une application. On peut l'employer pour envoyer des messages de commande appartenant au protocole ICMP (*Internet Control Message Protocol*), comme les demandes d'écho émises par l'utilitaire ping. Néanmoins, la plupart du temps on fera appel à une couche supérieure. Nous étudierons ici les deux protocoles TCP (*Transmission Control Protocol*) et UDP (*User Datagram Protocol*), qui ont des rôles complémentaires.

Le protocole TCP sert à fiabiliser la communication entre deux hôtes. Pour cela, il assure les fonctionnalités suivantes :

- Connexion : avec ce protocole, une liaison s'établit par une concertation de l'émetteur et du récepteur. On dit que la communication s'effectue de manière connectée. Une fois le canal de communication établi, il reste en vigueur jusqu'à ce qu'on le referme.

- **Fiabilité** : le protocole TCP garantit que – tant que la connexion sera valide – les données qui y transitent arriveront dans l'ordre et que leur intégrité sera vérifiée.
- **Contrôle de flux** : en complément de la fiabilité du protocole TCP, il est possible de l'employer comme un flux d'octets, à la manière d'un tube de communication. L'écriture peut devenir bloquante si le récepteur ne lit pas suffisamment vite de son côté.

À l'opposé, le protocole UDP fournit un service de transmission de paquets (*datagram*) sans assurer de fiabilité :

- **Pas de connexion** : l'émetteur peut envoyer des données sans s'assurer qu'un processus est à l'écoute. Aucun acquittement n'est nécessaire.
- **Pas de fiabilité** : le paquet transmis ne contient qu'optionnellement une somme de contrôle. Si des données sont perdues ou erronées, elles ne sont pas répétées.
- **Transmission en paquets** : les données envoyées n'arrivent pas nécessairement dans le même ordre qu'au départ (bien que ce cas ne se rencontre apparemment jamais en pratique).

En fait, le protocole IP brut offre les mêmes fonctionnalités que UDP, à la somme de contrôle près, qui n'existe pas dans la couche réseau (en fait IP vérifie l'intégrité de ses propres informations mais pas celles des données du paquet). L'implémentation de TCP effectue donc de nombreuses tâches afin d'assurer un mécanisme de communication fiable. Entre autres, TCP vérifie l'état des paquets qui arrivent, s'assure qu'ils sont dans le bon ordre au moyen d'un numéro de séquence, gère un délai maximal de transmission, des acquittements, etc. Si un paquet est endommagé ou absent, la couche TCP du destinataire demande à la couche TCP de l'émetteur de renvoyer les données. Tout ceci offre donc une sécurité de transmission des informations mais au prix d'une charge réseau supplémentaire.

Le protocole UDP de son côté permet d'envoyer des paquets de données sans se soucier vraiment du récepteur. Ceci est particulièrement utile dans les applications qui veulent diffuser des informations sous forme de fonctionnalité annexe du logiciel. Il n'est pas question dans ce cas de perdre du temps à gérer les connexions des correspondants ni de risquer de rester bloqué si le récepteur ne lit pas assez vite. L'émetteur peut envoyer ses paquets de données et passer immédiatement à autre chose, il est de la responsabilité du récepteur d'être à l'écoute au bon moment.

Résolution de nom

Un élément essentiel de la programmation réseau est la conversion des informations humainement compréhensibles (comme une URL) en données interprétables par l'ordinateur. L'opération inverse est également nécessaire pour afficher par exemple au propriétaire d'un serveur des informations sur ses clients.

Pendant longtemps, cette résolution s'appuyait sur plusieurs fonctions dont certaines sont désormais considérées comme obsolètes. De nos jours, la résolution de nom repose sur une unique fonction :

```
int getaddrinfo(const char *host, const char *service,
               const struct addrinfo *hints,
               struct addrinfo **result);
```

Elle traduit les chaînes de caractères `host` et `service` décrivant respectivement une machine et un service TCP ou UDP en une liste de structures `addrinfo` contenant toutes les informations nécessaires pour démarrer un serveur ou établir une connexion. La fonction inverse existe, il s'agit de `getnameinfo()` qui convertit les informations contenues dans une structure `sockaddr` en chaînes de caractères intelligibles pour un utilisateur.

Concentrons-nous tout d'abord sur `getaddrinfo()`. L'entité principale qu'elle manipule est la structure `addrinfo` dont les champs sont décrits ci-après. La première structure `hints` contient des indications sur le type d'information que nous désirons, la seconde structure `result` reçoit le résultat de la résolution de nom.

L'argument `host` de `getaddrinfo()` contient le nom de la machine recherchée. Il peut s'agir d'un nom d'hôte, mais également d'une adresse numérique au format IPv4 ou IPv6. Dans ce cas, il est conseillé d'indiquer `AI_NUMERICHOST` dans le champ `ai_flags` de la structure `hints` pour simplifier la recherche.

L'argument `service`, s'il est différent de `NULL`, est une chaîne de caractères contenant un numéro de port ou le nom d'un service comme HTTP. Si nous sommes certains que la chaîne contient un numéro de port, on peut indiquer `AI_NUMERICSERV` dans le champ `ai_flags` de `hints`.

En retour, `getaddrinfo()` renvoie zéro si tout s'est bien passé et un code d'erreur sinon. Ce code d'erreur peut être traduit en un message intelligible par `gai_strerror()` :

```
const char * gai_strerror(int err);
```

La fonction `getaddrinfo()` alloue une liste chaînée de structures `addrinfo` qu'elle nous renvoie dans son argument `result`. Bien entendu, après utilisation, il nous faudra libérer cette liste de structures, ce qui est réalisé en passant le pointeur `result` à la fonction `freeaddrinfo()`.

```
void freeaddrinfo(struct addrinfo *result);
```

Lors de l'appel de `getaddrinfo()`, seuls les champs `ai_flags`, `ai_family`, `ai_socktype` et `ai_protocol` sont pris en considération dans la structure `hints` précisant notre recherche. Il est néanmoins important, pour des raisons d'évolution et de portabilité, d'initialiser à zéro tous les autres champs (ou `NULL` pour les pointeurs). Les significations des champs pris en compte sont les suivantes :

Nom	Type	Signification
<code>ai_family</code>	<code>int</code>	Famille d'adresses souhaitée (par exemple, <code>AF_INET</code> ou <code>AF_INET6</code> pour IPv4 et IPv6). La valeur <code>AF_UNSPEC</code> signifie « n'importe quelle famille d'adresses ».
<code>ai_socktype</code>	<code>int</code>	Type de socket que nous souhaitons utiliser en relation avec cette adresse, par exemple <code>SOCK_STREAM</code> (communication connectée comme TCP/IP) ou <code>SOCK_DGRAM</code> (mode non connecté comme UDP/IP). Ces valeurs seront décrites plus en détail dans le prochain chapitre. On utilise 0 pour indiquer « n'importe quel type ».
<code>ai_protocol</code>	<code>int</code>	Le protocole utilisé comme <code>IPPROTO_TCP</code> , <code>IPPROTO_UDP</code> , etc. Nous en reparlerons plus en détail dans le prochain chapitre. La plupart du temps, on indique 0, signifiant « n'importe quel protocole compatible avec la famille et le type de socket indiqués précédemment ».
<code>ai_flags</code>	<code>int</code>	Plusieurs paramètres peuvent être associés ici en utilisant un OU binaire : <code>AI_NUMERICHOST</code> (l'argument <code>node</code> contient une adresse numérique plutôt qu'un nom d'hôte), <code>AI_NUMERICSERV</code> (l'argument <code>service</code> contient un numéro de port, et non pas un nom de service), <code>AI_PASSIVE</code> (l'adresse demandée sera utilisée pour y attacher un serveur avec <code>bind()</code> que nous verrons dans le prochain chapitre), <code>AI_CANONNAME</code> (nous voulons obtenir le nom canonique officiel de l'hôte demandé), etc.

Au retour de `getaddrinfo()`, notre pointeur `result` est initialisé au début d'une liste chaînée de toutes les structures `addrinfo` correspondant à notre demande. La liste est chaînée à l'aide du champ `ai_next` :

```
ai_next      struct addrinfo * Structure addrinfo suivante de la liste renvoyée par
                                getaddrinfo().
```

En effet, il peut y avoir plusieurs réponses à notre demande de résolution de nom : la même machine peut être accessible par l'intermédiaire de deux familles d'adresses comme IPv4 et IPv6. Elle peut aussi proposer le même service suivant plusieurs protocoles comme TCP/IP et UDP/IP, ou encore le même hôte peut être relié à plusieurs réseaux différents. La liste est renvoyée triée suivant un algorithme (décrit dans le document *Request For Comments 3484*) qui présente les meilleures réponses en premier. On essaiera donc d'utiliser les résultats dans l'ordre de la liste.

Les champs remplis seront les suivants :

ai_addrlen	socklen_t	Taille de la structure d'adresse (dépend de la famille indiquée dans ai_family).
ai_addr	struct sockaddr *	Pointeur vers la structure d'adresse. La struct sockaddr est un type générique. Le pointeur correspond en réalité à une structure spécifique à la famille d'adresses comme sockaddr_in pour AF_INET.
ai_canonname	char *	Nom « canonique » de l'hôte. Ce champ est rempli si l'attribut AI_CANONNAME était présent dans le champ ai_flags de hints.

Avant d'examiner notre premier exemple, nous devons nous intéresser à getnameinfo() qui nous permettra de présenter de manière compréhensible les résultats de getaddrinfo().

```
int getnameinfo(const struct sockaddr *saddr, socklen_t socklen,
                char *host, size_t hostlen,
                char *service, size_t servicelen,
                int flags);
```

La fonction getnameinfo() prend en premier argument un pointeur sur une structure sockaddr, comme celui fourni dans le champ ai_addr de la structure addrinfo. Ce pointeur est en réalité forcé (par un cast) vers une structure spécifique correspondant à la famille d'adresses (AF_INET, AF_INET6, AF_UNIX, etc.). C'est pourquoi on fournit en second paramètre la longueur de la structure d'adresses, que l'on a obtenue par exemple dans le champ ai_addrlen de la structure addrinfo.

Le troisième paramètre est une chaîne de caractères qui sera remplie pendant l'appel avec le nom de l'hôte. Cette chaîne doit être préallouée et sa longueur indiquée dans le quatrième argument. Pour être sûr d'utiliser une longueur de chaîne raisonnable, la bibliothèque C définit une constante symbolique (dans <netdb.h>) nommée NI_MAXHOST. Sous Linux, elle vaut 1025.

De même, les cinquième et sixième champs sont des chaînes de caractères et leur longueur (par exemple, NI_MAXSERV qui vaut 32 sous Linux) permet de recevoir le nom du service concerné. Naturellement, si le nom d'hôte ou le nom de service ne nous intéresse pas, nous fournissons un pointeur NULL et une longueur nulle.

Dans le dernier argument, on peut fournir une ou plusieurs constantes :

NI_NUMERICHOST	C'est la forme numérique de l'adresse qui nous intéresse.
NI_NUMERICSERV	Nous souhaitons la représentation numérique du service.
NI_DGRAM	Nous souhaitons le nom du service basé sur datagramme (UDP plutôt que TCP, par exemple).

NI_NAMEREQD	Nous voulons obtenir le nom d'hôte, sinon la fonction doit renvoyer une erreur.
NI_NOFQDN	Pour la machine du domaine local, nous ne voulons que la partie nom d'hôte de l'adresse.

Voici un exemple très général où notre programme affiche les informations sur les hôtes dont les noms ou adresses IP ont été indiqués sur la ligne de commande.

```
exemple-getaddrinfo-01.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>

int main (int argc, char * argv [])
{
    char host[NI_MAXHOST];
    struct addrinfo *result;
    struct addrinfo *info;
    struct addrinfo hints;
    int i;
    int err;

    for (i = 1; i < argc; i++) {
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET;
        fprintf(stdout, "%s : ", argv[i]);
        err = getaddrinfo(argv[i], NULL, &hints, &result);
        if (err != 0) {
            fprintf(stdout, "ERROR: %s\n", gai_strerror(err));
            continue;
        }
        for (info = result; info != NULL; info = info->ai_next) {
            getnameinfo(info->ai_addr, info->ai_addrlen,
                        host, sizeof(host), NULL, 0, 0);
            fprintf(stdout, "%s ", host);
        }
        freeaddrinfo(result);
        fprintf(stdout, "\n");
    }
    return EXIT_SUCCESS;
}
```

Voici quelques exemples d'utilisation avec tout d'abord un nom d'hôte et une adresse dans le domaine local :

```
$ ./exemple-getaddrinfo-01 LabPC3 192.168.3.15
LabPC3 : LabPC3 LabPC3 LabPC3
192.168.3.15 : 192.168.3.15 192.168.3.15 192.168.3.15
```

La machine 192.168.3.15 n'a pas de nom, c'est donc une simple adresse IP qui est renvoyée. Essayons à présent avec des adresses et noms distants, qui nécessitent la consultation de serveurs DNS :

```
$ ./exemple-getaddrinfo-01 www.kernel.org 192.30.252.120
www.kernel.org : yul-korg-pub.kernel.org yul-korg-pub.kernel.org
yul-korg-pub.kernel.org pub2.kernel.org pub2.kernel.org
pub2.kernel.org tiz-korg-pub.kernel.org tiz-korg-pub.kernel.org tiz-
korg-pub.kernel.org
192.30.252.120 : github.com github.com github.com
```

Les deux hôtes existent, on voit que l'adresse de kernel.org renvoie une liste de structures `addrinfo` qui représentent des machines différentes. Pour répartir la charge réseau, l'ordre de cette liste change à chaque appel.

Plusieurs structures sont renvoyées dans les appels précédents car plusieurs types de sockets et de protocoles peuvent être employés avec ces adresses. En examinant plus en détail les structures `addrinfo`, nous verrons des différences dans les champs `ai_socktype` et `ai_protocol`.

Nous pouvons examiner un exemple de la version Python de ces fonctions : il s'agit des fonctions `socket.getaddrinfo()` et `socket.getnameinfo()`. La première renvoie une liste de n-uplets et la seconde un couple de chaînes de caractères :

```
exemple-getaddrinfo-01.py :
#!/usr/bin/python
from __future__ import print_function
import socket
import sys

sys.argv.pop(0)
for var in sys.argv:
    try:
        print(var, end=': ')
        result = socket.getaddrinfo(var, 0, socket.AF_INET, 0, 0)
    except:
        print("Error")
        continue
    for (family, socktype, proto, canon, sockaddr) in result:
        (host, port) = socket.getnameinfo(sockaddr, 0)
        print(host, end=' ')
    print('')
```

Les résultats d'exécution sont exactement les mêmes que les précédents.

Services et numéros de ports

Lorsqu'un logiciel désire converser avec un correspondant qui est sur une autre machine, nous avons vu que le protocole IP, se trouvant sous les couches TCP ou UDP, permet de transmettre un paquet de données vers la station cible.

Toutefois, il peut y avoir beaucoup d'applications différentes qui fonctionnent simultanément sur la machine visée, et plusieurs d'entre elles peuvent offrir des fonctionnalités réseau. Il faut donc trouver le moyen de préciser quel correspondant nous désirons atteindre parmi les processus tournant sur l'ordinateur récepteur. Ceci est assuré par une fonctionnalité de la couche IP : les numéros de ports.

Chaque application voulant utiliser les services au-dessus de la couche IP se verra affecter un numéro de port, c'est-à-dire un entier sur 16 bits qui permettra d'identifier le canal de communication au sein de la machine. C'est ici que se différencient les fonctionnalités de transmission de la couche IP et celles de la couche UDP. Lorsqu'on envoie un paquet UDP, on lui affecte une adresse de destination mais également un numéro de port. La couche IP de la machine réceptrice passera les données à l'application associée à ce port.

Le processus émetteur est lui aussi doté d'un numéro de port, qui est d'ailleurs inscrit dans l'en-tête du paquet transmis, mais on s'intéresse généralement au numéro de port du récepteur plutôt qu'à celui de l'émetteur.

Les numéros de ports inférieurs à 255 sont strictement réservés à des services bien définis, disponibles sur de nombreux systèmes. Par exemple, le port 80 est réservé au service HTTP (*Hyper Text Transfer Protocol*), c'est-à-dire aux serveurs web habituels. On peut se connecter directement avec l'utilitaire `telnet` en lui précisant le numéro de port.

```
$ telnet www.logilin.fr 80
Trying 217.16.1.202...
Connected to www.logilin.fr.
Escape character is '^]'.
GET http://www.logilin.fr HTTP/1.0 (Entrée)
(Entrée)
HTTP/1.1 200 OK
[...]
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="fr">
<head>
  <title>Logilin - Formations et conseil sur Linux et les logiciels
  libres</title>
  [...]
</body>
</html>
```



```
Connection closed by foreign host.
$
```

Le texte renvoyé par le serveur est le contenu brut de la page HTML d'accueil du site, sans la mise en forme qui est habituellement réalisée par le navigateur web (Firefox, etc.).

Les numéros de ports inférieurs à 1024 ne peuvent être associés qu'à des processus ayant un UID effectif nul ou la capacité `CAP_NET_BIND_SERVICE`. Ceci permet à un correspondant de savoir qu'il a bien affaire à un service officiel de la machine cible et pas à une application pouvant être faussée par un pirate. Les autres numéros peuvent être employés par n'importe quel utilisateur.

Précisons bien que les numéros de ports TCP et les numéros de ports UDP sont tout à fait distincts. On peut rencontrer simultanément des canaux de communication TCP et UDP ayant le même numéro tout en étant totalement indépendants.

On doit donc préciser pour joindre un processus distant :

- l'adresse IP de la machine sur laquelle il s'exécute ;
- le numéro de port de réception ;
- le protocole (UDP ou TCP) employé.

Lorsqu'une machine dispose de plusieurs interfaces réseau (une ou plusieurs cartes), les numéros de ports sur chaque interface sont également indépendants.

Pour connaître l'association entre un numéro de port et un service particulier, on peut consulter le fichier `/etc/services`. Celui-ci contient les numéros attribués à plusieurs milliers de services. Ce fichier est consultable par tous les utilisateurs :

```
$ cat /etc/services
# /etc/services:
# $Id: services,v 1.49 2010/03/26 13:06:35 ovasik Exp $
[...]
tcpmux    1/tcp                # TCP port service multiplexer
tcpmux    1/udp                # TCP port service multiplexer
rje       5/tcp                # Remote Job Entry
rje       5/udp                # Remote Job Entry
echo      7/tcp
echo      7/udp
[...]
http      80/tcp    www www-http    # WorldWideWeb HTTP
http      80/udp    www www-http    # HyperText Transfer Protocol
[...]
$
```

Sur certains systèmes, le fichier `/etc/services` est complété par des données provenant d'un serveur NIS, accessible avec `yycat -k services`.

On remarque que certains services offrent à la fois une interface en TCP et en UDP, alors que d'autres ne travaillent que dans un seul mode. Il peut exister des alias, par exemple le service `http` vu précédemment peut être aussi invoqué sous le nom `www`. Certains services (comme `http`) réservent les ports TCP et UDP simultanément, bien qu'ils n'emploient que l'interface TCP. Ceci permet d'éviter les confusions de services qui auraient le même numéro de port sur deux protocoles différents. Les trois lignes suivantes sont équivalentes.

```
$ telnet www.logilin.fr 80
$ telnet www.logilin.fr http
$ telnet www.logilin.fr www
```

Pour ouvrir une communication avec un serveur, il est nécessaire de connaître le numéro de port à utiliser. Toutefois, on ne peut pas demander à l'utilisateur d'indiquer lui-même la valeur numérique. Il faut l'autoriser à employer un mot-clé indiqué dans le fichier `/etc/services`, même s'il s'agit d'une application locale.

Notre second programme employant `getaddrinfo()` et `getnameinfo()` va afficher les numéros des services dont on lui passe les noms sur la ligne de commande. Les seules différences par rapport au programme précédent sont :

```
exemple-getaddrinfo-02.c :
[...]  
int main (int argc, char * argv [])  
{  
    char service[NI_MAXSERV];  
    [...]  
    err = getaddrinfo(NULL, argv[i], &hints, &result);  
    [...]  
    getnameinfo(info->ai_addr, info->ai_addrlen,  
                NULL, 0, service, NI_MAXSERV, NI_NUMERICSERV);  
    fprintf(stdout, "%s ", service);  
    [...]  
}
```

Voici un exemple avec quelques noms (et numéros) de services :

```
$ ./exemple-getaddrinfo-02 http ftp ssh 8080 inexistant  
http : 80  
ftp : 21  
ssh : 22  
8080 : 8080  
inexistent : ERROR: Servname not supported for ai_socktype
```

Le script Python conduisant au même résultat est :

```
exemple-getaddrinfo-02.py :
#!/usr/bin/python
from __future__ import print_function
import socket
import sys

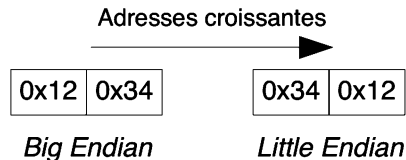
sys.argv.pop(0)
for var in sys.argv:
    try:
        print(var, end=': ')
        result = socket.getaddrinfo(None, var, socket.AF_INET, 0, 0)
    except:
        print("Error")
        continue
    for (family, socktype, proto, canon, sockaddr) in result:
        (host, port) = socket.getnameinfo(sockaddr, socket.NI_NUMERICSERV)
        print(port, end=' ')
    print('')
```

Nous verrons d'autres exemples d'utilisation de `getaddrinfo()` et `getnameinfo()` dans le prochain chapitre, aussi bien pour installer un serveur que pour connecter un client.

Ordre des octets

Les communications et les échanges de données entre ordinateurs hétérogènes sont souvent confrontés au problème d'ordre des octets dans les valeurs entières. Pour stocker en mémoire une valeur tenant sur 2 octets, certains processeurs placent en première position l'octet de poids faible, puis celui de poids fort. Comme les données commencent par leur plus petite extrémité, cette organisation est nommée *Little Endian*. À l'opposé, il existe des machines rangeant d'abord l'octet de poids fort, suivi de celui de poids faible. On les qualifie de *Big Endian*.

Figure 26-3
Stockage en mémoire
de la valeur 0x1234



Le programme suivant affiche la représentation en mémoire d'une valeur entière.

```
ordre-octets.c :
#include <stdio.h>
```

```
#include <stdlib.h>

int main (int argc, char * argv[])
{
    unsigned short int s_i;
    unsigned char *   ch;
    int               i;

    if ((argc != 2) || (sscanf (argv[1], "%hi", & s_i) != 1)) {
        fprintf (stderr, "Syntaxe : %s entier \n", argv[0]);
        exit (EXIT_FAILURE);
    }
    ch = (unsigned char *) & s_i;
    fprintf(stdout, "0x%04X représente ainsi ", s_i);
    for (i = 0; i < sizeof(short int); i++)
        fprintf(stdout, "%02X ", ch[i]);
    fprintf(stdout, "\n");
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution sur une machine *Little Endian*, un PC en l'occurrence :

```
$ ./ordre-octets 0x1234
0x1234 représente ainsi 34 12
$
```

Et à présent sur un processeur *SH4Big Endian*, on obtient :

```
$ ./ordre-octets 0x1234
0x1234 représente ainsi 12 34
$
```

La différence peut même être encore plus accentuée avec des données sur 32 bits, car la valeur 0x12345678 peut être stockée sous 4 formes : 1234 5678, 5678 1234, 3412 7856, ou 7856 3412. Naturellement, la première et la dernière sont les plus fréquentes, mais rien n'interdit l'existence des autres.

Lorsque des données binaires doivent être écrites sur une machine et relues sur une autre, ce problème peut compliquer sérieusement le travail du développeur car cela interdit entre autres l'emploi de `fwrite()` et de `fread()`. Toutefois le problème reste au niveau de l'application, qui peut utiliser différentes techniques pour y remédier³.

3. Par exemple stocker une valeur connue comme 0x1234 en début de fichier et la lire pour déterminer l'ordre employé lors de l'enregistrement, c'est la méthode utilisée dans plusieurs formats graphiques.

Là où la situation peut devenir vraiment gênante c'est lorsque des valeurs numériques sont employées dans les zones de données du protocole réseau lui-même. Par exemple, la couche IP utilise une valeur numérique entière pour coder le protocole employé par la couche supérieure. Lorsque la machine de destination reçoit un paquet de données au niveau de la couche IP, il faut qu'elle puisse décoder le numéro de protocole pour transmettre les données à la bonne couche de transport, UDP en l'occurrence. Le fait de devoir déterminer à chaque paquet l'ordre des octets de la machine émettrice est une surcharge de travail inacceptable.

La solution employée dans les protocoles fondés sur IP – et d'autres comme XNS – consiste à figer l'ordre des octets dans tous les en-têtes des paquets de données circulant sur le réseau. La forme retenue est *Big Endian*. Cela signifie que chaque machine doit convertir éventuellement l'ordre des données. L'avantage est que la bibliothèque C connaît à la compilation l'ordre des octets sur la station où elle est installée, et qu'elle n'a pas de questions à se poser : soit il faut toujours convertir, soit il ne le faut jamais.

Pour le programmeur, cela implique toutefois une opération supplémentaire. Toutes les valeurs numériques qui seront transmises au protocole réseau devront passer par une étape de conversion éventuelle.

Attention, nous parlons bien des valeurs transmises au protocole réseau. Les valeurs contenues dans les données de l'application ne sont pas concernées, quoique rien n'interdise d'employer le même mécanisme.

La bibliothèque C met à notre disposition quatre fonctions permettant de transformer un entier long ou court depuis l'ordre des octets de l'hôte vers celui du réseau, et inversement. Ces routines sont déclarées dans `<netinet/in.h>` :

```
unsigned long int htonl (unsigned long int valeur);
unsigned short int htons (unsigned short int valeur);
unsigned long int ntohl (unsigned long int valeur);
unsigned short int ntohs (unsigned short int valeur);
```

Les fonctions `htonl()` et `htons()` convertissent respectivement depuis l'ordre des octets de l'hôte (h) vers (*to*) celui du réseau (*network*) des entiers long (l) et court (*shorts*). Parallèlement, `ntohl()` et `ntohs()` convertissent les entiers depuis l'ordre des octets du réseau vers celui de l'hôte. Nous emploierons les conversions d'entiers longs pour les adresses IP (qui sont sur 32 bits en version 4), et les conversions courtes pour les numéros de ports que nous allons voir dans la prochaine section.

Il faut noter que ces fonctions étaient très utiles lorsque le programmeur devait remplir « manuellement » le contenu de la structure `sockaddr` avec des données provenant de l'interface utilisateur ou, inversement, les décoder pour présenter les informations sur les communications en cours. Depuis l'apparition de `getaddrinfo()` et

`getnameinfo()`, ce travail est plus rare car une grande partie des conversions est automatiquement réalisée dans ces routines. Nous pouvons le vérifier dans le résultat du programme exemple `_getaddrinfo()` : les numéros sont correctement affichés, quelle que soit la machine utilisée.

Conclusion

Nous avons examiné dans ce chapitre l'essentiel des moyens d'accès aux bases de données permettant la résolution de noms – tant au niveau des hôtes que des services et des protocoles. Une grosse partie du travail dans les programmes réseau repose sur ces routines.

Pour obtenir plus d'informations, on peut consulter bien entendu [Stevens 1990] *Unix Network Programming* – qui est un grand classique de ce domaine.

En ce qui concerne l'installation et l'administration d'un réseau sous Linux, on conseillera à nouveau le *Linux-Networking-HOWTO*.

Enfin, on trouvera ci-dessous quelques références des documents RFC définissant les principaux protocoles utilisés sur Internet.

Numéro RFC	Auteur et date	Sujet
RFC 791	J. Postel, 01/09/1981	IP : Internet Protocol.
RFC 792	J. Postel, 01/09/1981	ICMP : Internet Control Message Protocol.
RFC 793	J. Postel, 01/09/1981	TCP : Transmission Control Protocol.
RFC 768	J. Postel, 28/08/1980	UDP : User Datagram Protocol.
RFC 959	J. Postel, J. Reynolds, 01/10/1985	FTP : File Transfert Protocol.
RFC 783	K.R.Sollins, 01/06/1981	TFTP : Trivial File Transfert Protocol.
RFC 821	J. Postel, 01/08/1982	SMTP : Simple Mail Transfert Protocol.
RFC 977	B. Kantor, 01/02/1986	NNTP : Network News Transfer Protocol.
RFC 854	J. Postel, J. Reynolds, 01/05/1983	Telnet Protocol.
RFC 1918	Y. Rekhter et al. 01/02/1996	Adresses IP utilisables sur un réseau privé.
RFC 2500	J. Reynolds, 01/06/1999	Protocoles standard sur Internet.
RFC 1700	J.Reynolds, J. Postel, 01/10/1994	Noms et numéros standard pour tout ce qui concerne les communications sur Internet.

À présent que nous savons déterminer l'adresse d'un correspondant et le service qui nous intéressent, nous allons pouvoir enfin établir la communication avec un processus distant dans le prochain chapitre.

Utilisation des sockets

Concept de socket

Les sockets¹ sont apparues dans 4.2BSD, en 1983. Elles sont à présent disponibles sur tous les Unix courants, et il en existe des variantes sur les autres principaux systèmes d'exploitation. Il s'agit approximativement d'une extension de la portée des tubes nommés, pour pouvoir faire dialoguer des processus s'exécutant sur différentes machines.

On peut donc écrire des données dans une socket après l'avoir associée à un protocole de communication, et les couches réseau des deux stations s'arrangeront pour que les données ressortent à l'autre extrémité. La seule complication introduite par rapport aux tubes classiques est la phase d'initialisation, car il faut indiquer l'adresse et le numéro de port du correspondant. Une fois que la liaison est établie, le comportement ne sera pas très différent de ce qu'on a étudié dans le chapitre 23.

Les sockets sont représentées dans un programme par des entiers, comme les descripteurs de fichiers. On peut leur appliquer les appels système usuels, `read()`, `write()`, `select()`, `close()`, etc. Nous verrons dans ce chapitre les primitives spécifiques qui s'appliquent aux sockets.

1. Le mot socket se traduit par « prise » en français (dans le sens de « prise électrique »), mais j'utiliserai le mot original, qui est devenu un terme consacré.

Création d'une socket

La première étape consiste à créer une socket. Ceci s'effectue à l'aide de l'appel système `socket()`, défini dans `<sys/socket.h>` :

```
| int socket (int domaine, int type, int protocole);
```

En Python, l'appel est identique :

```
socket.socket(domaine, type, protocole)
```

Le premier argument de cette routine est le domaine de communication. Il s'agit d'une constante symbolique pouvant prendre plusieurs valeurs. En voici quelques exemples :

- `AF_INET` : protocole fondé sur IP ;
- `AF_INET6` : protocole IPv6 soumis à des options de compilation dans le noyau ;
- `AF_UNIX` : communication limitée aux processus résidant sur la même machine. Dans certains cas cette constante est remplacée par le synonyme `AF_LOCAL`, qui appartient d'ailleurs à la terminologie Posix.

Nous ne nous intéresserons ici qu'au domaine `AF_INET`, qui regroupe toutes les communications réseau avec IP, TCP, UDP ou ICMP.

Le second argument est le type de socket. Nous ne considérerons que trois cas :

- `SOCK_STREAM` : le dialogue s'effectue en mode connecté, avec un contrôle de flux d'une extrémité à l'autre de la communication ;
- `SOCK_DGRAM` : la communication a lieu sans connexion, par transmission de paquets de données ;
- `SOCK_RAW` : la socket sera utilisée pour dialoguer de manière brute avec le protocole.

Nous reviendrons ultérieurement sur les communications en mode connecté ou non. Finalement, le troisième argument indique le protocole désiré. Il s'agit du champ `p_proto` de la structure `protoent` examinée dans le chapitre précédent. Si on indique une valeur nulle, les combinaisons suivantes seront automatiquement réalisées.

Domaine	Type	Socket obtenue	Protocole équivalent
AF_INET	SOCK_STREAM	Socket de dialogue avec le protocole TCP/IP.	IPPROTO_TCP
AF_INET	SOCK_DGRAM	Socket utilisant le protocole UDP/IP.	IPPROTO_UDP

Pour les sockets de type `SOCK_RAW`, il faut utiliser l'un des deux protocoles suivants :

- `IPPROTO_RAW` : communication directe avec la couche IP ;
- `IPPROTO_ICMP` : communication utilisant le protocole ICMP. Ceci est utilisé par exemple dans l'utilitaire `/bin/ping`.

La création d'une socket de type `SOCK_RAW` nécessite la capacité `CAP_NET_RAW`. Les utilitaires qui en emploient (`traceroute`, `ping`...) et qui sont ouverts à tous les utilisateurs (à la différence de `tcpdump`) sont donc normalement installés *Set-UID root*.

La création d'une socket à l'aide de l'appel système éponyme ne fait que réserver un emplacement dans la table des descripteurs du noyau. Au retour de cette routine, nous disposons d'un entier permettant de distinguer la socket, mais aucun dialogue réseau n'a pris place. Il n'y a même pas eu d'échange d'informations avec les protocoles de communication du noyau. Celui-ci a simplement accepté de nous attribuer un emplacement dans sa table de sockets.

Le descripteur est supérieur ou égal à zéro : une valeur de retour négative indique une erreur. Comme aucune communication n'a commencé, les seules erreurs possibles sont :

- `EINVAL`, si le domaine est invalide ;
- `EPROTONOSUPPORT`, si le type est incohérent avec le protocole ou le domaine ;
- `EACCES`, si on n'a pas l'autorisation de créer une socket du type demandé (par exemple `AF_INET` et `SOCK_RAW`).

À cela s'ajoutent comme toujours `EMFILE`, `ENFILE`, `ENOMEM` si l'espace disponible dans la mémoire du noyau est insuffisant.

En Python, la valeur renvoyée est un objet et non pas un entier. Si on souhaite accéder au descripteur de fichier associé, on utilise la méthode `fileno()` de la classe `socket`.

Avant de pouvoir l'utiliser, il faut identifier la socket, c'est-à-dire définir l'adresse complète de notre extrémité de communication. Le nom affecté à la socket doit permettre de la trouver sans ambiguïté en employant le protocole réseau indiqué lors de sa création. Pour les communications fondées sur le protocole IP, l'identité d'une socket contient l'adresse IP de la machine et le numéro de port employé. En fait, un processus ne sera obligé d'identifier sa socket que s'il doit être joint par un autre programme. Si le processus doit lui-même contacter un serveur, il lui faut connaître l'identité de l'autre extrémité de la communication, mais l'extrémité locale sera automatiquement identifiée par le noyau.

Pour stocker l'adresse complète d'une socket, on emploie la structure `sockaddr`, définie dans `<sys/socket.h>` et contenant les membres suivants.

Nom	Type	Signification
<code>sa_family</code>	<code>unsigned short int</code>	Famille de communication.
<code>sa_data</code>	<code>char []</code>	Données propres au protocole.

En réalité, cette structure est une coquille vide, permettant d'employer un type homogène pour toutes les communications réseau. Pour indiquer véritablement l'identité d'une socket, on utilise une structure dépendant de la famille de communication, puis on emploie une conversion de type (`struct sockaddr *`) lors des appels système. C'est ce que nous avons aperçu dans le chapitre précédent.

Pour les sockets de la famille `AF_INET` reposant sur le protocole IP, la structure utilisée est `sockaddr_in`, définie dans `<netinet/in.h>`.

Nom	Type	Signification
<code>sin_family</code>	<code>short int</code>	Famille de communication <code>AF_INET</code> .
<code>sin_port</code>	<code>unsigned short</code>	Numéro de port, dans l'ordre des octets du réseau.
<code>sin_addr</code>	<code>struct in_addr</code>	Adresse IP de l'interface (dont le membre <code>s_addr</code> est dans l'ordre des octets du réseau).

Nous ne les étudierons pas ici, mais on peut signaler que d'autres familles de protocoles utilisent les structures suivantes.

Structure	Fichier	Famille	Utilisation
<code>sockaddr_ax25</code>	<code><netax25/ax25.h></code>	<code>AF_AX25</code>	Radioamateurs
<code>sockaddr_in6</code>	<code><netinet/in.h></code>	<code>AF_INET6</code>	IPv6
<code>sockaddr_ipx</code>	<code><netipx/ipx.h></code>	<code>AF_IPX</code>	Novell IPX
<code>sockaddr_un</code>	<code><sys/un.h></code>	<code>AF_UNIX</code>	Interne système Unix

L'intérêt de `getaddrinfo()`, que nous avons vu dans le chapitre précédent, est de nous fournir directement une structure `sockaddr` prête à transmettre aux appels système l'utilisant. Dans les versions précédentes de l'API réseau, il fallait procéder à des conversions rébarbatives entre types.

Nous savons créer une nouvelle socket et obtenir la structure décrivant entièrement une adresse complète. Nous pouvons maintenant examiner comment établir la communication.

Mentionnons auparavant l'existence d'un appel système nommé `socketpair()`, permettant de créer en une seule fois deux sockets, à la manière de `pipe()` :

```
| int socketpair (int domaine, int type, int proto,int sock [2]);
```

Les deux descripteurs de socket sont stockés dans le tableau passé en quatrième argument. À la différence de `pipe()`, les deux sockets sont bidirectionnelles. De plus, cet appel système n'est disponible que dans le domaine `AF_UNIX`, que nous n'examinerons pas ici. Son intérêt est assez limité car il sert essentiellement à la transmission de des-

cripteurs de fichiers ouverts entre processus. Ceci permet notamment à un serveur privilégié d'ouvrir des fichiers pour le compte de processus non privilégiés qui lui en ont fait la demande, le contrôle des accès se faisant par une procédure interne au serveur, généralement plus complexe que les autorisations gérées par le noyau. Il existe une fonction `socket.socketpair()` en Python équivalente à cet appel système.

Affectation d'adresse

Nous allons d'abord examiner comment affecter une identité à notre socket. Ceci s'effectue à l'aide de l'appel `systemebind()` que l'on emploie essentiellement côté serveur :

```
| int bind (int sock, struct sockaddr * adr, socklen_t longueur);
```

La socket représentée par le descripteur passé en premier argument est associée à l'adresse passée en seconde position que nous avons obtenue grâce à `getaddrinfo()`.

Le dernier argument représente la longueur de l'adresse. Cette valeur est indispensable, car avant d'analyser le type de la socket (et de connaître le détail de la structure d'adresse utilisée), le noyau doit copier l'adresse depuis l'espace de l'utilisateur vers son propre espace mémoire, et doit donc connaître la longueur réelle de la structure en deuxième position.

L'implémentation en Python est légèrement différente, `bind()` est une méthode de la classe `socket` qui prend en argument un tuple représentant l'adresse dans le format qui correspond au domaine sélectionné.

En C, le schéma habituel est donc pour une socket serveur en mode connecté :

```
| // Créer la socket serveur et lui
| // attribuer un numéro de service.
| int err;
| int sock_serveur;
| struct addrinfo hints;
| struct addrinfo *results;
|
| if ((sock_serveur = socket(AF_INET, SOCK_STREAM,
|                           IPPROTO_TCP)) < 0) {
|     perror("socket");
|     return -1;
| }
|
| memset(&hints, 0, sizeof(hints));
| hints.ai_family = AF_INET;
```

```
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
if ((err = getaddrinfo(NULL, nom_service,
                      &hints, &results)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
    close(sock_serveur);
    return -1;
}

err = bind(sock_serveur,
          results->ai_addr, results->ai_addrlen);
freeaddrinfo(results);
if (err < 0) {
    perror("bind");
    close(sock_serveur);
    return -1;
}
```

Notre socket est donc créée et elle possède un nom. Nous allons voir dans les prochains paragraphes comment le serveur peut se mettre à l'écoute en attendant que des processus le contactent.

Lorsqu'on crée une socket située du côté client, il n'est pas indispensable de mentionner explicitement notre identité. Le noyau attribuera de toute façon une adresse correcte lorsqu'on entrera en communication avec un autre processus.

L'appel système `bind()` peut échouer en renvoyant `-1`, avec les conditions d'erreur suivantes.

Nom	Signification
EBADF ENOTSOCK	Le descripteur de socket est invalide.
EACCESS	L'adresse demandée ne peut être employée que par un processus ayant la capacité <code>CAP_NET_BIND_SERVICE</code> .
EINVAL	La socket a déjà une adresse ou elle est déjà connectée, ou encore la longueur indiquée est inexacte.
EADDRINUSE	L'adresse est déjà utilisée.

L'erreur `EADDRINUSE` est souvent déclenchée lorsqu'on redémarre un serveur qu'on vient d'arrêter. Nous reviendrons en détail sur ce phénomène lorsque nous étudierons les options des sockets.

Il est possible également de rechercher l'adresse d'une socket. Ceci peut être utile si elle a été identifiée automatiquement par le noyau mais qu'on désire quand même connaître ses caractéristiques, pour les fournir à l'utilisateur par exemple.

L'appel système `getsockname()` renvoie l'adresse d'une socket :

```
int getsockname (int sock, struct sockaddr * adresse,
                 socklen_t * longueur);
```

Le second argument est un pointeur sur une structure correspondant au type de socket employée. Dans le domaine `AF_INET`, on utilise une `sockaddr_in` en convertissant explicitement le pointeur. Le troisième argument doit pointer sur une variable qui contient la longueur de cette structure. Au retour de la fonction, cette variable comprendra le nombre d'octets réellement écrits. En voici un exemple :

```
// Afficher notre adresse locale.
struct sockaddr_in * addr_in;
socklen_t length = 0;
char hostname [NI_MAXHOST];
char servname [NI_MAXSERV];

fprintf(stdout, "Mon adresse >> ");
length = sizeof(struct sockaddr_in);
if (getsockname(sock_serveur,
               (struct sockaddr *) &addr_in, &length) < 0) {
    perror ("getsockname");
    return -1;
}
if (getnameinfo((struct sockaddr *) &addr_in, length,
               hostname, NI_MAXHOST,
               servname, NI_MAXSERV,
               NI_NUMERICHOST | NI_NUMERICSERV) == 0)
    fprintf (stdout, "IP = %s, Port = %s \n", hostname, servname);
```

En Python, `getsockname()` est une méthode de la classe `socket`, qui renvoie un tuple représentant l'adresse. Par exemple, en IPv4, il s'agira d'un couple (hôte, service).

L'utilisation de `getsockname()` est plutôt rare. En revanche, l'appel `getpeername()` est beaucoup plus utile car cette méthode fournit les informations sur le correspondant, notamment son adresse réseau. On utilisera par exemple `getpeername()` pour connaître l'identité d'un client après une connexion et filtrer l'accès qu'on lui autorise. Il est vrai que l'appel système `accept()` qui permet de recevoir une connexion d'un client permet de récupérer directement les identifications du client. Néanmoins, il est fréquent qu'un serveur lance un processus fils ou un nouveau thread en lui transmettant simplement la nouvelle socket de dialogue obtenue avec `accept()`.

C'est donc ce processus fils ou ce thread qui appellera `getpeername()` pour en savoir plus sur le client en communication.

```
int getpeername (int sock, struct sockaddr * adresse,
                 socklen_t * longueur);
```

Dans le cas où le type d'adresse (INET, INET6, etc.) n'est pas connu à l'avance, on peut allouer dynamiquement la mémoire pour la structure `sockaddr`. Pour cela, on appelle deux fois `getpeername()` : la première fois avec une longueur nulle pour déterminer la taille nécessaire, la seconde fois avec un buffer fraîchement alloué :

```
// Afficher l'adresse du correspondant
struct sockaddr * sockaddr;
socklen_t length = 0;
char buffer [256];

getpeername(sock, NULL, &length);
if (length == 0)
    return;
sockaddr = malloc(length);
if (getpeername(sock, sockaddr, & length) < 0) {
    perror ("getpeername");
    free(sockaddr);
    return;
}
if (getnameinfo(sockaddr, length,
                hostname, NI_MAXHOST,
                servname, NI_MAXSERV,
                NI_NUMERICHOST | NI_NUMERICSERV) == 0) {
    snprintf (buffer, 256, "IP = %s, Port = %s \n",
              hostname, servname);
    fprintf(stdout, "Connexion distante %s", buffer);
}
free(sockaddr);
```

De même, lorsqu'une socket est connectée, il est possible d'obtenir des informations sur son correspondant en employant `getpeername()` :

```
int getpeername (int sock, struct sockaddr * adresse,
                 socklen_t * longueur);
```

Cet appel système fonctionne comme `getsockname()`, mais il nous renseigne sur le correspondant distant. Ceci n'est possible qu'en mode connecté, lorsqu'une communication s'établit de manière organisée entre deux correspondants. Si la socket n'est pas connectée, elle peut avoir une multitude de correspondants successifs, puisqu'on pourra changer d'interlocuteur à chaque écriture, et que tout un chacun pourra lui envoyer des données.

Mode connecté et mode non connecté

Lorsqu'on emploie des sockets fonctionnant en mode non connecté (`SOCK_DGRAM` dans le domaine `AF_INET`), le travail d'initialisation est déjà terminé. Le processus qui attend de recevoir des requêtes de la part d'autres programmes – appelons-le serveur – a identifié sa socket avec `bind()` : elle est accessible de l'extérieur. Du côté des clients, l'identification est établie automatiquement par le noyau. Au contraire, pour une communication connectée, il reste encore du travail à accomplir, tant du côté serveur que du côté client.

La différence essentielle entre les communications en mode connecté et celles en mode non connecté est que cette dernière technique nécessite d'indiquer le destinataire du message à chaque envoi. Au contraire, lorsqu'une connexion est établie, il n'y a plus que deux interlocuteurs face à face, et il n'y a pas d'ambiguïté lors de l'émission ou de la réception d'un message.

Lors d'une communication non connectée, on utilise les appels système `sendto()` et `recvfrom()`, qui permettent d'envoyer un paquet à destination d'un processus passé en argument ou de recevoir un paquet en récupérant l'adresse de l'émetteur. Nous reviendrons plus loin sur ces primitives. Par contre, en mode connecté un processus peut utiliser `send()` et `recv()`, qui ne précisent pas l'adresse du correspondant, ou même `read()` et `write()` comme avec un tube classique.

Il existe une bonne analogie qu'on retrouve souvent lorsqu'il s'agit de définir la notion de connexion. Dans le monde « réel », hors d'Internet, pour faire parvenir des informations à un proche, nous pouvons utiliser soit le téléphone, soit le service postal.

- Pour communiquer par téléphone nous devons établir la liaison en appelant notre correspondant. Celui-ci décroche et nous nous identifions mutuellement. Le respect de ce protocole nous permet de dialoguer en mode connecté. Tant que nous n'avons pas raccroché, il est possible d'envoyer des informations sans avoir besoin de préciser le destinataire. Si une interférence se produit, notre correspondant nous demande de répéter la phrase. La communication en mode connecté nous garantit ici que les messages seront bien délivrés à notre interlocuteur, indemnes et dans l'ordre. Le téléphone et ses états caractéristiques (en attente, non disponible, fin de communication), associés à un protocole comportant quelques mots standard « Allô ? », « Pardon, pouvez-vous répéter ? », « Au revoir ! », est un service de communication fiable.
- Lorsqu'on envoie des messages par la poste, nous devons écrire l'adresse du correspondant sur chaque enveloppe. Celui-ci regarde régulièrement dans sa boîte et peut y découvrir simultanément des lettres provenant de plusieurs émetteurs. Pour répondre à notre courrier, il lui faudra écrire une nouvelle lettre et la cacheter en inscrivant notre adresse de retour sur l'enveloppe. Il n'y a pas de connexion établie,

l'identité du destinataire doit être indiquée dans chaque message. Rien ne garantit non plus que les messages nous parviendront dans l'ordre ni même qu'ils arriveront un jour. Si une lettre est détrempée par la pluie et quasi illisible, notre correspondant devra nous contacter et nous demander de lui réécrire le message. Ceci n'est pas compris dans le protocole mais se trouve dans la couche applicative de la communication. La poste propose un service de communication non connecté, non fiable.

Dans le domaine AF_INET, les communications connectées (TCP) sont fiables, alors que les non-connectées (UDP) ne le sont pas. Ceci n'est toutefois pas une règle absolue, car des méthodes de contrôle et de séquençement peuvent être associées à un protocole sans connexion pour le fiabiliser encore plus. Pour continuer notre analogie, l'envoi de lettres postales en recommandé avec accusé de réception transforme ce service en communication non connectée fiable, mais avec un surcoût sensible.

Nous allons examiner à présent la fin de l'initialisation d'une communication en mode connecté avant d'analyser l'utilisation proprement dite de la socket, pour envoyer ou recevoir des données.

Attente de connexions

Nous allons tout d'abord nous intéresser à un serveur acceptant des connexions. Dans notre domaine AF_INET, il s'agira donc d'un serveur TCP. Supposons que nous désirions écrire une application fonctionnant comme un démon, à la manière d'un serveur FTP, telnet, http, etc. Notre application doit tout d'abord créer une socket de communication TCP, puis lui associer l'adresse IP et le numéro de port sur lesquels les clients tenteront de la contacter. Nous utiliserons la portion de code décrite précédemment.

Ensuite, nous devons indiquer au noyau que nous attendons des connexions sur cette socket. Ceci s'effectue en appelant l'appel système `listen()` :

```
| int listen (int sock, int nb_en_attente);
```

Le second argument de cet appel système demande au noyau de dimensionner une file d'attente des requêtes de connexions. Si une demande de connexion arrive et si le serveur est occupé, elle sera mise dans une file. Si la file est pleine, les nouvelles connexions seront rejetées. En général, on emploie la constante 5 car c'était la limite dans l'implémentation originale des sockets BSD, mais Linux accepte une file contenant jusqu'à 128 connexions en attente.

Attention à ne pas attacher trop d'importance à ce paramètre. Il sert simplement à dimensionner la tolérance du système lorsque plusieurs demandes arrivent simultanément. Ce n'est significatif que pour des serveurs acceptant de très nombreuses requêtes avec un rythme élevé (serveur HTTP par exemple).

En Python, la méthode `listen()` de la classe `socket` prend le même argument que la fonction `listen()` en C.

L'appel `listen()` n'est pas bloquant, il revient immédiatement. Les erreurs éventuelles concernent uniquement les descripteurs de socket invalides ou les tentatives d'utilisation de `listen()` sur des sockets fonctionnant en mode non connecté.

Une fois que le noyau est informé que le processus désire recevoir des connexions, il faut mettre effectivement le programme en attente. Pour cela on invoque l'appel système `accept()` :

```
int accept (int sock, struct sockaddr * adresse,  
            socklen_t * longueur);
```

Nous allons atteindre ici un point subtil de la programmation réseau, nécessitant un peu d'attention. Lorsqu'on crée un serveur TCP, on veut pouvoir recevoir des connexions en provenance de plusieurs clients. Et de surcroît, de façon simultanée. La socket que nous avons créée est attachée à une adresse IP et à un numéro de port connus des clients. Il s'agit donc d'une socket servant à établir le contact. Toutefois, on ne peut pas se permettre de la monopoliser ensuite pour assurer réellement la communication. Imaginons un serveur FTP qui a installé sa socket sur le port 21 de l'interface réseau de la machine. Un correspondant contacte notre serveur sur ce port et la connexion s'établit. Néanmoins, il est hors de question de bloquer ce port pour transférer toutes les données que notre correspondant désire, car aucun autre client ne pourrait nous contacter pendant ce temps.

Le principe de l'appel système `accept()` est donc de prendre une demande de connexion en attente – dans la file dimensionnée avec `listen()` –, puis d'ouvrir une nouvelle socket du côté serveur et d'établir la connexion sur celle-ci. La socket originale, celle qui a été passée en argument, reste donc intacte, prête à servir à nouveau pour une demande de connexion. La nouvelle socket créée est renvoyée par `accept()`. Le processus emploiera donc celle-ci pour toute la communication ultérieure.

Les deuxième et troisième arguments de `accept()` fonctionnent comme ceux de `getpeername()` et fournissent l'identité du client.

En Python, le comportement est légèrement différent. La méthode `accept()` de la classe `socket` ne prend pas d'argument, mais renvoie un couple constitué de la nouvelle socket connectée et d'un tuple représentant l'adresse du correspondant.

La plupart du temps, un serveur veut pouvoir dialoguer avec plusieurs clients simultanément. Pour cela le plus simple est d'invoquer `fork()` au retour de `accept()`, et de laisser le processus fils traiter la communication, alors que le père retourne en attente sur `accept()`. Voici un tel schéma :

```
int serveur_tcp (const char * nom_service)
{
    int sock_connectee;
    [...]

    // Créer la socket serveur et
    // lui attribuer un numéro de service.
    [...]

    // Afficher notre adresse locale.
    [...]

    listen(sock_serveur, 5);

    while (! quitter_le_serveur()) {
        sock_connectee = accept(sock_serveur, NULL, NULL);
        if (sock_connectee < 0) {
            perror("accept");
            return -1;
        }
        switch (fork()) {
            case 0 : // fils
                close(sock_serveur);
                traite_connexion(sock_connectee);
                exit(EXIT_SUCCESS);
            case -1 :
                perror("fork");
                return -1;
            default : // pere
                close(sock_connectee);
        }
    }
    return 0;
}
```

On notera que les processus fils qui se terminent deviennent zombies, car leur processus père ne lit pas les codes de retour. Pour éviter cette situation, on peut éventuellement ajouter une ligne :

```
signal(SIGCHLD, SIG_IGN);
```

ou – mieux – installer un petit gestionnaire pour le signal SIGCHLD.

L'exemple suivant va utiliser les routines développées précédemment.

Ici, la routine `quitter_le_serveur()` renvoie toujours zéro ; nous arrêterons le processus avec *Contrôle-C*. Le travail du processus fils consiste à déterminer l'adresse de son correspondant à l'aide de `getpeername()`, à afficher les adresses des deux extrémités de la socket, et à transmettre avec `write()` sa propre adresse au correspondant. Le programme est donc :

```
exemple-serveur-tcp.c :
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>

int serveur_tcp (const char * service_name);
int quitter_le_serveur (void);
void traite_connexion (int sock);

int serveur_tcp (const char * nom_service)
{
    [...]
}

int quitter_le_serveur (void)
{
    return 0;
}

void traite_connexion (int sock)
{
    // Afficher l'adresse du correspondant
    [...]
    close(sock);
}

int main (int argc, char * argv[])
{
    return serveur_tcp ("1234");
}
```

Pour nous connecter au serveur, nous utiliserons le programme `telnet` en lui indiquant en argument le numéro de port que le programme affiche au démarrage. Nous allons présenter le serveur sur la partie gauche de l'écran et les clients sur la moitié droite. Les deux premières connexions ont lieu depuis la même machine que le serveur, la troisième depuis une autre station. La première connexion emploie l'adresse `loopback` `127.0.0.1`, alors que les autres passent par l'interface réseau de cette machine `192.168.3.1`. Lors de l'appel `getaddrinfo()`, le fait de préciser un nom d'hôte `NULL` et l'attribut `AI_PASSIVE` dans la structure `hints` permet d'obtenir une adresse de serveur `0.0.0.0`, que l'on nomme également `INADDR_ANY`, et qui correspond à une écoute sur toutes les interfaces disponibles.

```
$ ./exemple-serveur-tcp
Mon adresse >> IP = 0.0.0.0, Port = 1234
$ telnet localhost 1234
Connexion distante IP = 127.0.0.1, Port = 56828
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Votre adresse : IP = 127.0.0.1, Port = 56828
Connection closed by foreign host.
$ telnet 192.168.3.1 1234
Connexion distante IP = 192.168.3.1, Port = 45559
Trying 192.168.3.1...
Connected to 192.168.3.1.
Escape character is '^]'.
Votre adresse : IP = 192.168.3.1, Port = 45559
$ telnet 192.168.3.1 1234
Connexion distante IP = 192.168.3.4, Port = 49271
Trying 192.168.3.1...
Connected to 192.168.3.1.
Escape character is '^]'.
Votre adresse : IP = 192.168.3.4, Port = 49271

(Contrôle-C)
$
```

Un programme peut avoir besoin d'attendre des connexions simultanément sur plusieurs sockets de contact – par exemple sur plusieurs ports. Or, `accept()` est un appel système bloquant. Il est donc possible d'utiliser auparavant `select()`, en attendant l'arrivée de données en lecture sur toutes les sockets surveillées. Les données reçues correspondront à une demande de connexion, et on pourra alors appeler `accept()` en sachant qu'on ne restera pas bloqué.

Comme nous l'avons souvent observé, le script Python permettant de réaliser le même travail que le programme C est beaucoup plus concis :

```
exemple-serveur-tcp.py :
#!/usr/bin/python
from __future__ import print_function
import os
import socket
import sys

def tcp_server(service_name):
    try:
        result = socket.getaddrinfo(None, service_name,
                                     socket.AF_INET, 0, 0, socket.AI_PASSIVE)
    except:
        print("Invalid service name:", service_name)
        sys.exit(1)

    (family, socktype, proto, canon, sockaddr) = result[0]

    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        server_sock.bind(sockaddr)
    except:
        print("Error in bind")

    (addr, port) = server_sock.getsockname()
    print("My port number is", port)

    server_sock.listen(5)

    while True:
        (connected_sock, address) = server_sock.accept()

        if (os.fork() == 0):
            server_sock.close()
            print("Connection from", address)
            os.write(connected_sock.fileno(),
                    "Connection from "
                    +str(address)+"\n")
            sys.exit(0)
        else:
            connected_sock.close()

tcp_server("5000")
```

Demander une connexion

Nous allons à présent nous intéresser à la socket située du côté client, en étudiant l'appel système `connect()`.

```
int connect (int sock, struct sockaddr * adresse,
             socklen_t longueur);
```

Celui-ci fonctionne de manière évidente, en contactant le serveur dont l'adresse est passée en argument et en établissant la connexion sur la socket indiquée. Nous pouvons l'employer pour créer un petit utilitaire, `tcp-2-stdout`, qui se connecte sur un serveur TCP et recopie tout ce qu'il reçoit sur sa sortie standard. Il acceptera les options `-a` et `-p` servant à indiquer respectivement l'adresse et le numéro de port du serveur. Nous créerons donc une routine analysant les arguments en ligne de commande et remplissant une structure `addrinfo`, afin de pouvoir la réutiliser dans d'autres programmes. On précise également dans le troisième argument (booléen) de cette routine s'il s'agit d'un serveur.

```
int lecture_arguments (int argc, char * argv[], int server, struct
addrinfo **results)
{
    char * liste_options = "a:p:h";
    int    option;
    int    err;
    char * hote = "localhost";
    char * port = "2000";

    struct addrinfo hints;

    while ((option = getopt(argc, argv, liste_options)) != -1) {
        switch (option) {
            case 'a' :
                hote = optarg;
                break;
            case 'p' :
                port = optarg;
                break;
            case 'h' :
                fprintf(stderr, "Syntaxe : %s [-a adresse] [-p port] \n",
                    argv[0]);
                return -1;
            default :
                break;
        }
    }
}
```

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if (server)
    hints.ai_flags = AI_PASSIVE;
if ((err = getaddrinfo(hote, port, &hints, results)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
    return -1;
}
return 0;
}
```

Le client TCP devient donc simplement :

```
tcp-2-stdout.c :
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>

#define LG_BUFFER 1024

int main (int argc, char * argv[])
{
    int    sock;
    char   buffer[LG_BUFFER];
    int    nb_lus;
    struct addrinfo *results;

    if (!lecture_arguments(argc, argv, 0, &results) < 0)
        exit(EXIT_FAILURE);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, results->ai_addr, results->ai_addrlen) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    freeaddrinfo(results);
```

```
setvbuf(stdout, NULL, _IONBF, 0);
while (1) {
    if ((nb_lus = read(sock, buffer, LG_BUFFER)) == 0)
        break;
    if (nb_lus < 0) {
        perror("read");
        break;
    }
    write(STDOUT_FILENO, buffer, nb_lus);
}
return EXIT_SUCCESS;
}
```

L'appel `read()` renvoie zéro lorsque la communication est coupée, et -1 en cas d'erreur, ce qui explique les deux cas traités dans la boucle. On remarque que le buffer de sortie de `stdout` a été supprimé avec `setvbuf()`. Ceci sert principalement lorsque cet utilitaire est employé pour transférer des données binaires, afin qu'elles soient transmises au processus en aval au rythme de leur arrivée depuis le réseau².

L'exécution de ce programme donne les mêmes résultats que ce que nous observions avec `telnet` :

```
$ ./exemple-serveur-tcp
Mon adresse >> IP = 0.0.0.0, Port = 1234
$ ./tcp-2-stdout -p 1234
Votre adresse : IP = 127.0.0.1, Port
= 57092
Connexion distante IP = 127.0.0.1, Port = 57092
(Contrôle-C)
$
$ ./tcp-2-stdout -p 1234
connect: Connection refused
$
```

Nous constatons lors de la seconde invocation que l'appel système `connect()` échoue s'il n'y a pas de serveur sur le port indiqué.

-
2. Comme je l'ai déjà mentionné dans un autre chapitre, j'ai déjà utilisé professionnellement les utilitaires de transfert entre le réseau et `stdin` ou `stdout` développés ici. Durant des phases de débogage ou de prototypage d'applications aéroportuaires, ils me servaient à transporter et à convertir des données provenant de radars vers des applications de visualisation. Il ne s'agit donc pas d'exemples totalement artificiels.

Un équivalent en Python peut se présenter ainsi :

```
tcp-2-stdout.py :
#!/usr/bin/python
from __future__ import print_function
import os
import socket
import sys

def lecture_arguments():
    server_name = "localhost"
    service_name = "2000"
    sys.argv.pop(0)
    while len(sys.argv) > 0:
        if sys.argv[0] == "-a":
            sys.argv.pop(0)
            server_name = sys.argv[0]
            sys.argv.pop(0)
            continue
        if sys.argv[0] == "-p":
            sys.argv.pop(0)
            service_name = sys.argv[0]
            sys.argv.pop(0)
            continue
        print("Unkown option:", sys.argv[0])
        sys.exit(1)
    try:
        result = socket.getaddrinfo(server_name, service_name,
                                     socket.AF_INET, 0, 0)
    except:
        print("Invalid host or service name")
        sys.exit(1)
    return result

result = lecture_arguments()
(family, socktype, proto, canon, sockaddr) = result[0]
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    sock.connect(sockaddr)
except:
    print("Unable to connect to the server")

while True:
    buffer = os.read(sock.fileno(), 4096)
    if buffer == "":
        break
    sys.stdout.write(buffer)
```

On peut très bien utiliser ce programme pour se connecter sur des services du système :

```
$ ./tcp-2-stdout.py -p daytime
07 MAY 2016 18:09:37 CEST
$
```

Le service `daytime` est implémenté directement dans le démon super-serveur `xinetd`. Il renvoie simplement la date et l'heure du système. Nous pouvons en implémenter une version très facilement en modifiant le programme `exemple_tcp_serveur.c` pour qu'il utilise la routine suivante :

```
void traite_connexion (int sock)
{
    char    buffer [256];
    time_t  heure;
    heure = time(NULL);
    sprintf(buffer, "%s", ctime(& heure));
    write(sock, buffer, strlen(buffer));
    close(sock);
}
```

Bien entendu, pour un véritable serveur système il faudrait employer le numéro de port approprié, 13 en l'occurrence. On peut vérifier que ce serveur se comporte comme l'original :

```
$ ./exemple-serveur-daytime
Mon adresse >> IP = 0.0.0.0, Port = 3000
$ ./tcp-2-stdout -p 3000
Sat May 7 18:10:43 2016
$

(Contrôle-C)
$
```

L'appel système `connect()` peut aussi être employé sur des sockets UDP. Cela sert à indiquer au noyau que nous désirons dialoguer sur cette socket exclusivement avec le correspondant indiqué en argument. Il sera alors possible d'utiliser directement `read()` et `write()`, ou `recv()` et `send()`, sans avoir besoin de s'occuper à nouveau de l'interlocuteur. Le noyau dirigera toutes nos écritures vers l'adresse et le port indiqués. Parallèlement, tous les messages ne provenant pas de ce correspondant seront éliminés par le noyau. Ceci permet de filtrer la communication lorsqu'on ne veut pas être dérangé par d'autres processus. L'utilisation de `connect()` sur une socket UDP

présente aussi l'avantage de pouvoir mieux gérer les erreurs grâce au protocole ICMP, comme nous le verrons plus bas.

Attention

La connexion d'une socket UDP n'est qu'une opération interne au processus et au noyau. Ce dernier mémorise l'adresse du correspondant préférentiel, mais aucun dialogue réseau n'a lieu. L'interlocuteur n'est aucunement concerné par cette action.

Fermeture d'une socket

Pour refermer une socket, on emploie en général l'appel système `close()`, qui est adapté à tous les descripteurs de fichiers sous Unix. Cette primitive est automatiquement invoquée lorsqu'un processus se termine. La socket est donc refermée et devient inutilisable.

Toutefois, avec un protocole connecté, il se peut que certaines données n'aient pas encore été transmises ou que l'accusé de réception ne soit pas encore arrivé. Si des données sont toujours en train de circuler sur le réseau, elles peuvent arriver endommagées et le destinataire peut nous demander de les répéter. Le protocole étant fiable, il doit garantir la bonne transmission des données, même si le processus s'est terminé juste après les avoir écrites.

La fermeture immédiate d'une communication TCP n'est donc pas possible. La terminaison est une opération à part entière du protocole, nécessitant des acquittements complets des deux correspondants. Cela signifie que la fermeture d'une socket TCP n'a pas de répercussion immédiate sur les interfaces réseau du noyau. La socket continue d'exister pendant un certain temps, afin de s'assurer que toutes les données restantes ont été transmises. Cette socket est encore visible avec l'utilitaire `netstat`. Si on essaye de réutiliser immédiatement l'adresse en relançant le processus, `bind()` nous renvoie l'erreur `EADDRINUSE`.

Pour en avoir le cœur net, nous pouvons créer un programme qui ouvre une socket serveur TCP, attend une connexion et la referme immédiatement. Ensuite, ce processus va essayer d'invoquer `bind()`, en bouclant jusqu'à ce qu'il réussisse. Il nous affichera alors la durée écoulée.

```
delai-close.c :  
#include <errno.h>  
#include <netdb.h>  
#include <stdio.h>  
#include <stdlib.h>
```

```
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>

int main (int argc, char * argv[])
{
    int    sock;
    time_t debut;
    time_t fin;
    struct addrinfo *results;

    if (lecture_arguments(argc, argv, 1, &results) < 0)
        exit(EXIT_FAILURE);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    if (bind(sock, results->ai_addr, results->ai_addrlen) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    listen(sock, 5);
    close(accept(sock, NULL, 0));
    close(sock);
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    time(& debut);
    while (1) {
        if (bind(sock, results->ai_addr, results->ai_addrlen) == 0)
            break;
        if (errno != EADDRINUSE) {
            perror("bind 2");
            exit(EXIT_FAILURE);
        }
        sleep(1);
    }
    time(& fin);
    fprintf(stdout, "Duree de persistance apres fermeture : %ld \n", fin -
debut);
    return 0;
}
```

Le comptage ne commence que lorsque la première socket est refermée, c'est-à-dire après la fin du telnet exécuté sur un autre terminal. On profite du délai pour observer la socket avec netstat.

```
$ ./delai-close -a 127.0.0.1 -p 1234
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
$ netstat -nt
Connexions Internet actives (sans serveurs)
Proto Recv-Q Send-Q Local Address      Foreign Address     State
tcp        0      0 127.0.0.1:1234     127.0.0.1:57272     TIME_WAIT
$
```

Et une minute plus tard...

```
$ Duree de persistance apres fermeture : 60
```

L'état TIME_WAIT indiqué par netstat correspond à l'attente intégrée dans le protocole TCP. Nous voyons qu'avec ce noyau, les sockets TCP persistent pendant une minute après leur fermeture. Nous verrons plus loin comment demander à réutiliser immédiatement une adresse, afin de pouvoir relancer un serveur sans attendre pendant une minute.

Il existe un appel système nommé shutdown(), permettant de contrôler plus finement la fin de l'utilisation d'une socket :

```
| int shutdown (int sock, int mode);
```

Si on appelle cette routine avec un second argument nul, la socket ne permettra plus de recevoir de données. Si cet argument vaut 1, shutdown() interdit l'émission de données sur la socket, et un argument valant 2 est équivalent à close(). L'utilisation typique de cette primitive est la suivante.

- Nous envoyons des données à un serveur TCP. Il s'agit d'un flux d'octets, dont la longueur est arbitraire (donc le serveur ne peut pas déterminer leur fin). Une fois que toutes les informations ont été envoyées, nous fermons le côté écriture de la socket avec shutdown(sock,1).

- Les appels système `read()` invoqués sur le serveur renverront 0 dès que les données auront toutes été envoyées et que nous aurons appelé `shutdown()`. Sachant que toutes les informations sont arrivées, le serveur peut les traiter et nous envoyer la réponse. À la fin de la réponse, le serveur ferme sa socket (et le processus fils se termine).
- Le client peut lire la réponse car sa socket n'est pas fermée en lecture. Quand toute la réponse aura été reçue, nos appels `read()` renverront 0. On pourra alors fermer la socket totalement.

L'appel système `shutdown()` ne peut être utilisé que sur une socket connectée.

Recevoir ou envoyer des données

Pour envoyer ou recevoir des données, nous avons jusqu'à présent utilisé `write()` et `read()`, car d'une part nous connaissions déjà ces fonctions et d'autre part nos programmes fonctionnaient en mode connecté avec TCP. Si nous choisissons UDP, il nous faut employer des routines plus générales, permettant d'avoir accès à l'identité de l'interlocuteur. Les fonctions `recvfrom()` et `sendto()` remplissent ce rôle :

```
int recvfrom (int sock, char * buffer, int taille_buffer,  
              int attributs, struct sockaddr * source, socklen_t * taille);  
int sendto  (int sock, char * buffer, int taille_buffer,  
              int attributs, struct sockaddr * source, socklen_t taille);
```

La seule différence entre ces deux prototypes est le dernier argument. Il s'agit d'un pointeur dans le cas de `recvfrom()` et d'une valeur dans le cas de `sendto()`. Les autres arguments correspondent à la socket employée, au buffer à transmettre ou à remplir, ainsi qu'à sa taille et à des attributs que nous détaillerons plus bas. Les deux derniers arguments définissent une adresse et sa taille.

Dans le cas de `recvfrom()`, la structure `sockaddr` transmise est remplie lors de l'appel système avec l'adresse de l'émetteur du message lu. Si ce pointeur est `NULL`, il est ignoré.

Dans le cas de `sendto()`, la structure `sockaddr` doit contenir l'adresse du destinataire du message. Si la socket est connectée, ce pointeur peut être `NULL`.

Il existe d'ailleurs deux fonctions plus courtes, `send()` et `recv()`, équivalentes de `sendto()` et `recvfrom()` avec des pointeurs d'adresse `NULL`.

```
int send (int sock, char * buffer, int taille_buffer,  
          int attributs);  
int recv (int sock, char * buffer, int taille_buffer,  
          int attributs);
```

L'appel `send()` ne peut être utilisé qu'après avoir invoqué `connect()` sur la socket (en TCP ou en UDP). Il faut en effet que le noyau connaisse l'adresse du correspondant. La primitive `recv()` peut en revanche être employée aussi sur une socket non connectée, encore que ce soit inhabituel – si on ne désire pas connaître l'adresse de l'émetteur et si on ne veut donc pas lui répondre.

Les valeurs les plus fréquentes qu'on peut associer par un *OU* binaire dans le champ attributs sont les suivantes.

Nom	Signification
MSG_DONTROUTE	Cette option sert surtout à déboguer les communications sur un réseau. Elle permet de négliger les procédures de routage mises en service par le noyau et de diriger directement le message vers l'interface qui correspond au sous-réseau de l'adresse du destinataire. Ne sert qu'avec <code>sendto()</code> ou <code>send()</code> .
MSG_OOB	Le message doit être considéré comme des données TCP hors bande. Elles sont émises avec une priorité supérieure à celle des informations normales. Au niveau du récepteur, elles seront reçues sans passer par une file d'attente. Suivant la configuration de la socket, il peut être nécessaire d'utiliser cette option pour lire les données hors bande.
MSG_PEEK	Lire les données désirées sur une socket TCP, sans les extraire de la file d'attente. Ne sert qu'avec <code>recvfrom()</code> ou <code>recv()</code> .

Il peut exister d'autres options spécifiques au noyau. Comme on le voit, la plupart du temps on n'utilise pas ces valeurs. Il est alors possible d'employer directement `write()` et `read()`, qui sont exactement équivalents à `send()` et `recv()` avec des arguments attributs nuls. Ces routines ont l'avantage d'être utilisables sur tout type de descripteurs, depuis les fichiers spéciaux de périphériques aux sockets, en passant par les fichiers normaux, les tubes, etc.

Il faut signaler l'existence de deux appels système très puissants mais assez complexes, `sendmsg()` et `recvmsg()`. Ceux-ci utilisent des structures permettant de regrouper plusieurs lectures ou plusieurs écritures, à la manière de `readv()` et de `writenv()`. Ils permettent également de transmettre un descripteur de fichier ouvert entre processus. Ces opérations sortent des limites de notre propos, qui est simplement d'expliquer comment faire communiquer des processus répartis.

Lorsqu'un processus tente d'écrire sur une socket n'ayant pas d'interlocuteur, le signal SIGPIPE est déclenché. On aura habituellement tout intérêt à l'ignorer, car dans ce cas l'appel système concerné – `write()`, `recv()` ou `recvfrom()` – renverra une erreur EPIPE, plus facile à traiter dans le cours du programme que de manière asynchrone dans un gestionnaire de signaux.

Lorsque la lecture se fait sur une socket connectée dont le correspondant a fermé l'autre extrémité, l'appel système renvoie une valeur nulle. Si on utilise `select()` sur

une socket en lecture et si l'interlocuteur la referme de son côté, cet appel système signale que des données sont disponibles en lecture. Ceci est dû à l'arrivée d'un caractère EOF. La lecture suivante renverra zéro octet.

Pour observer un peu `recvfrom()` et `sendto()` sur des sockets UDP, nous allons créer deux utilitaires : `udp-2-stdout`, qui permet de recevoir des données sur un port UDP et de les transmettre sur sa sortie standard, et en parallèle `stdin-2-udp`, qui sert à diriger des données vers un serveur en écoute.

```
stdin-2-udp.c :
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>

#define LG_BUFFER 1024

int main (int argc, char * argv[])
{
    int    sock;
    char   buffer[LG_BUFFER];
    int    nb_lus;
    struct addrinfo *results;

    if (lecture_arguments(argc, argv, 0, &results) < 0)
        exit(EXIT_FAILURE);

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    while (1) {
        if ((nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        sendto(sock, buffer, nb_lus, 0, results->ai_addr, results-
>ai_addrlen);
    }
    return 0;
}
```


On peut l'écrire en Python en modifiant légèrement le script précédent.

```
stdin-2-udp.py :
#!/usr/bin/python
from __future__ import print_function
import os
import socket
import sys

def lecture_arguments():
    # [...] (idem tcp-2-stdout.py)

result = lecture_arguments()
(family, socktype, proto, canon, sockaddr) = result[0]
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    buffer = os.read(sys.stdin.fileno(), 4096)
    if buffer == "":
        break
    sock.sendto(buffer, sockaddr)
```

Le second programme est :

```
udp-2-stdout.c :
[...]
int main (int argc, char * argv[])
{
    int sock;
    char buffer[LG_BUFFER];
    int nb_lus;
    struct addrinfo *results;

    if (lecture_arguments(argc, argv, 0, &results) < 0)
        exit(EXIT_FAILURE);

    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (bind(sock, results->ai_addr, results->ai_addrlen) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }
    freeaddrinfo(results);

    setvbuf(stdout, NULL, _IONBF, 0);
```

```

while (1) {
    if ((nb_lus = recv(sock, buffer, LG_BUFFER, 0)) == 0)
        break;
    if (nb_lus < 0) {
        perror("read");
        break;
    }
    write(STDOUT_FILENO, buffer, nb_lus);
}
return EXIT_SUCCESS;
} }

```

En Python, ce script devient :

```

udp-2-stdout.py :
#!/usr/bin/python
from __future__ import print_function
import os
import socket
import sys

def lecture_arguments():
    # [...] (idem tcp-2-stdout.py)

result = lecture_arguments()
(family, socktype, proto, canon, sockaddr) = result[0]
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
try:
    sock.bind(sockaddr)
except:
    print("Unable to use this address")

while True:
    (buffer, addr) = sock.recvfrom(4096)
    if buffer == "":
        break
    sys.stdout.write(buffer)

```

On notera que l'adresse et le numéro de port indiqués en arguments doivent dans les deux cas correspondre au processus récepteur. Nous pouvons essayer ces applications en lançant simultanément deux `stdin-2-udp` et un `udp-2-stdout`. L'enchaînement n'est pas très facile à représenter :

```

$ ./udp-2-stdout -a 192.168.3.1 -p 1234
$ ./stdin_2_udp -a 192.168.3.1 -p 1234
$ ./stdin_2_udp -a 192.168.3.1 -p 1234

```

```

    Première chaîne depuis station 1
Première chaîne depuis station 1
    Première chaîne depuis station 2
Première chaîne depuis station 2
    Deuxième chaîne, depuis station 1
Deuxième chaîne, depuis station 1
    (Contrôle-C)
$
    Deuxième chaîne, depuis station 2
Deuxième chaîne, depuis station 2
    (Contrôle-C)
$
    (Contrôle-C)
$

```

Nous n'avons pas utilisé jusqu'à présent l'aspect bidirectionnel des sockets. Nous allons maintenant le mettre en œuvre en créant un serveur TCP qui reçoit des chaînes de caractères émises par un client, les traite et les renvoie au même client qui les affiche. Pour avoir quelque chose à faire avec les chaînes, nous allons à nouveau créer une application particulièrement utile : un serveur d'anagrammes...

Le serveur est une variation sur `exemple-serveur-tcp.c`, dans lequel nous modifions la routine de traitement des connexions ainsi :

```

void traite_connexion (int sock)
{
    char buffer[256];
    int longueur;

    while (1) {
        longueur = read(sock, buffer, 256);
        if (longueur < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        if (longueur == 0)
            break;
        buffer[longueur] = '\0';
        strfry(buffer);
        write(sock, buffer, longueur);
    }
    close(sock);
}

```

De son côté, le client est construit à partir de `tcp-2-stdout.c`, en modifiant la routine principale :

```
int main (int argc, char * argv[])
{
    int    sock;
    char    buffer [LG_BUFFER];
    int    nb_lus;

    struct addrinfo *results;

    if (lecture_arguments(argc, argv, 0, &results) < 0)
        exit(EXIT_FAILURE);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, results->ai_addr, results->ai_addrlen) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    freeaddrinfo(results);
    while (1) {
        if (fgets(buffer, 256, stdin) == NULL)
            break;
        if (buffer[strlen(buffer) - 1] == '\n')
            buffer[strlen(buffer) - 1] = '\0';
        if (write(sock, buffer, strlen(buffer)) < 0) {
            perror("write");
            break;
        }
        if ((nb_lus = read(sock, buffer, LG_BUFFER)) == 0)
            break;
        if (nb_lus < 0) {
            perror("read");
            break;
        }
        fprintf(stdout, "%s\n", buffer);
    }
    return EXIT_SUCCESS;
}
```

L'exécution se déroule comme prévu.

```
$ ./exemple-serveur-anagramme
Mon adresse >> IP = 0.0.0.0, Port = 3815
$ ./exemple-client-anagramme -a 192.168.3.1 -p 3815
```

```
anagramme
gmraenaam
linux
lunxi
fin
ifn
(Contrôle-C)
$
(Contrôle-C)
$
```

Accès aux options des sockets

Il existe de multiples options configurables concernant les sockets. La plupart ne sont utiles qu'à des fins de débogage des protocoles réseau ou pour des applications très spécifiques. Pourtant, certaines d'entre elles sont couramment employées. Il existe deux appels système, `getsockopt()` et `setsockopt()`, permettant de lire l'état d'une option de configuration ou de la modifier.

```
int getsockopt (int sock, int niveau, int option,
               void * valeur, socklen_t * longueur);
int setsockopt (int sock, int niveau, int option,
               const void * valeur, socklen_t longueur);
```

Le premier argument de ces routines est l'identificateur de la socket concernée. Le second correspond au niveau auquel s'applique l'option. Ce niveau représente en fait la couche de protocole correspondant à l'option désirée. Pour nous, il s'agira uniquement des valeurs `SOL_SOCKET` indiquant qu'il s'agit de la socket elle-même, `IPPROTO_IP` correspondant à la couche réseau IP, ou `IPPROTO_TCP` pour la couche de transport TCP. Pour chaque option présentée page suivante, nous préciserons le niveau d'application.

Le troisième argument représente l'option elle-même. En quatrième argument on trouve un pointeur sur une variable contenant la valeur associée à l'option. Avec `getsockopt()`, cette variable sera remplie, avec `setsockopt()` elle sera lue. Enfin, on trouve en dernier argument la longueur de la variable employée pour stocker la valeur. Cette longueur doit dans tous les cas être initialisée avant l'appel, même si elle peut être modifiée par `getsockopt()`. Toutes les options présentées page suivante utilisent une valeur de type `int`, considérée comme vraie si elle est non nulle, à l'exception de `SO_LINGER` qui emploie une structure `linger`.

Les options les plus courantes pour le niveau `SOL_SOCKET` sont les suivantes.

Option	Signification
<code>SO_BROADCAST</code>	Autorisation de diffusion de messages <i>broadcast</i> sur une socket UDP. Nous décrirons ce mécanisme plus bas.
<code>SO_BSDCOMPAT</code>	Lorsqu'une socket UDP est connectée, une tentative d'écriture vers un port où personne n'écoute renverra une erreur ICMP. Ceci est également vrai sous Linux avec une socket UDP non connectée. Cette option – spécifique à Linux – force le noyau à adopter un comportement BSD, en n'envoyant pas cette erreur sur les socket non connectées.
<code>SO_DEBUG</code>	Activation des procédures de débogage dans les couches réseau du noyau.
<code>SO_DONTROUTE</code>	Contournement des procédures de routage, les messages étant directement dirigés vers l'interface correspondant à la partie sous-réseau de l'adresse du destinataire.
<code>SO_ERROR</code>	Uniquement avec <code>getsockopt()</code> : renvoie la valeur d'erreur correspondant à la socket (les erreurs sont identiques à celles de <code>errno</code>).
<code>SO_KEEPALIVE</code>	Activation d'un envoi périodique de messages sur une socket TCP connectée pour tester sa validité. Si le correspondant ne les acquitte pas, la communication est rompue. Ceci n'est généralement pas intéressant car le délai entre deux messages est de l'ordre de plusieurs heures.
<code>SO_LINGER</code>	Activation d'un délai de latence lors d'un appel <code>close()</code> s'il reste des données non émises. Rarement utile.
<code>SO_OOBINLINE</code>	Autorisation pour que les données hors bande arrivant sur une socket soient placées dans le flux normal de lecture, sans nécessiter d'option particulière de <code>recv()</code> . Ceci permet de transmettre des messages avec des priorités supérieures à d'autres.
<code>SO_RCVBUF</code> <code>SO_SNDBUF</code>	Indique la taille du buffer de réception ou d'émission.
<code>SO_RCVLOWAT</code> <code>SO_SNDLOWAT</code>	Ces valeurs correspondent à des seuils inférieurs dans les buffers de réception et d'émission, qui déclenchent – lorsqu'ils sont dépassés – une réponse positive de <code>select()</code> pour la socket.
<code>SO_RCVTIMEO</code> <code>SO_SNDTIMEO</code>	Uniquement avec <code>getsockopt()</code> , ces valeurs représentent un délai maximal en réception et en émission.
<code>SO_REUSEADDR</code>	Autorisation de réutiliser une adresse déjà affectée. Cette option est présentée ci-après.
<code>SO_TYPE</code>	Uniquement avec <code>getsockopt()</code> , renvoie la valeur correspondant au type de socket <code>SOCK_STREAM</code> , <code>SOCK_DGRAM</code> , <code>SOCK_RAW</code> ...

Les options qui intéressent en général le programmeur applicatif sont donc essentiellement `SO_BROADCAST` et `SO_REUSEADDR`, et parfois `SO_BSDCOMPAT`.

L'option `SO_REUSEADDR` permet notamment de relancer immédiatement un serveur TCP qu'on vient d'arrêter sans obtenir l'erreur `EADDRINUSE` lors du `bind()`. On insère l'appel `setsockopt()` avant le `bind()` :

```
int autorisation;  
  
autorisation = 1;  
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,  
           & autorisation, sizeof(int));
```

Notez bien que cette option est très utile, et que l'on peut considérer que tous les serveurs TCP/IP devraient l'activer.

L'option `SO_BROADCAST` permet d'effectuer de la diffusion globale, ce qui consiste à envoyer un message UDP en direction de tout un sous-réseau. L'ensemble des stations ayant une adresse IP dans ce sous-réseau recevra le paquet de données et le fera remonter jusqu'à la couche UDP. Si une application est en écoute sur le bon numéro de port, elle recevra les informations. Ce mécanisme permet d'arroser tout un ensemble de machines avec des données.

L'adresse de diffusion correspondant à un sous-réseau est obtenue en remplissant tout l'espace réservé pour les adresses des stations par des 1 binaires. Ainsi, sur un sous-réseau de classe C `192.1.1.x`, la diffusion *broadcast* s'obtient en envoyant des données à l'adresse `192.1.1.255`. Pour être sûr de ne pas effectuer cette opération de manière fortuite, il faut l'indiquer explicitement dans la configuration de la socket :

```
int autorisation;  
  
sock = socket(AF_INET, SOCK_DGRAM, 0);  
autorisation = 1;  
setsockopt(sock, SOL_SOCKET, SO_BROADCAST,  
           & autorisation, sizeof(autorisation));
```

Ensuite, on peut envoyer des messages à l'adresse de diffusion. La réception se fait de manière transparente, il suffit de lire les données sur le port adéquat.

La diffusion *broadcast* est un mécanisme très utile dans certaines situations, quand toutes les machines du même sous-réseau assurent une tâche similaire (affichage ou calculs en parallèle) avec des données identiques. Cependant, même les stations non intéressées par les données sont obligées de les faire remonter jusqu'à la couche UDP, où elles seront rejetées à cause de leur numéro de port. Ceci implique une surcharge de travail parfois importante.

Afin d'affiner le filtrage, il existe un autre mécanisme de diffusion, employant des adresses multicast qui sont traitées directement au niveau de l'interface réseau et de la couche IP.

Le principe de diffusion multicast consiste à obliger une application désireuse de recevoir les données à s'inscrire explicitement dans le groupe de diffusion. Cette inscription se fait au niveau de la couche réseau. Ainsi une machine dans laquelle aucun

processus n'est intéressé par ces informations n'a pas besoin de les laisser remonter dans ses couches IP et UDP.

Les adresses de groupes multicast se trouvent dans l'intervalle IP 224.0.0.0 à 239.255.255.255. Pour envoyer des données à tout un groupe de diffusion, il suffit donc d'écrire dans une socket UDP dirigée sur l'une de ces adresses.

Pour gérer son inscription et recevoir ainsi les informations, un processus doit renseigner une structure `ip_mreq`, définie ainsi.

Nom	Type	Signification
<code>imr_multiaddr</code>	<code>struct in_addr</code>	Adresse du groupe de diffusion qu'on désire rejoindre.
<code>imr_interface</code>	<code>struct in_addr</code>	Interface réseau à employer pour joindre le groupe. En général, on utilise <code>INADDR_ANY</code> .

Pour joindre un groupe, il faut utiliser l'option `IP_ADD_MEMBERSHIP` du niveau `IPPROTO_IP` de la socket (ce niveau d'options est présenté dans un tableau plus bas). Cette option prend en argument une structure `ip_mreq`. À partir de ce moment, les données à destination de l'adresse indiquée dans `ip_mreq.imr_multiaddr` remonteront jusqu'à la couche UDP de la machine réceptrice. Ensuite, elles seront disponibles sur le numéro de port qui leur est attribué. Le processus récepteur doit donc invoquer également `bind()`, pour préciser le port sur lequel il écoute.

Sous Linux, on peut invoquer `bind()` avant ou après l'inscription dans le groupe multicast ; cela n'a pas d'importance. Sous d'autres Unix et sous Windows, il faut que le `bind()` soit appelé en premier.

Le programme `udp-2-stdout.c` peut être modifié pour recevoir des données multicast.

```
exemple-reception-multicast.c :
[...]
```

```
int main (int argc, char * argv[])
{
    int sock;
    char buffer[LG_BUFFER];
    int nb_lus;
    struct addrinfo *results;
    struct ip_mreq requete_multicast;

    if (lecture_arguments(argc, argv, 0, &results) < 0)
        exit(EXIT_FAILURE);
```



```
if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(EXIT_FAILURE);
}
if (bind(sock, results->ai_addr, results->ai_addrlen) < 0) {
    perror("bind");
    exit(EXIT_FAILURE);
}

requete_multicast.imr_multiaddr.s_addr =
    ((struct sockaddr_in *)results->ai_addr)->sin_addr.s_addr;
requete_multicast.imr_interface.s_addr = htonl(INADDR_ANY);

if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &
requete_multicast, sizeof(struct ip_mreq)) < 0) {
    perror("setsockopt(IP_ADD_MEMBERSHIP)");
    exit(EXIT_FAILURE);
}

freeaddrinfo(results);

setvbuf(stdout, NULL, _IONBF, 0);
while (1) {
    if ((nb_lus = recv(sock, buffer, LG_BUFFER, 0)) == 0)
        break;
    if (nb_lus < 0) {
        perror("read");
        break;
    }
    write(STDOUT_FILENO, buffer, nb_lus);
}
return EXIT_SUCCESS;
}
```

On notera que l'inscription dans un groupe multicast nous oblige à manipuler les champs internes des structures d'adresses, ce que l'emploi de `getaddrinfo()` nous évitait jusqu'à présent.

Pour que la réception fonctionne, il faut que le système sache que les paquets dirigés vers l'adresse multicast du groupe choisi doivent être transférés à la couche IP, sinon ils seront rejetés dès la couche réseau. Il faut donc utiliser `route`. Supposons que nous voulions utiliser le groupe multicast 224.0.0.0. Nous allons configurer la réception, puis passer en écoute, une autre machine enverra des données multicast.

```
$ su -
Password:
# route add 224.0.0.0 dev eth0
# exit
exit
$ ./exemple_reception_multicast -a 224.0.0.0 -p 1234
$ ./stdin_2_udp -a 224.0.0.0 -p 1234
Chaine emise en multicast
(Contrôle-C)
$
$
```

Les options du niveau IPPROTO_IP qui nous concernent sont celles qui ont trait à la diffusion multicast, les autres étant à un niveau trop bas dans le protocole.

Options	Signification
IP_ADD_MEMBERSHIP IP_DROP_MEMBERSHIP	Demande l'inscription ou le désistement d'une socket UDP dans un groupe <i>multicast</i> . La valeur est représentée sous forme d'une structure <i>ip_mreq</i> décrite plus haut.
IP_MULTICAST_IF	Permet de préciser l'interface réseau à utiliser pour recevoir les données. La valeur est une structure <i>in_addr</i> .
IP_MULTICAST_LOOP	Activation ou non d'un écho via l'interface <i>loopback</i> des messages <i>multicast</i> émis si l'émetteur est membre du groupe de réception.
IP_MULTICAST_TTL	Configuration du champ <i>Time-To-Live</i> du paquet IP diffusé en <i>multicast</i> . Il s'agit d'une valeur entière indiquant approximativement le nombre de routeurs que le paquet peut franchir avant d'être détruit. Normalement, cette valeur vaut 1 pour limiter la portée des messages au même réseau physique.

Les options du niveau IPPROTO_TCP sont essentiellement les suivantes.

Option	Signification
TCP_MAXSEG	Configuration de la taille maximale des segments de données transmis par le protocole. Doit être inférieur à la valeur de MTU de l'interface réseau (en général 1500 pour les cartes Ethernet).
TCP_NODELAY	Cette option empêche TCP de mettre en attente – quelques centièmes de secondes – les petits paquets de données pour essayer de les regrouper en un seul gros paquet, afin de limiter la surcharge due aux en-têtes TCP. Ceci n'est utile que si de toutes petites quantités de données doivent être traitées très rapidement (des mouvements de souris par exemple).

Seule l'option TCP_NODELAY peut parfois avoir une utilité dans les applications courantes.

Programmation d'un démon ou utilisation de inetd

Lorsqu'un serveur de données TCP a atteint un niveau de maturité fonctionnelle suffisant pour présenter un intérêt global au niveau du système et du réseau (par exemple, un serveur d'anagrammes), il est souvent intéressant de le faire fonctionner en tant que démon.

Un *démon* est un processus tournant en arrière-plan sur le système, sans terminal de contrôle. En général, les démons sont démarrés lors de l'initialisation du système, et on les laisse s'exécuter jusqu'à l'arrêt de la machine. Pour transformer un serveur classique en démon, il faut respecter certaines règles.

- Tout d'abord, le démon doit passer en arrière-plan. Pour cela on utilise :

```
if (fork() != 0)
    exit(EXIT_SUCCESS);
```

- Le démon ne doit bloquer aucune partition du système – sauf s'il s'agit de ses propres répertoires comme `/var/spool/lpd` pour le démon `lpd`. Aussi il faudra en général remonter à la racine du système de fichiers :

```
chdir("/");
```

- Le processus doit créer une nouvelle session et s'assurer qu'il n'a pas de terminal de contrôle. Nous avons déjà observé ceci dans le chapitre 2 :

```
setsid();
if (fork() != 0)
    exit(EXIT_SUCCESS);
```

Finalement, le démon doit fermer tous les descripteurs de fichiers que le shell aurait pu lui transmettre. Une méthode courante est d'utiliser :

```
for (i = 0; i < OPEN_MAX; i++)
    close(i);
```

Naturellement, le démon ne pourra plus afficher de message sur `stderr`, il lui faudra employer le mécanisme `syslog()` que nous étudierons dans le chapitre 30.

Le programme `exemple-demon-anagramme.c` est une réplique de `exemple-serveur-anagramme.c` dans lequel nous avons remplacé toutes les occurrences de :

```
perror("xxx");
```

par :

```
syslog(LOG_ERR, "xxx: %m");
```


En fait, ce programme gagnerait à employer un numéro de port figé, inscrit dans `/etc/services`, plutôt que de nous obliger à regarder le fichier de messages de `syslog()` pour le trouver.

Une alternative à la programmation d'un démon est l'emploi du super-serveur réseau `xinetd`. Ce démon lit au démarrage sa configuration dans `/etc/xinetd.conf` et assure toute la gestion de l'aspect serveur TCP. Lorsqu'une connexion a été établie, il invoque directement l'utilitaire demandé, en ayant redirigé – grâce à `dup()` – son entrée et sa sortie standard vers la socket obtenue.

Sur la plupart des distributions Linux l'ancien super-serveur `inetd` a été remplacé par `xinetd`, apportant de nombreuses options supplémentaires notamment en ce qui concerne la sécurité du système.

Notre serveur d'anagrammes peut alors être réécrit tout simplement ainsi :

```
exemple-inet-anagramme.c :
#define _GNU_SOURCE
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main (void)
{
    char chaine[256];
    int n;
    while (1) {
        if ((n = read(STDIN_FILENO, chaine, 256)) <= 1)
            break;
        while (isspace(chaine[n - 1]))
            n --;
        chaine[n] = '\0';
        strfry(chaine);
        write(STDOUT_FILENO, chaine, n);
    }
    return EXIT_SUCCESS;
}
```

Il nous faut alors ajouter un port dédié dans `/etc/services` et un fichier de lancement dans `/etc/xinetd.d`. On se reportera aux pages de manuel `xinetd(8)` et `xinetd.conf(8)` pour plus de détails sur la syntaxe.

```
$ su
Mot de passe :
# cp exemple-inet-anagramme /usr/local/bin/
```

Édition de `/etc/services` et ajout, en fin de fichier, de la ligne :

```
| anagramme      98765/tcp
```

Édition d'un fichier `/etc/xinetd.d/anagramme` se présentant ainsi :

```
| service anagramme
| {
|     disable      = no
|     id           = anagramme
|     wait         = no
|     socket_type  = stream
|     user         = root
|     group        = root
|     server       = /usr/local/bin/exemple-inet-anagramme
| }
```

Relancer le super-serveur :

```
| # /etc/init.d/xinetd restart
| Arrêt de xinetd : [ OK ]
| Démarrage de xinetd : [ OK ]
| # exit
| $
```

On peut à présent tester le serveur aussi bien avec le logiciel client dédié qu'avec telnet.

```
$ ./exemple-client-anagramme -p 98765
linux
lxniu
client
neilct
  (Contrôle-C)
$ telnet localhost 98765
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
unix
nix
serveur
sueerrv
  (Contrôle-D)
Connection closed by foreign host.
$
```

Conclusion

Nous avons étudié dans ce chapitre les rudiments de la programmation réseau en TCP/IP et UDP/IP, en soulignant bien la puissance du concept de socket, puisque ce mécanisme simple et qui hérite logiquement des méthodes classiques de communication entre processus permet de donner une dimension nouvelle aux applications en leur offrant de dialoguer avec des stations réparties dans le monde entier.

Nous avons limité notre présentation aux informations les plus utiles pour un programmeur applicatif. Bien entendu, il existe encore de nombreux points qui n'ont pas été abordés. La référence essentielle dans ce domaine est [Stevens 1990] *Unix Network Programming*.

On se reportera également aux documents Linux Networking-HOWTO.

Les RFC restent la source ultime – mais peu digeste – de référence en ce qui concerne le réseau. Nous en avons indiqué quelques-unes dans le chapitre précédent.

Accès au contenu des répertoires

Il est souvent important pour un programme de pouvoir afficher la liste des fichiers contenus dans un répertoire. Ceci ne concerne pas uniquement les utilitaires du genre `ls` ou les gestionnaires de fichiers, mais peut servir à toute application proposant l'enregistrement et la récupération de données.

Les interfaces graphiques actuelles permettent de plus en plus facilement de disposer de boîtes de dialogue pour la sauvegarde ou la lecture de fichiers, sans avoir à écrire manuellement le code de parcours d'un répertoire. Toutefois, dans certaines situations, l'accès au contenu d'un répertoire est indispensable, notamment lorsque le nom d'un fichier constitue une information importante pour analyser son contenu. Je peux citer le cas d'une application recevant des fichiers de données météorologiques, dont le nom permet de retrouver la zone couverte ainsi que l'heure de capture. Ces informations pourraient à profit être intégrées dans une section d'en-tête du fichier, mais on n'est pas toujours responsable du format des données fournies en amont d'un système, principalement dans un environnement hétérogène incluant des dispositifs provenant de divers fournisseurs.

Nous allons donc dans ce chapitre nous concentrer sur l'accès au contenu d'un répertoire, la lecture de la liste des fichiers s'y trouvant, la modification du répertoire de travail, la suppression de sous-répertoires ou de fichiers, ainsi que la recherche de noms de fichiers par des caractères génériques et le parcours récursif, à la manière de l'utilitaire `find`.

Lecture du contenu d'un répertoire

Sous Unix, un répertoire est un fichier spécial, contenant pour chaque fichier ou sous-répertoire une structure opaque variant suivant le type de système de fichier. À titre d'exemple, avec le système ext4, les répertoires comprennent des structures `ext4_dir_entry` définies dans `<linux/ext4/ext4.h>`. Chaque structure dispose du nom du fichier, de son numéro d'i-nœud, ainsi que des champs servant à la gestion interne des structures.

Au niveau applicatif, les fonctions `opendir()`, `readdir()`, `closedir()` nous permettent d'accéder au contenu d'un répertoire sous forme de structures `dirent`. Pour assurer la portabilité d'une application, nous nous limiterons à l'utilisation des deux seuls champs qui soient définis par SUSv4, `char d_name[]`, qui contient le nom du fichier ou du sous-répertoire et `ino_t d_ino`, le numéro d'i-nœud du fichier (ce dernier champ est accessible si la constante symbolique `_XOPEN_SOURCE` contient la valeur 500).

Ces fonctions sont définies dans `<dirent.h>` :

```
DIR *      opendir (const char * repertoire);
struct dirent * readdir (DIR * dir);
int        closedir (DIR * dir);
```

Elles étaient implémentées en Python directement dans le module `dircache`, mais celui-ci est devenu obsolète et a disparu dans Python 3. La fonction `os.listdir()` standard peut les remplacer.

Le type `DIR`, défini dans `<sys/types.h>`, est une structure opaque, comparable au flux `FILE`, mais on l'emploie sur des répertoires au lieu des fichiers. À la manière de `fopen()`, la fonction `opendir()` renvoie un pointeur `NULL` en cas d'échec. La fonction `readdir()` renvoie l'entrée suivante ou `NULL` une fois arrivée à la fin du répertoire. Lorsqu'on a fini d'utiliser le répertoire, on le referme avec `closedir()` :

```
exemple-opendir.c :
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/types.h>

void affiche_contenu (const char * repertoire);

int main (int argc, char * argv[])
{
    int i;
    if (argc < 2)
        affiche_contenu(".");
```

```
    else for (i = 1; i < argc; i ++)  
        affiche_contenu(argv[i]);  
  
    return EXIT_SUCCESS;  
}  
  
void affiche_contenu (const char * repertoire)  
{  
    DIR * dir;  
    struct dirent * entree;  
  
    dir = opendir(repertoire);  
    if (dir == NULL)  
        return;  
    fprintf(stdout, "%s :\n", repertoire);  
    while ((entree = readdir(dir)) != NULL)  
        fprintf(stdout, "    %s\n", entree->d_name);  
    fprintf(stdout, "\n");  
    closedir(dir);  
}
```

Ce programme affiche le contenu des répertoires dont le nom est passé en argument.

```
$ ./exemple-opendir /etc/X11/ /proc/tty/  
/etc/X11/ :  
.  
prefdm  
aplnk  
..  
Xmodmap  
Xresources  
fontpath.d  
xorg.conf.d  
xinit  
  
/proc/tty/ :  
.  
..  
drivers  
ldiscs  
driver  
ldisc  
  
$
```

Le pointeur renvoyé par `readdir()` est une variable statique, qui peut être écrasée à chaque appel. Cette fonction n'est donc pas réentrante et ne doit pas être utilisée dans un contexte multithread. Pour cela, on peut employer la fonction `readdir_r()` qui prend deux arguments supplémentaires pour stocker la valeur de retour et mémoriser la position suivante dans le répertoire.

```
int readdir_r (DIR * dir,
               struct dirent * entree,
               struct dirent ** memorisation) ;
```

Cette fonction transmet 0. Arrivée à la fin du répertoire, elle renvoie également 0, mais l'argument `memorisation` vaut `NULL`. Voici comment l'utiliser :

```
struct dirent resultat;
struct dirent * memorisation;
DIR * dir;

while (1) {
    if (readdir(dir, & resultat, & memorisation) != 0)
        return -1;
    if (memorisation == NULL)
        break;
    fprintf(stdout, "    %s\n", entree->d_name);
}
fprintf(stdout, "\n");
```

On peut s'interroger sur l'allocation mémoire nécessaire pour stocker la chaîne de caractères contenant le nom des éléments. En fait, cette chaîne dispose automatiquement d'une taille maximale, définie par la constante `NAME_MAX`. La structure `dirent` comprend aussi sur la plupart des systèmes (mais pas tous) un membre `d_namlen` contenant la longueur du membre `d_name` comme la valeur renvoyée par `strlen()` (donc caractère nul final non compté). Ce champ n'est toutefois pas défini par SUSv4, et on évitera autant que possible de l'employer.

On observe que `readdir()` et `readdir_r()` renvoient les entrées « . » et « .. » correspondant respectivement au répertoire courant et à son parent. Ce comportement n'est pas garanti par SUSv4. Toutefois, ces deux entrées sont toujours valables pour `opendir()` ou pour des commandes de changement de répertoire de travail que nous verrons plus loin.

Comme avec les flux de fichiers, il est possible de se déplacer au sein des répertoires `DIR` en utilisant `rewinddir()`, qui revient au début du répertoire, `telldir()`, qui renvoie la position courante, ou `seekdir()`, qui permet de sauter à une position donnée, renvoyée précédemment par `telldir()`. Les prototypes de ces fonctions sont :

```
void rewinddir (DIR * dir);
void seekdir (DIR * dir, off_t offset);
off_t telldir (DIR * dir);
```

Ces trois fonctions sont définies par SUSv4.

Il existe également une fonction puissante permettant de sélectionner une partie du contenu d'un répertoire, de la trier et d'en fournir le contenu dans une table allouée automatiquement. Cette fonction est nommée `scandir()`, et son prototype peut paraître un peu inquiétant au premier coup d'œil :

```
int scandir(const char * dir, struct dirent ***namelist,
            int (* selection) (const struct dirent * entree),
            int (* comparaison) (const struct dirent ** entree_1,
                                const struct dirent ** entree_2));
```

La fonction `scandir()` commence par lire entièrement le contenu du répertoire dont le nom lui est fourni en premier argument. Ensuite, elle invoque pour chaque entrée du répertoire la fonction `selection` sur laquelle on lui passe un pointeur en troisième argument. Si la fonction `selection()` renvoie une valeur nulle, l'entrée considérée est rejetée. Sinon, elle est sélectionnée.

Puis, `scandir()` trie la table des entrées restantes, en invoquant la routine `qsort()` que nous avons étudiée au chapitre 17. Pour pouvoir trier la table, on utilise comme fonction de comparaison celle dont le pointeur est fourni en dernier argument de `scandir()`.

Une fois la table triée, `scandir()` met à jour le pointeur passé en second argument pour le diriger dessus. Les allocations ayant lieu avec `malloc()`, il faudra libérer ensuite le contenu de cette table.

Si on désire sélectionner tout le contenu du répertoire, il est possible de transmettre un pointeur `NULL` en guise de fonction de sélection. La bibliothèque Glibc met également à notre disposition une fonction `alphasort()` qui permet de trier automatiquement les entrées du répertoire par ordre alphabétique :

```
int alphasort (const struct dirent ** entree_1,
               const struct dirent ** entree_2);
```

Nous allons utiliser `scandir()` et `alphasort()` pour créer un exemple permettant de sélectionner les éléments correspondant à une expression rationnelle dans un répertoire donné. Les fonctions `regcomp()` et `regexexec()` traitant les expressions rationnelles ont été présentées dans le chapitre 18.

```
exemple-scandir.c :
#include <dirent.h>
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

regex_t motif_recherche;
```

```

int selection (const struct dirent * entree)
{
    if (regexexec(& motif_recherche, entree->d_name, 0, NULL, 0) == 0)
        return 1;
    return 0;
}

int main (int argc, char * argv[])
{
    struct dirent ** liste;
    int      nb_entrees;
    int      i;

    if (argc != 3) {
        fprintf(stderr, "Syntaxe : %s repertoire motif\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (regcomp(& motif_recherche, argv[2], REG_NOSUB) !=0) {
        fprintf(stderr, "Motif illegal\n");
        exit(EXIT_FAILURE);
    }
    nb_entrees = scandir(argv[1], & liste, selection, alphasort);
    if (nb_entrees <= 0)
        return EXIT_SUCCESS;
    for (i = 0; i < nb_entrees; i++) {
        fprintf(stdout, " %s\n", liste[i]->d_name);
        free(liste[i]);
    }
    fprintf(stdout, "\n");
    free(liste);

    return EXIT_SUCCESS;
}

```

Dans l'exécution suivante, on remarque deux choses :

- les expressions régulières ne sont pas identiques aux motifs génériques du shell (principalement en ce qui concerne le métacaractère *). Nous verrons plus loin dans ce chapitre des fonctions permettant d'obtenir le même comportement ;
- il ne faut pas oublier de protéger du shell les caractères spéciaux [,], |...

```

$ ./exemple-scandir /etc/ shad
gshadow
gshadow-
shadow
shadow-

```

Recherchons les fichiers qui contiennent un b, un c ou un d, suivi d'un a, suivi d'un caractère quelconque, suivi d'un caractère entre e et h ; il faut utiliser des backslashes \ pour protéger les caractères spéciaux [] qui encadrent les listes de caractères.

```
$ ./exemple-scandir /bin/ \[bcd\]a.\[e-h\  
  basename  
  bash  
  dash  
  date  
  
$
```

Un répertoire est, nous l'avons dit, considéré comme un fichier particulier, mais un fichier quand même. Il est donc possible d'ouvrir avec `open()` un descripteur sur ce fichier. L'ouverture ne peut se faire qu'en lecture, car seul le noyau a le droit de modifier le contenu véritable du répertoire, pour être sûr de garder l'ensemble cohérent. Le droit d'écriture sur un répertoire correspond simplement à l'autorisation d'y créer un nouveau fichier – avec `open()` par exemple – ou un sous-répertoire, mais en passant toujours par l'intermédiaire d'un appel système qui permet au noyau de contrôler les données.

Il n'est pas possible de lire directement le contenu d'un descripteur de répertoire avec `read()`. Il faut utiliser des appels système compliqués, comme `getdents()` ou la version bas niveau de `readdir()`. Ces appels système peuvent varier d'une version à l'autre du noyau, et les structures de données qu'ils manipulent ne sont pas portables. Nous ne les présenterons donc pas dans cet ouvrage. Le lecteur ayant véritablement besoin d'employer ces fonctions pourra se reporter aux sources du noyau ou à celles de la bibliothèque C pour y étudier les détails d'implémentation.

Changement de répertoire de travail

Chaque processus dispose en permanence d'un répertoire de travail. Ce répertoire est hérité du processus père et peut être modifié avec l'appel système `chdir()`. Le changement n'est toutefois visible que dans le processus courant et ses futurs descendants, pas dans le processus père.

Lors de la connexion d'un utilisateur, `login` lit dans le fichier `/etc/passwd` le répertoire personnel de l'utilisateur et s'y place, avant d'invoquer le shell. Il configure également la variable d'environnement `HOME`, qui restera donc correctement renseignée, même si l'utilisateur se déplace dans l'arborescence du système de fichiers.

Il existe deux appels système permettant de modifier le répertoire courant d'un processus : `chdir()`, qui prend en argument le nom du répertoire destination, et `fchdir()`, qui utilise un descripteur sur le répertoire cible. Ces deux appels système sont déclarés dans `<unistd.h>` et définis par SUSv4.

```
int chdir(const char * nom);  
int fchdir (int descripteur);
```

Ils renvoient tous deux 0 en cas de réussite, et -1 en cas d'erreur.

En Python, les fonctions `os.chdir(path)` et `os.fchdir(descripteur)` jouent un rôle identique.

Nous avons indiqué qu'un processus dispose toujours d'un répertoire de travail, mais aussi surprenant que cela puisse paraître, il n'existe pas d'appel système permettant d'obtenir directement le nom de ce répertoire. Il faut pour cela s'adresser à une fonction de bibliothèque comme `getcwd()`, `getwd()`, ou `get_current_working_dir_name()`. Cette dernière fonction n'est pas définie par SUSv4. Les trois prototypes sont déclarés dans `<unistd.h>` :

```
char * getcwd (char * buffer, size_t taille);  
char * getwd(char * buffer);  
char * get_current_working_dir_name (void);
```

La fonction `getcwd()` copie le chemin du répertoire courant dans le buffer transmis, dont on précise également la taille. Si le buffer n'est pas suffisamment grand, `getcwd()` échoue avec l'erreur `ERANGE`. Nous verrons plus bas le moyen de gérer cette situation. Avec la bibliothèque Glibc, il est possible de transmettre un `bufferNULL`, avec une taille valant 0, pour que `getcwd()` assure elle-même l'allocation mémoire nécessaire. Ce comportement n'est malheureusement pas portable sur d'autres systèmes.

En Python, `os.getcwd()` renvoie directement la chaîne contenant le chemin du répertoire courant.

La fonction `get_current_working_dir_name()` est une extension Gnu (requérant donc la définition de la constante `_GNU_SOURCE`). Elle alloue automatiquement la taille requise pour le stockage du chemin d'accès, en appelant `malloc()`. Il faudra donc libérer le pointeur obtenu.

La fonction `getwd()` est un héritage de BSD. Il faut donc définir la constante `_BSD_SOURCE`. Cette fonction suppose que le buffer transmis contient au moins `PATH_MAX` octets. Si ce n'est pas le cas, elle risque de déclencher silencieusement un débordement. Il faut donc éviter à tout prix d'employer cette routine.

Le fonctionnement interne traditionnel de `getcwd()` sous Unix est surprenant. Pour obtenir le nom du répertoire courant, `getcwd()` commence par mémoriser le numéro

d'i-nœud de l'entrée « . » du répertoire courant. Ensuite, elle analyse toutes les entrées du répertoire « .. » pour retrouver celle dont le numéro d'i-nœud correspond. Nous verrons au chapitre suivant le moyen d'accéder à cette information. Ensuite, le procédé est répété en remontant jusqu'à la racine du système de fichiers. Il est alors possible de reconstituer le chemin complet.

Toutefois, sous Linux, le noyau met à la disposition de la bibliothèque le pseudo-système de fichiers `/proc`, qui contient des informations diverses sur le système. Il existe un sous-répertoire pour chaque processus en cours (par exemple `/proc/524/`). Dans ce sous-répertoire, on trouve divers fichiers, dont un lien symbolique nommé `cwd` (pour *current working directory*) qui pointe vers le répertoire courant du processus. Il suffit alors de lire le contenu de ce lien symbolique (que nous étudierons dans le prochain chapitre) pour connaître le chemin recherché. Cette information n'est toutefois disponible que pour le propriétaire du processus concerné.

Pour simplifier encore le travail, il existe un sous-répertoire `/proc` nommé `self`, qui correspond au processus appelant. Voici donc un moyen simple d'accéder au répertoire courant :

```
$ cd /etc/
$ ls -l /proc/self/
[...]
-r--r--r-- 1 cpb cpb 0 19 août 08:00 cmdline
-rw-r--r-- 1 cpb cpb 0 19 août 08:00 comm
-rw-r--r-- 1 cpb cpb 0 19 août 08:00 coredump_filter
-r--r--r-- 1 cpb cpb 0 19 août 08:00 cpuset
lrwxrwxrwx 1 cpb cpb 0 19 août 08:00 cwd -> /etc
-r----- 1 cpb cpb 0 19 août 08:00 environ
[...]
$ cd /usr/local/sbin/
$ ls -l /proc/self/
-r--r--r-- 1 cpb cpb 0 19 août 08:01 cmdline
-rw-r--r-- 1 cpb cpb 0 19 août 08:01 comm
-rw-r--r-- 1 cpb cpb 0 19 août 08:01 coredump_filter
-r--r--r-- 1 cpb cpb 0 19 août 08:01 cpuset
lrwxrwxrwx 1 cpb cpb 0 19 août 08:01 cwd -> /usr/local/sbin
-r----- 1 cpb cpb 0 19 août 08:01 environ
lrwxrwxrwx 1 cpb cpb 0 19 août 08:01 exe -> /bin/ls
$
```

Il faut remarquer que lorsque le pseudo-système de fichiers `/proc` n'est pas accessible, la bibliothèque C doit se rabattre sur la méthode usuelle en remontant de répertoire en répertoire jusqu'à la racine.

Les fonctions de lecture du chemin courant renvoient un pointeur sur le buffer contenant le résultat, ou `NULL` en cas d'échec. Dans ces cas-là, la variable globale `errno`

renferme le type d'erreur. Généralement, il s'agit de `EINVAL` si on a transmis un pointeur illégal, `ERANGE` si le buffer est trop petit, mais on peut aussi rencontrer `EACCES`. Ce dernier cas est assez rare ; c'est une situation où on se trouve dans un répertoire sur lequel on a le droit d'exécution (donc de parcours) mais pas de lecture, et où le pseudo-système de fichiers `/proc` n'est pas monté. Normalement, ceci ne devrait pas se produire sur les systèmes Linux actuels.

Les applications courantes ont rarement besoin de changer de répertoire de travail. Les boîtes de dialogue graphiques pour le chargement ou la sauvegarde de données travaillent en effet avec des chemins d'accès absolus (depuis la racine) ou relatifs (depuis le répertoire courant), mais ne nécessitent pas de changement de répertoire.

Il existe toutefois des processus qui fonctionnent, comme les démons, pendant de longues périodes, en arrière-plan, en se faisant oublier de l'utilisateur. Il faut absolument qu'une telle application revienne à la racine du système de fichiers lors de son initialisation. En effet, dans le cas contraire, il serait impossible de démonter le système de fichiers sur lequel elle se trouve. Par exemple, un démon lancé par un utilisateur depuis son répertoire `/home/abc` ne doit en aucun cas empêcher l'administrateur de démonter temporairement la partition `/home` si le besoin se fait sentir. Celui-ci serait obligé d'avoir recours à l'utilitaire « `fuser -k` » pour tuer le processus bloquant le système de fichiers. Dans ce type de logiciel, on introduira donc un `chdir("/")` en début de programme.

Le programme d'exemple ci-dessous va servir à montrer le comportement de `fchdir()`, qui est légèrement plus compliqué que celui de `chdir()` puisqu'il faut passer par l'ouverture du répertoire avec `open()`. Sous Linux, il existe d'ailleurs un attribut `O_DIRECTORY` pour `open()`, servant à faire échouer cet appel système s'il est invoqué sur autre chose qu'un répertoire. Nous n'avons pas employé cet argument car il n'est pas portable, et les développeurs du noyau précisent qu'il ne doit être utilisé que pour l'implémentation de la fonction de bibliothèque `opendir()`.

Un deuxième point intéressant avec cet exemple est la manière de traiter l'erreur `ERANGE` de `getcwd()` pour augmenter la taille du buffer fourni. Nous utilisons délibérément une taille ridiculement petite au début (16 caractères) pour obliger la routine à réallouer automatiquement une nouvelle zone mémoire.

```
exemple-fchdir.c :
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

void affiche_chemin_courant (void)
{
```

```
char * chemin = NULL;
char * nouveau = NULL;
int    taille = 16;

while (1) {
    if ((nouveau = realloc(chemin, taille)) == NULL) {
        perror("realloc");
        break;
    }
    chemin = nouveau;
    if (getcwd(chemin, taille) != NULL) {
        fprintf(stdout, "%s\n", chemin);
        break;
    }
    if (errno != ERANGE) {
        perror("getcwd");
        break;
    }
    taille *= 2;
}
if (chemin != NULL)
    free(chemin);
}

void change_chemin_courant(const char * nom)
{
    int fd;
    if ((fd = open(nom, O_RDONLY)) < 0) {
        perror(nom);
        return ;
    }
    if (fchdir(fd) < 0)
        perror(nom);
    close(fd);
}

int main (int argc, char * argv[])
{
    int i;

    affiche_chemin_courant();
    for (i = 1; i < argc; i++) {
        change_chemin_courant(argv[i]);
        affiche_chemin_courant();
    }
    return EXIT_SUCCESS;
}
```

Lors de l'exécution, nous allons nous déplacer dans plusieurs répertoires, dont les noms mesurent plus de 16 caractères.

```
$ cp exemple-fchdir /tmp/
$ cd /tmp/
$ ./exemple-fchdir /usr/share/doc/man-pages-fr-3.23/ /etc/ /etc/passwd /bin
/tmp
/usr/share/doc/man-pages-fr-3.23
/etc
/etc/passwd: Not a directory
/etc
/bin
$ pwd
/tmp
$
```

La tentative de déplacement vers `/etc/passwd` (qui est un fichier et pas un répertoire) échoue évidemment. Nous voyons aussi que les changements de répertoire courant du processus exécutable n'ont bien entendu pas affecté le répertoire de travail du shell, comme le montre la commande `pwd` invoquée finalement.

Lorsqu'un programme reçoit un nom de fichier (quelle que soit la méthode utilisée), il arrive qu'il ait besoin de connaître son emplacement précis sur le système. Le chemin transmis peut en effet contenir des références relatives au répertoire courant (`.././src/`) ou utiliser des liens symboliques entre différents répertoires. Pour « nettoyer » un chemin d'accès, il existe une fonction `realpath()` issue de l'univers BSD, mais définie par SUSv4. Suivant les systèmes Unix et les versions de la bibliothèque C, elle peut être déclarée dans `<unistd.h>` ou dans `<stdlib.h>`. Les dernières versions de la Glibc emploient ce dernier fichier d'en-tête.

```
| char * realpath (char * chemin, char * chemin_exact);
```

Le premier argument est la chaîne contenant le chemin qu'on désire traiter. Le second argument est un tableau comprenant au minimum `MAXPATHLEN` caractères, cette constante étant définie dans `<sys/param.h>`. Ce tableau sera rempli par la fonction `realpath()`, qui renverra un pointeur sur lui si elle réussit, ou `NULL` en cas d'erreur.

```
| exemple-realpath.c :
| #include <stdio.h>
| #include <stdlib.h>
| #include <sys/param.h>
```

```
int main (int argc, char * argv[])
{
    char chemin_complet [MAXPATHLEN];
    int i;

    for (i = 0; i < argc; i ++) {
        fprintf(stderr, "%s : ", argv[i]);
        if (realpath(argv[i], chemin_complet) == NULL)
            perror("");
        else
            fprintf(stderr, "%s\n", chemin_complet);
    }
    return EXIT_SUCCESS;
}
```

L'exécution suivante montre que `realpath()` peut aussi bien résoudre les références relatives, comme celles qui sont contenues dans le nom du programme exécutable transmis en argument `argv[0]` de `main()`, que les liens symboliques comme `/usr/tmp` qui pointe traditionnellement vers `/var/tmp`.

```
$ ./exemple-realpath /usr/tmp/
./exemple-realpath : /home/cpb/Documents/Livres/
Developpement_Systeme_sous_Linux/Exemples/chapitre-28/exemple-realpath
/usr/tmp/ : /var/tmp
$
```

Cette fonction peut être très commode dans certains cas, mais elle est toutefois peu conseillée car sa portabilité n'est pas assurée du fait qu'elle n'est pas définie par Posix.

Changement de répertoire racine

Chaque processus dispose d'un pointeur sur son répertoire racine dans le système de fichiers. Pour la plupart des processus, il s'agit du véritable répertoire de départ de toute l'arborescence du système. Il peut toutefois être utile dans certaines conditions de modifier le répertoire qu'un processus considère comme la racine du système de fichiers.

Dans [Cheswick 1991] *An Evening With Berferd*, Bill Cheswick décrit un piège qu'il a construit pour étudier un pirate. Il établit dans un répertoire banal une fausse arborescence avec les sous-répertoires habituels minimaux, et utilise l'appel système `chroot()` pour que son visiteur indésirable croie se trouver dans le véritable système de fichiers complet.

Cet article célèbre est une des premières descriptions des *honeypots* (pots de miel) – système mis en place spécifiquement pour leurrer et piéger les pirates.

L'appel système `chroot()` est une fonction privilégiée demandant la capacité `CAP_SYS_CHROOT`. Il n'y aurait probablement pas de grand risque à la laisser à la disposition des utilisateurs courants, sauf peut-être pour sa capacité à construire des chevaux de Troie destinés à piéger les mots de passe d'un autre utilisateur (en écrivant par exemple un faux `/bin/su`). L'implémentation en Python est `os.chroot()`.

L'application la plus courante de cet appel système est celle qui est utilisée dans le démon de ftp lorsqu'il fonctionne en mode anonyme. Lorsqu'une connexion est établie, le processus bascule sur une nouvelle racine du système de fichiers en `/var/ftp`. Dans ce répertoire, on retrouve les utilitaires indispensables de `/bin` et les bibliothèques partagées de `/lib` (respectivement dans `/var/ftp/bin` et dans `/var/ftp/lib`).

Dans l'exemple suivant, nous allons créer un programme *Set-UID root* qui se déplace dans le répertoire indiqué en premier argument, en fait son répertoire racine, et exécute les commandes passées dans les arguments suivants.

```
exemple-chroot.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    if (argc < 3) {
        fprintf(stderr, "Syntaxe %s chemin commande...\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (chdir(argv [1]) != 0){
        perror("chdir");
        exit(EXIT_FAILURE);
    }
    if (chroot(argv[1]) != 0) {
        perror("chroot");
        exit(EXIT_FAILURE);
    }
    if (seteuid(getuid()) < 0) {
        perror("seteuid");
        exit(EXIT_FAILURE);
    }
    execvp(argv[2], argv + 2);
    perror("execvp");
    return EXIT_FAILURE;
}
```

Nous allons demander à ce programme, après l'avoir installé *Set-UID root*, de lancer la commande `sh`. Pour que cela fonctionne, il faut qu'il puisse trouver ce fichier exéc-

cutable dans `/bin` et les bibliothèques partagées nécessaires dans `/lib`. Nous allons commencer à créer notre fausse arborescence :

```
$ mkdir /tmp/racine
$ mkdir /tmp/racine/bin
$ mkdir /tmp/racine/lib
```

Puis copions-y les fichiers exécutables dont nous aurons besoin (le shell et la commande `ls` par exemple) :

```
$ cp /bin/sh /tmp/racine/bin/
$ cp /bin/ls /tmp/racine/bin/
```

À présent, nous devons déterminer la liste des bibliothèques nécessaires pour ces exécutables. La commande `ldd` est là pour nous aider :

```
$ ldd /tmp/racine/bin/ls /tmp/racine/bin/sh
/tmp/racine/bin/ls:
    linux-gate.so.1 => (0x003ab000)
    libselinux.so.1 => /lib/libselinux.so.1 (0x0049f000)
    librt.so.1 => /lib/librt.so.1 (0x00377000)
    libcap.so.2 => /lib/libcap.so.2 (0x00c78000)
    libacl.so.1 => /lib/libacl.so.1 (0x070bc000)
    libc.so.6 => /lib/libc.so.6 (0x00184000)
    libdl.so.2 => /lib/libdl.so.2 (0x0035b000)
    /lib/ld-linux.so.2 (0x00162000)
    libpthread.so.0 => /lib/libpthread.so.0 (0x0033e000)
    libattr.so.1 => /lib/libattr.so.1 (0x0015a000)
/tmp/racine/bin/sh:
    linux-gate.so.1 => (0x00aef000)
    libtinfo.so.5 => /lib/libtinfo.so.5 (0x07074000)
    libdl.so.2 => /lib/libdl.so.2 (0x0035b000)
    libc.so.6 => /lib/libc.so.6 (0x00184000)
    /lib/ld-linux.so.2 (0x00162000)
$
```

Et copions pour finir toutes les bibliothèques nécessaires (fichiers lib???.so.?):

```
$ cp /lib/libselinux.so.1 /tmp/racine/lib/
$ cp /lib/librt.so.1      /tmp/racine/lib/
$ cp /lib/libcap.so.2     /tmp/racine/lib/
$ cp /lib/libacl.so.1     /tmp/racine/lib/
$ cp /lib/libc.so.6       /tmp/racine/lib/
$ cp /lib/libdl.so.2      /tmp/racine/lib/
$ cp /lib/libpthread.so.0 /tmp/racine/lib/
$ cp /lib/libattr.so.1    /tmp/racine/lib/
$ cp /lib/libtinfo.so.5   /tmp/racine/lib/
$ cp /lib/ld-linux.so.2   /tmp/racine/lib/
```

Nous pouvons démarrer notre exécutable, après l'avoir rendu *Set-UID root*.

```
$ su
Mot de passe :
# chown root.root exemple-chroot
# chmod u+s exemple-chroot
# exit
$ ./exemple-chroot /tmp/racine/ sh
sh-4.1$ ls
bin lib
sh-4.1$ cd /
sh-4.1$ ls
bin lib
sh-4.1$ ls bin/
ls sh
sh-4.1$ ls lib/
ld-linux.so.2 libattr.so.1 libcap.so.2 libpthread.so.0 libselinux.so.1
libacl.so.1   libc.so.6   libdl.so.2  librt.so.1  libtinfo.so.5
sh-4.1$ ls /tmp
ls: cannot access /tmp: No such file or directory
sh-4.1$ exit
exit
$ rm -rf /tmp/racine/
$
```

Nous voyons que le shell lancé avec un répertoire racine spécifique fonctionne parfaitement, mais il a une vue très réduite du système et ne peut en aucun cas accéder aux fichiers et répertoires en dehors de sa nouvelle racine.

Création et suppression de répertoire

Pour créer un nouveau répertoire, ou en supprimer un, il existe deux appels système, `mkdir()` et `rmdir()`, dont le fonctionnement est assez intuitif et rappelle les deux commandes `/bin/mkdir` et `/bin/rmdir` qui sont construites à partir de ces fonctions. Leurs prototypes sont déclarés ainsi dans `<unistd.h>` :

```
int mkdir (const char * repertoire, mode_t mode);
int rmdir (const char * repertoire);
```

En Python, on dispose bien entendu de `os.mkdir()` et `os.rmdir()` aux fonctionnalités identiques.

L'emploi du type `mode_t` pour le second argument de `mkdir()` nécessite l'inclusion supplémentaire de `<fcntl.h>` et de `<sys/types.h>`, comme avec `open()`.

Ces deux appels système renvoient 0 s'ils réussissent, et -1 en cas d'échec. En plus des erreurs liées aux autorisations d'accès ou aux irrégularités concernant le nom fourni, `mkdir()` peut échouer avec le code `ENOSPC` dans `errno` si le disque est saturé ou si le quota attribué à l'utilisateur est rempli, ou avec l'erreur `EEXIST` si le répertoire existe déjà. De son côté, `rmdir()` peut renvoyer surtout les erreurs `EACCES` ou `EPERM` liées aux autorisations d'accès, `ENOTEMPTY` si le répertoire à supprimer n'est pas vide, ou `EBUSY` si on essaye de supprimer le répertoire de travail courant d'un autre processus. Cette dernière erreur n'est pas respectée sur tous les systèmes.

La profondeur des sous-répertoires dans une arborescence n'est pas limitée. Il est donc possible de créer des sous-répertoires imbriqués jusqu'à la saturation du disque. Il peut toutefois y avoir des limitations liées au système de fichiers sous-jacent. Par exemple, les systèmes Iso 9660, sans les extensions *Rock Ridge*, ne permettent pas plus de huit niveaux de sous-répertoires. Ce système de fichiers n'est toutefois utilisé que pour les CD-Rom, et il n'y a donc pas de raisons d'invoquer `mkdir()` dessus.

Le mode fourni en second argument de `mkdir()` sert à indiquer les autorisations d'accès du répertoire nouvellement créé. Comme pour `open()`, on utilise les constantes `S_Ixxx` ou leurs valeurs octales que nous avons vues dans le chapitre 22. Avec un répertoire, les différents bits d'autorisation ont les significations suivantes.

Bit	Valeur	Signification
<code>S_ISGID</code>	02000	Bit <i>Set-GID</i> : les fichiers ou les sous-répertoires créés dans ce répertoire appartiendront automatiquement au même groupe que lui.
<code>S_ISVTX</code>	01000	Bit « <i>Sticky</i> » : les fichiers créés dans ce répertoire ne pourront être écrasés ou effacés que par leur propriétaire ou celui du répertoire. C'est utile pour des répertoires comme <code>/tmp</code> ou des zones de stockage publiques comme <code>/pub/incoming</code> d'un serveur ftp anonyme.

Bit	Valeur	Signification
S_IR...	0..4..	Lecture : on a accès au contenu du répertoire.
S_IW...	0..2..	Écriture : on peut créer un fichier ou un sous-répertoire dans le répertoire.
S_IX...	0..1..	Exécution : on peut entrer dans le répertoire pour en faire son répertoire de travail.

On trouvera donc le plus souvent les valeurs suivantes :

- 00755 : répertoire normal, lisible, accessible en déplacement pour tous, écriture uniquement par le propriétaire ;
- 00700 : répertoire privé, accessible uniquement par son propriétaire (parfois /root) ;
- 01777 : répertoire /tmp par exemple.

Bien entendu, le fait d'interdire le parcours ou à plus forte raison la lecture d'un répertoire empêche également l'accès à tous ses sous-répertoires.

Lors de la création d'un nouveau répertoire, les autorisations fournies sont passées au travers du umask du processus. Plus précisément, la valeur du umask est extraite des permissions demandées. Si le umask vaut 0002 (ce qui est courant) et qu'on demande une création 00777, le répertoire aura en réalité la permission 0775. Il faut donc faire attention de modifier son propre umask (nous le détaillerons dans le prochain chapitre) si on essaye de créer des répertoires accessibles à tous.

L'exemple suivant met en relief ce comportement. Nous essayons à deux reprises de créer un répertoire en mode 00777 et nous vérifions le résultat en invoquant `ls`. La première tentative se fait sans modifier le umask, la seconde après l'avoir ramené à zéro.

```
exemple-mkdir.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main (void)
{
    fprintf(stderr, "Creation repertoire mode rwxrwxrwx : ");
    if (mkdir("repertoire", 0777) != 0) {
        perror("");
        exit(EXIT_FAILURE);
    } else {
        fprintf(stderr, "Ok\n");
    }
    system("ls -ld repertoire");
    fprintf(stderr, "Suppression repertoire : ");
    if (rmdir("repertoire") != 0) {
        perror("");
    }
}
```

```
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Ok\n");
}
fprintf(stderr, "Modification umask\n");
umask(0);
fprintf(stderr, "Creation repertoire mode rwxrwxrwx : ");
if (mkdir("repertoire", 0777) != 0) {
    perror("");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Ok\n");
}
system("ls -ld repertoire");
fprintf(stderr, "Suppression repertoire : ");
if (rmdir("repertoire") != 0) {
    perror("");
    exit(EXIT_FAILURE);
} else {
    fprintf(stderr, "Ok\n");
}
return EXIT_SUCCESS;
}
```

Voici l'exécution de ce programme, montrant bien l'influence du umask.

```
$ ./exemple-mkdir
Creation repertoire mode rwxrwxrwx : Ok
drwxrwxr-x 2 cpb cpb 4096 20 août 09:43 repertoire
Suppression repertoire : Ok
Modification umask
Creation repertoire mode rwxrwxrwx : Ok
drwxrwxrwx 2 cpb cpb 4096 20 août 09:43 repertoire
Suppression repertoire : Ok
$
```

Suppression ou déplacement de fichiers

Pour bien comprendre le comportement des fonctions de suppression ou de déplacement de fichiers, il est nécessaire d'observer la structure des données sur un système de fichiers Unix. Sur le disque, les répertoires sont en réalité de simples listes de noms de fichiers, auxquels sont associés des numéros d'i-nœuds. Un i-nœud est un identifiant unique pour un fichier sur le disque. En revanche, un même fichier peut

avoir plusieurs noms. Il existe une table globale des i-nœuds, permettant de retrouver le contenu réel du fichier.

Dans l'exemple suivant, nous allons créer deux fichiers – en copiant des fichiers système accessibles à tous –, puis utiliser l'utilitaire `ln` (*link*) pour ajouter plusieurs autres liens physiques sur le même fichier. L'option `-i` de la version Gnu de `ls` nous permettra d'observer les numéros d'i-nœuds associés aux entrées du répertoire. Nous vérifierons donc que le même fichier physique dispose de plusieurs noms indépendants.

```
$ ls -li exemple-mkdir
8552928 -rwxrwxr-x 1 cpb cpb 8462 20 août 09:42 exemple-mkdir
$ ln exemple-mkdir azerty
$ ls -li exemple-mkdir azerty
8552928 -rwxrwxr-x 2 cpb cpb 8462 20 août 09:42 azerty
8552928 -rwxrwxr-x 2 cpb cpb 8462 20 août 09:42 exemple-mkdir
$ rm exemple-mkdir
$ ls -li exemple-mkdir azerty
ls: impossible d'accéder à exemple-mkdir: Aucun fichier ou dossier de ce type
8552928 -rwxrwxr-x 1 cpb cpb 8462 20 août 09:42 azerty
$ ln azerty exemple-mkdir
$ ls -li exemple-mkdir azerty
8552928 -rwxrwxr-x 2 cpb cpb 8462 20 août 09:42 azerty
8552928 -rwxrwxr-x 2 cpb cpb 8462 20 août 09:42 exemple-mkdir
$ rm azerty
$
```

Dans le i-nœud correspondant à un fichier est mémorisé le nombre d'entrées de répertoires faisant référence à ce fichier, c'est-à-dire le nombre de noms différents dont un fichier dispose. Ce nombre est affiché dans la troisième colonne de la commande `ls -li`. Lorsque le nombre de liens tombe à zéro, le fichier est effectivement effacé du disque s'il n'est ouvert par aucun processus, mais pas avant.

L'appel système permettant d'effacer un fichier est donc nommé `unlink()`, et non `erase()`, `delete()` ou quelque chose dans ce goût-là, car il sert uniquement à supprimer le lien entre un nom de fichier (une entrée de répertoire) et l'i-nœud correspondant au contenu du fichier. Cet appel est déclaré dans `<unistd.h>` ainsi :

```
| int unlink (const char * nom_fichier);
```

En Python, il existe deux fonctions dont le comportement est identique sous Linux : `os.unlink(nom)` et `os.remove(nom)`.

Dans l'exemple suivant, nous allons créer un fichier, puis le supprimer tout en le conservant ouvert. Nous contrôlerons que son nom disparaît du répertoire (en invoquant `ls -l`), mais que nous pouvons continuer à accéder à son contenu.

```
exemple-unlink.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    FILE * fp;
    char  chaine[27];

    printf("Creation fichier\n");
    fp = fopen("essai.unlink", "w+");
    if (fp == NULL) {
        perror("fopen");
        exit(EXIT_FAILURE);
    }
    fprintf(fp, "ABCDEFGHJKLMNOPQRSTUVWXYZ");
    fflush(fp);
    system("ls -l essai.unlink");
    printf("Effacement fichier\n");
    if (unlink("essai.unlink") < 0) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }
    system("ls -l essai.unlink");
    printf("Relecture du contenu du fichier\n");
    if (fseek(fp, 0, SEEK_SET) < 0) {
        perror("fseek");
        exit(EXIT_FAILURE);
    }
    if (fgets(chaine, 27, fp) == NULL) {
        perror("fgets");
        exit(EXIT_FAILURE);
    }
    printf("Lu : %s\n", chaine);
    printf("Fermeture fichier\n");
    fclose(fp);
    return EXIT_SUCCESS;
}
```

L'exécution confirme nos attentes, le fichier a bien disparu du répertoire lors du second appel de `ls`, mais on peut continuer à en lire le contenu tant qu'on ne l'a pas refermé.

```
$ ./exemple-unlink
Creation fichier
-rw-rw-r-- 1 cpb cpb 26 20 août 09:55 essai.unlink
Effacement fichier
ls: impossible d'accéder à essai.unlink: Aucun fichier ou dossier de
ce type
Relecture du contenu du fichier
Lu : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Fermeture fichier
$
```

L'appel système `rmdir()` permet de supprimer des répertoires, et l'appel `unlink()` des fichiers. Il existe une fonction de la bibliothèque C nommée `remove()` qui leur sert de frontal en invoquant l'appel système correspondant au type d'objet concerné. Elle est déclarée dans `<stdio.h>` :

```
| int remove (const char * nom)
```

Son emploi est très simple, le programme suivant efface les fichiers et répertoires (vides uniquement) dont les noms sont transmis en argument.

Pour déplacer ou renommer un fichier ou un répertoire, il existe un appel système unique, `rename()`, facile d'utilisation :

```
| int rename (const char * ancien_nom, const char * nouveau_nom);
```

Cette routine renvoie 0 si elle réussit ou -1 en cas d'erreurs, qui, en-dehors des problèmes habituels de pointeurs invalides ou de permissions d'accès, peuvent être :

- **EBUSY** : le répertoire qu'on veut écraser ou celui qu'on veut déplacer est utilisé comme répertoire de travail par un processus.
- **EINVAL** : on essaye de déplacer un répertoire vers un de ses propres sous-répertoires. C'est impossible.
- **EISDIR** : on essaye d'écraser un répertoire existant avec un fichier régulier.
- **ENOTEMPTY** : le répertoire qu'on veut écraser n'est pas vide.
- **EXDEV** : on essaye de déplacer un fichier ou un répertoire vers un système de fichiers différent. C'est impossible, il faut passer par une étape de copie, puis de suppression.

Si le fichier ou le répertoire cible existe déjà, il est écrasé. Le noyau s'arrange pour que le déplacement soit atomique et que le nouveau nom ne soit jamais absent du système de fichiers.

Notons que `rmkdir()`, `remove()`, `rename()` ne sont capables de supprimer ou d'écraser un répertoire non vide. Pour cela, il est nécessaire de descendre récursivement jusqu'au dernier sous-répertoire, puis de remonter en effaçant tout le contenu de chaque sous-répertoire, avec `remove()` par exemple. C'est ce que fait par exemple la commande « `rm -r` ».

Fichiers temporaires

Il est très fréquent d'avoir besoin de fichiers temporaires. Ne serait-ce que pour insérer des données au milieu d'un fichier existant, la méthode la plus simple consiste à recopier le fichier original dans un fichier temporaire en ajoutant au passage les nouvelles informations, puis à recopier ou à renommer le fichier temporaire pour écraser l'original.

Il existe plusieurs fonctions pouvant nous aider à obtenir un fichier temporaire, mais leur utilisation correcte est loin d'être triviale. En effet, une application écrite proprement doit éviter à tout prix de laisser des fichiers traîner bien après sa terminaison (dans `/tmp` ou ailleurs). Ce n'est pas toujours simple, surtout si le programme peut être tué abruptement par un signal.

Le premier point consiste à obtenir un nom de fichier unique. Ce nom doit être créé par le système, ce qui nous garantit qu'il ne sera pas réattribué lors d'une autre demande de fichier temporaire.

Par exemple les fonctions `tempnam()`, `tmpnam()` et `mktemp()` créent un nom de fichier unique à partir d'un nom de répertoire de travail et d'un préfixe pour le nom du fichier.

```
char * tempnam (const char * repertoire, const char * prefixe);  
char * tmpnam(char * chaîne);  
char * mktemp(char * motif);
```

Toutefois, ces fonctions sont considérées aujourd'hui comme dangereuses, et ne doivent plus être utilisées. C'est également le cas pour `os.tmpnam()` et `os.tempnam()` en Python. En effet, rien ne garantit qu'entre le moment où le nom du fichier est généré et le moment où on réalise l'ouverture avec `open()`, un autre processus n'a pas utilisé explicitement le même nom de fichier (volontairement ou non), induisant ainsi des risques de collisions entre les deux contenus.

Afin de s'assurer de cet accès unique, le système nous propose une fonction `mkstemp()` qui crée un nom de fichier temporaire unique, et ouvre le fichier pour nous, en garantissant l'atomicité des deux opérations au niveau du système de fichiers. Elle est définie dans `<stdlib.h>` :

```
int mkstemp (char * motif) ;
```

Le motif fourni en argument est une chaîne de caractères représentant le chemin d'accès du fichier temporaire, ainsi qu'un éventuel préfixe, et doit se terminer par six caractères X. La fonction `mkstemp()` remplace ces X par une chaîne aléatoire puis ouvre le fichier, et renvoie un descripteur, ou -1 en cas d'erreur. L'application peut employer ensuite `fdopen()` sur le descripteur si elle désire travailler avec un flux de type `FILE` *.

Une fonction complémentaire `mkdtemp()` réalise la même opération pour la création d'un répertoire temporaire, et renvoie un pointeur sur la chaîne de caractères contenant le nom du répertoire :

```
| char * mkdtemp (char * motif) ;
```

Le second point important pour un fichier temporaire, est d'être impérativement effacé lorsque le programme se termine. Il est particulièrement agaçant pour un administrateur système de voir le répertoire `/tmp` contenir une myriade de fichiers difficiles à distinguer les uns des autres et qu'il faut effacer manuellement de temps à autre (même si on peut automatiser la suppression en vérifiant la date du dernier accès au fichier à l'aide de la commande `find`).

Une application soignée doit s'assurer d'effacer tous les fichiers temporaires qu'elle crée. Pour cela, la méthode la plus simple consiste à demander au système d'éliminer le fichier aussitôt après son ouverture. On se souvient en effet que l'appel système `unlink()` ne fait disparaître le contenu d'un fichier qu'une fois qu'il n'est plus ouvert par aucun processus et qu'il n'a plus de nom dans le système de fichiers. Tant que nous conserverons notre descripteur ou notre flux ouvert, le fichier temporaire sera donc utilisable. En revanche, dès sa fermeture ou la fin du programme, le fichier disparaîtra définitivement. Ceci permet de pallier le problème d'une terminaison violente de l'application par un signal.

Rappelons-nous que `mkstemp()` modifie la chaîne de caractères qui lui est transmise en argument (et qui ne doit donc pas être une chaîne constante allouée dans en mémoire statique bien entendu). Au retour de cette fonction, la chaîne contiendra le nom du fichier, tel que nous pourrions directement le transmettre à `unlink()`.

Nous utiliserons donc un code comme celui-ci :

```
| int fd;  
| char motif[20];  
  
| strcpy(motif, "/tmp/XXXXXX");  
| fd = mkstemp(motif);  
| if (fd < 0) {  
|     perror("mkstemp");  
|     exit(EXIT_FAILURE);  
| }
```



```
}  
unlink(motif);  
/* Utilisation de fd */  
...
```

Le fichier sera donc éliminé automatiquement lors de la fin du programme.

Notons qu'il existe également une fonction `tmpfile()`, définie par SUSv4, qui réalise le même travail, en renvoyant un flux de données. Elle est déclarée dans `<stdio.h>` :

```
FILE * tmpfile (void);
```

En Python, elle est proposée sous le nom `os.tmpfile()`.

Cette routine gère entièrement la création du nom de fichier, l'ouverture exclusive d'un flux et la suppression automatique du fichier temporaire. Son seul défaut c'est que le nom du fichier n'est pas accessible. Il n'est pas possible de le fournir en argument lors d'une invocation avec `system()` d'une autre application (comme `sort` pour trier le fichier). On notera que `tmpfile()` existe sur tous les environnements compatibles Ansi C, mais qu'à la différence de SUSv4, un programme se terminant anormalement sur ces systèmes ne détruira pas nécessairement ses fichiers temporaires.

Recherche de noms de fichiers

Correspondance simple d'un nom de fichier

Lorsqu'on recherche l'ensemble des fichiers dont les noms correspondent à un motif donné, il est possible d'utiliser les routines de manipulation d'expressions régulières que nous avons vues dans le chapitre 18. Toutefois, comme nous l'avons déjà fait remarquer, la syntaxe des expressions régulières n'est pas celle qui est communément adoptée par les shells pour identifier les fichiers. Pour répondre à ce besoin, la bibliothèque C met à notre disposition la fonction `fnmatch()`, mieux adaptée à la comparaison des noms de fichiers et définie dans `<fnmatch.h>` :

```
int fnmatch (const char * motif, const char * nom, int attributs);
```

Cette fonction compare tout simplement le motif transmis en premier argument avec le nom de fichier fourni en seconde position, et renvoie 0 si les chaînes correspondent, ou `FNM_NOMATCH` sinon. Sur certains systèmes, cette routine peut également renvoyer une valeur non nulle autre que `FNM_NOMATCH` en cas d'erreur. Ce n'est pas le cas avec la Glibc.

Le troisième argument permet de configurer certaines options par un *OU* binaire.

Attribut	Signification
FNM_PATHNAME	Avec cette option, les caractères slash '/' sont traités de manière particulière : ils ne sont jamais mis en correspondance avec des caractères génériques. Ce comportement est généralement celui désiré quand on cherche à mettre en correspondance des noms de fichiers.
FNM_FILE_NAME	Il s'agit d'un synonyme de FNM_PATHNAME spécifique à Gnu.
FNM_PERIOD	Le caractère point '.' est traité spécifiquement s'il se trouve en début de nom. Dans ce cas en effet, il ne peut pas être mis en correspondance avec un motif générique. Ce comportement est également celui attendu habituellement lors du traitement des noms de fichiers.
FNM_NOESCAPE	Cette option désactive l'utilisation du caractère backslash '\' pour supprimer la signification particulière d'un caractère (comme * pour indiquer un astérisque).
FNM_CASEFOLD	Cet attribut est une extension Gnu permettant d'ignorer la différence entre les majuscules et les minuscules durant la mise en correspondance.
FNM_LEADING_DIR	Cette extension Gnu permet d'autoriser la mise en correspondance si le motif est la partie initiale du nom et que le reste de ce nom commence par '/'. Ceci revient à accepter le motif /tmp pour /tmp/abcd par exemple. Cette méthode n'est toutefois pas la meilleure pour traiter des descentes de sous-répertoires.

Classiquement, sur un système Unix, les options qu'on utilise sont FNM_PATHNAME et FNM_PERIOD puisqu'elles permettent de comparer les noms de fichiers de la même manière que les interpréteurs de commandes usuels. Dans l'exemple ci-dessous, nous utiliserons la fonction `scandir()` que nous avons déjà étudiée, mais cette fois la sélection des fichiers à afficher sera réalisée en employant `fnmatch()` et non plus `regexec()`.

```
exemple-fnmatch.c :
#include <dirent.h>
#include <fnmatch.h>
#include <stdio.h>
#include <stdlib.h>

static char * motif = NULL;

int selection (const struct dirent * entree)
{
    if (fnmatch(motif, entree->d_name, FNM_PATHNAME | FNM_PERIOD) == 0)
        return 1;
    return 0;
}

int main (int argc, char * argv[])
{
```

```
struct dirent ** liste;
int    nb_entrees;
int    i;

if (argc != 3) {
    fprintf(stderr, "Syntaxe : %s repertoire motif\n", argv[0]);
    exit(EXIT_FAILURE);
}
motif = argv[2];
nb_entrees = scandir(argv[1], & liste, selection, alphasort);
if (nb_entrees <= 0)
    return EXIT_SUCCESS;
for (i = 0; i < nb_entrees; i++) {
    fprintf(stdout, " %s\n", liste[i]->d_name);
    free(liste[i]);
}
fprintf(stdout, "\n");
free(liste);
return EXIT_SUCCESS;
}
```

Nous vérifions la comparaison sur les points spécifiques aux mises en correspondance des noms de fichiers.

```
$ ./exemple-fnmatch /dev "tty[0-9]"
tty0
tty1
tty2
tty3
tty4
tty5
tty6
tty7
tty8
tty9

$ ./exemple-fnmatch /dev "tty[0-9]*"
tty0
tty1
tty10
tty11
tty12
[...]
tty60
tty61
```

```

tty62
tty63
tty7
tty8
tty9

```

Nous voyons bien que le comportement est celui qui est attendu. Bien entendu, les caractères spéciaux comme « * » ou « . » doivent être protégés du shell à l'aide des guillemets pour arriver intacts au cœur de notre programme.

Recherche sur un répertoire total

L'utilisation conjointe de `scandir()` et de `fnmatch()` nous a permis d'extraire une liste de noms de fichiers appartenant à un répertoire donné. Pour accomplir cette tâche automatiquement, la bibliothèque C met à notre disposition les fonctions `glob()` et `globfree()` qui sont également bien plus puissantes. Elles sont déclarées dans `<glob.h>` ainsi :

```

int glob      (const char * motif, int attribut,
               int (* f_erreur) (const char *chemin, int erreur),
               glob_t * vecteur);
void globfree (glob_t * vecteur);

```

La fonction `glob()` prend successivement les arguments suivants :

- le motif qu'on désire mettre en correspondance ;
- des attributs regroupés par un *OU* binaire, que nous détaillerons plus bas ;
- une éventuelle fonction d'erreur qui sera invoquée en cas de problème ;
- une structure de type `glob_t` dans laquelle le résultat sera stocké.

Cette fonction recherche tous les fichiers correspondant au motif transmis, depuis le répertoire de travail courant. Bien entendu, si le motif commence par des références relatives (`../home/`, `bin/`) ou absolues (`/var/tmp/`), le répertoire de recherche est modifié en conséquence. L'ensemble des fichiers sélectionnés est stocké dans une table contenue dans la structure `glob_t` fournie en dernier argument. Cette structure contient les membres suivants.

Nom	Type ^a	Signification
<code>gl_pathc</code>	<code>int</code>	Ce membre contient le nombre de noms ayant été mis en correspondance.
<code>gl_pathv</code>	<code>char **</code>	Ce champ représente un pointeur sur une table de noms de fichiers ayant été sélectionnés.

Nom	Type ^a	Signification
<code>gl_offs</code>	<code>int</code>	Ce champ est rempli avant d'appeler <code>glob()</code> . Il contient le nombre d'emplacements libres que la fonction doit laisser au début de la table <code>gl_pathv</code> . Il n'est utilisé que si la constante <code>GLOB_DOOFS</code> est présente dans les attributs de <code>glob()</code> . Sinon, il est ignoré, même s'il n'est pas nul.
<code>gl_opendir</code>	fonction	Ce membre est une extension Gnu. Il s'agit d'un pointeur sur une fonction permettant de remplacer <code>opendir()</code> . Le prototype de cette fonction doit être compatible avec celui de <code>opendir()</code> . Ceci est principalement utile pour insérer des routines d'encadrement de débogage.
<code>gl_closedir</code>	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant <code>closedir()</code> .
<code>gl_readdir</code>	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant <code>readdir()</code> .
<code>gl_stat</code>	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant <code>stat()</code> , que nous étudierons dans le prochain chapitre.
<code>gl_lstat</code>	fonction	Ce membre est une extension Gnu représentant un pointeur sur une fonction remplaçant <code>lstat()</code> , que nous étudierons dans le prochain chapitre.

a. Les types exacts des pointeurs de fonction ne sont pas développés. On se reportera au besoin au prototype de la fonction remplacée pour implémenter une routine ayant la même interface.

Il est conseillé de se limiter à l'emploi des trois premiers membres uniquement puisqu'ils sont définis par SUSv4.

Les attributs qu'on peut détailler pour paramétrer le fonctionnement de `glob()` sont les suivants.

Nom	Signification
<code>GLOB_APPEND</code>	Le résultat doit être ajouté à celui qui a déjà été obtenu dans la structure <code>glob_t</code> par un appel antérieur à <code>glob()</code> . Ceci permet de combiner le résultat de plusieurs recherches (équivalent ainsi à un <i>OU</i> logique). Cet attribut ne doit pas être utilisé lors de la première invocation de <code>glob()</code> . Le pointeur <code>gl_pathv</code> peut être modifié par <code>realloc()</code> , et l'ancien pointeur n'a peut-être plus de signification lors du retour de <code>glob()</code> . Il faut donc bien relire le contenu de ce membre, sans le sauvegarder entre deux appels.
<code>GLOB_ALTDIRFUNC</code>	Cet attribut est une extension Gnu qui indique que <code>glob()</code> doit utiliser les pointeurs de fonctions des membres <code>gl_opendir</code> , <code>gl_readdir</code> , etc., de la structure <code>glob_t</code> . Ceci n'a pas d'utilité dans les applications courantes, mais peut servir à gérer de manière uniforme des répertoires normaux et des pseudo-systèmes de fichiers comme une liaison <code>ftp</code> ou le contenu d'une archive <code>tar</code> .
<code>GLOB_BRACE</code>	Cette extension Gnu demande que les accolades soient employées à la manière du <code>shellcsh</code> , c'est-à-dire qu'elles indiquent une liste des différentes possibilités, séparées par des virgules.

Nom	Signification
GLOB_DOOFS	<p>Lorsque cet attribut est signalé, la valeur du membre <code>gl_offs</code> de la structure <code>glob_t</code> est utilisée pour réserver des emplacements au début de la table <code>gl_pathv</code>. Les pointeurs ainsi réservés sont initialisés à <code>NULL</code>. Si on se sert de cet attribut, il faut le mentionner à chaque invocation successive éventuelle lors d'un <code>GLOB_APPEND</code>.</p> <p>Ceci est utile pour glisser ensuite dans les emplacements libres des chaînes représentant le nom d'un fichier exécutable à invoquer et ses éventuelles options, avant d'appeler <code>execvp()</code> avec le tableau <code>gl_pathv</code>. Ainsi, on peut simuler le développement des caractères génériques du shell avant de lancer un programme.</p>
GLOB_ERR	<p>Quand <code>glob()</code> rencontre une difficulté lors de la lecture d'un répertoire, il abandonne immédiatement si cet attribut est présent. Sinon, il tente de continuer quand même. Nous verrons plus bas qu'on peut indiquer un pointeur sur un gestionnaire d'erreur dans l'invocation de <code>glob()</code> afin d'affiner la détection de problèmes.</p>
GLOB_MARK	<p>Lorsqu'un sous-répertoire correspond au motif transmis, on le stocke en ajoutant un <i>slash</i> à la fin de son nom.</p>
GLOB_NOCHECK	<p>Si aucune correspondance n'a pu être établie, renvoyer le motif original en guise de résultat plutôt que d'indiquer un échec.</p>
GLOB_NOESCAPE	<p>Cette option est équivalente à <code>FNМ_NOESCAPE</code> de <code>fnmatch()</code> que <code>glob()</code> invoque de manière interne. Elle sert donc à désactiver le comportement particulier du backslash '<code>\</code>', qui permet autrement de protéger les caractères spéciaux.</p>
GLOB_NOMAGIC	<p>Cette extension Gnu permet de renvoyer le motif original si aucune correspondance n'est trouvée, à la manière de <code>GLOB_NOCHECK</code>, mais uniquement si le motif ne contient pas de caractères spéciaux. Dans ce cas, on peut par exemple décider de créer le fichier, chose qui est plus compliquée avec <code>GLOB_NOCHECK</code> seul.</p>
GLOB_NOSORT	<p>Ne pas trier les chemins d'accès par ordre alphabétique. Ceci permet théoriquement de gagner du temps mais, en pratique, le tri en mémoire des noms de fichiers consomme avec les processeurs modernes une durée infime par rapport à la consultation du contenu du répertoire sur le disque.</p>
GLOB_PERIOD	<p>Cette extension Gnu est équivalente à <code>FNМ_PERIOD</code> de <code>fnmatch()</code>. Le caractère point '<code>.</code>' en début de nom ne peut pas être mis en correspondance avec un caractère générique.</p>
GLOB_TILDE	<p>Avec cette extension Gnu, le caractère tilde '<code>~</code>' est traité spécialement lorsqu'il apparaît en tête de motif. Comme avec le shell, le tilde seul ou suivi d'un <i>slash</i> correspond au répertoire personnel de l'utilisateur. Si le tilde est suivi d'un nom d'utilisateur, il représente alors son répertoire personnel. Par exemple les chaînes <code>~/bashrc</code> ou <code>~www/html/</code> sont traitées comme le fait le shell.</p> <p>Si le répertoire personnel n'est pas accessible quelle qu'en soit la raison, le tilde est alors considéré comme un caractère normal appartenant au nom du fichier.</p>

Nom	Signification
GLOB_TILDE_CHECK	Dans cette extension Gnu, le comportement est le même que GLOB_TILDE, à la différence que <code>glob()</code> échoue si la mise en correspondance du tilde avec un répertoire personnel n'est pas possible plutôt que de considérer le tilde comme un caractère normal.

Si une erreur se produit alors que `glob()` tente de lire le contenu d'un répertoire et si le pointeur de fonction fourni en troisième argument n'est pas NULL, celle-ci sera invoquée avec en arguments le nom du chemin d'accès dont la lecture a échoué et le contenu de la variable globale `errno` telle qu'elle a été remplie par les fonctions `opendir()`, `readdir()`, `stat()` ou `lstat()`. Si la fonction d'erreur renvoie une valeur non nulle, ou si l'attribut `GLOB_ERR` a été indiqué, la fonction `glob()` se terminera immédiatement. Sinon, elle tentera de passer à la mise en correspondance suivante.

La valeur de retour de `glob()` est nulle si tout s'est bien passé, ou la fonction renvoie l'une des constantes suivantes en cas d'échec :

- `GLOB_ABORTED` : la routine a été arrêtée à la suite d'une erreur ;
- `GLOB_NOMATCH` : aucune correspondance n'a pu être établie ;
- `GLOB_NOSPACE` : un manque de mémoire a empêché l'allocation de l'espace nécessaire.

Finalement, les données allouées avec `glob()` au sein de la structure `glob_t` peuvent être libérées à l'aide de la fonction `globfree()`.

En Python, l'implémentation de `glob.glob(motif)` est plus simple : elle renvoie directement une liste – éventuellement vide – de tous les noms de fichiers correspondant au motif.

Le programme suivant va simplement afficher les mises en correspondance avec la chaîne passée en argument.

```
exemple-glob.c :
#include <stdio.h>
#include <stdlib.h>
#include <glob.h>

int main (int argc, char * argv[])
{
    glob_t chemins;
    int i;
    int erreur;

    if (argc != 2) {
        fprintf(stderr, "Syntaxe : %s motif\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    erreur = glob(argv[1], 0, NULL, & chemins);
    if ((erreur != 0) && (erreur != GLOB_NOMATCH))
        perror(argv[1]);
    for (i = 0; i < chemins.gl_pathc; i++)
        fprintf(stdout, "%s\n", chemins.gl_pathv[i]);
    globfree(& chemins);
    return EXIT_SUCCESS;
}
```

Ou un équivalent en Python :

```
exemple-glob.py :
#!/usr/bin/python
from __future__ import print_function
import glob
import sys

for f in glob.glob(sys.argv[1]):
    print(f)
```

L'exécution confirme le fonctionnement de `glob()` dans la vérification de répertoires.

```
$ ./exemple-glob '/dev/tty1*'
/dev/tty1
/dev/tty10
/dev/tty11
[...]
/dev/tty18
/dev/tty19
$ ./exemple-glob '*lob*'
exemple-glob
exemple-glob.c
$
```

Développement complet à la manière d'un shell

Le shell offre bien d'autres fonctionnalités que le simple remplacement des caractères génériques. La bibliothèque C propose un couple de fonctions, `wordexp()` et `wordfree()`, particulièrement puissantes, qui assurent l'essentiel des tâches accomplies habituellement par le shell. Ces fonctions travaillent sur un modèle assez semblable à celui de `glob()` et de `globfree()`, mais en utilisant une structure de données de type `wordexp_t`. Elles sont d'ailleurs déclarées dans `<wordexp.h>`. Le concept ici est en effet de remplacer des mots par leur signification après les interprétations suivantes :

Développement du tilde

En début de chaîne, le caractère ‘~’ seul ou suivi d’un slash représente le répertoire personnel de l’utilisateur appelant, déterminé grâce à la variable d’environnement HOME. Si le tilde est directement suivi d’un nom d’utilisateur – déterminé avec la fonction getpwnam() que nous étudierons dans le chapitre 30 –, il s’agit du répertoire personnel de celui-ci. Lorsque le tilde apparaît au cœur d’un nom, il est considéré comme un caractère normal.

Substitution des variables

Les chaînes commençant par un \$ sont remplacées par la variable d’environnement correspondante, avec plusieurs syntaxes possibles.

Syntaxe	Substitution
\$VARIABLE	La valeur de la variable est renvoyée. Le nom de la variable est délimité par le premier caractère blanc rencontré après le \$.
\${VARIABLE}	La valeur de la variable est directement renvoyée. Les accolades permettent de délimiter le nom, pour pouvoir le joindre à d’autres éléments sans insérer d’espace. Ainsi, si VAR vaut TERNIT, E\${VAR}E correspond à ETERNITE.
\${#VARIABLE})	Renvoie le nombre de lettres contenues dans la variable. Ainsi, si la variable VAR contient le mot ETERNITE, \${#VAR} correspond à 8.
\${VARIABLE:-DEFAULT}	Si la variable n’est pas définie ou si elle est vide, renvoyer la valeur par défaut. Sinon renvoyer la valeur de la variable.
\${VARIABLE:=DEFAULT}	Si la variable n’est pas définie ou si elle est vide, la remplir avec la valeur par défaut. Renvoyer la valeur de la variable.
\${VARIABLE:?MESSAGE}	Si la variable n’est pas définie ou si elle est vide, afficher le message sur stderr et échouer. Sinon renvoyer sa valeur.
\${VARIABLE:+VALEUR}	Renvoyer la valeur indiquée si la variable est définie et non vide. Sinon ne rien substituer.
\${VARIABLE##PREFIXE}	Renvoyer la valeur de la variable en ayant retiré le maximum de caractères correspondant au préfixe fourni. On essaye de supprimer le plus grand nombre de caractères possibles. Ainsi si VAR vaut ETERNITE, \${VAR##*T} renvoie E, car on supprime tous les caractères jusqu’au second T.
\${VARIABLE#PREFIXE}	Comme pour le cas précédent, on supprime le préfixe indiqué, mais en retirant le minimum de lettres. Si VAR vaut ETERNITE, \${VAR#*T} renvoie ETERNITE.
\${VARIABLE%%SUFFIXE}	Cette fois-ci, on supprime le suffixe indiqué en essayant de retirer le maximum de caractères. \${VAR%%T*} renvoie uniquement E car on retire tout à partir du premier T.
\${VARIABLE%SUFFIXE}	Symétriquement, on retire le plus petit suffixe possible. \${VAR%T*} renvoie ETERNI.

Évaluation arithmétique et exécution de commande

Les chaînes du type ``commande`` ou `$(commande)` sont remplacées par le résultat de la commande qui est exécutée dans un shell, comme avec la commande `system()`.

Les chaînes du type `$(calcul)` ou `=$((calcul))` sont remplacées par le résultat du calcul. On trouvera le détail des opérations arithmétiques possibles dans la page de manuel du shell. Les expressions sont évaluées de gauche à droite.

La première syntaxe `$(calcul)` est considérée comme obsolète et ne doit dorénavant plus être employée.

Découpage des mots et développement des noms de fichiers

Finalement, la chaîne est découpée en mots en employant les séparateurs du shell, puis les noms de fichiers sont développés en remplaçant tout mot contenant des caractères génériques par la liste des fichiers dont les noms lui correspondent.

Les fonctions que la bibliothèque C nous fournit pour analyser une chaîne à la manière du shell sont les suivantes :

```
int wordexp(const char * chaîne,
            wordexp_t * mots, int attributs);
void wordfree (wordexp_t * mots);
```

La fonction `wordexp()` prend la chaîne qu'on lui fournit en premier argument, effectue toutes les transformations que nous avons aperçues précédemment, et renvoie une liste des mots trouvés dans la structure `wordexp_t` sur laquelle on passe un pointeur en second argument. Cette structure contient les membres suivants.

Nom	Type	Signification
<code>we_wordc</code>	<code>int</code>	Le nombre de mots contenus dans le tableau suivant.
<code>we_wordv</code>	<code>char **</code>	Le tableau proprement dit contenant des pointeurs sur des chaînes de caractères correspondant aux différents mots.
<code>we_offs</code>	<code>int</code>	Comme le champ <code>gl_offs</code> de la structure <code>glob_t</code> , ce membre permet de réserver de l'espace au début de la table <code>we_wordv</code> , à condition d'utiliser l'attribut <code>WRDE_DOOFFS</code> .

Les différents attributs qu'on peut transmettre à `wordexp()` sont combinés avec un *OU* binaire parmi les constantes suivantes.

Nom	Rôle
WRDE_APPEND	Ajouter les mots trouvés à ceux qui sont déjà présents dans la structure <code>wordexp_t</code> . Cette option ne doit pas être utilisée lors du premier appel de <code>wordexp()</code> .
WRDE_DOOFFS	Réserver dans la table <code>we_wordv</code> la place indiquée dans le membre <code>we_offs</code> de la structure <code>wordexp_t</code> .
WRDE_NOCMD	Ne pas effectuer la substitution de commandes. Ceci évite qu'un programme <i>Set-UID</i> exécute des commandes arbitraires fournies par l'utilisateur. Si on essaye de transmettre une chaîne contenant <code>`commande`</code> ou <code>\$(commande)</code> , <code>wordexp()</code> échoue.
WRDE_REUSE	Réutiliser une structure <code>wordexp_t</code> ayant déjà servi. Ceci évite de libérer les données à chaque fois.
WRDE_SHOWERR	Lors de la substitution de commandes, pour utiliser le même flux d'erreur standard que le processus en cours. Ceci permet d'afficher des éventuels messages de diagnostic. Par défaut, ces erreurs ne sont pas visibles.
WRDE_UNDEF	Considérer qu'il y a une erreur si on essaye de consulter une variable d'environnement non définie.

La fonction `wordexp()` renvoie zéro si elle réussit, ou l'une des constantes suivantes en cas d'erreur :

- `WRDE_BADCHAR` : la chaîne contient un caractère interdit, comme `<`, `>`, `&`, `;`, `|` ou `\n` ;
- `WRDE_BADVAL` : une variable est indéfinie et on a utilisé l'option `WRDE_UNDEF` ;
- `WRDE_CMDSUB` : on a essayé de faire une substitution de commandes alors que l'option `WRDE_NOCMD` a été demandée ;
- `WRDE_NOSPACE` : pas assez de mémoire pour allouer la table ;
- `WRDE_SYNTAX` : une erreur de syntaxe a été détectée, comme des accolades manquantes par exemple.

La fonction `wordfree()` permet bien sûr de libérer la mémoire occupée par les tables contenues dans la structure passée en argument. Dans l'exemple suivant, nous allons construire un microshell n'ayant qu'une seule commande interne, `set`, permettant de configurer une variable d'environnement. Toutes les autres commandes seront exécutées en employant `execvp()`.

```
exemple-wordexp.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <wordexp.h>
#include <sys/wait.h>

#define LG_LIGNE 256
```

```

void affiche_erreur (int numero)
{
    switch (numero) {
        case WRDE_BADCHAR :
            fprintf(stderr, "Caractere interdit\n");
            break;
        case WRDE_BADVAL :
            fprintf(stderr, "Variable indefinie\n");
            break;
        case WRDE_CMDSUB :
            fprintf(stderr, "Invocation de commande interdite\n");
            break;
        case WRDE_NOSPACE :
            fprintf(stderr, "Pas assez de memoire\n");
            break;
        case WRDE_SYNTAX :
            fprintf(stderr, "Erreur de syntaxe\n");
            break;
        default :
            break;
    }
}

int main (void)
{
    char ligne[LG_LIGNE];
    wordexp_t mots;
    int erreur;
    pid_t pid;

    while (1) {
        fprintf(stdout, "-> ");
        if (fgets(ligne, LG_LIGNE, stdin) == NULL)
            break;
        if (strlen(ligne) == 0)
            continue;
        if (ligne[strlen(ligne) - 1] == '\n')
            ligne[strlen(ligne) - 1] = '\0';
        if ((erreur = wordexp(ligne, & mots, WRDE_SHOWERR)) != 0) {
            affiche_erreur(erreur);
            goto fin_boucle;
        }
        if (mots.we_wordc == 0)
            goto fin_boucle;
        if (strcmp(mots.we_wordv[0], "set") == 0) {
            if (mots.we_wordc != 3) {
                fprintf(stderr, "syntaxe : set variable valeur\n");
            }
        }
    }
}

```

```

        goto fin_boucle;
    }
    if (setenv(mots.we_wordv[1], mots.we_wordv[2], 1) < 0)
        perror("");
    goto fin_boucle;
}
if ((pid = fork()) < 0) {
    perror("fork");
    goto fin_boucle;
}
if (pid == 0) {
    execvp(mots.we_wordv[0], mots.we_wordv);
    perror(mots.we_wordv[0]);
    exit(EXIT_FAILURE);
} else {
    wait(NULL);
}
fin_boucle :
    wordfree(& mots);
}
fprintf(stdout, "\n");
return EXIT_SUCCESS;
}

```

Ce petit programme est simpliste, mais il est déjà étonnamment puissant :

```

$ ./exemple-wordexp
-> set VAR ETERNITE
-> echo $VAR
ETERNITE
-> echo ${VAR#*T}
ERNITE
-> ls /var/www
cgi-bin  error   html    icons
-> set X 1
-> set Y $(( $X + 2 ))
-> echo $Y
3
-> echo $(( $Y * 25 ))
75
-> set DMY $(date "+%d_%m_%Y")
-> echo $DMY
20_08_2010
-> (Contrôle-D)
$

```

Bien entendu, nous sommes loin de la réalisation d'un véritable shell, capable d'interpréter les caractères spéciaux de redirection (< ou >), les lancements de commandes en arrière-plan (&), etc. Malgré tout, nous voyons qu'avec quelques lignes de code il est déjà possible d'utiliser facilement la puissance des fonctions `wordexp()` et `wordfree()` de la bibliothèque C. Répétons qu'il faut être très prudents avec la substitution de commande, qui est un moyen très efficace pour introduire un trou de sécurité dans un programme *Set-UID* ou dans un démon. On privilégiera donc systématiquement l'option `WRDE_NOCMD`, à moins d'avoir vraiment besoin de cette fonctionnalité.

Descente récursive de répertoires

Pour l'instant, nous avons observé le moyen d'accéder au contenu d'un unique répertoire avec la fonction `scandir()`. Il est parfois nécessaire de descendre récursivement une arborescence en explorant tous ses sous-répertoires. Ceci peut se réaliser à l'aide de la commande `ftw()` ou de son dérivé `nftw()`, déclarées toutes deux dans `<ftw.h>`. Leurs prototypes sont :

```
int ftw (const char * depart,
        int (* fonction) (const char * nom,
                          const struct stat * etat,
                          int attributs),
        int profondeur);
```

Pour pouvoir utiliser `nftw()`, il faut définir la constante symbolique `_XOPEN_SOURCE` et lui donner la valeur 500 avant d'inclure `<ftw.h>`.

```
int nftw (const char * depart,
         int (* fonction) (const char * nom,
                          const struct stat * etat,
                          int attributs,
                          struct FTW * status),
         int profondeur,
         int options);
```

Ces deux fonctions partent du répertoire dont le chemin leur est fourni en premier argument. Elles parcourent son contenu en invoquant la fonction fournie en second argument pour chaque point d'entrée du répertoire. Ensuite, elles descendent récursivement dans toute l'arborescence. Une fois arrivées à la profondeur indiquée en troisième argument, ces fonctions devront refermer des descripteurs pour les réemployer à nouveau. Le nombre total de descripteurs disponibles simultanément pour un processus est en effet limité. Il y a deux différences entre ces deux fonctions : la première tient à un argument supplémentaire dans la routine invoquée pour chaque

entrée des répertoires, la seconde réside dans le quatrième argument de `nftw()`, qui permet de préciser son comportement.

Les routines appelées pour chaque entrée d'un répertoire reçoivent tout d'abord le nom de cet élément. Leur second argument est une structure `stat` que nous étudierons en détail dans le prochain chapitre, mais qui contient diverses informations comme les dernières dates de modification ou d'accès, le numéro d'i-nœud, la taille du fichier, etc. Le troisième argument est un indicateur du type d'entrée, qui peut prendre l'une des valeurs suivantes.

Nom	Signification
FTW_D	L'élément est un répertoire.
FTW_DNR	L'élément est un répertoire dont on ne pourra pas lire le contenu.
FTW_DP	L'élément est un répertoire dont on a visité tous les sous-répertoires. Ceci n'est défini qu'avec <code>nftw()</code> , lorsque l'option <code>FTW_DEPTH</code> est utilisée.
FTW_F	L'élément est un fichier. Il faut toutefois se méfier car <code>ftw()</code> considère comme fichier tout ce qui n'est pas un répertoire.
FTW_NS	L'appel système <code>stat()</code> a échoué, le second argument de la routine n'est pas valide. Ce cas ne devrait normalement jamais se produire.
FTW_SL	L'élément est un lien symbolique. Comme <code>ftw()</code> suit les liens symboliques, ceci ne peut apparaître que si le lien pointe vers une destination inexistante. En revanche, pour <code>nftw()</code> , cet attribut apparaît si l'option <code>FTW_PHYS</code> est utilisée.
FTW_SLN	L'élément est un lien symbolique pointant vers une destination inexistante. Cet argument n'apparaît qu'avec <code>nftw()</code> .

La fonction invoquée lors de la descente récursive de `nftw()` reçoit donc un quatrième argument se présentant sous la forme d'une structure contenant les membres suivants.

Nom	Type	Signification
<code>base</code>	<code>int</code>	Il s'agit de la taille de la partie nom du fichier reçu en premier argument. Le reste de la chaîne est le chemin d'accès au fichier.
<code>level</code>	<code>int</code>	Il s'agit de la profondeur d'exploration de l'arborescence. La profondeur du répertoire de départ vaut 0.

Les options supplémentaires que propose `nftw()` sont les suivantes.

Nom	Rôle
<code>FTW_CHDIR</code>	Le processus change son répertoire de travail pour aller dans le répertoire exploré, avant d'appeler la routine fournie en second argument.

Nom	Rôle
FTW_DEPTH	L'exploration se fait en profondeur d'abord, en descendant au plus bas avant de remonter dans les répertoires. Les répertoires seront alors détectés après leurs sous-répertoires (on recevra l'attribut FTW_DP et non FTW_D). Cette option permet de vider récursivement une arborescence à la manière de <code>rm -r</code> .
FTW_MOUNT	La fonction <code>nftw()</code> se limitera aux répertoires se trouvant sur le même système de fichiers que le répertoire de départ.
FTW_PHYS	Ne pas suivre les liens symboliques. La routine de l'utilisateur sera invoquée avec l'attribut <code>F_SLN</code> . Si le lien pointe sur une destination inexistante, l'attribut <code>F_SLN</code> sera alors utilisé.

Si la fonction appelée pour un élément renvoie une valeur non nulle, `ftw()` arrête son exploration, libère les structures de données dynamiques qu'elle utilisait, et renvoie cette valeur. Sinon, elle se terminera lorsque tout le parcours sera fini, et renverra 0.

L'exemple ci-dessous est simplement un effacement récursif d'une arborescence. On prend garde à effacer les liens symboliques sans les suivre et à descendre jusqu'au bout des sous-répertoires avant de commencer à les vider.

```
exemple-nftw.c :
#define _XOPEN_SOURCE 500
#include <ftw.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int routine (const char * nom, const struct stat * etat,
             int attribut, struct FTW * status)
{
    if (attribut == FTW_DP)
        return rmdir(nom);
    return unlink(nom);
}

int main (int argc, char * argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        if (nftw(argv[i], routine, 32,
                  FTW_DEPTH | FTW_PHYS | FTW_MOUNT) != 0)
            perror(argv[i]);
    return EXIT_SUCCESS;
}
```


Nous allons créer quelques fichiers et sous-répertoires pour pouvoir les supprimer par la suite.

```
$ mkdir -p tmp/repA tmp/repB tmp/repA/repAA
$ touch tmp/fic1 tmp/repA/ficA1 tmp/repB/ficB1 tmp/repA/repAA/ficAA1
$ find tmp/
tmp/
tmp/fic1
tmp/repB
tmp/repB/ficB1
tmp/repA
tmp/repA/repAA
tmp/repA/repAA/ficAA1
tmp/repA/ficA1
$ ./exemple-nftw tmp/ /etc
/etc: Permission denied
$ find tmp/
find: "tmp/": Aucun fichier ou dossier de ce type
$
```

Conclusion

Nous avons vu dans ce chapitre l'essentiel des fonctions permettant de travailler au niveau d'un répertoire, que ce soit pour en lire le contenu, créer des sous-répertoires, effacer ou déplacer des fichiers.

Les fonctions de mises en correspondance que nous avons étudiées pour rechercher des noms de fichiers sont très performantes, et permettent d'ajouter facilement à une application une interface puissante avec le système.

Pour avoir plus de détails sur la syntaxe des commandes arithmétiques du shell ou de la substitution des variables, on pourra se reporter à [BLAESS 2008] *Scripts Shells Linux et Unix par la pratique*.

Attributs des fichiers

Pour le moment, nous avons étudié les fichiers sous l'angle de leur contenu, des moyens d'y accéder et d'un point d'entrée dans un répertoire. Les fichiers existent pourtant également en tant qu'entités propres sur le disque, et c'est sous ce point de vue que nous allons les observer dans ce chapitre. Nous examinerons tout d'abord les différentes informations que le système peut nous fournir sur un fichier, puis nous nous intéresserons successivement à tout ce qui concerne la taille du fichier, ses permissions d'accès, ses propriétaire et groupe, ainsi que les divers horodatages qui lui sont associés.

Informations associées à un fichier

Les informations que nous traitons dans ce chapitre sont indépendantes du contenu et du nom du fichier. Comme nous le verrons plus loin, un même fichier peut avoir plusieurs noms, dans un ou plusieurs répertoires. Pourtant, toutes les représentations de ce fichier partagent un certain nombre d'informations communes. Ces données peuvent être obtenues avec les appels système `stat()`, `fstat()` ou `lstat()`. Tous trois fournissent leurs résultats dans une structure `stat`, définie dans `<sys/stat.h>`, que nous avons déjà rencontrée dans le chapitre précédent, à propos de `ftw()`. Cette structure renfermant en effet toutes les caractéristiques principales d'un fichier, on la retrouve très souvent. En Python, les fonctions `os.stat()`, `os.fstat()` et `os.lstat()` renvoient un objet dont les membres ont les mêmes noms que les champs de cette structure.

La structure `stat` est définie par SUSv4 comme contenant les membres suivants :

Nom	Type	Signification
<code>st_mode</code>	<code>mode_t</code>	Ce champ contient les permissions d'accès au fichier ainsi que le type de ce dernier (répertoire, socket, fichier normal...). Les autorisations d'accès peuvent être modifiées avec l'appel système <code>chmod()</code> . Pour déterminer le type du fichier, il existe des macros décrites plus bas.
<code>st_ino</code>	<code>ino_t</code>	La norme SUSv4 parle de numéro de référence du fichier. Il s'agit d'un identifiant unique permettant d'accéder au contenu du fichier. En pratique, sous Linux comme avec la majorité des Unix, on le nomme plutôt numéro d'i-nœud. Ce numéro est unique au sein d'un même système de fichiers.
<code>st_dev</code>	<code>dev_t</code>	Ce membre comprend le numéro du périphérique qui contient le système de fichiers auquel se rapporte le numéro d'i-nœud. Le couple <code>st_ino</code> et <code>st_dev</code> permet de définir de manière unique un fichier. La valeur <code>st_dev</code> n'est pas obligatoirement conservée entre deux redémarrages de la machine. Elle peut par exemple dépendre de l'ordre de détection des disques. On ne doit donc pas considérer qu'elle a une durée de vie plus longue que celle de l'exécution d'un processus.
<code>st_nlink</code>	<code>nlink_t</code>	Un fichier pouvant avoir plusieurs noms, ce champ en conserve le nombre. Il s'agit donc du nombre de liens physiques sur l'i-nœud. Lors d'un appel système <code>unlink()</code> , cette valeur est décrémentée. Le fichier n'est véritablement supprimé que lorsque <code>st_nlink</code> arrive à zéro.
<code>st_uid</code>	<code>uid_t</code>	Ce champ contient l'UID du propriétaire du fichier. Il n'y a qu'un seul propriétaire pour un fichier, même si celui-ci dispose de plusieurs noms. Ce champ peut être modifié par l'appel système <code>chown()</code> .
<code>st_gid</code>	<code>gid_t</code>	Comme <code>st_uid</code> , ce membre identifie l'appartenance du fichier, mais cette fois-ci à un groupe. La valeur est modifiée également par l'appel système <code>chown()</code> .
<code>st_size</code>	<code>off_t</code>	La taille du fichier est ici mesurée en octets. Elle n'a de véritable signification que pour les fichiers normaux, pas pour les liens symboliques ni pour les fichiers spéciaux de périphérique.
<code>st_atime</code>	<code>time_t</code>	Ce membre contient la date du dernier accès au fichier. Elle est mise à jour lors de toute lecture ou écriture du contenu du fichier.
<code>st_ctime</code>	<code>time_t</code>	La date de changement du statut du fichier est mise à jour à chaque consultation ou modification du contenu du fichier, mais également lors de la modification de ses caractéristiques (avec <code>chmod()</code> , <code>chown()</code> ...).
<code>st_mtime</code>	<code>time_t</code>	Cette date est celle de la dernière modification du contenu du fichier. Elle n'est pas affectée par les changements de propriétaire, de permissions...
<code>st_rdev</code>	<code>dev_t</code>	Pour un fichier spécial représentant un périphérique, il s'agit des numéros d'identification majeur et mineur. Le numéro majeur est contenu dans le poids fort, le mineur dans le poids faible.

Nom	Type	Signification
st_blksize	long	Il s'agit de la taille de bloc la mieux adaptée pour les entrées-sorties sur ce fichier. Elle est mesurée en octets. Cette valeur est très utile, nous l'avons vu au chapitre 21, lorsqu'on désire configurer la taille d'un buffer de sortie avec <code>setvbuf()</code> . Nous sommes assurés en utilisant un buffer, dont la taille est un multiple de <code>st_blksize</code> , d'avoir des entrées-sorties par flux optimales pour ce système de fichiers.
st_blocks	long	Cette valeur représente la taille effectivement allouée pour le fichier, telle qu'elle est mesurée par l'utilitaire <code>du</code> . Ce champ est évalué en nombre de blocs, mais la taille même des blocs n'est pas disponible de manière portable. On évitera d'utiliser ce membre.

Les prototypes des fonctions de la famille `stat()` sont déclarés dans `<unistd.h>`.

```
int stat(const char * nom_fichier, struct stat * infos);
int fstat (int descripteur, struct stat * infos);
int lstat (const char * nom_fichier, struct stat * infos);
```

La fonction `stat()` prend en premier argument un nom de fichier et remplit la structure `stat` sur laquelle on lui a transmis un pointeur en seconde position. Pour accéder aux informations concernant un fichier, il faut simplement avoir un droit de parcours (exécution) dans le répertoire le contenant, ainsi que dans les répertoires parents.

L'appel système `lstat()` fonctionne comme `stat()`, mais lorsque le nom correspond à un lien symbolique, il fournit les informations concernant le lien lui-même et pas celles correspondant au fichier visé par le lien.

Enfin, `fstat()` utilise un descripteur de fichier déjà ouvert, ce qui peut permettre par exemple de vérifier le type de descripteur associé aux flux d'entrée ou de sortie standard (`STDIN_FILENO`, `STDOUT_FILENO`).

Pour vérifier le type d'un fichier, il faut utiliser une macro qui prend en argument le champ `st_mode` de la structure `stat`. Ces macros sont définies par SUSv4 et prennent une valeur vraie si le fichier correspond au type indiqué.

Macro	Signification
<code>S_ISBLK(infos->st_mode)</code>	Fichier spécial de périphérique bloc.
<code>S_ISCHR(infos->st_mode)</code>	Fichier spécial de périphérique caractère.
<code>S_ISDIR(infos->st_mode)</code>	Répertoire.
<code>S_ISFIFO(infos->st_mode)</code>	FIFO (tube nommé).
<code>S_ISLNK(infos->st_mode)</code>	Lien symbolique.
<code>S_ISREG(infos->st_mode)</code>	Fichier régulier.
<code>S_ISSOCK(infos->st_mode)</code>	Socket.

En Python, les fonctions `stat.S_ISBLK()`, `stat.S_ISCHR()`, etc., jouent le même rôle. Pour connaître les autorisations d'accès du fichier, on prend des constantes symboliques qu'on compare en utilisant un *ET* binaire au champ `st_mode` de la structure `stat`. Ces constantes ont été présentées dans le chapitre 22, lors de la description du troisième argument de `open()`. Il s'agit des valeurs `S_IRUSR`, `S_IXGRP`, etc. On peut éventuellement se permettre, au risque d'une portabilité légèrement amoindrie, de consulter directement la valeur `st_mode`, en effectuant un *ET* binaire avec le masque octal 07777, comme nous l'avions signalé à propos de `open()`.

Rappelons qu'un fichier ne possédant pas l'autorisation d'exécution pour son groupe `S_IXGRP`, mais ayant en revanche l'attribut *Set-GID* (`I_SGID`) est en réalité un fichier sur lequel un verrouillage strict s'applique, comme nous en avons vu des exemples à la fin du chapitre 22.

Le programme suivant permet de connaître le type et les autorisations d'accès à un fichier. Si on lui transmet un ou plusieurs noms en arguments, il utilise `stat()` pour obtenir les informations. Si on ne lui envoie rien, il invoque `fstat()` pour afficher les données correspondant à ses flux d'entrée et de sortie standard.

```
exemple-stat.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

void affiche_status (struct stat * status)
{
    if (S_ISBLK(status->st_mode))
        fprintf(stderr, "bloc ");
    else if (S_ISCHR(status->st_mode))
        fprintf(stderr, "caractere ");
    else if (S_ISDIR(status->st_mode))
        fprintf(stderr, "repertoire ");
    else if (S_ISFIFO(status->st_mode))
        fprintf(stderr, "fifo ");
    else if (S_ISLNK(status->st_mode))
        fprintf(stderr, "lien ");
    else if (S_ISREG(status->st_mode))
        fprintf(stderr, "fichier ");
    else if (S_ISSOCK(status->st_mode))
        fprintf(stderr, "socket ");
    fprintf(stderr, "u:");
    fprintf(stderr, status->st_mode & S_IRUSR ? "r" : "-");
    fprintf(stderr, status->st_mode & S_IWUSR ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXUSR ? "x" : "-");
    fprintf(stderr, " g:");
    fprintf(stderr, status->st_mode & S_IRGRP ? "r" : "-");
```

```

    fprintf(stderr, status->st_mode & S_IWGRP ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXGRP ? "x" : "-");
    fprintf(stderr, " o:");
    fprintf(stderr, status->st_mode & S_IROTH ? "r" : "-");
    fprintf(stderr, status->st_mode & S_IWOTH ? "w" : "-");
    fprintf(stderr, status->st_mode & S_IXOTH ? "x" : "-");
    fprintf(stderr, "\n");
}

int main (int argc, char * argv[])
{
    struct stat status;
    int i;

    if (argc == 1) {
        fprintf(stderr, "stdin : ");
        if (fstat(STDIN_FILENO, & status) < 0)
            perror("");
        else
            affiche_status(& status);
        fprintf(stderr, "stdout : ");
        if (fstat(STDOUT_FILENO, & status) < 0)
            perror("");
        else
            affiche_status(& status);
    } else {
        for (i = 1; i < argc ; i++) {
            fprintf(stderr, "%s : ", argv[i]);
            if (stat(argv[i], & status) < 0)
                perror("");
            else
                affiche_status(& status);
        }
    }
    return EXIT_SUCCESS;
}

```

Lors de l'exécution de ce programme, nous essayons d'examiner les principaux types de fichiers, périphériques de type bloc ou caractère, répertoires, fichiers normaux et FIFO obtenus grâce à une redirection du shell.

```

$ ./exemple-stat /dev/sda1 /dev/ttyS0 /etc/passwd
/dev/sda1 : bloc u:rw- g:rw- o:---
/dev/ttyS0 : caractere u:rw- g:rw- o:---
/etc/passwd : fichier u:rw- g:r-- o:r--
$ ./exemple-stat /etc/
/etc/ : repertoire u:rw- g:r-x o:r-x
$ ./exemple-stat | cat

```

```

stdin : caractere u:rw- g:-w- o:---
stdout : fifo u:rw- g:--- o:---
$ cat /dev/null | ./exemple-stat
stdin : fifo u:rw- g:--- o:---
stdout : caractere u:rw- g:-w- o:---
$

```

Un exemple assez proche en Python :

```

exemple-stat.py :
#!/usr/bin/python
from __future__ import print_function
import os
import stat
import sys

sys.argv.pop(0)
for f in sys.argv:
    try:
        s = os.stat(f)
        if (stat.S_ISBLK(s.st_mode)):
            print(f, "bloc")
        if (stat.S_ISCHR(s.st_mode)):
            print(f, "caractere")
        if (stat.S_ISDIR(s.st_mode)):
            print(f, "repertoire")
        if (stat.S_ISFIFO(s.st_mode)):
            print(f, "fifo")
        if (stat.S_ISLNK(s.st_mode)):
            print(f, "lien")
        if (stat.S_ISREG(s.st_mode)):
            print(f, "fichier")
        if (stat.S_ISSOCK(s.st_mode)):
            print(f, "socket")
    except:
        print(f, "erreur")

```

Autorisations d'accès

Pour modifier les autorisations d'accès à un fichier, on utilise l'un des appels système `chmod()` ou `fchmod()` déclarés dans `<sys/stat.h>` :

```

int chmod(const char * nom_fichier, mode_t mode);
int fchmod (int descripteur, mode_t mode);

```


On voit que `fchmod()` agit directement sur un descripteur de fichier déjà ouvert alors que `chmod()` travaille sur un nom de fichier. Il en est de même pour `os.fchmod()` et `os.chmod()` en Python. Pour être autorisé à changer les autorisations associées à un fichier, il faut que l'UID effectif du processus appelant soit égal à 0 (root) ou à celui du propriétaire du fichier (indiqué dans le champ `st_uid` de la structure `stat`). Si toutefois le GID du fichier (champ `st_gid`) n'est égal à aucun des groupes auxquels appartient le processus appelant, et si l'UID effectif de ce dernier n'est pas nul, le bit `S_ISGID` (*Set-GID*) sera silencieusement effacé.

Lorsqu'un processus accède véritablement à un fichier grâce aux appels système `open()` ou `execve()`, l'UID pris en compte pour vérifier les autorisations est l'UID effectif du processus et pas son UID réel. Cela pose un problème pour les processus *Set-UID*. Supposons que nous écrivons un programme pilotant une interface spécifique personnalisée (automate industriel, instrument de mesure scientifique...). Cette application, pour dialoguer avec notre dispositif, doit employer des appels système privilégiés. Pour avoir le droit d'exploiter ces appels et pour pouvoir être employée par n'importe quel utilisateur, l'application doit être installée *Set-UID root*. Notre processus dispose donc de la toute-puissance de root. Toutefois, nous désirons également que l'utilisateur puisse sauvegarder des données dans ses propres fichiers. On ne peut pas utiliser directement l'appel `open()`, car l'UID effectif étant celui de root, notre utilisateur pourrait écraser n'importe quel fichier. Il est possible d'abandonner temporairement nos privilèges, comme nous l'avons étudié dans le chapitre 2, mais c'est fastidieux si on alterne régulièrement des entrées-sorties avec `inb()`-`outb()` et des ouvertures de fichiers.

Il existe donc un appel système nommé `access()` qui permet de vérifier si un processus peut exécuter ou non un accès particulier à un fichier en se fondant sur son UID réel (celui de l'utilisateur qui a lancé le processus). En Python, il s'agit de `os.access()`. En C, il est déclaré dans `<unistd.h>` :

```
| int access (const char * nom_fichier, int mode);
```

Le mode qu'on transmet en second argument correspond à l'utilisation qu'on désire faire du fichier. Il existe quatre constantes symboliques :

Nom	Signification
F_OK	Le fichier existe-t-il ?
R_OK	Puis-je lire le contenu du fichier ?
W_OK	Puis-je écrire dans le fichier ?
X_OK	Puis-je exécuter le fichier ?

Attention, la vérification n'a lieu qu'en ce qui concerne les bits d'autorisation du fichier, le test d'exécution peut très bien réussir alors que `execve()` échouera si le fichier n'est pas dans un format exécutable correct.

La valeur renvoyée par cet appel système est nulle si l'accès est autorisé, et vaut -1 sinon. La variable globale `errno` est dans ce cas remplie.

On emploie donc `access()` immédiatement avant l'appel `open()` ou `execve()` correspondant. Il faut être conscient du risque potentiel concernant la sécurité, car il existe un petit délai entre la vérification des autorisations avec l'UID réel et l'ouverture du fichier avec l'UID effectif. Un utilisateur malintentionné pourrait profiter de ce délai pour supprimer le fichier banal qu'il proposait de modifier et le remplacer par un lien matériel vers un fichier système (`/etc/passwd` par exemple) que l'UID effectif du processus pourra ouvrir. Pour éviter ce genre de désagrément, on préférera autant que possible perdre temporairement nos privilèges pour retrouver l'identité effective de l'utilisateur ayant lancé le programme, en employant les appels système `setreuid()` ou `setresuid()`.

Propriétaire et groupe d'un fichier

Lorsqu'un processus *Set-UID* vérifie avec `access()` s'il peut écrire ou créer un fichier et qu'il le crée effectivement, avec `open()` ou `creat()`, ce nouveau fichier possède les UID et GID effectifs du processus appelant. Il lui faut donc modifier les appartenances du nouveau fichier pour les faire correspondre à celles de l'utilisateur.

Seul un processus ayant un UID effectif nul ou la capacité `CAP_CHOWN` peut modifier le propriétaire d'un fichier. En revanche, le propriétaire d'un fichier peut l'affecter à n'importe quel groupe auquel il appartient lui-même. Les identités du propriétaire et du groupe sont communes à toutes les occurrences du fichier à travers ses différents noms (via des liens physiques).

Pour modifier l'UID ou le GID d'un fichier, on emploie les appels système `chown()`, `fchown()` et `lchown()`, déclarés dans `<unistd.h>` :

```
int chown(const char * nom_fichier, uid_t propr, gid_t grpe);
int fchown (int descripteur_fichier, uid_t propr, gid_t grpe);
int lchown (const char * nom_fichier, uid_t propr, gid_t grpe);
```

En Python, on dispose de `os.chown()`, `os.fchown()` et `os.lchown()`. Les appels système `chown()` et `lchown()` modifient l'appartenance d'un fichier dont le nom leur est fourni en premier argument. La différence concerne les liens symboliques ; `chown()` modifie l'appartenance du fichier visé par le lien alors que `lchown()` s'applique au lien lui-même – ce qui ne présente pas beaucoup d'intérêt. De son côté, `fchown()` agit sur le descripteur d'un fichier déjà ouvert. Si l'un des UID ou GID indiqués vaut -1, cette valeur n'est pas changée.

Lors de la modification du propriétaire, le bit Set-UID éventuel est effacé. Lors de la modification du groupe, le bit Set-GID est effacé si ce fichier possède également le bit d'exécution pour son groupe (sinon, c'est un fichier avec un verrouillage strict).

On reprend l'exemple d'un logiciel de communication avec un périphérique spécifique et d'une séquence `access()` suivie de `open()` comme nous l'avons décrit précédemment pour créer un fichier d'enregistrement. La modification du propriétaire de ce fichier nécessiterait en théorie que l'application soit Set-UID root. Toutefois, il existe d'autres possibilités, notamment on peut utiliser un exécutable Set-GID appartenant à un groupe autorisé à accéder aux ports de communication (`dialout` par exemple). La modification de l'appartenance du nouveau fichier est alors restreinte à celle de son groupe, ce qui peut être envisagé avec n'importe quel UID effectif. Il vaut mieux, pour des raisons de sécurité, employer dans ce cas `fchown()` directement sur le descripteur du fichier qu'on vient de créer plutôt que `chown()` sur son nom, car l'utilisateur pourrait à nouveau exploiter le délai entre la création du fichier et la modification de son appartenance pour l'effacer et le remplacer par un lien matériel sur un fichier système.

Taille du fichier

La taille d'un fichier est indiquée par le champ `st_size` de la structure `stat`. Nous reprenons le programme `exemple-stat.c` et nous le modifions pour obtenir `exemple-tailles.c` :

- On remplace l'affichage des modes `status->st_mode` par l'affichage de la taille `status->st_size`.
- On remplace l'appel `stat()` par `lstat()` pour ne pas suivre les liens symboliques, mais s'intéresser au lien lui-même.

Le programme ainsi obtenu nous permet de formuler plusieurs observations :

```
$ ./exemple-tailles /dev/sda1 /dev/ttyS0
/dev/sda1 : bloc 0
/dev/ttyS0 : caractere 0
$
```

Les fichiers spéciaux (en mode caractère ou bloc) ont une taille nulle. En réalité, il s'agit simplement d'indicateurs pour le noyau. Ils occupent une entrée de répertoire, mais pas d'autre place sur le disque. Nous reviendrons sur ces fichiers plus loin.

```
$ ./exemple-tailles /etc /usr /var
/etc : repertoire 12288
/usr : repertoire 4096
/var : repertoire 4096
$ mkdir vide
```

```
$ ./exemple-tailles vide
vide : repertoire 4096
$ rmdir vide/
$
```

Un répertoire occupe une taille (multiple de 4 096 sur ce système de fichiers) correspondant à son contenu, c'est-à-dire les noms des fichiers et les pointeurs vers les i-nœuds.

```
$ ls -l /etc/services
-rw-r--r-- 1 root root 644302 16 août 12:15 /etc/services
$ ln -sf /etc/services ./services
$ ./exemple-tailles /etc/services ./services
/etc/services : fichier 644302
./services : lien 13
$
```

Un fichier normal occupe la taille nécessaire pour stocker son contenu. Un lien symbolique n'emploie que la taille indispensable pour enregistrer le nom du fichier vers lequel il pointe (en l'occurrence « /etc/services » comporte 13 caractères).

```
$ cat /dev/null | ./exemple-tailles | cat
stdin : fifo 0
stdout : fifo 0
$
```

Les tubes et les FIFO sont des structures particulières n'ayant pas de taille attribuée (bien qu'elles aient toutefois une dimension maximale).

Remarquons bien que les informations fournies par le membre `st_size` de `stat` correspondent à la taille des données contenues dans un fichier, et pas forcément à son occupation sur le disque. En voici un exemple :

```
$ ./exemple-tailles /etc/services
/etc/services : fichier 644302
$ du -h /etc/services
632K  /etc/services
$ echo $(( 632*1024 ))
647168
$
```

L'utilitaire `du` calcule en effet la place occupée en prenant en compte la taille des blocs du système de fichiers et le nombre de blocs employés.

Lorsqu'on désire augmenter la dimension d'un fichier, on utilise simplement les fonctions d'écriture. Pour diminuer sa taille, le travail est plus compliqué : il faut procéder en réalisant une copie partielle du fichier original, qu'on renommera ensuite. Une autre possibilité consiste à utiliser des fonctions `truncate()` ou `ftruncate()`, déclarées dans `<unistd.h>` :

```
int truncate(const char * nom_fichier, off_t longueur);
int ftruncate (int descripteur_fichier, off_t longueur);
```

Dans le cas d'une réduction de taille, les données supplémentaires se trouvant en fin de fichier sont simplement éliminées. Si la longueur demandée est plus grande que la taille actuelle du fichier, la zone intermédiaire est remplie de zéros.

```
exemple-truncate.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    long longueur;
    if ((argc != 3) || (sscanf(argv[2], "%ld", & longueur) != 1)) {
        fprintf(stderr, "Syntaxe : %s fichier longueur\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (truncate(argv[1], longueur) < 0)
        perror(argv[1]);
    return EXIT_SUCCESS;
}
```

Nous utilisons le programme `exemple-getchar` du chapitre 20 pour examiner le contenu d'un fichier que nous fabriquons et dont nous modifions la taille.

```
$ printf "abcdefghijklmnopqrstuvwxy" > essai.truncate
$ ls -l essai.truncate
-rw-rw-r-- 1 cpb cpb 26 21 août 14:30 essai.truncate
$ ./exemple-truncate essai.truncate 10
$ ls -l essai.truncate
-rw-rw-r-- 1 cpb cpb 10 21 août 14:30 essai.truncate
$ ../chapitre-20/exemple-getchar < essai.truncate
00000000 61 62 63 64 65 66 67 68-69 6A                                abcdefghi
$ ./exemple-truncate essai.truncate 20
$ ../chapitre-20/exemple-getchar < essai.truncate
00000000 61 62 63 64 65 66 67 68-69 6A 00 00 00 00 00 00 00 00 abcdefghij
00000010 00 00 00 00
```

```
$ rm essai.truncate
$
```

`ftruncate()` peut être très utile pour fixer précisément la longueur d'un fichier qu'on désire projeter en mémoire. Cela s'applique principalement aux applications qui veulent projeter simultanément plusieurs zones du même fichier, en ajoutant ainsi des blocs complets en fin de fichier.

Horodatages d'un fichier

Nous avons observé dans la structure `stat` que trois dates sont associées à un fichier :

- `st_atime`, la date du dernier accès au contenu du fichier, en lecture ou en écriture ;
- `st_mtime`, la date de dernière modification du contenu du fichier avec une primitive d'écriture ;
- `st_ctime`, la date de dernière modification de la structure `stat` associée au fichier, ce qui inclut le changement de propriétaire, de mode...

Le type utilisé pour enregistrer ces dates est `time_t`, qui s'exprime en secondes écoulées depuis le 1^{er} janvier 1970. Nous en avons parlé dans le chapitre 8.

Les horodatages sont mis à jour automatiquement par le noyau, mais on peut avoir besoin pour de nombreuses raisons de modifier les dates `st_atime` ou `st_mtime`. Le champ `st_ctime` ne peut être mis à jour que par le noyau. Les appels `utime()` et `utimes()` sont déclarés respectivement dans `<utime.h>` et `<sys/time.h>`. Ils servent tous deux à mettre à jour les dates `st_atime` et `st_mtime`, mais `utimes()` permet d'accéder à une précision de l'ordre de la microseconde, alors que `utime()` est limité à la seconde près par le type `time_t`.

```
int utime(const char * nom_fichier, struct utimbuf * dates);
int utimes(const char * nom_fichier, struct timeval dates [2]);
```

La structure `utimbuf` contient les champs suivants :

Nom	Type	Signification
<code>actime</code>	<code>time_t</code>	Date du dernier accès au contenu du fichier.
<code>modtime</code>	<code>time_t</code>	Date de dernière modification du contenu du fichier.

Pour l'appel `utimes()`, il faut passer un tableau de deux structures, la première correspondant au dernier accès, et la seconde à la dernière modification. Les membres de la structure `timeval` – que nous avons déjà rencontrée – sont :

Nom	Type	Signification
tv_sec	long	Nombre de secondes écoulées depuis le 1 ^{er} janvier 1970.
tv_usec	long	Nombre de microsecondes (0 à 999 999).

Linux ne permet pas de mémoriser une telle précision, et `utimes()` est ainsi implémenté en remplissant simplement les champs d'une structure `timebuf` avant d'appeler `utime()`.

Si le pointeur passé en second argument de `utime()` ou de `utimes()` est `NULL`, les dates sont mises à jour au moment de l'appel.

Liens physiques

Nous avons déjà évoqué à plusieurs reprises les liens physiques ou liens matériels (*hard links*). Cette notion de lien est un peu trompeuse, car elle suggère une entité inamovible, originelle, à laquelle on rattache des avatars de moindre importance. Si cette image peut s'appliquer aux liens symboliques que nous verrons dans le prochain paragraphe, elle est totalement erronée dans le cas des liens physiques.

Lorsqu'on crée un lien physique sur un fichier, on ajoute simplement un nouveau nom dans le système de fichiers, qui pointe vers le même i-nœud que le nom original. Dès lors, il n'y a plus aucun moyen de distinguer le nom qui était le premier et celui qui a été créé ensuite. Les deux noms sont traités avec égalité par le système. En fait, il faut considérer que tout nom présent dans le système de fichiers est un lien physique vers le contenu même du fichier.

Il n'est pas possible de créer dans un répertoire un nom lié à un fichier se trouvant sur un autre système de fichiers. De plus, certains systèmes Unix, dont Linux, n'autorisent pas la création de liens physiques sur un répertoire. On pourrait en effet concevoir une boucle dans le système de fichiers (un répertoire contenant un sous-répertoire correspondant à son père), or le noyau n'est pas prêt à détecter de tels conflits (contrairement à ce qui se passe avec les liens symboliques).

Pour créer un lien matériel, il existe un appel système nommé `link()`, décrit par SUSv4, et déclaré dans `<unistd.h>` :

```
int link (const char * nom_original,  
          const char * nouveau_nom);
```

Il est possible de l'invoquer avec `os.link()` en Python. Cet appel système établit donc un nouveau lien sur le fichier transmis en second argument, créant ainsi un nouveau nom dans le système de fichiers. Le champ `st_nlink` de la structure `stat` correspondant à l'i-nœud est incrémenté. Cette valeur est également visible dans la

seconde colonne affichée lors d'un `ls -l`. Notons que le nouveau nom ne sera pas écrasé s'il existe déjà. Si on veut forcer la création, il faut l'effacer auparavant.

Nous comprenons à présent pourquoi l'appel système d'effacement d'un nom de fichier se nomme `unlink()`, puisqu'il sert simplement à supprimer un lien physique sur le contenu du fichier et à décrémenter ainsi le champ `st_nlink`. Lorsque ce dernier arrive à zéro, l'espace occupé sur le disque est réellement libéré.

Il existe une application `/bin/ln` qui sert de frontal à l'appel système `link()`. Lorsque nous ne précisons aucune option, cet utilitaire crée un lien physique. En voici un exemple d'utilisation :

```
$ ls -l exemple-tailles.c
-rw-r--r-- 1 cpb cpb 1195 21 août 14:19 exemple-tailles.c
$ ln exemple-tailles.c deuxieme
$ ls -l exemple-tailles.c deuxieme
-rw-r--r-- 2 cpb cpb 1195 21 août 14:19 deuxieme
-rw-r--r-- 2 cpb cpb 1195 21 août 14:19 exemple-tailles.c
$ rm deuxieme
$ ls -l exemple-tailles.c
-rw-r--r-- 1 cpb cpb 1195 21 août 14:19 exemple-tailles.c
$
```

Nous remarquons que le champ `st_nlink` de la structure `stat`, affiché en seconde colonne du résultat de `ls -l`, passe bien à 2, puis revient à 1 après la destruction de l'un des noms. Vérifions à présent que les liens matériels sont interdits sur les répertoires et entre plusieurs systèmes de fichiers différents :

```
$ ln /tmp ./essai
ln: « /tmp »: lien direct non permis pour un répertoire
$ ln /proc/cpuinfo cpu
ln: création d'un lien direct de « cpu » vers « /proc/cpuinfo »: Lien
croisé de périphérique invalide
$
```

Les liens physiques sont souvent utilisés pour donner plusieurs noms différents à la même application, de manière transparente vis-à-vis de l'utilisateur. Sur les systèmes embarqués par exemple, on utilise très fréquemment l'application `busybox` pour émuler les principales commandes Unix. Il s'agit d'un seul exécutable, compilé et optimisé, qui est capable de se comporter au choix comme `ls`, `cp`, `mv`, `vi`, `find`, `sh`, `telnet`, etc. (plus de 200 commandes !).

Dans les installations classiques, on copie l'exécutable sur le système de fichiers de la cible, et on crée autant de liens physiques qu'il y a de noms d'utilitaires à émuler. Lorsque le programme démarre, il examine son argument `argv[0]` de ligne de commande (le nom sous lequel il a été invoqué) et ajuste son comportement en conséquence.

Liens symboliques

Contrairement à leurs homonymes physiques, les liens symboliques (*soft links* ou *symbolic links*) sont soumis à un nombre moins important de contraintes. Ils sont également implémentés conceptuellement à un niveau plus élevé dans l'organisation du système de fichiers.

Un lien symbolique n'est rien de plus qu'un petit fichier de texte comprenant le chemin d'accès et le nom du fichier vers lequel il pointe. Le lien est également marqué par un type spécial – qu'on détermine grâce à la macro `S_ISLNK()` appliquée au champ `st_mode` de la structure `stat` – et qui permet au noyau de le reconnaître. Avec certains appels système, le noyau agira alors sur le contenu du lien, en opérant sur le fichier visé, alors que d'autres primitives fonctionneront directement sur le lien symbolique lui-même.

Un lien symbolique est créé grâce à l'appel `symlink()`, déclaré dans `<unistd.h>` :

```
| int symlink (const char * nom_original, const char * nouveau_nom);
```

En Python, on emploiera `os.symlink()`. On peut utiliser aussi l'utilitaire `/bin/ln` avec l'option `-s` pour créer un lien symbolique.

Attention, il est tout à fait possible de créer un lien symbolique pointant vers un fichier inexistant. Le système indiquera une erreur lors de la tentative d'ouverture. De même, si le fichier visé est supprimé, les liens symboliques qui pointaient vers lui ne sont pas concernés.

En conséquence, on peut alors créer un lien symbolique entre différents systèmes de fichiers et créer des liens sur des répertoires. On peut aussi concevoir des boucles dans les liens, ce que le système détectera lors des tentatives d'accès.

Voici la création d'un lien symbolique normal et sa suppression :

```
$ ls -l Makefile
-rw-r--r-- 1 cpb cpb 233 21 août 14:29 Makefile
$ ln -s Makefile Makefile.2
$ ls -l Makefile*
-rw-r--r-- 1 cpb cpb 233 21 août 14:29 Makefile
```

```
lrwxrwxrwx 1 cpb cpb 8 21 août 14:54 Makefile.2 -> Makefile
$ rm Makefile.2
$ ls -l Makefile*
-rw-r--r-- 1 cpb cpb 233 21 août 14:29 Makefile
$
```

La taille d'un lien symbolique est limitée à la longueur du chemin qu'il contient (toujours, l'allocation de l'espace sur le disque est assurée par blocs beaucoup plus gros). Voici un exemple de lien symbolique pointant vers un fichier inexistant :

```
$ ln -s je_n_existe_pas ici
$ ls -l ici
lrwxrwxrwx 1 cpb cpb 15 21 août 14:55 ici -> je_n_existe_pas
$ cat ici
cat: ici: Aucun fichier ou dossier de ce type
$ rm ici
$
```

Un lien symbolique peut contenir un chemin absolu depuis la racine du système de fichiers ou un chemin relatif à base de `./` ou `../`. Cette dernière solution est souvent préférable, surtout si le répertoire est susceptible d'être exporté par le système NFS pour être visible sur d'autres systèmes n'ayant pas la même arborescence. Notons que certains systèmes de fichiers (par exemple `msdos` ou `vfat`) ne permettent pas la création de liens symboliques.

Les permissions d'accès à un lien symbolique ne sont jamais prises en compte, aussi sont-elles fixées automatiquement à `rw-rw-rw-`. Le nom du propriétaire d'un lien symbolique n'a que rarement de l'intérêt. Le seul cas où cette information est utile est celui où le lien réside dans un répertoire public ayant son bit *Sticky* à 1, ce qui signifie que seul root ou le propriétaire d'un fichier peuvent l'effacer ou le modifier.

Les liens symboliques sont une méthode très commode pour configurer un système, particulièrement dans une arborescence de fichiers source. On utilise par exemple souvent des liens symboliques dans des applications portées sur plusieurs plateformes, pour faire pointer un seul fichier `Makefile` au choix vers `Makefile.linux`, `Makefile.aix`, `Makefile.solaris`, `Makefile.hpux`, etc. De même, chaque équipe de développement a souvent une petite bibliothèque de fichiers source réutilisés dans plusieurs projets. Ces fichiers peuvent être conservés en un seul exemplaire, permettant ainsi une mise à jour automatique en cas de correction à apporter, tout en ayant des liens symboliques dans les arborescences de chaque projet qui les utilise.

Le lien symbolique présente l'avantage d'être une indirection explicite, facilement visible, au contraire des liens physiques. Cela permet de changer l'emplacement réel

d'un fichier, tout en le laissant accessible à partir d'un autre répertoire où il était placé historiquement. C'est le cas du répertoire `/usr/tmp` qu'on conserve pour des raisons historiques, mais qui est généralement lié symboliquement au répertoire `/var/tmp`. La partition `/usr` doit en effet pouvoir être montée en lecture seule (voire depuis un serveur NFS). Alors que `/var` est souvent une partition locale, comme `/tmp`, sur laquelle on peut éventuellement recréer le système de fichiers à chaque redémarrage de la machine. Les liens symboliques sont alors des outils précieux pour les administrateurs qui gèrent un parc de plusieurs machines hétérogènes utilisant la même arborescence `/usr` depuis un serveur NFS, puisqu'ils permettent d'employer un lien symbolique dans la partition commune vers un fichier dépendant de chaque machine.

Lorsqu'on utilise `open()` sur un lien symbolique, cet appel système tente d'accéder au fichier visé. Pour connaître le véritable contenu du lien (le chemin vers lequel il pointe), il faut employer un appel système différent, nommé `readlink()`, déclaré dans `<unistd.h>` :

```
int readlink (const char * nom_lien, char * buffer,
              size_t taille);
```

Cet appel système recopie dans le buffer passé en deuxième argument le contenu du lien dont le nom est passé en première position. Il limite la longueur de la copie à la taille fournie en troisième argument, mais n'ajoute pas de caractère nul.

La valeur renvoyée vaut -1 en cas d'échec. Si `readlink()` réussit, il renvoie le nombre de caractères copiés. Si ce nombre est égal à la taille maximale, on recommencera donc l'appel avec un buffer plus grand, comme dans l'exemple suivant :

```
exemple-readlink.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void lecture_contenu (const char * nom)
{
    char * buffer = NULL;
    char * nouveau = NULL;
    int taille = 0;
    int nb_copies;

    while (1) {
        taille += 16;
        if ((nouveau = realloc(buffer, taille)) == NULL) {
            perror(nom);
            break;
        }
    }
}
```

```

    buffer = nouveau;
    if ((nb_copies = readlink(nom, buffer, taille - 1)) == -1) {
        perror(nom);
        break;
    }
    if (nb_copies < taille - 1) {
        buffer[nb_copies] = '\0';
        fprintf(stdout, "%s : %s\n", nom, buffer);
        break;
    }
}
free(buffer);
}

int main (int argc, char * argv[])
{
    int i;
    for (i = 1; i < argc; i++)
        lecture_contenu(argv[i]);
    return EXIT_SUCCESS;
}

```

Dans l'exécution suivante, nous créons un lien dont le contenu est largement plus long que 16 caractères pour vérifier que notre routine fonctionne :

```

$ ln -s /usr/lib/gcc/i686-redhat-linux/4.4.4/include/gcj/libgcj-
config.h config.h
$ ./exemple-readlink config.h
config.h : /usr/lib/gcc/i686-redhat-linux/4.4.4/include/gcj/libgcj-
config.h
$ rm config.h
$

```

En Python, l'utilisation de la fonction `os.readlink()` est plus simple car elle renvoie directement une chaîne de caractères correspondant à la destination du lien symbolique.

```

exemple-readlink.py :
#!/usr/bin/python
from __future__ import print_function
import os
import sys

sys.argv.pop(0)
for f in sys.argv:
    try:
        print(f, os.readlink(f))
    
```

Il n'est pas toujours évident de savoir si un appel système s'applique au lien symbolique lui-même ou à son contenu. En règle générale, le comportement des appels système est dicté par le bon sens. En voici toutefois un récapitulatif rapide pour les principales primitives de traitement des fichiers :

Appel système	concerne le lien lui-même	concerne le fichier visé
<code>access()</code>		X
<code>chdir()</code>		X
<code>chmod()</code>		X
<code>chown()</code>	jusqu'à Linux 2.0	depuis Linux 2.2
<code>execve()</code>		X
<code>lchown()</code>	X	
<code>lstat()</code>	X	
<code>open()</code>		X
<code>readlink()</code>	X	
<code>rename()</code>	X	
<code>stat()</code>		X
<code>truncate()</code>		X
<code>unlink()</code>	X	
<code>utime()</code>	X	
<code>utimes()</code>	X	

Nœud générique du système de fichiers

Il existe, nous l'avons constaté dans la structure `stat`, plusieurs types de nœuds, qu'on peut rencontrer dans un système de fichiers. Tout d'abord, on trouve les fichiers réguliers qu'on crée avec l'appel système `open()` ou `creat()` ; bien entendu, il existe également les répertoires, créés avec `mkdir()`, ainsi que les liens physiques et symboliques issus de `link()` et `symlink()`. Les sockets ne se trouvent généralement pas dans le système de fichiers¹, mais leurs descripteurs – fournis par l'appel système `socket()` que nous avons examiné dans le chapitre 27 – sont manipulés comme les descripteurs de fichiers et peuvent être transmis en argument de `fstat()`. On peut encore trouver trois types de nœuds : les files FIFO, les fichiers spéciaux de périphé-

1. Certaines sockets – celles dans le domaine `AF_UNIX` – peuvent avoir un nom dans le système de fichiers, par exemple `/dev/log`. Leur comportement s'apparente assez à celui d'une FIFO.

rique de type caractère, et ceux de type bloc. Pour créer ce genre de nœuds, on utilise l'appel système `mknod()`, déclaré dans `<sys/stat.h>` :

```
int mknod (const char * nom, mode_t mode, dev_t periph);
```

En Python, on dispose de `os.mknod()`. Le premier argument de cet appel système indique le nom du nœud à créer, et le mode précisé à la suite doit être l'une des constantes suivantes :

Nom	Signification
<code>S_IFREG</code>	Création d'un fichier régulier vide, équivalent d'un <code>open()</code> suivi d'un <code>close()</code> . Le troisième argument de <code>mknod()</code> est ignoré. Ce mode de fonctionnement ne nous intéressera pas ici.
<code>S_IFIFO</code>	Création d'une file FIFO. Ce type de fichier est habituellement créé à l'aide de la fonction de bibliothèque <code>mkfifo()</code> , que nous avons vue dans le chapitre 23.
<code>S_IFBLK</code>	Création d'un fichier spécial de périphérique de type bloc.
<code>S_IFCHR</code>	Création d'un fichier spécial de périphérique de type caractère.

Dans le cas d'un fichier spécial de périphérique (bloc ou caractère), le troisième argument indique au noyau le type du pilote de périphérique désiré. Cette valeur est composée de deux nombres, qu'on nomme numéro majeur et numéro mineur du périphérique. Le numéro majeur permet au noyau de déterminer quel pilote de périphérique est concerné lorsqu'on tente une ouverture, une lecture ou une écriture sur le nœud dont il est question. Le numéro mineur est réservé au pilote lui-même, pour pouvoir différencier plusieurs dispositifs matériels par exemple.

Lorsqu'un pilote est chargé en mémoire, il indique au noyau le type de périphérique pour lequel il est compétent, en lui transmettant une structure `file_operations` décrite dans le fichier d'en-tête `<linux/fs.h>`. Ce mécanisme est interne au noyau, mais il est intéressant de le comprendre pour bien saisir le rôle des fichiers spéciaux. À l'instar des méthodes définies pour les classes en programmation orientée objet, la structure `file_operations` contient des pointeurs sur les fonctions que le pilote est capable de fournir pour le périphérique :

Nom du champ	Rôle
<code>lseek()</code>	Fonction appelée pour déplacer la position de lecture ou d'écriture sur le périphérique.
<code>read()</code>	Fonction de lecture depuis le périphérique.
<code>write()</code>	Fonction d'écriture sur le périphérique.
<code>readdir()</code>	Fonction de lecture du contenu d'un répertoire servant à homogénéiser l'implémentation des systèmes de fichiers, mais non utilisée sur les périphériques.

Nom du champ	Rôle
<code>poll()</code>	Fonction permettant de surveiller la disponibilité des données en lecture ou en écriture, comme nous l'avons vu dans le chapitre 25.
<code>ioctl()</code>	Point d'entrée permettant d'assurer des opérations particulières sur un périphérique autres que les lectures ou écritures, par exemple éjection d'un CD, programmation de la parité d'une interface série, etc.
<code>mmap()</code>	Fonction demandant la projection du contenu du périphérique en mémoire.
<code>open()</code>	Fonction d'ouverture et d'initialisation du périphérique.
<code>flush()</code>	Demande de vidage des buffers associés à un périphérique.
<code>release()</code>	Fonction de fermeture et libération d'un périphérique, équivalent de <code>close()</code> .
<code>fsync()</code>	Fonction de synchronisation du contenu du périphérique et de ses buffers associés.
<code>fasync()</code>	Demande de fonctionnement asynchrone du périphérique.
<code>check_media_change()</code>	Fonction vérifiant si le support amovible contenu dans le périphérique a été modifié (par exemple un CD).
<code>revalidate()</code>	Fonction de gestion du buffer cache.
<code>lock()</code>	Fonction de verrouillage du périphérique.

Lorsqu'on demande l'ouverture d'un fichier spécial de périphérique, par exemple un port série, le noyau vérifie le numéro majeur et appelle la fonction `open()` du pilote correspondant, en lui transmettant diverses informations, dont le numéro mineur désiré. Bien sûr, certains pilotes de périphériques n'implémentent pas toutes les fonctions indiquées ci-dessus (on ne peut pas déplacer avec `lseek()` la position de lecture sur un port de communication série), aussi existe-t-il des routines par défaut, permettant de renvoyer une erreur par exemple.

Les numéros majeur et mineur d'un fichier spécial sont donc essentiels pour la compréhension entre le noyau et le pilote de périphérique, contrairement au nom du fichier spécial qui n'a aucune importance. Les numéros réservés sont décrits dans le fichier `Documentation/devices.txt` accompagnant les sources du noyau. Par exemple, les ports de communication série sont gérés avec le numéro majeur 4 et les numéros mineurs 64, 65, 66 pour les ports COM1, COM2, COM3 (nommés généralement `ttyS0`, `ttyS1`, `ttyS2` sous Linux). On peut le constater en examinant les cinquième et sixième colonnes de la commande `ls -l` sur les fichiers spéciaux correspondants :

```
$ cd /dev
$ ls -l /dev/ttyS*
crw-rw---- 1 root dialout 4, 64 20 août 14:34 /dev/ttyS0
crw-rw---- 1 root dialout 4, 65 20 août 14:34 /dev/ttyS1
crw-rw---- 1 root dialout 4, 66 20 août 14:34 /dev/ttyS2
crw-rw---- 1 root dialout 4, 67 20 août 14:34 /dev/ttyS3
```

Le caractère 'c' en tête de la première colonne indique qu'il s'agit d'un périphérique spécial de type caractère. D'autres peuvent être de type bloc :

```
$ ls -l /dev/sda*
brw-rw---- 1 root disk 8, 0 20 août 12:34 /dev/sda
brw-rw---- 1 root disk 8, 1 20 août 12:34 /dev/sda1
brw-rw---- 1 root disk 8, 2 20 août 12:34 /dev/sda2
[...]
brw-rw---- 1 root disk 8, 7 20 août 12:34 /dev/sda7
$
```

La différence entre périphériques caractère et bloc réside dans la manière d'accéder aux données. Dans un cas, elles arrivent octet par octet, dans l'autre des blocs complets sont affectés pour la lecture ou l'écriture. Un corollaire de cette distinction est qu'un périphérique de type bloc peut généralement contenir un système de fichiers, ce qui n'est pas possible avec un périphérique en mode caractère. L'essentiel des périphériques de type bloc est constitué de disques durs, de CD-Rom, de clé USB et de cartes Flash. Lorsqu'on développe un driver personnalisé pour assurer l'interface avec un périphérique qui est aussi personnalisé (par exemple un instrument de mesure), on utilise le plus souvent un pilote de type caractère.

Pour créer un nouveau nœud du système de fichiers qui représente un fichier spécial de périphérique, on utilise généralement l'utilitaire `/bin/mknod`, qui prend en arguments la lettre `b` ou `c` (suivant le type de périphérique), le numéro majeur et le numéro mineur. Cette application sert ainsi de frontal à l'appel système `mknod()`, dont le troisième argument regroupe les deux numéros mineurs sous forme d'une valeur `dev_t`, composée ainsi :

```
dev_t periph;
periph = (majeur << 8) | mineur;
```

Le numéro mineur est donc limité à l'espace allant de 0 à 255.

La création directe d'un fichier spécial, en employant l'appel système `mknod()`, est assez rare, puisqu'on préfère en général créer ces nœuds en utilisant `/bin/mknod`, éventuellement dans un script shell qui encadre le chargement du module pilote du périphérique correspondant.

L'appel système `mknod()` donnant un accès direct aux périphériques matériels reliés à l'ordinateur, il est naturellement réservé à l'administrateur. De même, les fichiers spéciaux de périphériques ne sont interprétés en tant que tels par le noyau que si la partition a été montée avec l'option adéquate. C'est le cas par défaut pour les systèmes de fichiers

montés automatiquement au démarrage, mais cela est désactivé pour les supports susceptibles d'être montés par n'importe quel utilisateur, comme les clés USB ou les CD-Rom.

Masque de création de fichier

Lorsqu'un processus crée un fichier, quel que soit son type, les permissions d'accès sont filtrées par un masque particulier, qui retire des autorisations. Ce masque peut être modifié avec l'appel système `umask()`, déclaré dans `<sys/stat.h>` :

```
| int umask (int masque);
```

Cet appel système (ou `os.umask()` en Python) permet de configurer le nouveau masque et renvoie sa valeur précédente. Lorsqu'un processus dispose par exemple d'un masque 0022 en octal, ce qui est courant, même s'il crée un fichier avec les autorisations 0777 (lecture, écriture, exécution pour tout le monde), les bits d'écriture (correspondant à 2) seront supprimés pour le groupe et le reste des utilisateurs. Le fichier ainsi créé aura l'autorisation 0755, ce qui est plus raisonnable. En voici un exemple très simple :

```
exemple-umask.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

int main (void)
{
    int fd;
    int masque;

    masque = umask(0);
    fprintf(stdout, "Ancien masque = %o, nouveau = 0 \n", masque);
    fprintf(stdout, "Tentative de creation deessai.umask \n");
    fd = open("essai.umask", O_RDWR | O_CREAT | O_EXCL, 0777);
    if (fd < 0)
        perror("open");
    else
        close(fd);
    system("ls -l essai.umask");
    unlink("essai.umask");
```

```
umask(masque);
fprintf(stdout, "Remise masque = %o \n", masque);
fprintf(stdout, "Tentative de creation de essai.umask \n");
fd = open("essai.umask", O_RDWR | O_CREAT | O_EXCL, 0777);
if (fd < 0)
    perror("open");
else
    close(fd);
system("ls -l essai.umask");
unlink("essai.umask");

return EXIT_SUCCESS;
}
```

L'exécution montre bien la différence des modes lors de la création d'un fichier :

```
$ ./exemple-umask
Ancien masque = 2, nouveau = 0
Tentative de creation de essai.umask
-rwxrwxrwx 1 cpb cpb 0 21 août 10:16 essai.umask
Remise masque = 2
Tentative de creation de essai.umask
-rwxrwxr-x 1 cpb cpb 0 21 août 10:16 essai.umask
$
```

Le masque étant hérité au cours d'un `fork()` et conservé au cours d'un `exec()`, il s'agit avant tout d'un dispositif de sécurité qu'on peut mettre en place dans le script de configuration du shell (qui dispose d'une commande de configuration `umask` interne) pour s'assurer de la confidentialité des fichiers créés ultérieurement.

Surveillance du système de fichiers

Pour terminer ce chapitre, nous allons étudier un mécanisme implémenté assez récemment dans Linux, puisqu'il n'a été introduit que dans le noyau 2.6.14 (en 2005) : *inotify*. Ce système a été développé pour remplacer *dnotify* qui consommait trop de descripteurs de fichiers pour fonctionner efficacement.

Le principe d'*inotify* consiste à demander au noyau de nous fournir un descripteur de fichier spécial, sur lequel il enverra des notifications chaque fois qu'il y aura une modification sur les éléments du système de fichiers que nous voulons surveiller. Cela est très pratique par exemple pour les utilitaires qui doivent présenter le contenu d'un répertoire (navigateur de fichiers, outils de transferts par réseau, etc.) et qui veu-

lent faire apparaître dynamiquement les modifications qu'on y apporte (création ou suppression de fichier ou de sous-répertoire, modification du contenu d'un fichier...). Le descripteur est fourni par l'appel système `inotify_init()` ou `inotify_init1()`, déclarés dans `<sys/inotify.h>` :

```
int inotify_init (void);
int inotify_init1 (int attributs);
```

Les attributs acceptés par `inotify_init1()` sont une combinaison de :

Attribut	Signification
<code>IN_CLOEXEC</code>	Activer l'attribut <code>CLOSE-ON-EXEC</code> sur le descripteur fourni.
<code>IN_NONBLOCK</code>	Activer l'attribut <code>O_NONBLOCK</code> sur le descripteur renvoyé.

L'appel `inotify_init()` est équivalent à `inotify_init1(0)`.

Le descripteur renvoyé (-1 en cas d'erreur) pourra être refermé ultérieurement avec `close()`. On peut ajouter de nouvelles surveillances à ce descripteur avec `inotify_add_watch()` ou en supprimer avec `inotify_rm_watch()`.

```
int inotify_add_watch (int fd, const char *nom_fichier, uint32_t masque);
int inotify_rm_watch (int fd, uint32_t watch);
```

Le premier argument de `inotify_add_watch()` est le descripteur global obtenu précédemment. Le second est le nom du fichier ou du répertoire à surveiller, et le troisième est le masque des événements dont on veut être notifié. Les événements suivants sont pris en charge :

Événement	Signification
<code>IN_ACCESS</code>	Accès au fichier surveillé.
<code>IN_ATTRIB</code>	Modification des attributs (permissions, etc.) du fichier.
<code>IN_CLOSE_WRITE</code>	Fermeture d'un fichier ouvert précédemment en écriture.
<code>IN_CLOSE_NOWRITE</code>	Fermeture d'un fichier ouvert précédemment en lecture seule.
<code>IN_CREATE</code>	Création d'un fichier ou d'un sous-répertoire dans le répertoire surveillé.
<code>IN_DELETE</code>	Suppression d'un fichier ou d'un sous-répertoire dans le répertoire surveillé.
<code>IN_DELETE_SELF</code>	Suppression du fichier ou du répertoire surveillé.
<code>IN_MODIFY</code>	Modification du fichier surveillé.
<code>IN_MOVE_SELF</code>	Déplacement ou renommage du fichier ou du répertoire surveillé.
<code>IN_MOVED_FROM</code>	Déplacement d'un fichier ou d'un sous-répertoire depuis le répertoire surveillé.

Événement	Signification
IN_MOVED_TO	Déplacement d'un fichier ou d'un sous-répertoire dans le répertoire surveillé.
IN_OPEN	Ouverture du fichier surveillé.

Il existe un événement `IN_ALL_EVENTS` qui peut servir à regrouper tous les événements précédents, ainsi que `IN_MOVE` et `IN_CLOSE` qui sont définis respectivement comme `(IN_MOVED_FROM | IN_MOVED_TO)` et `(IN_CLOSE_WRITE | IN_CLOSE_NOWRITE)`.

Le descripteur renvoyé est un descripteur de surveillance, qui correspond donc à un type d'événement particulier.

Une fois la liste des événements attendus enregistrée dans le descripteur global, on se met en attente sur ce dernier avec `read()`. L'appel `read()` est bloquant – sauf option spécifique à l'initialisation, comme nous l'avons indiqué plus haut – et renvoie des structures `inotify_event` comportant les champs suivants :

Champ	Type	Signification
wd	int	Descripteur de surveillance ayant déclenché la notification.
mask	uint32_t	Événements produits.
cookie	uint32_t	Identifiant d'événement servant à corréler les <code>IN_MOVED_FROM</code> et <code>IN_MOVED_TO</code> .
len	uint32_t	Longueur du champ <code>name</code> .
name	char[]	Nom du fichier concerné (événement sur répertoire).

Il faut que le buffer transmis à `read()` soit capable de contenir `sizeof(inotify_event)+len` octets, sinon l'appel échouera avec l'erreur `EINVAL`.

Dans le petit exemple ci-après, on surveille tous les événements qui se produisent sur le répertoire passé en argument :

```
exemple-inotify.c :
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/inotify.h>

void affiche_evenement(struct inotify_event * event)
{
    if (event->len > 0)
        fprintf(stdout, "%s : ", event->name);
    if (event->mask & IN_ACCESS)
        fprintf(stdout, "acces, ");
```

```
    if (event->mask & IN_ATTRIB)
        fprintf(stdout, "modification attributs, ");
    if (event->mask & IN_CLOSE_WRITE)
        fprintf(stdout, "close (read/write), ");
    if (event->mask & IN_CLOSE_NOWRITE)
        fprintf(stdout, "close (read only), ");
    if (event->mask & IN_CREATE)
        fprintf(stdout, "creation, ");
    if (event->mask & IN_DELETE)
        fprintf(stdout, "effacement, ");
    if (event->mask & IN_DELETE_SELF)
        fprintf(stdout, "destruction, ");
    if (event->mask & IN_MODIFY)
        fprintf(stdout, "modification, ");
    if (event->mask & IN_MOVE_SELF)
        fprintf(stdout, "deplacement, ");
    if (event->mask & IN_MOVED_FROM)
        fprintf(stdout, "depart, ");
    if (event->mask & IN_MOVED_TO)
        fprintf(stdout, "arrivee, ");
    if (event->mask & IN_OPEN)
        fprintf(stdout, "ouverture, ");
    fprintf(stdout, "\n");
}

int main(int argc, char * argv[])
{
    int fd;
    struct inotify_event * event;
    event = malloc(sizeof(struct inotify_event) + PATH_MAX);
    if (argc != 2) {
        fprintf(stderr, "usage: %s repertoire\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fd = inotify_init()) < 0) {
        perror("inotify_init");
        exit(EXIT_FAILURE);
    }
    if (inotify_add_watch(fd, argv[1], IN_ALL_EVENTS) < 0) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    while (read(fd, event, sizeof(struct inotify_event) + PATH_MAX) > 0) {
        affiche_evenement(event);
    }
    return EXIT_SUCCESS;
}
```

Voici un exemple d'exécution, où l'on surveille le répertoire courant, en y faisant des modifications depuis un autre terminal :

```
$ ./exemple-inotify .
passwd : creation,
passwd : modification,
ouverture,
passwd : effacement,
afac : creation,
afac : ouverture,
afac : modification attributs,
afac : close (read/write),
ouverture,
afac : effacement,
(Contrôle-C)
$
```

```
$ cp /etc/passwd .
$ rm passwd
$ touch afac
$ rm afac
```

Le lecteur est naturellement encouragé à expérimenter lui-même les possibilités de ce mécanisme très intéressant.

Conclusion

Ce chapitre nous a permis de faire le point sur les fichiers en tant qu'entités physiquement présentes sur les disques. De par les différences existant entre les divers types de systèmes de fichiers, nous sommes resté à un niveau de description assez élevé, correspondant à celui d'un développement applicatif.

Pour des informations sur l'implémentation interne du système de fichiers, je conseillerais l'excellent [Love 2003] *Linux Kernel Development* qui contient également des renseignements sur la plupart des autres sous-ensembles du noyau.

Accès aux informations du système

Les informations concernant l'état du système, les groupes et les utilisateurs inscrits, les systèmes de fichiers montés ou les derniers événements survenus sont principalement utilisés dans des applications de configuration (ajout ou suppression d'utilisateurs par exemple) et dans des utilitaires de surveillance du système à destination de l'administrateur. Dans la plupart des cas, un petit logiciel sert à encadrer l'appel système ou la fonction de bibliothèque correspondante, et les fonctionnalités de haut niveau sont assurées par un ou plusieurs scripts shell.

Nous allons examiner les fonctions offertes par le noyau et la bibliothèque C pour consulter ou configurer toutes ces données générales sur le paramétrage du système. Comme la plupart de ces données sont conservées dans des fichiers système à peu près similaires, nous verrons que les fonctions présentées dans ce chapitre suivent une inspiration commune.

Groupes et utilisateurs

Les informations concernant les groupes et les utilisateurs inscrits sur le système sont assez largement employées. Bien entendu ceci concerne les utilitaires permettant de gérer la liste des utilisateurs, mais également les applications de communication, la rédaction de courrier électronique, les écrans de connexion graphique au système X-Window, ou tout simplement l'affichage en clair des noms du propriétaire et du groupe d'un fichier.

Fichier des groupes

Les groupes d'utilisateurs sont enregistrés sous Linux dans le fichier `/etc/group`. Ce fichier contient une ligne pour chaque groupe, avec les champs suivants séparés par des deux-points :

- nom du groupe ;
- mot de passe ; ce champ n'est plus utilisé pour les groupes, on le laisse généralement vide ou on le remplit d'un caractère `x` marquant son invalidité ;
- GID du groupe, sous forme décimale ;
- liste des noms des utilisateurs appartenant au groupe, séparés par des virgules.

Voici un extrait d'un fichier `/etc/group` :

```
$ cat /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
...
nobody:x:99:
users:x:500:cpb
...
$
```

Un même nom pouvant être indiqué en troisième argument dans plusieurs lignes du fichier `/etc/group`, cela permet la configuration des groupes supplémentaires de l'utilisateur.

La consultation directe du fichier `/etc/group` est bien entendu déconseillée pour garantir une certaine portabilité du programme vers des systèmes employant d'autres méthodes. La bibliothèque C offre ainsi un certain nombre de fonctions de manipulation des groupes. Une partie de ces fonctions renvoie des données allouées statiquement ; elles disposent à présent d'homologues réentrantes, adaptées à une utilisation dans un contexte multithread.

Pour traiter le contenu des entrées présentes dans le fichier des groupes, on utilise une structure `group`, définie dans `<grp.h>` :

Nom	Type	Signification
<code>gr_name</code>	<code>char *</code>	Nom du groupe.
<code>gr_gid</code>	<code>gid_t</code>	GID du groupe.
<code>gr_mem</code>	<code>char **</code>	Table contenant les noms des utilisateurs, le dernier élément étant un pointeur NULL.

Les fonctions `getgrnam()` et `getgrgid()` – ainsi que `getgrnam_r()` et `getgrgid_r()` – permettent d’obtenir une structure `group` à partir d’un nom ou d’un `GID`.

```
struct group * getgrnam (const char * nom);
int getgrnam_r (const char * nom, struct group * retour,
               char * buffer, size_t taille_buffer,
               struct group ** pointeur_resultat);
struct group * getgrgid (gid_t gid);
int getgrgid_r (gid_t gid, struct group * retour,
               char * buffer, size_t taille_buffer,
               struct group ** pointeur_resultat);
```

En Python, `grp.getgrnam()` et `grp.getgrgid()` sont simples à utiliser : elles renvoient directement un objet – correspondant à la structure `group` – représentant le groupe recherché. En C, l’invocation est un peu plus compliquée. Les fonctions `getgrnam()` et `getgrgid()` renvoient un pointeur sur une zone de mémoire allouée statiquement ou un pointeur `NULL` si aucune entrée n’a été trouvée. Le fonctionnement de `getgrnam_r()` et de `getgrgid_r()` est légèrement plus compliqué du fait qu’il faut fournir un espace pour stocker les chaînes de caractères sur lesquelles la structure regroupe des pointeurs :

- La structure `group` sur laquelle on passe un pointeur en second argument est remplie avec les données lues.
- Le buffer passé en troisième argument, dont la taille est indiquée à la suite, est utilisé pour stocker les chaînes de caractères correspondant aux champs `gr_name` et `gr_mem` de la structure `group`. Si le buffer est trop petit, la fonction échoue et le code d’erreur `ERANGE` est inscrit dans la variable globale `errno`.
- Finalement, le pointeur transmis en dernier argument est rempli avec un pointeur sur la structure `group` passée en deuxième argument, ou avec `NULL` en cas d’échec.

La valeur de retour de `getgrnam_r()` et `getgrgid_r()` est nulle si elles réussissent.

Parfois, on peut être amené à examiner l’ensemble des groupes définis sur la machine, ne serait-ce que dans le cadre d’un utilitaire d’aide à l’administration système. En Python, c’est simple, il existe une fonction `grp.getgrall()` qui renvoie une liste de tous les objets `group`. En C, il faudra utiliser les fonctions `getgrent()` et `getgrent_r()` qui permettent de lire séquentiellement tous les enregistrements du fichier des groupes.

```
struct group * getgrent (void);
int getgrent_r (struct group * retour,
               char * buffer, size_t taille_buffer,
               struct group ** pointeur_resultat);
```

Ces fonctions utilisent de manière interne un flux correspondant au fichier `/etc/group`. Lors de la première invocation de `getgrent()`, ce flux est ouvert, puis les lignes sont analysées successivement à chaque appel. En fin de fichier, `getgrent()` renvoie un pointeur `NULL`, et `getgrent_r()` une valeur non nulle. Toutefois, on peut avoir besoin de revenir volontairement au début du fichier des groupes. Il existe deux fonctions, `setgrent()` et `endgrent()`, qui ont un fonctionnement antagoniste, mais avec finalement le même résultat :

```
void setgrent (void);  
void endgrent (void);
```

La fonction `setgrent()` ouvre le flux interne utilisé par `getgrent()` et `getgrent_r()`. Si le flux est déjà ouvert, sa position de lecture est ramenée au début. Si `setgrent()` n'est pas appelée explicitement, la première invocation de `getgrent()` le fera automatiquement. La fonction `endgrent()` referme le flux interne. En conséquence, l'une et l'autre de ces fonctions ont pour résultat de faire reprendre la prochaine lecture au début du fichier.

Il est parfois nécessaire de consulter les données se trouvant dans un autre fichier que `/etc/group`. Même si ce genre de situation est rare, on peut la rencontrer par exemple lorsqu'on administre un système distant dont la partition `/etc` est montée par NFS dans un autre emplacement de notre arborescence. Cette situation se présente aussi pour des raisons de sécurité lorsqu'un répertoire est employé comme racine – avec l'appel `chroot()` – pour un processus particulier (comme `/var/ftp`).

Pour cela on dispose des fonctions `fgetgrent()` et `fgetgrent_r()`, qui permettent de lire depuis un flux qu'on ouvre et qu'on ferme normalement avec `fopen()` et `fclose()`.

```
struct fgetgrent (FILE * flux);  
struct fgetgrent_r (FILE * flux, struct group * retour,  
                    char * buffer, size_t taille_buffer,  
                    struct group ** pointeur_resultat);
```

Ces deux routines fonctionnent exactement comme `getgrent()` et `getgrent_r()`, en se servant simplement du flux transmis en premier argument plutôt que de le gérer elles-mêmes.

On notera l'existence d'une fonction de bibliothèque nommée `initgroups()`, qui sert à consulter le fichier des groupes et à initialiser la liste des groupes supplémentaires d'un utilisateur donné.

```
int initgroups (const char * nom, gid_t gid);
```

Cette routine est privilégiée car elle invoque `setgroups()`, que nous avons étudiée dans le chapitre 2, et qui réclame la capacité `CAP_SETGID`. Cette fonction est appelée lors de la connexion d'un utilisateur par l'utilitaire `/bin/login`, ainsi que par `/bin/su`. On ne l'invoque normalement pas dans une application courante.

Fichier des utilisateurs

Comme le fichier des groupes, celui des utilisateurs est stocké dans le répertoire `/etc`. Typiquement, il s'agit du fichier `/etc/passwd`. Ce dernier est accessible en lecture pour tous les utilisateurs du système (ceci est nécessaire par exemple pour trouver le nom réel d'un utilisateur à partir de son UID). Toutefois, le mot de passe y apparaît de manière cryptée, comme nous en avons parlé dans le chapitre 18. L'évolution des processeurs rend à présent possible la recherche de mots de passe par force brute, en cryptant successivement tout le dictionnaire pour découvrir la chaîne chiffrée correspondant à celle du fichier `/etc/passwd`. Pour cette raison, la plupart des systèmes Linux utilisent à présent la technique des *shadow passwords*. Le mot de passe crypté n'est plus stocké dans `/etc/passwd` mais dans un autre fichier, comme `/etc/shadow`, accessible en lecture uniquement par un processus ayant un UID nul.

Le fichier contient une ligne pour chaque utilisateur, avec un certain nombre de paramètres. Pour manipuler ces enregistrements, on utilise la structure `passwd`, définie dans `<pwd.h>` :

Nom	Type	Signification
<code>pw_name</code>	<code>char *</code>	Nom de l'utilisateur, tel qu'il est employé pour la connexion.
<code>pw_passwd</code>	<code>char *</code>	Mot de passe crypté.
<code>pw_uid</code>	<code>uid_t</code>	UID de l'utilisateur.
<code>pw_gid</code>	<code>gid_t</code>	GID principal de l'utilisateur.
<code>pw_gecos</code>	<code>char *</code>	Commentaires sur l'utilisateur.
<code>pw_dir</code>	<code>char *</code>	Répertoire personnel de l'utilisateur.
<code>pw_shell</code>	<code>char *</code>	Le shell employé lors de la connexion de l'utilisateur.

Nous pouvons relever quelques points :

- Le membre `pw_passwd` n'est pas significatif sur les systèmes employant les *shadow passwords*. Si vous désirez écrire une application qui s'assure de l'identité d'un utilisateur en vérifiant son mot de passe, il faut que celle-ci puisse accéder au fichier `/etc/shadow` plutôt que `/etc/passwd`, et qu'elle puisse y lire les enregistrements.
- Le champ `pw_gid` correspond au groupe principal de l'utilisateur. L'accès aux GID de ses groupes supplémentaires est possible avec la fonction `getgroups()`, que nous avons vue dans le chapitre 2.

- Le champ `pw_gecos` peut contenir des informations plus ou moins pertinentes suivant les habitudes d'administration sur le système. Dans certains cas, il est vide ou ne comprend que le nom complet de l'utilisateur. À l'opposé, on peut rencontrer des systèmes où ce champ est lui-même scindé en plusieurs sous-enregistrements séparés par des virgules comportant adresse, numéro de téléphone, numéro de fax, etc.
- Le répertoire personnel de l'utilisateur indiqué dans `pw_dir` est normalement accessible en lecture, parcours et écriture par l'utilisateur. Pour certains comptes particuliers (serveurs), il peut s'agir d'un répertoire uniquement accessible en parcours, voire de la racine du système de fichiers.
- De même, le shell employé pour la connexion, indiqué dans le champ `pw_shell`, peut dans certains cas particuliers être un programme spécial (`/sbin/shutdown` par exemple).

L'accès à la structure `passwd` s'obtient par le biais de fonctions qui ressemblent largement à celles que nous avons rencontrées pour les groupes d'utilisateurs. Pour avoir l'entrée correspondant à un utilisateur donné, `getpwuid()` et `getpwnam()` permettent des recherches respectivement sur l'UID ou le nom de connexion. Leurs versions réentrantes, `getpwuid_r()` et `getpwnam_r()`, fonctionnent sur le même modèle que celui qui a été décrit pour `getgrnam_r()`.

```
struct passwd * getpwuid (uid_t uid);
int getpwuid_r (uid_t uid, struct passwd * retour,
               char * buffer, size_t taille_buffer,
               struct passwd ** pointeur_resultat);
struct passwd * getpwnam (const char * nom);
int getpwnam_r (const char * nom, struct passwd * retour,
               char * buffer, size_t taille_buffer,
               struct passwd ** pointeur_resultat);
```

De même que pour les groupes, nous pouvons utiliser les fonctions `pwd.getpwuid()`, `pwd.getpwnam()` et `pwd.getpwall()` en Python. Lorsqu'on veut balayer tout le fichier des utilisateurs en C, on peut utiliser la fonction `getpwent()` ou sa cousine réentrante, `getpwent_r()`.

```
struct passwd * getpwent (void);
int getpwent_r (struct passwd * retour,
               char * buffer, size_t taille_buffer,
               struct passwd ** pointeur_resultat);
```

Pour réinitialiser la lecture séquentielle du fichier des utilisateurs, on a le choix entre `setpwent()`, qui ouvre le flux interne, ou `endpwent()`, qui le ferme.

```
void setpwent (void);
void endpwent (void);
```

Quand on désire travailler avec un autre fichier que `/etc/passwd` (notamment `/etc/shadow` ou `/var/ftp/etc/passwd`), on emploie les fonctions `fgetpwent()` ou `fgetpwent_r()` en leur transmettant le pointeur de flux correspondant au fichier déjà ouvert.

```
struct passwd * fgetpwent (FILE * flux);  
int fgetpwent_r (FILE * flux, struct passwd * retour,  
                char * buffer, size_t taille_buffer,  
                struct passwd ** pointeur_resultat);
```

La fonction `putpwent()` permet de créer un enregistrement dans un fichier d'utilisateurs dont on lui fournit un pointeur de flux. L'emploi de cette routine est déconseillé car elle ajoute simplement l'enregistrement, sans vérifier s'il existait auparavant. Finalement, les applications devant modifier le fichier des mots de passe (comme `/usr/bin/passwd` bien entendu, mais aussi des utilitaires comme `/usr/bin/chsh`) devront être installées *Set-UID root* et modifieront directement le fichier.

```
int putpwent (const struct passwd * passwd, FILE * flux);
```

Une fonctionnalité importante du fichier des mots de passe concerne l'obtention du nom de connexion de l'utilisateur ayant lancé le processus courant. En réalité, les fonctions `getlogin()` et `getlogin_r()` donnent accès au nom de l'utilisateur connecté sur le terminal de contrôle du processus. Ces routines, à la différence des précédentes, sont déclarées dans `<unistd.h>` :

```
char * getlogin (void);  
int getlogin_r (char * buffer, size_t taille_buffer);
```

La fonction `cuserid()` fournit le nom de l'utilisateur correspondant à l'UID effectif du processus appelant. Elle est déclarée dans `<stdio.h>` :

```
char * cuserid (char * nom);
```

Si le buffer transmis en argument n'est pas `NULL`, il doit faire au moins `L_cuserid` octets de long, et le nom y est stocké. Sinon, la fonction renvoie un pointeur sur une zone de mémoire allouée statiquement.

Attention, cette routine n'est pas portable ; elle est implémentée différemment sur d'autres Unix. Elle est considérée désormais comme obsolète.

Fichier des interpréteurs shell

Il existe normalement un fichier `/etc/shells`, qui contient, ligne par ligne, la liste des interpréteurs de commandes disponibles. En voici un exemple sur une distribution Fedora 13 :

```
$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/ksh
/bin/tcsh
/bin/csh
/bin/zsh
$
```

Cette liste est employée principalement par l'utilitaire `/usr/bin/chsh` et par les programmes d'aide à l'administration système pour ajouter un utilisateur. Pour lire le contenu de ce fichier, les fonctions `getusershell()`, `setusershell()` et `endusershell()` sont déclarées dans `<unistd.h>` :

```
char * getusershell (void);
void setusershell (void);
void endusershell (void);
```

Comme nous l'avons déjà observé avec le fichier des groupes et celui des utilisateurs, la fonction `getusershell()` permet de lire le fichier des shells séquentiellement. `setusershell()` et `endusershell()` ramènent quant à elles la position de lecture au début.

Un utilisateur n'est pas forcé de choisir son shell de connexion dans cette liste. Par exemple, dans le cas d'un nom d'utilisateur fictif servant à arrêter le système, le programme de connexion indiqué dans le fichier des utilisateurs pourra être `/sbin/halt`. On notera également que si le fichier des shells n'est pas accessible, `getusershell()` se comporte comme si celui-ci contenait les lignes `/bin/sh` et `/bin/csh`.

Nom d'hôte et de domaine

Nom d'hôte

Le nom d'hôte est principalement employé pour identifier le système lors d'un dialogue avec un utilisateur humain. Ce nom ne sert généralement pas lors des commu-

nications entre ordinateurs, où on utilise plutôt des identifications numériques comme l'adresse IP ou l'adresse MAC. Pour obtenir le nom de la machine sur laquelle une application se déroule, on emploie la fonction `gethostname()`. La routine privilégiée `sethostname()` sert à configurer le nom d'hôte. Elle est généralement invoquée une seule fois dans un script de démarrage par le biais de l'utilitaire `/bin/hostname` qui lui sert d'interface.

```
int gethostname (char * buffer, size_t taille);  
int sethostname (char * buffer, size_t taille);
```

La taille du buffer contenant le nom est transmise en seconde position. Si le buffer est trop petit pour recevoir la chaîne de caractères, `gethostname()` échoue avec l'erreur `ENAMETOOLONG`. Python propose une fonction `socket.gethostname()` mais pas de fonction de modification directe du nom d'hôte.

Le nom d'hôte qu'on manipule avec `gethostname()` ou `sethostname()` doit être complet, c'est-à-dire qu'il doit contenir le domaine en entier. En voici un exemple d'utilisation :

```
exemple-gethostname.c :  
#include <errno.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int main (void)  
{  
    char * buffer = NULL;  
    size_t taille = 8;  
  
    buffer = malloc(taille);  
    while (gethostname(buffer, taille) != 0) {  
        if (errno != ENAMETOOLONG) {  
            perror("gethostname");  
            return EXIT_FAILURE;  
        }  
        taille += 8;  
        buffer = realloc(buffer, taille);  
    }  
    fprintf(stdout, "%s\n", buffer);  
    free(buffer);  
    return EXIT_SUCCESS;  
}
```

Nous pouvons vérifier que le nom de la machine est bien complet :

```
$ ./exemple-gethostname  
venux.logilin.fr  
$
```

Lorsqu'on utilise l'appel système `sethostname()`, on signale simplement en second argument la longueur du nom, telle qu'elle est fournie par `strlen()`. Si ce nom est trop long, la fonction renvoie l'erreur `EINVAL` dans `errno`. Pour être autorisé, le changement de nom d'hôte doit être réalisé par un processus ayant la capacité `CAP_SYS_ADMIN`.

Nom de domaine

Parallèlement, il existe deux appels système, `getdomainname()` et `setdomainname()`, qui permettent d'obtenir et de configurer le nom du domaine auquel appartient la machine.

```
int getdomainname (char * buffer, size_t taille);  
int setdomainname (char * buffer, size_t taille);
```

En fait, `getdomainname()` n'est plus un appel système sous Linux, il a été remplacé par une fonction de bibliothèque qui utilise l'appel `uname()`. On notera que `getdomainname()` renvoie toujours une chaîne vide si on ne prend pas le système NIS pour déterminer le domaine. Sur la plupart des stations Linux autonomes, cette fonction n'est donc pas utile.

Identifiant d'hôte

Pour essayer d'identifier une machine de manière unique, la bibliothèque C définit deux fonctions, `gethostid()` et `sethostid()`. La seconde est une fonction privilégiée qui enregistre l'identifiant qu'on lui transmet dans un fichier (généralement `/etc/hostid`). Lorsque `gethostid()` est invoquée sans qu'on ait appelé `sethostid()` auparavant, elle utilise l'adresse IP de la première interface réseau de la machine. Si cette opération se révèle impossible, elle renvoie 0.

```
long int gethostid (void);  
int sethostid (long int identifiant);
```

L'identifiant peut, sur certains systèmes autres que Linux, être construit directement à partir de l'adresse MAC de l'interface réseau.

Informations sur le noyau

Identification du noyau

Il peut être utile dans une application d'identifier la version du noyau en cours d'exécution (par exemple pour tirer parti de certaines nouvelles fonctionnalités ou pour éviter un bogue présent dans une ancienne version). L'appel système `uname()` permet d'obtenir plusieurs renseignements sur le système.

```
| int uname (struct utsname * utsname);
```

Les informations sont stockées dans une structure `utsname` définie dans `<sys/utsname.h>`, et dont tous les membres sont du type `char *` :

Nom	Signification
<code>sysname</code>	Nom du système d'exploitation. Pour nous, « Linux ».
<code>nodename</code>	Nom complet de la machine, comme avec <code>gethostname()</code> .
<code>release</code>	Numéro de version du noyau.
<code>version</code>	Numéro de révision du noyau au sein de la version courante.
<code>machine</code>	Type de machine. Il s'agit du nom du processeur suivi de celui du fabricant de l'ordinateur. Ce dernier nom n'est pas toujours disponible ; on obtient souvent quelque chose comme « i686-unknown ».
<code>domainname</code>	Nom du domaine auquel appartient la machine, comme dans <code>getdomainname()</code> . Pour que ce champ soit disponible, il faut définir la constante symbolique <code>_GNU_SOURCE</code> avant l'inclusion de l'en-tête <code><sys/utsname.h></code> .

En Python, la fonction `os.uname()` renvoie une liste des cinq premiers champs précédents (sauf `domainname`).

Voici un programme simple permettant de visualiser les différents champs :

```
| exemple-uname.c
| #define _GNU_SOURCE
| #include <stdio.h>
| #include <stdlib.h>
| #include <sys/utsname.h>
|
| int main (void)
| {
|     struct utsname utsname;
|     uname(& utsname);
|     fprintf(stdout, " sysname = %s \n nodename = %s \n"
```

```

        " release = %s \n version = %s \n"
        " machine = %s \n domain  = %s \n",
        utsname.sysname,
        utsname.nodename,
        utsname.release,
        utsname.version,
        utsname.machine,
        utsname.domainname);
    return EXIT_SUCCESS;
}

```

Et voici un exemple d'exécution :

```

$ ./exemple-uname
sysname = Linux
nodename = tr-a-1
release = 2.6.33.6-147.2.4.fc13.i686
version = #1 SMP Fri Jul 23 17:27:40 UTC 2010
machine = i686
domain  = (none)
$

```

Informations sur l'état du noyau

Il est rare dans un cadre applicatif d'avoir réellement besoin d'obtenir des renseignements pointus sur l'état du système. Toutefois, cela est possible sous Linux à l'aide de l'appel système `sysinfo()`, déclaré dans `<sys/sysinfo.h>` :

```
int sysinfo (struct sysinfo * info);
```

Cet appel système n'est absolument pas portable sur d'autres systèmes que Linux. La structure `sysinfo` qu'il remplit est définie dans `<linux/kernel.h>` avec les membres suivants :

Nom	Type	Signification
uptime	Long	Nombre de secondes écoulées depuis le boot de la machine.
loads	unsigned long [3]	Charge système durant les 1, 5 et 10 dernières minutes, 65 535 correspondant à 100 %.
freeram	unsigned long	Mémoire libre, exprimée en octets.
sharedram	unsigned long	Mémoire partagée entre plusieurs processus.
bufferram	unsigned long	Mémoire utilisée pour les buffers du noyau.

Nom	Type	Signification
totalswap	unsigned long	Taille totale du périphérique de swap, en octets.
freeswap	unsigned long	Quantité disponible sur le périphérique de swap.
procs	unsigned short	Nombre de processus en cours d'exécution.

Le programme suivant met en œuvre l'appel système sysinfo(), à la manière des utilitaires /usr/bin/uptime ou /usr/bin/top.

```

exemple-sysinfo.c :
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>
#include <linux/kernel.h>

int main (void)
{
    struct sysinfo info;
    if (sysinfo(& info) != 0) {
        perror("sysinfo");
        exit(1);
    }
    fprintf(stdout, "Nb secondes depuis boot      : %ld \n",
              info.uptime);
    fprintf(stdout, "Charge systeme depuis 1 mn : %.2f%%\n",
              info.loads[0] / 655.36);
    fprintf(stdout, "                               5 mn : %.2f%%\n",
              info.loads[1] / 655.36);
    fprintf(stdout, "                               10 mn : %.2f%%\n",
              info.loads[2] / 655.36);
    fprintf(stdout, "Memoire disponible           : %ld Mo\n",
              info.freeram >> 20);
    fprintf(stdout, "Memoire partagee            : %ld Mo\n",
              info.sharedram >> 20);
    fprintf(stdout, "Memoire dans buffers         : %ld Mo\n",
              info.bufferram >> 20);
    fprintf(stdout, "Espace de swap total         : %ld Mo\n",
              info.totalswap >> 20);
    fprintf(stdout, "Espace de swap libre         : %ld Mo\n",
              info.freeswap >> 20);
    fprintf(stdout, "Nb processus en cours       : %d \n",
              info.procs);
    return EXIT_SUCCESS;
}

```

Rappelons que cet appel système est spécifique à Linux et qu'il ne sera pas disponible sur d'autres architectures.

```
$ ./exemple-sysinfo
Nb secondes depuis boot      : 108453
Charge systeme depuis 1 mn   : 26.37%
                             5 mn : 32.28%
                             10 mn : 38.23%
Memoire disponible           : 96 Mo
Memoire partagee             : 0 Mo
Memoire dans buffers         : 131 Mo
Espace de swap total         : 2047 Mo
Espace de swap libre         : 2043 Mo
Nb processus en cours        : 300
$
```

Système de fichiers

Un système Linux normal gère au minimum deux partitions physiques : l'une constitue la racine de l'arborescence du système de fichiers et l'autre est utilisée comme périphérique de *swap*. On emploie également d'autres systèmes de fichiers pour accéder aux périphériques amovibles, comme les lecteurs de CD-Rom ou les clés USB. Il peut aussi être intéressant de se servir de plusieurs partitions physiques différentes sur le disque, qu'on monte à différents endroits du système de fichiers. Ceci permet par exemple d'utiliser un espace disque limité et figé pour les répertoires ne devant pas évoluer sensiblement (*/*, */bin*, */etc*, */dev*, */usr*), et de laisser le reste de la capacité disponible pour les zones susceptibles de subir des modifications importantes (*/home*, */tmp*). On peut également utiliser un découpage du disque en plusieurs partitions pour simplifier les travaux de sauvegardes systématiques en isolant les données modifiées fréquemment de celles qui n'évoluent pas une fois le système installé. Lorsqu'on administre un parc de plusieurs stations Unix, il est très important qu'un utilisateur puisse disposer de son environnement de travail personnel quelle que soit la machine devant laquelle il s'assoit. On organise alors une distribution des répertoires personnels */home/xxx* sur l'ensemble des machines, avec un montage au travers du réseau par le protocole NFS.

Enfin, on peut être amené à personnaliser le partitionnement pour des besoins spécifiques. À titre d'exemple, je peux citer un cas où je devais installer des stations Linux dans un environnement de production assez périlleux, sujet à de fréquents problèmes de distribution électrique et à des manipulations pour le moins maladroites. J'ai décidé d'employer un montage en lecture seule de tous les répertoires système (*/usr*,

/etc, ...). Les données propres à l'utilisateur ainsi que les fichiers de configuration susceptibles de changer se trouvaient dans des répertoires (/home et /usr/local) montés par NFS depuis un serveur administré par une équipe de maintenance opérationnelle. Enfin, une partition reformatée à chaque démarrage de la machine regroupait les fichiers à faible durée de vie dans /tmp. L'avantage d'une telle organisation était de permettre un arrêt brutal de la machine sans risque de perte de données.

Caractéristiques des systèmes de fichiers

Pour simplifier le travail de l'administrateur – afin d'autoriser l'insertion de périphériques amovibles par n'importe quel utilisateur et permettre le montage automatique de certaines partitions au démarrage de la machine –, on établit la liste des systèmes de fichiers disponibles et on la stocke dans /etc/fstab. Il sera alors possible de rattacher une partition au système de fichiers simplement en invoquant une commande du type :

```
$ mount /mnt/disque-A  
$ mount /mnt/flash
```

De plus, les gestionnaires graphiques de fichiers sauront monter ou démonter les nœuds correspondant en consultant cette table. Chaque ligne du fichier /etc/fstab contient les champs suivants, séparés par des tabulations ou par des espaces.

- Nom du fichier spécial en mode bloc représentant le périphérique (par exemple /dev/sda1 pour une partition disque, etc.). Pour les pseudo-systèmes de fichiers comme /proc ou /sys, on indique souvent le mot-clé none. Les répertoires provenant d'un serveur NFS sont mentionnés en signalant le nom du serveur (ou son adresse IP), suivi d'un deux-points et du chemin d'accès au répertoire. Par exemple srvr_home:/home/user. Sur les systèmes récents, on remplace le nom de la partition par son identifiant, qui est recherché lors du *boot*, par exemple : UUID=599e43fe-57f7-4f8b-9370-26f42d283c14.
- Le point de montage du système de fichiers dans l'arborescence générale (par exemple / pour la partition racine, /media/cdrom ou /mnt/cle pour des périphériques amovibles, /home/users pour un répertoire normal). Pour les partitions de *swap*, on utilise le mot-clé none.
- Le type du système de fichiers. Le noyau Linux reconnaît couramment les systèmes minix (héritage historique des premiers noyaux), ext (obsolète, à oublier), ext3 et ext4 (les standards Linux actuels), ext2 (leur prédécesseur, encore largement utilisé), iso9660 (pour les CD-Rom), udf (pour les DVD-Rom), swap, nfs, msdos, ntfs, et vfat (pour accéder aux partitions Dos ou Windows). On peut mentionner les pseudo systèmes de fichiers proc et sys (informations du noyau) et devpts (pseudo terminaux Unix 98 comme nous les étudierons dans le

chapitre 32). Il existe de nombreux autres systèmes, moins utilisés ou expérimentaux, qui sont généralement disponibles sous forme de modules du noyau.

- Des options concernant le montage du système de fichiers. Il existe des options générales, comme `noauto` qui empêche le montage automatique au démarrage (utile pour les périphériques amovibles), `user` qui autorise n'importe quel utilisateur à monter le système de fichiers, `ro` qui demande un montage en lecture seule, ou `mand` qui permet les verrouillages stricts, comme nous l'avons vu au chapitre 22. Il y a aussi des options spécifiques pour chaque type de système. On consultera au besoin les pages de manuel `mount(8)`, `fstab(5)` et `nfs(5)` pour avoir des détails supplémentaires.
- La fréquence des sauvegardes de la partition par l'utilitaire `dump`. Cette option n'est généralement plus utilisée, on la remplace par un zéro.
- L'ordre de vérification des systèmes de fichiers au démarrage de la machine. Si ce champ est absent ou nul, la partition n'est pas vérifiée. Sinon, le programme `fsck` traite les systèmes de fichiers séquentiellement dans l'ordre indiqué. Normalement, la partition racine doit être configurée avec la valeur 1, et les autres avec la valeur 2. Si les partitions sont gérées par des contrôleurs de disque distincts, elles sont vérifiées en parallèle.

Voici un extrait d'un tel fichier :

```
$ cat /etc/fstab
tmpfs                /dev/shm            tmpfs defaults      0 0
devpts               /dev/pts            devpts gid=5,mode=620 0 0
sysfs                /sys                sysfs defaults      0 0
proc                 /proc               proc  defaults      0 0
UUID=de3af5a8-c3fd-4fd3-9ee7-b840ad70530a / ext4  defaults      1 1
UUID=d5ca8352-37cb-4bf6-bc7a-7df2dd513250 swap swap default      0 0
/dev/sda6             /home               ext3  defaults,mand 1 2
192.168.3.4:/var/ftp/pub/Video /mnt/Video  nfs  defaults,noauto,ro 0 0
```

Naturellement, il est toujours possible d'invoquer directement la commande `mount` avec toutes les options en ligne de commande, mais il est beaucoup plus agréable de n'avoir à saisir que « `mount /media/flash` » par exemple. Il faut donc accorder une grande importance à la rédaction du fichier `/etc/fstab`, d'autant que cette tâche d'administration n'a lieu qu'une seule fois, lors de l'installation du système (ou en cas d'ajout d'un nouveau périphérique). La bibliothèque C propose un ensemble de fonctions permettant de consulter ce fichier. Les routines `setfsent()` et `endfsent()` fonctionnent comme d'habitude en ouvrant ou en fermant le fichier `/etc/fstab`, ce qui a donc pour conséquence de faire reprendre la lecture suivante au début.

```
int setfsent (void);
void endfsent (void);
```

Pour manipuler les enregistrements, une structure `fstab` est définie dans `<fstab.h>` :

Nom	Type	Signification
<code>fs_spec</code>	<code>char *</code>	Nom du fichier spécial représentant le périphérique concerné. Peut également être un nom d'hôte suivi d'un chemin d'accès pour les montages NFS.
<code>fs_file</code>	<code>char *</code>	Point de montage dans l'arborescence du système de fichiers.
<code>fs_vfstype</code>	<code>char *</code>	Type de système de fichiers.
<code>fs_mntops</code>	<code>char *</code>	Options de montage, globales ou spécifiques au type de système de fichiers employé.
<code>fs_type</code>	<code>char *</code>	Mode d'accès à la partition montée (voir plus bas).
<code>fs_freq</code>	<code>int</code>	Période (en jours) entre deux sauvegardes (souvent inutilisé).
<code>fs_passno</code>	<code>int</code>	Ordre de vérification de la partition au démarrage.

Le champ `fs_type` peut contenir l'une des chaînes de caractères décrites dans le tableau suivant.

Attention, il s'agit bien de chaînes de caractères et non de constantes symboliques. Il faut donc les examiner avec `strcmp()`.

Chaîne	Signification
<code>FSTAB_RW</code>	Partition à monter en lecture et écriture (par exemple <code>/tmp</code>).
<code>FSTAB_RQ</code>	Partition à monter en lecture et écriture avec un système de comptabilité par quotas (par exemple <code>/home</code>).
<code>FSTAB_RO</code>	Partition à monter en lecture seule (par exemple <code>/usr</code>).
<code>FSTAB_SW</code>	Partition de <i>swap</i> .
<code>FSTAB_XX</code>	Partition ignorée.

La fonction `getfsent()` renvoie l'entrée suivante du fichier, `getfsspec()` recherche l'entrée dont le champ `fs_spec` (nom du fichier spécial de périphérique) correspond à la chaîne transmise en argument. La fonction `getfsfile()` retourne quant à elle l'entrée dont le membre `fs_file` (point de montage) correspond à son argument.

```
struct fstab * getfsent(void);
struct fstab * getfsspec (const char * nom);
struct fstab * getfsfile (const char * nom);
```

Ces fonctions renvoient un pointeur sur une structure stockée dans une zone de mémoire statique, ou `NULL` en cas d'échec.

Pour analyser ces données, on utilise une structure `mntent` définie dans `<mntent.h>`, dont les membres sont mieux nommés que ceux de `fstab` :

Les fonctions `setmntent()` et `endmntent()` permettent d'ouvrir et de fermer un flux en fournissant le nom du fichier. Ces routines rendent possible la manipulation de `/etc/mntab` mais également d'autres fichiers ayant le même format, comme `/etc/fstab` ou `/proc/mounts`.

Les arguments de `setmntent()` sont identiques à ceux de `fopen()`. Attention, le pointeur de flux renvoyé par `setmntent()` doit être refermé à l'aide de `endmntent()` et surtout pas avec `fclose()`.

[illegible]

Le programme suivant va utiliser `getmntent_r()` – bien que `getmntent()` aurait largement suffi dans ce contexte mono-thread – afin de consulter le fichier dont le nom est fourni en argument.

```
exemple-mtab.c :
#include <stdio.h>
#include <stdlib.h>
#include <mntent.h>

int main (int argc, char * argv[])
{
    struct mntent mtab;
    char buffer[256];
    FILE * file;

    if (argc != 2) {
        fprintf(stderr, "%s <fichier mtab>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((file = setmntent(argv[1], "r")) == NULL) {
        perror("setmntent");
        exit(EXIT_FAILURE);
    }
    while (1) {
        if (getmntent_r(file, & mtab, buffer, 256) == NULL)
            break;
        fprintf(stdout, "fsname = %s \n dir = %s\n type = %s \n"
            " opts = %s \n freq = %d \n passno = %d \n",
            mtab.mnt_fsname, mtab.mnt_dir,
            mtab.mnt_type, mtab.mnt_opts,
            mtab.mnt_freq, mtab.mnt_passno);
    }
    endmntent(file);
    return EXIT_SUCCESS;
}
```

On peut utiliser ce programme sur `/etc/fstab`, `/etc/mtab`, `/proc/mounts`...

```
$ ./exemple-mtab /proc/mounts
fsname = rootfs
dir = /
type = rootfs
opts = rw
freq = 0
passno = 0
fsname = /proc
```

```

dir = /proc
type = proc
opts = rw,relatime
freq = 0
passno = 0
fsname = /sys
dir = /sys
type = sysfs
opts = rw,relatime
freq = 0
passno = 0
[...]
fsname = /dev/sda6
dir = /home
type = ext3
opts = rw,mand,relatime,errors=continue,user_xattr,acl,data=ordered
freq = 0
passno = 0
[...]
$

```

Il est aussi possible d'ajouter une nouvelle entrée dans un fichier, à l'aide de `addmntent()`, qui permet d'écrire une application incorporant des routines de montage et de démontage de systèmes de fichiers tout en restant compatible avec l'utilitaire `mount`. On peut également employer ces routines pour créer un éditeur de fichier `/etc/fstab`. Comme il n'existe pas de routine spécialisée, si on veut supprimer une ou plusieurs lignes, il faut recopier le fichier entrée par entrée, en sautant celles qu'on veut éliminer, et utiliser `rename()` pour remplacer le fichier original. Naturellement, l'ajout d'enregistrements dans un fichier nécessite l'ouverture en mode « r+ ».

```
int addmntent (FILE * fichier, const struct mntent * mntent);
```

Pour analyser le champ `mnt_opts` de la structure `mntent`, il est conseillé d'utiliser la routine `getsubopt()` que nous avons étudiée dans le chapitre 4. Toutefois, lorsqu'on désire simplement vérifier la présence d'une option bien déterminée dans une entrée, on peut plutôt choisir la fonction `hasmntopt()`.

```
char * hasmntopt (const struct mntent * mntent,
                 const char * option);
```

Si l'option indiquée en second argument se trouve dans le champ `mnt_opts` de la structure passée en première position, cette fonction renvoie un pointeur sur le premier caractère de cette option (dans la chaîne `mnt_opts`). Sinon, elle transmet un pointeur `NULL`.

Ainsi le programme suivant recherche les partitions mentionnées dans `/etc/fstab` qui possèdent l'attribut « `mand` » autorisant un verrouillage strict des fichiers.

```
exemple-hasmntopt.c :
#include <stdio.h>
#include <stdlib.h>
#include <mntent.h>

int main (void)
{
    FILE * fichier;
    struct mntent * mntent;

    fichier = setmntent("/etc/fstab", "r");
    if (fichier == NULL)
        exit(EXIT_FAILURE);
    while (1) {
        mntent = getmntent(fichier);
        if (mntent == NULL)
            break;
        if (hasmntopt(mntent, "mand") != NULL)
            fprintf(stdout, "%s (%s)\n",
                    mntent->mnt_fsname,
                    mntent->mnt_dir);
    }
    endmntent(fichier);
    return EXIT_SUCCESS;
}
```

On retrouve alors la partition correspondante de l'exemple précédent :

```
$ ./exemple-hasmntopt
/dev/sda6 (/home)
$
```

Informations sur un système de fichiers

Pour obtenir des informations concernant un système de fichiers particulier, s'il est monté, il est possible d'employer les appels système `statfs()` ou `fstatfs()`. Le premier fournit des renseignements sur le système contenant le fichier (ou le répertoire) dont le nom est passé en argument. Le second appel système utilise un descripteur de fichier, qui doit donc avoir été préalablement ouvert.

```
int statfs(const char * fichier, struct statfs * statfs);
int fstatfs (int descripteur, struct statfs * statfs);
```

Les informations sont transmises dans une structure `statfs`, définie dans `<sys/statfs.h>` avec au minimum les membres suivants :

Nom	Type	Signification
<code>f_type</code>	<code>int</code>	Type de système de fichiers.
<code>f_bsize</code>	<code>int</code>	Taille de bloc.
<code>f_blocks</code>	<code>long</code>	Nombre total de blocs.
<code>f_bfree</code>	<code>long</code>	Nombre de blocs libres.
<code>f_bavail</code>	<code>long</code>	Nombre de blocs vraiment disponibles.
<code>f_files</code>	<code>long</code>	Nombre d'i-nœuds.
<code>f_ffree</code>	<code>long</code>	Nombre d'i-nœuds libres.
<code>f_fsid</code>	<code>fsid_t</code>	Identifiant du système de fichiers (peu utilisé actuellement).
<code>f_namelen</code>	<code>int</code>	Longueur maximale des noms de fichiers.

Le type de système de fichiers est indiqué sous forme numérique. Il existe une constante symbolique pour chaque système connu par le noyau. Ces constantes sont définies dans le fichier d'en-tête `<linux/magic.h>`. La constante symbolique est alors `XXX_SUPER_MAGIC`. Ou `XXX` correspond au type de système de fichiers. Par exemple, le système de fichiers `ext3` est représenté par la constante symbolique `EXT3_SUPER_MAGIC` qui prend la valeur `0xEF53`. Une application n'a normalement pas besoin de connaître ces valeurs.

Nous pouvons utiliser cet appel système pour obtenir des informations statistiques sur l'utilisation du système auquel appartient le fichier dont le nom est passé en argument.

```
exemple-statfs.c :
#include <stdio.h>
#include <stdlib.h>
#include <sys/vfs.h>

int main (int argc, char * argv[])
{
    struct statfs etat;
    int i;
    for (i = 1; i < argc; i++) {
        if (statfs(argv[i], & etat) < 0) {
            perror(argv[i]);
            continue;
        }
        fprintf(stdout, "%s : 1 bloc = %d octets \n"
            "      total %ld blocs \n"
            "      libre %ld blocs \n"
            "      disponible %ld blocs \n",
            argv[i], etat.f_bsize,
            etat.f_blocks, etat.f_bfree, etat.f_bavail);
    }
}
```

```
}  
    return EXIT_SUCCESS;  
}
```

L'exécution donne un résultat comparable à la commande `/bin/df`, qui affiche les mêmes statistiques pour tous les systèmes de fichiers en employant `getmntent()` sur `/etc/mtab`, comme nous l'avons déjà fait.

```
$ ./exemple-statfs /etc/  
/etc/ : 1 bloc = 4096 octets  
    total 2405312 blocs  
    libre 538049 blocs  
    disponible 513620 blocs  
$ ./exemple-statfs /home  
/home : 1 bloc = 4096 octets  
    total 13490224 blocs  
    libre 1602977 blocs  
    disponible 906661 blocs  
$ ./exemple-statfs /proc/  
/proc/ : 1 bloc = 4096 octets  
    total 0 blocs  
    libre 0 blocs  
    disponible 0 blocs  
$
```

Les appels système `statfs()` et `fstatfs()` ne sont pas définis par SUSv4 et ils ne sont que moyennement portables. Si cela pose un problème, on peut s'inspirer des sources de l'application `/bin/df` qui emploie des appels système différents suivant les machines, pour obtenir finalement le même résultat. Notons également au passage la persistance d'un appel système `ustat()` provenant de Système V, offrant un sous-ensemble des informations fournies par `statfs()`, et donc quasi obsolète de nos jours.

Montage et démontage des partitions

Nous ne détaillerons pas les mécanismes utilisés pour monter ou démonter les partitions, car ils sont susceptibles de changer en fonction des versions du noyau. Précisons simplement rapidement la structure des appels système `mount()` et `umount()` qui remplissent ce rôle. Ils sont déclarés dans `<sys/mount.h>` :

```
int mount (const char * fichier_special,  
           const char * point_montage, const char * type_systeme,  
           unsigned long attribut, const void * options);  
int umount (const char * nom);
```

Les arguments principaux de `mount()` ressemblent aux champs du fichier `/etc/fstab`. Un argument supplémentaire est présent pour préciser des attributs de montage. Il doit être rempli partiellement avec une valeur magique (`0xC0ED`) et complété par des valeurs spéciales définies dans le fichier d'en-tête `<linux/fs.h>`, dépendant donc du noyau.

Ces appels système sont privilégiés, nécessitant la capacité `CAP_SYS_ADMIN`. L'utilitaire `/bin/mount` est donc installé *Set-UID root*, et il gère lui-même les options `user` ou `nouser` qui autorisent ou interdisent le montage d'un système par n'importe quel utilisateur. Ces options sont internes à l'application et ne concernent pas l'appel système.

Si on désire créer un logiciel qui permette à l'utilisateur de monter ou de démonter des partitions décrites dans `/etc/fstab` (comme un gestionnaire graphique de fichiers), on emploiera donc de préférence les invocations suivantes :

```
|  execl ("/bin/mount", "/bin/mount", point_de_montage, NULL);
```

et

```
|  execl ("/bin/umount", "/bin/umount", point_de_montage, NULL);
```

Cela simplifiera les vérifications des autorisations et rejettera sur l'application `mount` le problème de compatibilité des systèmes de fichiers avec la version du noyau. La configuration du logiciel – par le biais d'un fichier d'initialisation – ou d'une variable d'environnement peut permettre de préciser le chemin d'accès de `mount` et de `umount` pour le cas où ils ne se trouvent pas dans `/bin` (ce qui est peu probable car ils doivent être disponibles avec le minimum vital du système).

Journalisation

Linux incorpore plusieurs mécanismes de journalisation des informations. Tout d'abord, nous étudierons le mécanisme servant à connaître le nom des utilisateurs connectés au système. Ensuite, nous examinerons le fichier permettant de garder une trace de toutes les connexions et redémarrages. Enfin, nous pourrions voir l'utilisation du système `syslog`, qui permet d'afficher et de mémoriser tous les événements importants en provenance d'une application système.

Journal utmp

Le fichier `utmp` se trouve généralement dans le répertoire `/var/run`. On peut parfois le rencontrer dans `/var/adm`, voire dans `/etc` suivant les distributions. Il sert à mémoriser l'état présent du système en stockant des lignes horodatées contenant une description d'un certain nombre d'événements.

Le fichier `utmp` est mis à jour par tous les utilitaires système (`init`, `getty`, `login`, `xterm`, `sshd`...). Chacun de ces processus renseignant au fur et à mesure la ligne de `utmp` qui le concerne.

Pour décrire ces données, la structure `utmp` est définie dans `<utmp.h>` :

Nom	Type	Signification
<code>ut_type</code>	<code>short int</code>	Description de l'enregistrement. Il existe dix types différents, que nous examinerons ci-dessous.
<code>ut_pid</code>	<code>pid_t</code>	PID du processus concerné.
<code>ut_line</code>	<code>char[32]</code>	Fichier spécial de terminal (sans le préfixe <code>/dev</code>).
<code>ut_id</code>	<code>char[4]</code>	Chaîne d'identification au sein du fichier <code>/etc/inittab</code> .
<code>ut_user</code>	<code>char[32]</code>	Nom de connexion de l'utilisateur.
<code>ut_host</code>	<code>char[256]</code>	Nom de l'hôte d'où provient la connexion.
<code>ut_exit</code>	<code>struct exit_status</code>	État de fin du processus.
<code>ut_session</code>	<code>long int</code>	Identifiant de session.
<code>ut_tv</code>	<code>struct timeval</code>	Horodatage de l'événement.
<code>ut_addr_v6</code>	<code>int32_t[4]</code>	Adresse IP de l'hôte distant.

Bien entendu, ces champs n'ont pas tous une signification simultanément. Leur utilisation dépend du type d'enregistrement.

Les différents types d'événements possibles sont les suivants :

Nom	Signification
<code>EMPTY</code>	Enregistrement vide.
<code>RUN_LVL</code>	Changement de niveau d'exécution du système.
<code>BOOT_TIME</code>	Démarrage de la machine. Permet d'enregistrer l'heure de <i>boot</i> .
<code>OLD_TIME</code>	Ancienne heure, juste avant une modification de l'horloge interne. Cet enregistrement est suivi d'un enregistrement <code>NEW_TIME</code> .
<code>NEW_TIME</code>	Nouvelle heure, juste après la modification de l'horloge interne.
<code>INIT_PROCESS</code>	Processus lancé par <code>init</code> .
<code>LOGIN_PROCESS</code>	Processus <code>login</code> .
<code>USER_PROCESS</code>	Connexion d'un utilisateur.
<code>DEAD_PROCESS</code>	Fin d'un processus.
<code>ACCOUNTING</code>	Début de comptabilisation.

Pour comprendre le principe, reprenons à la mise en route de la machine :

- Un enregistrement `BOOT_TIME` est ajouté dès que le noyau a lancé le processus `init`. Celui-ci détermine alors son niveau d'exécution et ajoute un enregistrement `RUN_LVL`.

- Le processus `init` consulte alors le fichier `/etc/inittab` et, pour chaque entrée valide, il invoque `fork()` et `exec()` pour lancer l'application attendue (par exemple `getty` ou `xdm`), en ajoutant à chaque fois un enregistrement `INIT_PROCESS` avec les champs `ut_id` et `ut_pid` remplis.
- L'utilitaire `getty` recherche l'enregistrement correspondant à son `PID`, remplit son champ `ut_line` avec le nom du terminal qu'il surveille, et modifie le champ `ut_type` pour qu'il contienne `LOGIN_PROCESS`.
- Lorsqu'un utilisateur se connecte, `getty` exécute `login`, qui recherche l'enregistrement correspondant à son `PID`, remplit le champ `ut_user` et modifie le champ `ut_type` avec le type `USER_PROCESS`.

L'utilitaire `login` peut aussi remplir les champs `ut_host` et `ut_addr_v6` lorsqu'il a été invoqué par `sshd` plutôt que par `getty`.

- Lorsqu'un processus se termine, `init` en est informé grâce à l'appel système `wait()`, il remplit l'enregistrement correspondant à son `PID` avec les codes de retour et lui met un type `DEAD_PROCESS`. Les applications comme `getty` ou `xterm` recherchent d'abord s'il existe un enregistrement de ce type correspondant à leur fichier spécial avant d'en créer un nouveau.

Il n'y a en effet pas de moyen d'effacer un enregistrement. Leur nombre est limité de fait par la quantité de pseudo-terminaux permettant des connexions simultanées.

Pour lire les informations de `utmp`, on utilise les fonctions `setutent()` et `endutent()`, qui initialisent et terminent la lecture, puis les routines `getutent()` ou sa contrepartie réentrante, `getutent_r()`, pour balayer le fichier séquentiellement.

```
void setutent(void);
void endutent(void);
struct utmp * getutent (void);
int          getutent_r (struct utmp * utmp, struct utmp **ret);
```

Lorsqu'on recherche un enregistrement correspondant à un `PID` donné, comme le fait `login`, on utilise `getutid()` ou `getutid_r()`, qui prennent en argument une structure `utmp`. Celle-ci doit avoir des champs `ut_pid` et `ut_type` remplis. Si le champ `ut_type` contient une constante `xxx_PROCESS`, on cherche un enregistrement ayant le même `ut_pid` et un `ut_type` correspondant également à un `xxx_PROCESS`. Sinon, on recherche un enregistrement ayant le même `ut_type` et le même `ut_pid`. Si aucune entrée ne correspond, `getutid()` renvoie `NULL`, et `getutid_r()` -1.

```
struct utmp * getutid (const struct utmp * utmp);
int          getutid_r (struct utmp * utmp, struct utmp **ret);
```


On peut aussi rechercher un enregistrement correspondant à un terminal particulier. Les fonctions `getutline()` et `getutline_r()` prennent une structure `utmp` en argument et renvoient l'enregistrement correspondant à la même valeur de `ut_line`, avec un `ut_type` contenant `LOGIN_PROCESS` ou `USER_PROCESS`.

```
struct utmp * getutline (const struct utmp * utmp);  
int          getutline_r (struct utmp * utmp,  
                           struct utmp ** ret);
```

Finalement, nous pouvons ajouter un enregistrement dans la base de données `utmp`. La fonction `pututline()` prend une structure `utmp` en argument et met à jour le fichier en recherchant un éventuel enregistrement avec le même PID. Le pointeur renvoyé est dirigé vers une copie de la structure ajoutée, ou est `NULL` si le processus n'a pas les autorisations nécessaires pour modifier la base de données `utmp`.

```
struct utmp * pututline (const struct utmp * utmp);
```

Le programme suivant affiche une partie des informations contenues dans la base de données `utmp`.

```
exemple-getutent.c :  
#include <stdio.h>  
#include <stdlib.h>  
#include <time.h>  
#include <utmp.h>  
  
void affiche_utmp (struct utmp * utmp)  
{  
    struct tm * tm;  
    char      heure[80];  
  
    tm = localtime(& (utmp->ut_tv.tv_sec));  
    strftime(heure, 80, "%x %X", tm);  
    switch (utmp->ut_type) {  
        case EMPTY :  
            break;  
        case RUN_LVL :  
            printf("%s : ", heure);  
            printf("Run-level \n");  
            break;  
        case BOOT_TIME :  
            printf("%s : ", heure);  
            printf("Boot \n");  
            break;  
        case OLD_TIME :  
            printf("%s : ", heure);  
            printf("Old Time\n");  
            break;
```

```

        case NEW_TIME :
            printf("%s : ", heure);
            printf("New Time\n");
            break;
        case INIT_PROCESS :
            printf("%s : ", heure);
            printf("Init process, ");
            printf("PID = %u, ", utmp->ut_pid);
            printf("inittab = %s\n", utmp->ut_id);
            break;
        case LOGIN_PROCESS :
            printf("%s : ", heure);
            printf("Login process, ");
            printf("PID = %u, ", utmp->ut_pid);
            printf("TTY = %s\n", utmp->ut_line);
            break;
        case USER_PROCESS :
            printf("%s : ", heure);
            printf("User process, ");
            printf("PID = %u, ", utmp->ut_pid);
            printf("TTY = %s, ", utmp->ut_line);
            printf("%s\n", utmp->ut_user);
            break;
        case DEAD_PROCESS :
            break;
        default :
            printf("?");
            break;
    }
}

int main (void)
{
    struct utmp * utmp;
    while ((utmp = getutent()) != NULL)
        affiche_utm(utmp);
    return EXIT_SUCCESS;
}

```

Voici un exemple sur un PC Linux portable :

```

$ ./exemple-getutent
08/20/10 12:34:46 : Boot
08/20/10 12:34:46 : Run-level
08/20/10 12:34:56 : Login process, PID = 1735, TTY = tty2
08/20/10 12:34:56 : Login process, PID = 1737, TTY = tty3
08/20/10 12:34:56 : Login process, PID = 1739, TTY = tty4

```

```
08/20/10 12:34:56 : Login process, PID = 1741, TTY = tty5
08/20/10 12:34:56 : Login process, PID = 1743, TTY = tty6
08/20/10 12:35:13 : User process, PID = 1919, TTY = tty1, cpb
08/20/10 14:32:05 : User process, PID = 5159, TTY = pts/0, cpb
08/21/10 18:11:14 : User process, PID = 5159, TTY = pts/1, cpb
$
```

Fonctions X/Open

Une partie des routines que nous avons étudiées dispose d'équivalents portables car ils sont déclarés dans SUSv4. Ces fonctions manipulent les enregistrements utmp par l'intermédiaire d'une structure utmpx ayant les membres suivants (identiques à ceux de la structure utmp) :

Nom	Type
ut_type	short int
ut_pid	pid_t
ut_line	char [32]
ut_id	char [4]
ut_user	char [32]
ut_tv	struct timeval

Les fonctions suivantes sont équivalentes à leurs homologues traitant les structures utmp. Elles sont d'ailleurs définies dans la bibliothèque Glibc par des alias. Notons qu'il n'existe pas de fonction réentrante :

```
void setutxent (void);
void endutxent (void);
struct utmpx * getutxent(void);
struct utmpx * getutxid (const struct utmpx * utmpx);
struct utmpx * getutxline (const struct utmpx * utmpx);
struct utmpx * pututxline (const struct utmpx * utmpx);
```

Toutes ces fonctions sont déclarées dans <utmpx.h>.

Journal wtmp

Le fichier utmp est effacé à chaque démarrage de la machine, et lorsqu'un processus de connexion se termine, son enregistrement est marqué comme DEAD_PROCESS avant d'être réutilisé ensuite. Il existe également sur le système Linux un fichier nommé wtmp qui n'est pas effacé. Les enregistrements y sont ajoutés successivement. Ce

fichier sert d'historique des connexions. On le trouve en général dans `/var/log`, mais on peut aussi le rencontrer dans `/var/adm` ou `/etc`. Sur certaines machines, il est également copié automatiquement sur une imprimante système afin de conserver une trace de toutes les connexions des utilisateurs, à des fins de sécurité. Le fichier `wtmp` est souvent archivé ou effacé de façon automatique une fois par mois par un script d'administration déclenché par le démon `crond`.

Pour examiner le contenu de `wtmp`, on peut utiliser les mêmes routines que celles que nous avons déjà étudiées. Pour cela, il existe une routine nommée `utmpname()` qui permet de préciser le nom du fichier qu'on veut lire. Par défaut, c'est le fichier `utmp` du système qui est utilisé, mais cette fonction permet d'en indiquer un autre.

```
| int utmpname (const char * fichier);
```

Pour savoir où se trouve le fichier à lire, on peut employer l'une des macros `_PATH_UTMP` et `_PATH_WTMP`, qui se transforment en chaînes de caractères représentant le chemin du fichier `utmp` ou `wtmp`.

Dans l'exemple suivant, nous fournissons les noms de fichiers sur la ligne de commande. Le programme emploie la routine `affiche_utmp()` de l'exemple précédent, que nous ne réécrivons pas ici.

```
exemple-utmpname.c :
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <utmp.h>

void affiche_utmp (struct utmp * utmp)
{
    [...]
}

int main (int argc, char * argv[])
{
    struct utmp * utmp;
    int i;
    for (i = 1; i < argc; i ++) {
        if (utmpname(argv[i]) != 0)
            continue;
        while ((utmp = getutent()) != NULL)
            affiche_utmp(utmp);
    }
    return EXIT_SUCCESS;
}
```

Ce programme offre le même genre de résultat que l'utilitaire `last`.

```
$ ./exemple-utmpname /var/log/wtmp
06/21/10 14:25:29 : Boot
06/21/10 14:25:29 : Run-level
06/21/10 14:53:49 : Login process, PID = 1483, TTY = tty2
06/21/10 14:53:49 : Login process, PID = 1487, TTY = tty4
06/21/10 14:53:49 : Login process, PID = 1485, TTY = tty3
[...]
08/21/10 17:07:28 : User process, PID = 5159, TTY = pts/2, cpb
08/21/10 17:09:11 : User process, PID = 5159, TTY = pts/2, cpb
08/21/10 18:11:14 : User process, PID = 5159, TTY = pts/1, cpb
$
```

Notre routine d'affichage des enregistrements n'est peut-être pas très adaptée à la présentation de traces de connexions, car nous devrions plutôt éliminer les lignes contenant les enregistrements `INIT_PROCESS` ou `LOGIN_PROCESS` qui correspondent aux `getty` et `login`, et conserver les `DEAD_PROCESS` qui représentent la déconnexion d'un utilisateur.

Pour ajouter un enregistrement au fichier `wtmp`, on n'utilisera pas `pututline()`, car elle écrase les entrées inutilisées, mais plutôt `updwtmp()`, qui écrit simplement la nouvelle ligne de données à la fin du fichier.

```
| void updwtmp (const char * fichier, const struct utmp * utmp);
```

Journal syslog

Une application, même si elle fonctionne en arrière-plan, doit pouvoir communiquer des informations de temps à autre. L'écriture sur `stdout` ou `stderr` n'est pas toujours possible, notamment pour les logiciels fonctionnant sous forme de démons. Pour pouvoir transmettre des indications sur son état, un programme peut alors employer plusieurs techniques, comme l'émission d'un courrier électronique à destination de l'utilisateur qui l'a lancé, ou proposer une connexion réseau (par `telnet` ou même par navigateur web) et afficher ainsi sa configuration.

Une autre méthode, plus souple, consiste à utiliser le démon `syslogd`. Celui-ci est lancé au démarrage par les scripts d'initialisation du système et il reste en attente de messages. Les fonctions de bibliothèque `openlog()`, `syslog()` et `closelog()` permettent de lui transmettre des données. En fonction de la configuration du démon (via `/etc/syslog.conf`) et de la gravité du message, celui-ci peut être stocké dans un fichier, envoyé dans un tube nommé vers un autre programme (en général `mail`), affiché sur la console et sur les écrans des utilisateurs, ou même transmis par le réseau à destination d'un autre démon `syslogd` fonctionnant sur une machine de supervision.

Les distributions Linux récentes utilisent plutôt le démon `rsyslogd` (*Reliable Syslog Daemon*) plus fiable que le `syslogd` initial, son fichier de configuration est naturellement `/etc/rsyslog.conf`.

Les trois routines d'accès sont disponibles dans `<syslog.h>` :

```
void openlog(char * identificateur, int option, int type);
void syslog (int urgence, char * format, ...);
void closelog (void);
```

En Python, il s'agit de `syslog.openlog()`, `syslog.syslog()` et `syslog.closelog()`. La fonction `openlog()` permet d'ouvrir une session de journalisation. Le premier argument est un identificateur qui sera ajouté à chaque message pour le distinguer. En général, on choisit le nom du programme.

Le second argument peut contenir une ou plusieurs des constantes symboliques suivantes, liées par un *OU* binaire :

Nom	Signification
LOG_CONS	Écrire les messages sur la console système si une erreur se produit lors de leur traitement.
LOG_NDELAY	Ouvrir tout de suite la communication avec le démon <code>syslogd</code> , sans attendre l'arrivée du premier message.
LOG_PERROR	Envoyer sur la sortie d'erreur une copie des messages.
LOG_PID	Ajouter le PID du processus appelant dans chaque message.

L'utilisation de `LOG_PERROR` permet de simplifier la mise au point d'un programme qu'on fera fonctionner ensuite sous forme de démon. De même, `LOG_PID` est très utile car cette information est souvent indispensable, et on évite ainsi de devoir l'inscrire explicitement dans chaque message.

Enfin, le troisième argument de `openlog()` est une valeur numérique servant à classer le programme dans une catégorie de logiciels. Cela permet de filtrer les messages, par exemple en redirigeant tous ceux qui concernent le courrier vers l'utilisateur `postmaster`. Les constantes suivantes sont déclarées dans `<syslog.h>` :

Nom	Utilisation
LOG_KERN	Message provenant du noyau.
LOG_USER	Message provenant d'une application utilisateur.
LOG_MAIL	Système de gestion du courrier électronique.
LOG_DAEMON	Ensemble des démons du système.
LOG_AUTH	Système d'authentification des utilisateurs.

Nom	Utilisation
LOG_SYSLOG	Démon <code>syslogd</code> lui-même.
LOG_LPR	Système de gestion des impressions.
LOG_NEWS	Système des <i>news</i> Usenet.
LOG_UUCP	Message provenant d'un démon <code>uucp</code> .
LOG_CRON	Exécution différée par <code>cron</code> .
LOG_AUTHPRIV	Système d'authentification personnel.
LOG_FTP	Démon <code>ftpd</code> .
LOG_LOCAL0 ...LOG_LOCAL7	Message provenant d'une application spécifique du système.

On emploiera généralement `LOG_USER` ou `d` à `LOG_LOCAL7` pour les logiciels personnels. En fait, `openlog()` ne fait qu'initialiser des champs qui seront utilisés ensuite lors de la transmission effective des messages. Pour en envoyer un, on emploie `syslog()`. Cette fonction prend un premier argument qui correspond à l'urgence du message. On définit les niveaux de priorité suivants :

Nom	Signification
LOG_EMERG	Le système concerné n'est plus utilisable.
LOG_ALERT	L'intervention immédiate d'un administrateur est indispensable ou le système va devenir inutilisable.
LOG_CRIT	Des conditions critiques se présentent, pouvant nécessiter une intervention.
LOG_ERR	Des erreurs ont été détectées.
LOG_WARNING	Des conditions rares ou inattendues ont été observées.
LOG_NOTICE	Information importante, mais fonctionnement normal.
LOG_INFO	Information sans importance renseignant sur l'état du système.
LOG_DEBUG	Données utiles pour le débogage, à ignorer sinon.

On notera que le terme « système » dans ce tableau fait référence à l'application invoquant `syslog()` et éventuellement à ses interlocuteurs, mais qu'il ne regroupe pas l'ensemble des fonctionnalités de la machine comme on l'entend habituellement.

Le format se trouvant en second argument de `syslog()` ainsi que les autres arguments éventuels correspondent exactement à ceux de `printf()`. La seule différence est l'existence d'un code `%m` qui est remplacé par la chaîne `strerror(errno)`.

Si `openlog()` n'a pas encore été appelée, `syslog()` l'invoque automatiquement avec les arguments :

- `identificateur = NULL` ;

- option = 0 ;
- type = LOG_USER.

La routine `closelog()` ferme la session de communication avec `syslogd`. Cette fonction n'est pas indispensable, la session étant terminée automatiquement à la fin du programme.

Pour des raisons de sécurité – comme nous l'avons déjà mentionné avec `printf()` – il faut toujours utiliser `syslog(urgence, "%s", message)` et jamais `syslog(urgence, message)` si le message est susceptible d'être fourni par l'utilisateur. Ceci concerne surtout les démons et les applications réseau.

Voici un exemple simple utilisant `syslog()`.

```
exemple-syslog.c :
#include <stdio.h>
#include <stdlib.h>
#include <syslog.h>

int main (int argc, char * argv[])
{
    int    i;

    openlog(argv[0], LOG_PID, LOG_USER);
    for (i = 1; i < argc; i++)
        syslog(LOG_INFO, argv[i]);
    closelog();
    return EXIT_SUCCESS;
}
```

À l'exécution de ce programme, les messages sont redirigés, sur notre système, vers le fichier `/var/log/messages`, lisible uniquement par `root`.

```
$ ./exemple-syslog "Premier message"
$ ./exemple-syslog "Second message"
$ su
Mot de passe :
# tail /var/log/messages
[...]
Aug 22 13:52:47 tr-a-1 ./exemple-syslog[27722]: Premier message
Aug 22 13:52:51 tr-a-1 ./exemple-syslog[27725]: Second message
#
```

L'utilisation de `syslog()` est fortement recommandée pour le développement d'applications fonctionnant essentiellement en arrière-plan, car cette routine laisse à l'administrateur du système le choix du comportement vis-à-vis des messages de diagnostic et d'erreur. Cette souplesse est très appréciable car on peut ainsi plus facile-

ment décider d'éliminer tous les messages peu importants, de les stocker dans un fichier ou de les rediriger vers une console réservée à cet usage.

Conclusion

Nous avons vu dans ce chapitre les méthodes pour accéder à de nombreuses informations concernant le système. Une large partie d'entre elles sont suffisamment portables pour être disponibles sur l'essentiel des systèmes Unix.

Les fonctions présentées ici sont certainement suffisantes pour réaliser la plus grande partie des tâches d'administration du système. On trouvera plus de détails sur ces travaux ainsi que sur les différents outils d'aide à l'administrateur dans [Frisch 2003] *Les bases de l'administration système*.

Internationalisation

Nous avons déjà signalé dans le chapitre 16, à propos des caractères larges, que les développeurs se soucient de plus en plus des possibilités d'internationalisation de leurs logiciels. En dehors des applications « maison », destinées à un usage unique et très spécifique, la plupart des programmes peuvent voir subitement leur portée étendue à une échelle internationale grâce à Internet, par exemple.

En raison du volume de discussion qu'engendrent les problèmes d'internationalisation, un sigle a même été créé, *i18n*, signifiant « i suivi de 18 lettres puis d'un n », afin d'éviter les guerres de clans entre les partisans du mot *internationalisation* et ceux – américains – du terme *internationalization*.

Après avoir présenté les principes de l'internationalisation, nous examinerons des méthodes permettant d'offrir des messages d'interface dans la langue de l'utilisateur. Malgré tout, l'internationalisation d'un logiciel ne consiste pas uniquement en la traduction des messages de l'interface utilisateur, même s'il s'agit probablement du point le plus important dans la plupart des cas. En fait, la langue n'est qu'une partie des conventions culturelles propres à un peuple, et l'ordre de présentation des éléments d'une date est par exemple un autre aspect de l'internationalisation d'une application.

Pour permettre la transposition d'un système vers d'autres pays, la bibliothèque C autorise l'utilisateur à configurer ces éléments culturels et linguistiques à son gré. L'adaptation aux désirs de l'utilisateur se fait par le biais de la localisation¹. Nous verrons dans ce chapitre comment employer l'ensemble des éléments configurés dans la localisation.

1. Le mot anglais *locale* est traduit différemment suivant les auteurs. Je conserverai le terme localisation, qui est le plus répandu même s'il n'est pas très esthétique.

Principe

La localisation est un ensemble de règles, réparties par catégories, que la bibliothèque C applique dans les routines qui doivent réagir différemment suivant les choix de l'utilisateur. Par exemple, il existe une catégorie dans laquelle on indique le caractère qu'on préfère utiliser pour séparer la partie entière d'un nombre réel de ses décimales. Dans la localisation anglo-saxonne, il s'agit du point, alors que dans la localisation française on préfère la virgule. Certaines routines d'affichage comme `printf()` prennent cette information en considération pour présenter leurs résultats.

La plupart des utilitaires du système sont sensibles à la localisation, du moins en ce qui concerne la traduction des messages. Nous pourrions donc observer directement quelques effets des modifications apportées. Pour configurer sa localisation, un utilisateur remplit des variables d'environnement qui seront consultées par les applications lancées par la suite.

Comme nous l'avons précisé dans le chapitre 4, les variables d'environnement ne concernent que le processus qui les configure et ses descendants. Si l'utilisateur définit sa localisation dans une session shell, toutes les applications lancées ensuite grâce à ce shell en bénéficieront, mais pas les logiciels démarrés depuis un autre shell ou depuis un environnement graphique X-Window. L'administrateur du système configure souvent une localisation par défaut dans les fichiers d'initialisation communs à tous les utilisateurs. Il s'agit généralement de la localisation correspondant à l'implantation physique de la station. Chaque utilisateur peut toutefois modifier cette configuration dans ses propres scripts de connexion afin de l'adapter à ses préférences.

Pour bénéficier automatiquement de l'internationalisation des routines de la bibliothèque C, il suffit d'insérer deux lignes dans une application :

- Le fichier d'en-tête `<locale.h>` doit être inclus en début de module.
- L'instruction `setlocale(LC_ALL, "")` doit être appelée en début de programme.
En Python, on invoquera `locale.setlocale(locale.LC_ALL, "")`.

Rien qu'avec ces deux lignes, un programme est capable de s'adapter correctement à la plupart des conventions culturelles de l'utilisateur, hormis la langue bien entendu. Pour obtenir une internationalisation au niveau du langage, il faut stocker les messages et leurs traductions dans des catalogues, comme nous le verrons plus loin.

Catégories de localisations disponibles

Nous avons indiqué que la localisation se faisait par l'intermédiaire de plusieurs catégories différentes. Ceci permet à l'utilisateur de configurer indépendamment plusieurs aspects de l'interface de l'application. Par exemple, il est possible de demander

que les messages soient affichés en français pour faciliter la lecture, mais que les dates et les valeurs numériques soient affichées en respectant les normes américaines, afin de récupérer directement ces données pour les transmettre à des collègues étrangers.

Chaque catégorie est représentée par une variable d'environnement et par une constante symbolique du même nom, disponible au sein de l'application. Les catégories sont les suivantes :

Nom	Signification
LC_ALL	Cette catégorie écrase toutes les autres. On l'utilise pour donner une valeur immédiate de localisation à toutes les catégories. En réalité, l'emploi de cette variable d'environnement comme configuration est un peu abusif, elle ne devrait être utilisée qu'en tant que constante symbolique pour consulter la localisation.
LC_COLLATE	Dans cette catégorie se trouvent les règles employées par les routines devant ordonner des chaînes de caractères, comme <code>strcoll()</code> , que nous avons étudiée dans le chapitre 16.
LC_CTYPE	Cette catégorie concerne les routines de classification des caractères comme <code>isalpha()</code> , ainsi que celles de conversion comme <code>tolower()</code> . Elle sert également à déterminer les règles employées pour les conversions entre caractères larges et séquences multioctets, comme nous l'avons indiqué dans le chapitre 16.
LC_MESSAGES	La traduction des messages réclamée par la catégorie LC_MESSAGES concerne l'interface avec l'utilisateur. Il ne s'agit pas nécessairement de la même langue que celle qui est employée dans les données elles-mêmes ni surtout du même jeu de caractères.
LC_MONETARY	Cette catégorie configure la manière de représenter des valeurs monétaires, tant du point de vue du symbole évoquant la monnaie que pour la position de ce symbole, et la séparation entre partie entière et décimale.
LC_NUMERIC	Avec cette catégorie, on indique les coutumes de représentation des valeurs numériques, comme la séparation des chiffres par milliers ou le symbole utilisé comme séparateur décimal.
LANG	La variable LANG sert à définir la langue utilisée pour l'ensemble des messages et des textes, mais c'est surtout une valeur par défaut, qui permet de configurer toutes les catégories qui ne sont pas remplies explicitement.

La localisation est donc l'ensemble de toutes ces catégories, représentant chacune des règles usuelles appliquées à l'emplacement où se trouve l'utilisateur. Pour remplir une catégorie, on emploie une chaîne de caractères indiquant en premier lieu la langue choisie. Il s'agit de deux caractères minuscules, dont voici quelques exemples :

Nom	Langue
da	danois
de	allemand
el	grec

Nom	Langue
en	anglais
es	espagnol
fi	finnois

Nom	Langue
fr	français
it	italien
nl	hollandais

Nom	Langue
pt	portugais
sv	suédois

Le nom de la localisation est ensuite précisé par un emplacement géographique si plusieurs pays emploient la même langue, mais avec des différences de coutumes dans d'autres catégories. Par exemple, si on demande l'affichage de la valeur monétaire 2000 avec une localisation francophone de France (fr_FR), on obtient

| 2 000,00 EUR

alors que l'affichage avec la localisation francophone du Canada (fr_CA) donne

| 2 000,00\$

La même valeur en anglais du Canada (en_CA) est affichée :

| \$2,000.00

En anglais de Grande-Bretagne (en_GB), le résultat est :

| £2,000.00

Alors qu'en anglais commun au Royaume-Uni (en_UK), on voit :

| 2000.00

L'emplacement géographique est donc précisé avec un code de deux lettres majuscules. Voici quelques pays européens :

Pays	Code
Allemagne	DE
Angleterre	GB
Autriche	AT
Belgique	BE
Danemark	DK
Grèce	EL
Espagne	ES
Finlande	FI

Pays	Code
France	FR
Irlande	IR
Italie	IT
Luxembourg	LU
Hollande	NL
Portugal	PT
Suède	SV

Une source fréquente d'erreur lors de la configuration de la localisation est l'inversion entre le pays et la langue². Ainsi la disposition des majuscules dans "FR_fr" paraît plus naturelle que dans "fr_FR", mais c'est pourtant ce second cas seulement qui fonctionne, le premier étant inconnu (donc ignoré et équivalent à la localisation américaine par défaut). Dans certaines situations, on peut ajouter le nom d'un jeu de caractères à la suite, mais c'est plutôt rare.

Attention, les noms des localisations proprement dites peuvent varier suivant les systèmes d'exploitation. Nous décrivons ici leur aspect avec la *Glibc*, mais il ne faut pas faire de suppositions hâtives quant au contenu des variables sur d'autres systèmes.

Le troisième élément de configuration qui peut intervenir est l'encodage à utiliser pour les caractères. On l'ajoute en fin de chaîne, précédé d'un point, ainsi : « fr_FR.UTF-8 ».

Voyons donc les effets de la localisation sur quelques utilitaires, en commençant par les messages d'erreur de `/bin/ls` :

```
$ unset LANG
$ unset LC_ALL
$ export LANG=fr_FR
$ ls inexistant
ls: impossible d'acc?der ? inexistant: Aucun fichier ou dossier de ce type
$ export LANG=fr_FR.UTF-8
$ ls inexistant
ls: impossible d'accéder à inexistant: Aucun fichier ou dossier de ce type
$ export LANG=en_EN.UTF-8
$ ls inexistant
ls: cannot access inexistant: No such file or directory
$
```

Suivant la configuration de la localisation du système, l'utilisation de `fr_FR` peut suffire ou il peut être nécessaire d'utiliser `fr_FR.UTF-8`.

Nous allons observer à présent les répercussions de la localisation sur l'affichage de la date. On vérifiera que la catégorie `LC_ALL` a bien préséance sur `LANG`. Le format `"%x"` ordonne à l'utilitaire `/bin/date` d'afficher la représentation locale de la date. Le test a lieu le 22 août.

2. Disons la deuxième source d'erreur la plus fréquente, la première étant l'oubli pur et simple de l'appel à `setlocale(LC_ALL, "")` en début de programme...

```
$ env -i /bin/sh
$ export LANG=en_EN.utf-8
$ date +%X"
08/22/10
$ export LC_ALL=fr_FR.UTF-8
$ date +%X"
22/08/2010
$ exit
```

Pour connaître les règles applicables dans une localisation donnée, la bibliothèque C dispose de fichiers de configuration placés en général dans les répertoires `/usr/locale/` ou `/usr/share/locale/`. On y trouve normalement un nombre important de sous-répertoires, chacun représentant une localisation connue par le système. Tout cela peut varier légèrement en fonction de la distribution Linux choisie. Les fichiers employés par la bibliothèque C sont dans un format binaire. Pour modifier une localisation existante – ou en créer une nouvelle –, il faut installer les sources des localisations. Elles sont généralement établies avec l'ensemble des sources de la *Glibc*. On peut invoquer `localedef --help` pour savoir où les sources des localisations sont placées (par exemple `/usr/share/i18n/locale/`). Ce répertoire regroupe un ensemble de fichiers décrivant toutes les localisations connues par la bibliothèque. Ces fichiers sont tout à fait lisibles, leur format est assez intuitif. L'utilitaire `localedef` sert à compiler l'un de ces fichiers en créant les répertoires système et les fichiers binaires nécessaires pour que la nouvelle localisation soit reconnue par la bibliothèque C. Cette tâche est normalement réservée à l'administrateur du système.

Traduction de messages

Le fait de proposer une interface dans la langue de l'utilisateur est probablement le premier souhait en ce qui concerne l'internationalisation d'un programme. La traduction automatique des messages d'une langue à l'autre n'est pas encore possible, aussi un programme doit-il employer pour ses messages un stock de libellés, et afficher ceux qui correspondent à la langue de l'utilisateur.

L'ensemble de tous les messages et leurs traductions peuvent être directement insérés dans le code source de l'application. La sélection du libellé correspondant à la traduction d'un message dans la langue désirée se fera en fonction d'un paramétrage interne (choix dans un menu) ou externe (variable d'environnement). Cette méthode est parfois employée lorsqu'un logiciel doit être distribué sous forme binaire sur des systèmes d'exploitation totalement différents, n'offrant pas toujours des possibilités d'internationalisation. Si cette approche se justifie donc dans certains cas, elle est

quand même fortement déconseillée, car l'ajout du support d'une nouvelle langue ou la correction d'une faute de traduction d'un message nécessitent la recompilation de l'application.

On préfère donc regrouper les libellés dans un fichier externe, qu'on peut échanger au gré de la localisation.

Il y a un avantage supplémentaire au regroupement de tous les messages dans un unique fichier, même sans tenir compte des possibilités de traduction. Cela permet en effet d'avoir sous les yeux tous les libellés d'interface du logiciel et de s'assurer immédiatement de l'homogénéité de l'ensemble. Lorsqu'il y a plusieurs possibilités pour nommer un objet manipulé par le programme, ou s'il faut choisir entre traduire un nom ou laisser le terme original qu'on pourra retrouver dans d'autres logiciels, on s'assure, en voyant tous les messages côte à côte, que les mêmes décisions ont été prises pour toute l'interface.

La bibliothèque *Glibc* offre deux méthodes différentes pour gérer un ensemble de messages externes, stockés dans des fichiers qui pourront être mis à jour sans que l'application ait besoin d'être recompilée. Ces deux dernières nécessitent d'abord que la bibliothèque puisse trouver le fichier de messages lui appartenant, adapté à la langue choisie. Les différences apparaissent ensuite dans la manière d'accéder aux libellés contenus dans le fichier proprement dit.

Catalogues de messages gérés par `catgets()`

Ce premier mécanisme est plus ancien et plus répandu que le second. Il rend aussi le travail du développeur sensiblement plus compliqué. Les fonctions `catopen()`, `catgets()` et `catclose()` que nous allons examiner sont définies dans les spécifications SUSv4. Chaque message du logiciel doit être associé à une clé numérique unique. Ceci représente le point le plus complexe, principalement lorsque le développement d'une application se fait de manière répartie avec plusieurs équipes indépendantes.

Tout d'abord, le programme doit ouvrir le catalogue de messages. Ceci s'effectue avec la fonction `catopen()`, déclarée ainsi dans `<n1_types.h>` :

```
| n1_catdcatopen (const char * nom, int attribut);
```

Cette routine essaye d'ouvrir le catalogue dont le nom est passé en premier argument. Si ce nom contient un caractère « / », on considère qu'il s'agit d'un chemin d'accès entier. Sinon, on suppose qu'il s'agit du nom d'un catalogue qui est alors recherché dans divers répertoires suivant la configuration des variables d'environnement. Le fait d'employer un chemin figé ne se justifie que lors de la mise au point du programme, car le principe même de l'internationalisation consiste à laisser l'utilisateur configurer le répertoire correspondant à sa localisation.

La recherche se fait en employant la variable d'environnement `NLSPATH`. Celle-ci contient un ou plusieurs chemins d'accès séparés par des deux-points. Un chemin peut comprendre des codes spéciaux, qui seront remplacés automatiquement lors de la tentative d'ouverture :

Code	Signification
<code>%N</code>	Nom du catalogue tel qu'il a été transmis en premier argument de <code>catopen()</code> .
<code>%L</code>	Localisation configurée pour les messages d'interface.
<code>%l</code>	Langage configuré pour les messages d'interface, sans préciser l'emplacement ni le jeu de caractères. Par exemple <code>fr</code> dans la localisation <code>fr_BE.ISO-8859-1</code> .
<code>%t</code>	Emplacement configuré pour les messages d'interface, sans préciser la langue ni le jeu de caractères. <code>BE</code> dans la localisation <code>fr_BE.ISO-8859-1</code> .
<code>%C</code>	Jeu de caractères configuré pour les messages d'interface, sans préciser la langue ni l'emplacement géographique. <code>ISO-8859-1</code> dans la localisation <code>fr_BE.ISO-8859-1</code> .
<code>%%</code>	Le caractère <code>%</code> lui-même.

L'attribut indiqué en seconde position lors de l'appel de `catopen()` permet de préciser les variables prises en compte pour la localisation. Si cet attribut est nul, la fonction n'emploie que la variable `LANG`. Sinon, si l'attribut prend la valeur `NL_CAT_LOCALE`, la bibliothèque recherche la localisation successivement dans les variables `LC_ALL`, `LC_MESSAGES` et `LANG`. On emploiera donc toujours `NL_CAT_LOCALE` en second argument de `catopen()`.

Si la variable `NLSPATH` n'est pas définie, la fonction emploie une valeur par défaut, configurée lors de la compilation de la bibliothèque `C`, qui correspond généralement à :

```
/usr/share/locale/%L/%N:/usr/share/locale/%L/LC_MESSAGES/%N:/usr/share/
locale/%l/%N:/usr/share/locale/%l/LC_MESSAGES/%N
```

Cela signifie que lors d'une tentative d'ouverture du fichier de catalogue `msg`, dans la localisation `fr_FR`, le système recherche le fichier successivement dans :

```
/usr/share/locale/fr_FR/msg
/usr/share/locale/fr_FR/LC_MESSAGES/msg
/usr/share/locale/fr /msg
/usr/share/locale/fr/LC_MESSAGES/msg
```

Le descripteur de catalogue est du type opaque `nl_catd` et sera employé dans les fonctions `catgets()` et `catclose()`. Si aucun fichier n'est disponible, `catopen()` renvoie `(nl_catd) -1`.

Une fois que le catalogue de messages est ouvert, on peut accéder à son contenu à l'aide de la fonction `catgets()`, déclarée ainsi :

```
char * catgets (nl_catd catalogue,  
                int ensemble, int message,  
                const char * original);
```

Cette fonction recherche dans le catalogue décrit par son premier argument – obligatoirement obtenu grâce à `catopen()` – le message appartenant à l'ensemble indiqué en deuxième position, et dont le numéro est passé en troisième argument. Si le message n'est pas disponible, la chaîne transmise en dernier argument est renvoyée.

Cette chaîne est donc rédigée dans une langue par défaut, la plupart du temps en anglais. L'organisation du catalogue sous forme d'ensembles de messages permet de découper l'application en ensembles fonctionnels, en attribuant un numéro général à chaque équipe de développement afin qu'elle maintienne elle-même la numérotation dans son propre ensemble. Les numéros peuvent être choisis arbitrairement, ils n'ont pas besoin de se suivre. En revanche, la paire (ensemble, numéro) ne peut désigner qu'un seul message dans le catalogue.

Ceci complique sensiblement le travail des programmeurs, qui doivent gérer une nomenclature supplémentaire dans leurs applications.

Pour refermer un catalogue de messages qui n'est plus utilisé, on emploie `catclose()`, déclarée ainsi :

```
int catclose (nl_catd catalogue);
```

Cette fonction renvoie 0 si tout s'est bien passé et -1 si une erreur s'est produite (généralement, c'est le descripteur de catalogue qui est erroné).

Les catalogues sont créés à l'aide de l'utilitaire `/usr/bin/gencat`. Celui-ci prend en entrée un fichier de texte contenant les chaînes de caractères et fabrique un fichier binaire permettant l'accès rapide avec `catgets()`. Le détail de ce fichier binaire ne concerne que la bibliothèque C. Le format des fichiers lus par `gencat` est décrit en détail dans la documentation Gnu. Voyons-en les principales caractéristiques :

- Les lignes blanches ou commençant par un symbole \$ suivi d'un caractère blanc sont ignorées. On peut introduire ainsi des commentaires.
- Une ligne ayant la forme `$set <identifiant>` indique le début d'un ensemble de messages.
- Une ligne ayant la forme `<identifiant> <chaîne de caractères>` précise un message appartenant à l'ensemble en cours.

L'identifiant de l'ensemble ou celui du message peuvent être signalés sous forme numérique, mais également sous forme de mot-clé alphanumérique. Dans ce cas, gencat les remplacera par des numéros adéquats, et créera un fichier où les identifiants seront définis sous forme de constantes symboliques. On pourra donc inclure ce fichier en début de programme. Pour créer les constantes symboliques, gencat ajoute le suffixe `Set` à la fin des noms d'ensembles et insère le nom de l'ensemble devant les chaînes. Le programme suivant va permettre de mieux comprendre ce principe.

```
exemple-catgets.c :
#include <locale.h>
#include <nl_types.h>
#include <stdio.h>
#include <stdlib.h>
#include "exemple-catgets.h"

int main (void)
{
    nl_catd catalogue;
    char * chaine;

    setlocale(LC_ALL, "");
    if ((catalogue= catopen("msg-catgets",NL_CAT_LOCALE))
        == (nl_catd) -1) {
        perror("catopen");
    }
    chaine = catgets(catalogue, premier_Set, premier_chaine_1,
        "This is the first string in the first set");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, premier_Set, premier_chaine_2,
        "and here is the second string in the first set.");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, second_Set, second_chaine_1,
        "Now let's have a look at the 1st string in 2nd set,");
    fprintf(stdout, "%s \n", chaine);
    chaine = catgets(catalogue, second_Set, second_chaine_2,
        "and finally the second string in the second set.");
    fprintf(stdout, "%s \n", chaine);
    catclose(catalogue);
    return EXIT_SUCCESS;
}
```

Nous construisons aussi le fichier de messages traduits en français ainsi :

```
exemple-catgets.msg :
$ Voici le catalogue de messages pour
$ l'exemple_catgets.
```

```
$set premier_  
chaîne_1 Ceci est la première chaîne du premier ensemble,  
chaîne_2 et voici la seconde chaîne du premier ensemble.  
  
$ Nous pouvons insérer des commentaires  
$ qui seront ignorés  
  
$set second_  
chaîne_1 Maintenant voyons la 1ère chaîne du 2ème ensemble,  
chaîne_2 et finalement la seconde chaîne du second ensemble.
```

À présent, nous compilons le catalogue de messages, en demandant à gencat de nous fournir aussi un fichier de définition des constantes :

```
$ gencat -o msg-catgets -H exemple-catgets.h exemple-catgets.msg  
$ cat exemple-catgets.h  
#define second_Set 0x2          /* exemple-catgets.msg:11 */  
#define second_chaine_1 0x1     /* exemple-catgets.msg:12 */  
#define second_chaine_2 0x2     /* exemple-catgets.msg:13 */  
  
#define premier_Set 0x1         /* exemple-catgets.msg:4 */  
#define premier_chaine_1 0x1    /* exemple-catgets.msg:5 */  
#define premier_chaine_2 0x2    /* exemple-catgets.msg:6 */  
$ ls -l msg-catgets  
-rw-rw-r-- 1 cpb cpb 308 22 août 16:49 msg-catgets
```

Nous pouvons maintenant compiler l'application et installer le fichier msg-catgets dans le répertoire /usr/share/locale/fr/LC_MESSAGES/ afin qu'il soit trouvé par la bibliothèque C dans la localisation correspondante.

```
$ su  
Mot de passe :  
# cp msg-catgets /usr/share/locale/fr/LC_MESSAGES/  
# exit  
exit  
$
```

Essayons à présent de faire fonctionner notre application internationalisée :

```
$ env -i /bin/sh
$ ./exemple-catgets
catopen: No such file or directory
This is the first string in the first set
and here is the second string in the first set.
Now let's have a look at the 1st string in 2nd set,
and finally the second string in the second set.
$ export LANG=fr_FR.UTF-8
$ ./exemple-catgets
Ceci est la premiere chaine du premier ensemble,
et voici la seconde chaine du premier ensemble.
Maintenant voyons la 1ere chaine du 2eme ensemble,
et finalement la seconde chaine du second ensemble.
$ exit
```

Nous voyons que ce mécanisme fonctionne très bien mais qu'il est très lourd à mettre en œuvre dans le codage du programme, chaque manipulation de chaîne devant faire l'objet de vérifications dans la nomenclature pour connaître le nom ou le numéro de l'ensemble et celui du message.

Il existe pourtant une alternative plus simple : les fonctionnalités *GetText* du projet Gnu.

Catalogues de messages Gnu GetText

Le principe des catalogues de messages *GetText* est d'employer la chaîne originale comme clé d'accès dans le catalogue de traduction. Ainsi, il n'y a plus besoin de manipuler des identificateurs, puisque la chaîne se suffit à elle-même.

Le projet Gnu *GetText* est relativement ambitieux puisqu'il contient de nombreux outils pour aider à internationaliser des programmes qui n'étaient pas conçus pour l'être à l'origine. Nous allons simplement présenter ici les fonctionnalités qui concernent le programmeur désireux d'employer *GetText* comme une alternative plus pratique à *catgets*().

L'ensemble de la traduction repose essentiellement sur l'emploi d'une unique fonction, nommée *gettext*(), et déclarée dans `<libintl.h>` :

```
| char * gettext (const char * origine);
```

En Python, une API identique est proposée par le module `gettext`. On peut donc utiliser `gettext.gettext()` ainsi que les fonctions `gettext.textdomain()` et `gettext.bindtextdomain()` que nous verrons plus loin.

L'interface de cette routine se rapproche au maximum de celle que pourrait proposer – que proposera peut-être un jour – un traducteur automatique. On lui transmet la chaîne originale et elle renvoie un pointeur sur une zone de mémoire statique contenant la traduction adaptée à la localisation de l'utilisateur. Si la traduction est impossible ou si la localisation est la même que celle du concepteur du programme, le pointeur renvoyé est identique à celui de la chaîne transmise.

On peut donc écrire des choses comme :

```
fprintf(stdout, gettext("Vitesse : %d bits / sec \n"), vitesse);

fprintf(stdout, gettext("Parite = %s \n"),
        parite==P_PAIRE ? gettext("paire") : gettext("impaire"));
```

On peut traduire aussi bien des chaînes de caractères correspondant à des messages que des formats pour `printf()` par exemple.

On devine que l'éventail des possibilités offertes par une telle interface est assez large. En effet, dans l'implémentation actuelle, la traduction est simplement recherchée dans un fichier, mais il est possible d'imaginer que la fonction `gettext()` peut évoluer pour interroger – par réseau – une énorme base de données ou un logiciel de traduction automatique. Dans le cas d'un portage sur un système ne supportant pas ce mécanisme, on définit simplement une macro

```
#define gettext(X)    (X)
```

en tête de programme pour annuler la tentative de traduction.

Pour que la bibliothèque puisse faire correspondre une traduction à un message, il faut lui indiquer le catalogue de messages à employer. Ceci s'effectue à l'aide de deux fonctions, `textdomain()` et `bindtextdomain()`. La bibliothèque *GetText* introduit en effet le concept de domaine, qui permet de scinder la base de textes en plusieurs fichiers. En général, une application n'utilise qu'un seul domaine, qu'elle configure dès le démarrage du programme. Ceci s'effectue avec `textdomain()` :

```
char * textdomain (const char * domaine);
```

Cette fonction signale à la bibliothèque que les messages ultérieurs seront recherchés dans le domaine dont on passe le nom. Ce nom sera utilisé pour déterminer le fichier contenant les libellés des messages. La fonction `bindtextdomain()` permet d'indi-

quer le nom du répertoire dans lequel se trouve l'arborescence des fichiers correspondant à un domaine particulier :

```
char * bindtextdomain (const char * domaine,  
                        const char * repertoire);
```

En fait, le fichier de traduction est recherché avec le chemin d'accès composé ainsi :/ repertoire_de_bindtextdomain()/localisation/LC_MESSAGES/domaine.mo

Le répertoire de départ a été spécifié avec `bindtextdomain()`. Il s'agit en général de /usr/share/locale. La localisation est extraite successivement des variables LANGUAGE, LC_ALL, LC_MESSAGES et LANG. Le nom du fichier final est construit avec le nom de domaine et le suffixe .mo signifiant *Machine Object*. Ce fichier est binaire, compilé par l'utilitaire /usr/bin/msgfmt à partir du fichier de texte avec le suffixe .po signifiant *Portable Object*.

Le projet *GetText* incorpore des macros pour permettre l'édition facile du fichier .po, mais son format est tellement simple que nous pourrions le manipuler directement.

Nous allons utiliser le même principe qu'avec exemple-catgets, en prenant cette fois-ci les fonctionnalités *GetText*.

```
exemple-gettext.c :  
#include <libintl.h>  
#include <locale.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main (void)  
{  
    setlocale(LC_ALL, "");  
  
    textdomain("exemple-gettext");  
    bindtextdomain("exemple-gettext", "/usr/share/locale");  
    printf(gettext("This is the first string in the first set\n"));  
    printf(  
        gettext("and here is the second string in the first set.\n"));  
    printf(  
        gettext("Now let's have a look at the 1st string in 2nd set,\n"));  
    printf(  
        gettext("and finally the second string in the second set.\n"));  
    return EXIT_SUCCESS;  
}
```

On voit que l'impact sur le programme est beaucoup plus limité qu'avec `catgets()`. L'application peut être compilée et utilisée immédiatement sans avoir à définir des constantes symboliques. Seules deux lignes ont été ajoutées en début de programme.

Quant aux appels `gettext()`, ils pourraient être rendus encore plus discrets à l'aide d'une macro.

Nous créons un fichier `.po` de traduction en français :

```
exemple-gettext.po :
msgid "This is the first string in the first set\n"
msgstr "Ceci est la premiere chaîne du premier ensemble\n"

msgid "and here is the second string in the first set.\n"
msgstr "et voici la seconde chaîne du premier ensemble.\n"

msgid "Now let's have a look at the 1st string in 2nd set,\n"
msgstr "A present regardons la 1ere chaîne du 2eme ensemble,\n"

msgid "and finally the second string in the second set.\n"
msgstr "et finalement la seconde chaîne du second ensemble.\n"
```

Ce fichier est construit avec des séquences successives utilisant le mot-clé `msgid` pour indiquer la chaîne originale et `msgstr` pour sa traduction.

Remarquons, que nous pouvons exécuter notre application sans aucun fichier de message, et sans erreurs :

```
$ env -i /bin/sh
$ ./exemple-gettext
This is the first string in the first set
and here is the second string in the first set.
Now let's have a look at the 1st string in 2nd set,
and finally the second string in the second set.
$ exit
```

Nous allons compiler le fichier `.po`, puis nous l'installerons dans le répertoire système d'internationalisation :

```
$ msgfmt -o exemple-gettext.mo exemple-gettext.po
$ ls -l exemple-gettext.mo
-rw-rw-r-- 1 cpb cpb 511 Aug 22 17:11 exemple-gettext.mo
$ su
Password:
# cp exemple-gettext.mo /usr/share/locale/fr/LC_MESSAGES/
# exit
exit
$
```

Essayons notre application :

```
$ export LANG=fr_FR.UTF-8
$ ./exemple-gettext
Ceci est la premiere chaine du premier ensemble
et voici la seconde chaine du premier ensemble.
A present regardons la 1ere chaine du 2eme ensemble,
et finalement la seconde chaine du second ensemble.
$
```

On remarque que le fichier a été copié dans le répertoire de la localisation fr alors que la variable `LANG` a été configurée avec `fr_FR.UTF-8`. La bibliothèque *GetText* recherche en effet intelligemment les fichiers de traduction disponibles.

Il est ainsi possible, de manière simple, de traduire facilement les messages d'interface d'un logiciel. Le projet *GetText* contient également des utilitaires permettant d'analyser le fichier source d'une application existante, afin d'en extraire les chaînes à traduire. Le fichier `.po` est alors construit automatiquement, et il ne reste plus qu'à le soumettre à un traducteur.

Il ne faut toutefois pas oublier que la traduction des messages n'est qu'une partie de l'internationalisation d'un logiciel. De nombreuses conventions culturelles sont parfois aussi importantes que la langue utilisée pour l'interface utilisateur. Si une application affiche le libellé

| This message was received on 03.08.2010

et qu'on traduit uniquement le texte, obtenant ainsi

| Ce message a été reçu le 03.08.2010

il y a de fortes chances pour que le lecteur français lise 3 août au lieu du 8 mars original.

Un logiciel doit donc pouvoir s'adapter aux règles d'usage décrites dans les autres catégories de localisation.

Configuration de la localisation

Pour qu'une application soit sensible à la localisation, elle doit d'abord invoquer la fonction `setlocale()`, déclarée dans `<locale.h>` :

```
| char * setlocale (int categorie, const char * localisation);
```

Cette routine (et `locale.setlocale()` en Python) demande à la bibliothèque C que toutes les fonctions manipulant des données en rapport avec la catégorie précisée en premier argument prennent en compte le fait que l'utilisateur se trouve dans la localisation indiquée en second argument.

La catégorie est mentionnée sous forme d'une constante symbolique ayant le même nom que les variables d'environnement décrites plus haut : `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`, et surtout `LC_ALL`. Il n'y a pas de constante `LANG`, cette variable d'environnement étant utilisée comme valeur par défaut pour toutes les catégories. Il faut noter que dans l'avenir d'autres catégories seront peut-être ajoutées, et qu'il en existe déjà d'autres sur certains systèmes.

La localisation indiquée en second argument peut prendre l'une des formes suivantes :

- Une chaîne de caractères construite sur le même modèle que le contenu des variables d'environnement précédentes (par exemple `"fr_FR.UTF-8"`).
- Une chaîne renvoyée par un appel précédent à `setlocale()`, comme nous le décrirons plus bas.
- Les chaînes spéciales `"POSIX"` ou `"C"`, qui demandent à la bibliothèque d'adopter le comportement décrit exactement dans ces standards, sans s'occuper des variations dues à la localisation. Il s'agit en fait d'une commande d'anti-internationalisation, assurant que le programme fournisse partout le même résultat. Nous reviendrons également sur cette option.
- La chaîne vide `""`, qui demande à la bibliothèque d'adopter le comportement adapté à la localisation configurée par l'utilisateur dans ses variables d'environnement.

En fait, c'est bien entendu la dernière forme qui est la plus utilisée. Il me semble d'ailleurs n'avoir jamais invoqué `setlocale()` dans une application avec d'autres arguments que

```
| setlocale (LC_ALL, "");
```

qui réclament de la bibliothèque C une adaptation de ses fonctionnalités, dans toutes les catégories, suivant la localisation choisie par l'utilisateur.

Si on n'appelle pas `setlocale()`, le comportement de la bibliothèque C est le même qu'en ayant invoqué `setlocale(LC_ALL, "C")`, qui ne présente donc pas d'intérêt. En revanche, les localisations "C" et "POSIX" peuvent être utiles pour configurer une catégorie particulière.

Par exemple, une application peut autoriser un utilisateur à employer ses préférences en termes de langage, de présentation de la date ou de classification des caractères accentués, mais imposer que les saisies de nombres réels se fassent en employant le point comme séparateur décimal. Ceci afin de pouvoir relire automatiquement des fichiers de données déjà construits. On emploiera alors une séquence :

```
setlocale (LC_ALL, "");  
setlocale (LC_NUMERIC, "C");
```

Ce genre de restriction peut aussi s'appliquer à la catégorie `LC_CTYPE` pour les programmes qui s'appuient fortement sur les correspondances entre les caractères *Ascii* et leurs valeurs numériques.

Le fait de configurer directement une localisation avec une chaîne explicite ne se justifie que si l'application est lancée sans que les variables d'environnement n'aient pu être configurées par le shell (un démon par exemple). On laissera alors l'utilisateur inscrire ses préférences dans un fichier de configuration, et l'application devra invoquer `setlocale()` avec la chaîne indiquée.

Lorsqu'on passe un argument `NULL` en seconde position de `setlocale()`, cette fonction renvoie un pointeur sur une chaîne de caractères décrivant la localisation actuelle pour la catégorie concernée. Cette chaîne peut être copiée pour une utilisation ultérieure, au besoin. Le pointeur est dirigé vers une zone de mémoire statique interne à la bibliothèque, qui risque d'être écrasée par la suite, et qu'il faut donc copier si on désire la conserver. Si on réclame la valeur de `LC_ALL`, la chaîne renvoyée peut prendre différents formats, car elle représente l'ensemble des catégories (qui peuvent être configurées avec des localisations différentes). Cette chaîne n'est pas obligatoirement intelligible – quoique ce soit apparemment le cas avec la *Glibc* –, mais elle pourra dans tous les cas être réemployée comme second argument d'un appel `setlocale()` ultérieur.

Voici un programme qui va afficher l'état des localisations programmées :

```
exemple-setlocale.c :  
#include <locale.h>  
#include <stdio.h>  
#include <stdlib.h>
```

```
int main (void)
{
    setlocale(LC_ALL, "");
    printf("LC_COLLATE = %s \n", setlocale(LC_COLLATE, NULL));
    printf("LC_CTYPE = %s \n", setlocale(LC_CTYPE, NULL));
    printf("LC_MESSAGES = %s \n", setlocale(LC_MESSAGES, NULL));
    printf("LC_MONETARY = %s \n", setlocale(LC_MONETARY, NULL));
    printf("LC_NUMERIC = %s \n", setlocale(LC_NUMERIC, NULL));
    printf("LC_TIME = %s \n", setlocale(LC_TIME, NULL));
    printf("LC_ALL = %s \n", setlocale(LC_ALL, NULL));
    return EXIT_SUCCESS;
}
```

En fait, l'exécution nous permet de vérifier que la localisation par défaut est "C", et que les localisations respectent la hiérarchie LANG < LC_xxx < LC_ALL.

```
$ env -i /bin/sh
$ ./exemple-setlocale
LC_COLLATE = C
LC_CTYPE = C
LC_MESSAGES = C
LC_MONETARY = C
LC_NUMERIC = C
LC_TIME = C
LC_ALL = C
$ export LANG=fr_FR.UTF-8
$ ./exemple-setlocale
LC_COLLATE = fr_FR.UTF-8
LC_CTYPE = fr_FR.UTF-8
LC_MESSAGES = fr_FR.UTF-8
LC_MONETARY = fr_FR.UTF-8
LC_NUMERIC = fr_FR.UTF-8
LC_TIME = fr_FR.UTF-8
LC_ALL = fr_FR.UTF-8
$ export LC_CTYPE=fr_CA.UTF-8
$ ./exemple-setlocale
LC_COLLATE = fr_FR.UTF-8
LC_CTYPE = fr_CA.UTF-8
LC_MESSAGES = fr_FR.UTF-8
LC_MONETARY = fr_FR.UTF-8
LC_NUMERIC = fr_FR.UTF-8
LC_TIME = fr_FR.UTF-8
```

```

LC_ALL      = LC_CTYPE=fr_CA.UTF-8;LC_NUMERIC=fr_FR.UTF-8;LC_TIME=fr_
FR.UTF-8;LC_COLLATE=fr_FR.UTF-8;LC_MONETARY=fr_FR.UTF-8;LC_MESSAGES=
fr_FR.UTF-8;LC_PAPER=fr_FR.UTF-8;LC_NAME=fr_FR.UTF-8;LC_ADDRESS=fr_F
R.UTF-8;LC_TELEPHONE=fr_FR.UTF-8;LC_MEASUREMENT=fr_FR.UTF-8;LC_IDENT
IFICATION=fr_FR.UTF-8
$ export LC_MONETARY=fr_BE.UTF-8
$ ./exemple-setlocale
LC_COLLATE = fr_FR.UTF-8
LC_CTYPE   = fr_CA.UTF-8
LC_MESSAGES = fr_FR.UTF-8
LC_MONETARY = fr_BE.UTF-8
LC_NUMERIC = fr_FR.UTF-8
LC_TIME    = fr_FR.UTF-8
LC_ALL     = LC_CTYPE=fr_CA.UTF-8;LC_NUMERIC=fr_FR.UTF-8;LC_TIME=fr_
FR.UTF-8;LC_COLLATE=fr_FR.UTF-8;LC_MONETARY=fr_BE.UTF-8;LC_MESSAGES=
fr_FR.UTF-8;LC_PAPER=fr_FR.UTF-8;LC_NAME=fr_FR.UTF-8;LC_ADDRESS=fr_F
R.UTF-8;LC_TELEPHONE=fr_FR.UTF-8;LC_MEASUREMENT=fr_FR.UTF-8;LC_IDENT
IFICATION=fr_FR.UTF-8
$ export LC_ALL=es_ES
$ ./exemple-setlocale
LC_COLLATE = es_ES
LC_CTYPE   = es_ES
LC_MESSAGES = es_ES
LC_MONETARY = es_ES
LC_NUMERIC = es_ES
LC_TIME    = es_ES
LC_ALL     = es_ES
$

```

Localisation et fonctions de bibliothèques

Une fois que la localisation a été définie pour une ou plusieurs catégories, certaines fonctions de bibliothèque modifient leur comportement pour s'adapter aux coutumes en usage chez l'utilisateur. L'application peut continuer à utiliser `printf()`, par exemple pour afficher des réels, mais le symbole employé pour afficher le séparateur décimal sera modifié. Notons que `printf()` ne permet pas de séparer les chiffres par milliers, au contraire de `strfmon()` que nous étudierons plus loin.

Le programme suivant affiche la valeur numérique 2010,08 avec les variations dues à la localisation :

```

exemple-numeric.c :
#include <locale.h>
#include <stdio.h>

```

```
#include <stdlib.h>

int main (int argc, char * argv[])
{
    double d = 2010.08;
    setlocale(LC_ALL, "");
    fprintf(stdout, "%.2f\n", d);
    return EXIT_SUCCESS;
}
```

L'exécution nous montre que seul le point décimal est modifié :

```
$ unset LC_ALL LANG LC_NUMERIC
$ ./exemple-numeric
2010.08
$ export LANG=fr_FR.UTF-8
$ ./exemple-numeric
2010,08
$
```

De nombreuses routines sont affectées par les catégories `LC_CTYPE` et `LC_COLLATE`, qui concernent les caractères manipulés par le programme. Dans le chapitre 16, nous avons examiné le comportement des fonctions `strcascmp()`, `strcoll()` ou `strxfrm()` vis-à-vis de la localisation, et nous avons remarqué que les résultats pouvaient varier grandement suivant le jeu de caractères utilisé.

Dans le chapitre 8, nous avons observé aussi que plusieurs fonctions de manipulation des dates étaient sensibles à la localisation via la catégorie `LC_TIME`. Cela concerne aussi bien l'affichage avec `strftime()` par exemple que la saisie avec `getdate()`. On peut s'en rendre également compte avec le comportement de l'utilitaire `/bin/date` en l'invoquant avec l'argument `"%A %x"`, qui lui demande d'afficher le nom du jour de la semaine, suivi de la date complète.

```
$ unset LC_ALL LANG LC_TIME
$ date +"%A %x"
Sunday 08/22/10
$ export LC_TIME=fr_FR
$ date +"%A %x"
dimanche 22/08/2010
$ export LC_TIME=es_ES
$ date +"%A %x"
domingo 22/08/10
$
```

Pour afficher des valeurs monétaires, `printf()` n'est pas vraiment adapté car il ne prend pas en compte le symbole de la monnaie du pays ni certaines coutumes comme la séparation des valeurs par milliers. Pour le remplacer dans ce rôle, il existe une fonction nommée `strfmon()`, déclarée dans `<monetary.h>` :

```
ssize_t strfmon (char * buffer, size_t taille,  
                char * format, ...);
```

Cette fonction utilise le format indiqué en troisième argument et convertit les données à sa suite, en stockant le résultat dans le premier argument, dont la taille maximale est indiquée en seconde position. Cette routine se comporte donc un peu comme `snprintf()`, mais elle renvoie le nombre de caractères inscrits dans le buffer. La chaîne de format peut comprendre des caractères normaux, qui seront recopiés directement, ou des indicateurs de conversion commençant par le caractère `%`. Les conversions possibles sont :

Code	Type d'argument	Signification
<code>%i</code>	<code>double</code>	Représentation locale d'une valeur monétaire représentée sous sa forme internationale.
<code>%n</code>	<code>double</code>	Représentation locale d'une valeur monétaire représentée sous sa forme nationale.
<code>%%</code>		Affichage du caractère <code>%</code> .

Entre le `%` et le code de conversion peuvent se trouver plusieurs champs :

- un attribut précisant le formatage du nombre ;
- la largeur minimale de la représentation de la valeur ;
- un symbole `#` suivi de la largeur minimale de la partie entière, sans compter les éventuels séparateurs de milliers ;
- un point suivi du nombre de décimales à afficher.

L'attribut de formatage peut prendre les formes suivantes :

Code	Signification
<code>=<caractère></code>	Le caractère indiqué sera employé pour remplir les blancs avant le nombre lorsque le résultat est plus petit que la longueur minimale demandée. En général le caractère par défaut, espace, suffit. Parfois, on peut préférer des zéros ou des points.
<code>!</code>	Ne pas afficher le symbole de la monnaie.
<code>^</code>	Ne pas afficher les séparateurs des milliers.
<code>-</code>	Aligner les nombres à gauche plutôt qu'à droite.
<code>+</code>	Les valeurs sont précédées de leur signe, positif ou négatif.

Code	Signification
(Les valeurs négatives sont entourées de parenthèses. Cet attribut ne peut pas être employé en même temps que le précédent.

Le programme suivant va appeler `strfmon()` avec les arguments passés sur sa ligne de commande.

```
exemple-strfmon.c :
#include <locale.h>
#include <monetary.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char * argv[])
{
    double d;
    char  buffer[80];
    setlocale(LC_ALL, "");
    if ((argc != 3)
        || (sscanf(argv[2], "%lf", & d) != 1)) {
        fprintf(stderr, "%s format valeur \n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if (strfmon(buffer, 80, argv[1], d) > 0)
        fprintf(stdout, "%s\n", buffer);
    return EXIT_SUCCESS;
}
```

Nous allons tout d'abord observer les effets de la localisation sur une valeur donnée :

```
$ unset LC_ALL LANG LC_MONETARY
$ ./exemple-strfmon "%n" 1500
1500.00
$ export LANG=fr_FR
$ ./exemple-strfmon "%n" 1500
1 500,00 EUR
$ ./exemple-strfmon "%i" 1500
1 500,00 EUR
$ export LANG=fr_CH
$ ./exemple-strfmon "%n" 1500
Fr. 1'500.00
$ ./exemple-strfmon "%i" 1500
CHF 1'500.00
$ export LANG=en_US
```

```
$ ./exemple-strfmon "%n" 1500
$1,500.00
$ ./exemple-strfmon "%i" 1500
USD 1,500.00
$
```

On remarque la différence entre les représentations nationales et internationales des monnaies, définies par la norme *Iso-4217*. Le séparateur des milliers varie aussi. Regardons à présent l'effet des indicateurs de signe :

```
$ export LANG=fr_FR
$ ./exemple-strfmon "%n" -1500
-1 500,00 EUR
$ ./exemple-strfmon "%(n" 1500
1 500,00 EUR
$ ./exemple-strfmon "%(n" -1500
(1 500,00 EUR)
$
```

Nous laisserons le lecteur expérimenter lui-même les différentes possibilités, en précisant que le caractère '!' peut poser des problèmes avec certains shells et qu'il est préférable pour l'utiliser de basculer sur un interpréteur ne l'employant pas, comme ksh, plutôt que de compliquer la chaîne de format pour protéger le caractère :

En fait, `strfmon()` ne sait manipuler que des valeurs monétaires, tout comme `strftime()` ne traite que des dates et des heures. Si la chaîne finale doit contenir d'autres conversions, on peut se servir de `strfmon()` pour construire la chaîne de format qui sera employée dans `printf()`. En voici un exemple.

```
exemple-strfmon-2.c :
#include <locale.h>
#include <monetary.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int quantite[] = {
        1, 4, 3, 1, 1, 2, 0
    };
    char * reference[] = {
        "ABC", "DEF", "GHI", "JKL", "MNO", "PQR", NULL
    };
};
```

```

double prix[] = {
    1500, 2040, 560, 2500, 38400, 125, 0
};
int i;
char format[80];
double total = 0.0;

setlocale(LC_ALL, "");

for (i = 0; reference[i] != NULL; i++) {
    strfmon(format, 80, "%%5s : %%5n x %%d = %%5n\n",
            prix[i], prix[i] * quantite[i]);
    fprintf(stdout, format, reference[i], quantite[i]);
    total += prix[i] * quantite[i];
}
strfmon(format, 80, "                                Total = %%5n\n", total);
fprintf(stdout, format);
return EXIT_SUCCESS;
}

```

La chaîne de format transmise à `fprintf()` permet d'afficher un libellé et un nombre de pièces.

```

$ unset LC_ALL LANG LC_MONETARY
$ export LANG=fr_FR
$ ./exemple-strfmon-2
ABC :   1 500,00 EUR x 1 =   1 500,00 EUR
DEF :   2 040,00 EUR x 4 =   8 160,00 EUR
GHI :     560,00 EUR x 3 =   1 680,00 EUR
JKL :   2 500,00 EUR x 1 =   2 500,00 EUR
MNO : 38 400,00 EUR x 1 = 38 400,00 EUR
PQR :    125,00 EUR x 2 =    250,00 EUR
      Total = 52 490,00 EUR

$ export LANG=en_GB.UTF-8
$ ./exemple-strfmon-2
ABC : £ 1,500.00 x 1 = £ 1,500.00
DEF : £ 2,040.00 x 4 = £ 8,160.00
GHI : £   560.00 x 3 = £ 1,680.00
JKL : £ 2,500.00 x 1 = £ 2,500.00
MNO : £38,400.00 x 1 = £38,400.00
PQR : £   125.00 x 2 = £   250.00
      Total = £52,490.00

$

```

Localisation et fonctions personnelles

Il peut arriver qu'une application ait à construire elle-même la représentation locale d'une valeur numérique ou monétaire, sans que la fonction `strfmon()` soit suffisante. On peut par exemple avoir besoin de connaître le symbole monétaire employé localement pour l'afficher en tête de colonne d'une facture.

Informations numériques et monétaires avec `localeconv()`

Il existe une fonction nommée `localeconv()`, définie comme `setlocale()` par le standard *Iso C*, et déclarée dans `<locale.h>` ainsi :

```
| struct lconv * localeconv (void);
```

Cette routine renvoie un pointeur sur une zone de données statiques, interne à la bibliothèque, susceptible d'être modifiée, et qu'il ne faut pas écraser. En Python, `locale.localeconv()` renvoie un dictionnaire dont les clés correspondent aux membres de la structure `lconv` :

Nom	Type	Signification
<code>decimal_point</code>	<code>char *</code>	Chaîne contenant le caractère employé comme séparateur décimal. Par défaut, il s'agit du point.
<code>thousands_sep</code>	<code>char *</code>	Chaîne comportant le caractère employé comme séparateur des milliers. Par défaut, la chaîne est vide.
<code>mon_decimal_point</code>	<code>char *</code>	Comme <code>decimal_point</code> , mais appliqué uniquement aux valeurs monétaires.
<code>mon_thousands_sep</code>	<code>char *</code>	Comme <code>thousands_sep</code> pour les valeurs monétaires.
<code>currency_symbol</code>	<code>char *</code>	Symbole monétaire pour des échanges nationaux.
<code>int_curr_symbol</code>	<code>char *</code>	Symbole monétaire pour des échanges internationaux. Conforme à la norme <i>Iso-4217</i> .
<code>positive_sign</code>	<code>char *</code>	Signe employé pour les valeurs monétaires positives. Par défaut, cette chaîne est vide.
<code>negative_sign</code>	<code>char *</code>	Signe utilisé pour les valeurs monétaires négatives. Si cette chaîne est vide, comme c'est le cas par défaut, et si le membre précédent est également vide, il faut employer " - ".
<code>frac_digits</code>	<code>char</code>	Nombre de décimales à afficher dans une représentation monétaire nationale. La valeur par défaut, <code>CHAR_MAX</code> , signifie que le comportement n'est pas précisé.
<code>int_frac_digits</code>	<code>char</code>	Nombre de décimales à afficher dans une représentation monétaire internationale. La valeur par défaut, <code>CHAR_MAX</code> , signifie que le comportement n'est pas précisé.

Nom	Type	Signification
p_cs_precedes	char	Ce membre vaut 1 si le symbole monétaire doit précéder une valeur positive, 0 s'il doit la suivre, et CHAR_MAX si le comportement n'est pas précisé.
p_sep_by_space	char	Ce membre vaut 1 si le symbole monétaire doit être séparé d'une valeur positive par un espace, 0 sinon, et CHAR_MAX si le comportement n'est pas précisé.
p_sign_posn	char	Ce champ peut prendre les valeurs suivantes : 0 si aucun signe n'est affiché devant une valeur positive ; 1 si le signe doit précéder une valeur positive et son symbole ; 2 si le signe doit suivre la valeur positive et le symbole ; 3 si le signe doit se trouver immédiatement avant le symbole ; 4 si le signe doit se trouver immédiatement après le symbole.
n_cs_precedes	char	Comme p_cs_precedes, pour une valeur négative.
n_sep_by_space	char	Comme p_sep_by_space, pour une valeur négative.
n_sign_posn	char	Ce champ est équivalent à p_sign_posn pour des valeurs négatives, mais s'il vaut 0, la valeur négative et son symbole doivent être encadrés par des parenthèses.

Le programme suivant affiche les informations de la structure `lconv` correspondant à la localisation en cours.

```

exemple-localeconv.c :
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    struct lconv * lconv;
    setlocale(LC_ALL, "");
    lconv = localeconv();
    printf("decimal_point      = %s \n", lconv->decimal_point);
    printf("thousands_sep          = %s \n", lconv->thousands_sep);
    printf("mon_decimal_point        = %s \n", lconv->mon_decimal_point);
    printf("mon_thousands_sep        = %s \n", lconv->mon_thousands_sep);
    printf("currency_symbol          = %s \n", lconv->currency_symbol);
    printf("int_curr_symbol          = %s \n", lconv->int_curr_symbol);
    printf("positive_sign            = %s \n", lconv->positive_sign);
    printf("negative_sign            = %s \n", lconv->negative_sign);
    printf("frac_digits              = %d \n", lconv->frac_digits);
    printf("int_frac_digits          = %d \n", lconv->int_frac_digits);
    printf("p_cs_precedes            = %d \n", lconv->p_cs_precedes);
    printf("p_sep_by_space           = %d \n", lconv->p_sep_by_space);
    printf("p_sign_posn              = %d \n", lconv->p_sign_posn);

```

```

    printf("n_cs_precedes      = %d \n", lconv->n_cs_precedes);
    printf("n_sep_by_space    = %d \n", lconv->n_sep_by_space);
    printf("n_sign_posn       = %d \n", lconv->n_sign_posn);
    return EXIT_SUCCESS;
}

```

Un script quasi équivalent en Python serait :

```

exemple-localeconv.py :
#!/usr/bin/python
from __future__ import print_function
import locale

locale.setlocale(locale.LC_ALL, "")
database = locale.localeconv()
for key in database:
    print(key, database[key])

```

Rappelons qu'une valeur CHAR_MAX (127) dans un champ de type char signifie que l'information n'est pas disponible.

```

$ env -i /bin/sh
$ ./exemple-localeconv
decimal_point      = .
thousands_sep     =
mon_decimal_point  =
mon_thousands_sep =
currency_symbol    =
int_curr_symbol    =
positive_sign      =
negative_sign      =
frac_digits        = 127
int_frac_digits    = 127
p_cs_precedes      = 127
p_sep_by_space     = 127
p_sign_posn        = 127
n_cs_precedes      = 127
n_sep_by_space     = 127
n_sign_posn        = 127
$ export LC_ALL=fr_FR.UTF-8
$ ./exemple-localeconv
decimal_point      = ,
thousands_sep     =
mon_decimal_point  = ,
mon_thousands_sep =

```

```
currency_symbol = €
int_curr_symbol = EUR
positive_sign   =
negative_sign   = -
frac_digits     = 2
int_frac_digits = 2
p_cs_precedes   = 0
p_sep_by_space  = 1
p_sign_posn     = 1
n_cs_precedes   = 0
n_sep_by_space  = 1
n_sign_posn     = 1
$ export LC_ALL=en_US.utf-8
$ ./exemple-localeconv
decimal_point   = .
thousands_sep   = ,
mon_decimal_point = .
mon_thousands_sep = ,
currency_symbol  = $
int_curr_symbol  = USD
positive_sign    =
negative_sign    = -
frac_digits      = 2
int_frac_digits  = 2
p_cs_precedes    = 1
p_sep_by_space   = 0
p_sign_posn      = 1
n_cs_precedes    = 1
n_sep_by_space   = 0
n_sign_posn      = 1
$ exit
$7
```

Informations complètes avec `nl_langinfo()`

Il apparaît à l'usage que la fonction `localeconv()` n'est pas suffisante pour obtenir toutes les informations pertinentes concernant la localisation. La limitation aux valeurs numériques et monétaires est très restrictive par rapport au contenu complet des données localisées. Impossible en effet d'avoir accès aux noms des mois sans passer par `strftime()`, ou encore de vérifier si la réponse d'un utilisateur est affirmative ou négative.

Une autre fonction a donc été définie, un peu moins portable que `localeconv()` car elle n'est pas dans la norme *ISO C*. Nommée `nl_langinfo()`, elle se trouve quand même dans les spécifications SUSv4. Sa déclaration se trouve dans `<langinfo.h>` :

```
| char * nl_langinfo (nl_item objet);
```

Le type `nl_item` est numérique, il est défini dans `<nl_types.h>`. En Python, `locale.nm_langinfo()` prend en argument des constantes identiques à celles du langage C, préfixées par `locale`. Cette routine renvoie un pointeur sur une chaîne de caractères contenant la représentation locale de l'objet demandé. Contrairement à `localeconv()`, la fonction `nl_langinfo()` permet donc de réclamer uniquement les informations qui nous intéressent.

La chaîne de caractères renvoyée se trouve dans une zone de mémoire statique, susceptible d'être écrasée à chaque appel. L'argument de cette routine est une valeur numérique, qu'on choisit parmi les constantes symboliques suivantes, définies dans `<langinfo.h>` :

Nom	Catégorie	Signification
YESEXPR, NOEXPR	LC_MESSAGES	Chaîne représentant une réponse affirmative ou négative.
MON_DECIMAL_POINT	LC_MONETARY	Significations identiques à celles des champs ayant les mêmes noms dans la structure <code>lconv</code> fournie par <code>localeconv()</code> .
MON_THOUSANDS_SEP		
CURRENCY_SYMBOL		
INT_CUR_SYMBOL		
POSITIVE_SIGN		
NEGATIVE_SIGN		
FRAC_DIGITS		
INT_FRAC_DIGITS		
P_CS_PRECEDES		
P_SEP_BY_SPACE		
P_SIGN_POSN		
N_CS_PRECEDES		
N_SEP_BY_SPACE		
N_SIGN_POSN		
DECIMAL_POINT	LC_NUMERIC	
THOUSANDS_SEP		

Nom	Catégorie	Signification
ABDAY_1 ... ABDAY_7	LC_TIME	Abréviations des noms des jours de la semaine.
ABMON_1 ... ABMON_12		Abréviations des noms des mois.
DAY_1 ... DAY_7		Noms des jours de la semaine.
MON_1 ... MON_12		Nom des mois.
AM_STR, PM_STR		Chaînes représentant les symboles AM et PM.
D_FMT, D_T_FMT		Formats pour <code>strftime()</code> afin d'obtenir la date seule, ou la date et l'heure.
T_FMT, T_FMT_AMPM		Formats pour avoir l'heure, éventuellement avec les symboles AM et PM.

Il existe quelques autres objets dans la catégorie `LC_TIME`, si la localisation supporte un second calendrier. Ceci est rarement utilisé, et on laissera le lecteur se reporter à la documentation Gnu s'il a besoin de ces fonctionnalités.

Le programme suivant affiche le contenu des champs qui n'étaient pas définis dans la structure `lconv`.

```
exemple-nl-langinfo.c :
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
    int i;
    char * libelles[] = {
        "YESEXPR", "NOEXPR",
        "ABDAY_1", "ABDAY_7",
        "ABMON_1", "ABMON_12",
        "DAY_1", "DAY_7",
        "MON_1", "MON_12",
        "AM_STR", "PM_STR",
        "D_FMT", "D_T_FMT",
        "T_FMT", "T_FMT_AMPM",
        NULL
    };
    nl_item objet[] = {
        YESEXPR, NOEXPR,
        ABDAY_1, ABDAY_7,
        ABMON_1, ABMON_12,
```

```

    DAY_1,    DAY_7,
    MON_1,    MON_12,
    AM_STR,   PM_STR,
    D_FMT,    D_T_FMT,
    T_FMT,    T_FMT_AMPM,
    0
};
setlocale(LC_ALL, "");
for (i = 0; libelles[i] != NULL; i++)
    printf("%10s = \"%s\\\"\\n\", libelles[i], nl_langinfo(objet [i]));
return EXIT_SUCCESS;
}

```

Nous n'affichons pas tous les jours de la semaine ni tous les mois.

```

$ env -i /bin/sh
$ ./exemple-nl-langinfo
YESEXPR = "^[yY]"
NOEXPR = "^[nN]"
ABDAY_1 = "Sun"
ABDAY_7 = "Sat"
ABMON_1 = "Jan"
ABMON_12 = "Dec"
    DAY_1 = "Sunday"
    DAY_7 = "Saturday"
    MON_1 = "January"
    MON_12 = "December"
    AM_STR = "AM"
    PM_STR = "PM"
    D_FMT = "%m/%d/%y"
    D_T_FMT = "%a %b %e %H:%M:%S %Y"
    T_FMT = "%H:%M:%S"
T_FMT_AMPM = "%I:%M:%S %p"
$ export LC_ALL=fr_FR.UTF-8
$ ./exemple-nl-langinfo
YESEXPR = "^[oOyY].*"
NOEXPR = "^[nN].*"
ABDAY_1 = "dim."
ABDAY_7 = "sam."
ABMON_1 = "janv."
ABMON_12 = "déc."
    DAY_1 = "dimanche"
    DAY_7 = "samedi"
    MON_1 = "janvier"

```

```
MON_12 = "décembre"
AM_STR = ""
PM_STR = ""
D_FMT = "%d/%m/%Y"
D_T_FMT = "%a %d %b %Y %T %Z"
T_FMT = "%T"
T_FMT_AMPM = ""
$ exit
$
```

Nous voyons ainsi qu'on peut obtenir toutes les informations pertinentes concernant les différentes catégories de localisation susceptibles d'être employées dans une application.

Conclusion

Avec les possibilités de traduction des messages d'interface, en employant la bibliothèque Gnu *GetText* et en s'appuyant sur les fonctionnalités offertes par `ngettext()`, une application peut prétendre à une véritable portée internationale. La documentation Gnu, principalement celle de la bibliothèque *GetText*, offre des informations complémentaires qui intéresseront les développeurs confrontés à des situations plus complexes que celles décrites ici.

Gestion du terminal

La gestion des terminaux, sous Linux du moins, est souvent et principalement motivée par l'un des trois principes suivants :

- 1 Les entrées-sorties habituelles ne permettent pas de capturer des caractères au vol sans que l'utilisateur n'appuie sur la touche *Entrée*. Dans certaines applications, on peut avoir besoin de modifier le mode du terminal pour adopter ce type de comportement.
- 2 Lorsqu'on veut proposer une connexion à distance, à la manière des démons `sshd` ou `telnetd`, il est souvent nécessaire d'utiliser des fonctionnalités particulières offertes par le noyau, par le biais des pseudo-terminaux.
- 3 Sur un PC sous Linux, les lignes d'entrée-sortie série servent encore pour la connexion de périphériques (cartes d'acquisition, systèmes embarqués, etc.) et l'initialisation de ces ports RS-232 (souvent via un adaptateur USB) emploie les méthodes de configuration des terminaux.

Nous allons donc étudier les fonctions de manipulation des terminaux avec ces trois optiques successives.

Définition des terminaux

La notion de terminal Unix a été introduite dans les systèmes où une unité centrale accueillait les connexions en provenance d'une ou plusieurs dizaines de terminaux travaillant en mode texte. Depuis quelques années ces ensembles ont pratiquement disparu, remplacés par des stations de travail intégrant l'unité de calcul et un terminal

graphique haute résolution, connectés par réseau Ethernet afin de partager grâce au protocole NFS les ressources disque.

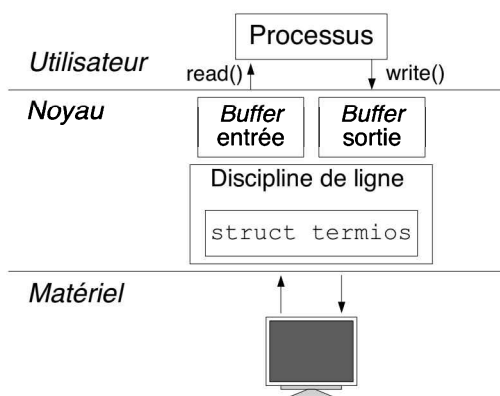
Cette évolution est encore plus marquée avec les PC sous Linux, sur lesquels il est bien rare qu'une machine grand public dispose encore d'un port RS-232, et encore plus rare que celui-ci serve à connecter un terminal en mode texte, sauf à titre de curiosité purement expérimentale. Toutefois, les mécanismes de configuration sont toujours disponibles et permettent de gérer les consoles virtuelles, les pseudo-terminals ou des liaisons RS-232 avec divers équipements généralement par le biais d'adapteurs USB-Série.

Chaque terminal qu'on pouvait connecter sur une unité centrale offrait des possibilités spécifiques, et sa configuration différait toujours quelque peu des autres modèles. Ces paramètres concernent par exemple les codes servant à effacer l'écran, à déplacer le curseur ou à changer la couleur du texte. Une liste imposante de terminaux différents avec leurs caractéristiques est donc disponible dans le fichier `/etc/termcap` des systèmes Unix.

Le noyau gère le terminal de manière à offrir une interface homogène aux processus lorsqu'ils y accéderont avec les appels système `read()` et `write()`. Pour configurer l'état d'un terminal, le noyau propose une série de paramètres regroupés dans une structure `termios`, qu'on nomme *discipline de ligne*. Nous reviendrons sur son contenu plus bas.

La discipline de ligne s'intercale donc entre le terminal et le processus. Le noyau incorpore aussi deux buffers pour conserver les touches appuyées afin qu'elles soient lues par le programme et pour enregistrer les caractères envoyés, jusqu'à ce que le terminal soit prêt à les afficher.

Figure 32-1
Organisation de la configuration du terminal



Les méthodes de configuration des terminaux étaient à l'origine fondées sur l'appel système `ioctl()`, qui offre une interface de bas niveau avec les périphériques

d'entrée-sortie. Ce mécanisme est complexe et peu portable. D'autres fonctions ont donc été définies par Posix, bien qu'il reste encore quelques extensions BSD ou Système V disponibles sous Linux.

Un terminal peut être utilisé dans deux modes nommés traditionnellement *canonique* et *non canonique* :

- En mode canonique, les saisies sont transmises au processus ligne par ligne. Ceci signifie que tant que l'utilisateur n'a pas appuyé sur la touche *Entrée*, le pilote de terminal n'envoie rien au processus qui effectue un `read()`. Cette configuration est celle établie par défaut sur les terminaux sous Unix, nous en avons déjà parlé à de multiples reprises dans ce livre.
- En mode non canonique ou brut (*raw*), le terminal transmet immédiatement les caractères sur lesquels on appuie. Un `read()` ne nous retournera donc pas nécessairement des lignes entières.

Chacun de ces modes présente des avantages :

- Le mode canonique permet de laisser le pilote de terminal gérer entièrement les corrections, effacement, etc. Lors d'une saisie avec `fgets()`, l'utilisateur pourra revenir en arrière avec la touche d'effacement et corriger sa ligne. Seule la pression sur la touche *Entrée* validera vraiment la saisie.
- Le mode non canonique permet de construire des applications d'édition de texte plus dynamiques, puisqu'on pourra gérer un curseur avec les touches fléchées, les sauts de page, etc. Le processus peut aussi capturer des caractères au vol dans des applications où les saisies de l'utilisateur se déroulent en parallèle avec d'autres tâches.
- Une ligne, en mode canonique, est soumise à une longueur maximale de `MAX_CANON` caractères, définie dans `<limits.h>`. Cette valeur, 255 sous Linux, est souvent utilisée pour dimensionner des chaînes de caractères avant d'appeler `fgets()`.

Configuration d'un terminal

La structure `termios` comporte cinq membres définis par SUSv4 :

Nom	Type	Signification
<code>c_iflag</code>	<code>tcflag_t</code>	Attributs définissant le mode d'entrée depuis le terminal.
<code>c_oflag</code>	<code>tcflag_t</code>	Attributs concernant le mode de sortie vers le terminal.
<code>c_cflag</code>	<code>tcflag_t</code>	Attributs permettant le contrôle du terminal.
<code>c_lflag</code>	<code>tcflag_t</code>	Attributs locaux pour le terminal.
<code>cc</code>	<code>cc_t[]</code>	Variables concernant le terminal. Essentiellement des affectations de touches.

Nous allons examiner en détail les attributs un peu plus loin. Indiquons tout de suite que la répartition des attributs dans les différents membres n'a rien d'intuitif et que certaines fonctionnalités, comme la gestion des signaux, nécessitent d'agir sur plusieurs attributs, en l'occurrence `c_iflag` et `c_cflag`.

La configuration du terminal s'effectue à l'aide des fonctions `tcgetattr()` et `tcsetattr()`, qui permettent de lire ou de fixer la valeur de la structure `termios`.

```
int tcgetattr (int terminal, struct termios * configuration);
int tcsetattr (int terminal, int option, struct termios *config);
```

L'implémentation réelle de la structure `termios` comportant des champs avec un grand nombre d'attributs, il est important de lire l'état du terminal avec `tcgetattr()` avant de modifier les membres voulus, puis de la réécrire avec `tcsetattr()`.

En Python, la fonction `termios.tcgetattr(terminal)` renvoie une liste de sept valeurs (*iflag*, *oflag*, *cflag*, *lflag*, *ispeed*, *ospeed* et *cc*) correspondant aux cinq champs de la structure `termios` et aux vitesses de communication en entrée et en sortie. La fonction `termios.tcsetattr(terminal, option, config)` prend en troisième argument une liste construite sur le même schéma.

Le premier argument de ces fonctions doit être un descripteur de terminal, par exemple `STDIN_FILENO`. La routine `tcsetattr()` – `termio.tcsetattr()` en Python – utilise une option qui précise à quel moment les modifications éventuelles doivent avoir lieu. Il peut s'agir de l'une des constantes suivantes :

Nom	Signification
TCSANOW	Les modifications sont appliquées immédiatement.
TCSADRAIN	Les modifications prendront effet lorsque tous les caractères en attente d'affichage auront été lus par le terminal.
TCSAFLUSH	Comme avec TCSADRAIN, on attend que le contenu du buffer de sortie ait été transmis, mais de plus le buffer d'entrée sera automatiquement purgé.

Tant qu'aucune information n'a été transmise, juste après l'ouverture du terminal, on pourra donc employer `TCSANOW` pour le configurer, mais ensuite on utilisera `TCSAFLUSH` ou `TCSADRAIN` en fonction de l'importance qu'on accorde aux données en attente de lecture.

Le retour de `tcsetattr()` est problématique, car cette fonction n'échoue que si aucune modification n'a pu être apportée à la structure `termios` originale. Pour vérifier que tout s'est bien passé, il est donc nécessaire de relire à nouveau la configuration avec `tcgetattr()` et de comparer le résultat avec les options demandées. L'attitude défensive, voire paranoïaque, à adopter est la suivante :

- lecture de la structure `termios` avec `tcgetattr()` ;
- modification des membres de la structure ;
- écriture des nouveaux attributs avec `tcsetattr()` ;
- lecture de la configuration réellement acceptée avec `tcgetattr()` ;
- comparaison entre les modifications demandées et celles qui ont été obtenues.

L'utilisation de `TCSAFLUSH` permet d'implémenter la fonctionnalité que de trop nombreux débutants espèrent obtenir en invoquant `fflush(stdin)`, qui n'a aucune signification. Il suffit de recopier la configuration du terminal sans la modifier. En voici un exemple :

```
exemple-flush.c :
#include <stdio.h>
#include <stdlib.h>
#include <termios.h>
#include <unistd.h>

int main (void)
{
    struct termios terminal;
    int i;

    fprintf(stdout, "FLUSH dans 5 secondes\n");
    sleep(5);
    fprintf(stdout, "FLUSH !\n");
    if (tcgetattr(STDIN_FILENO, & terminal) == 0)
        tcsetattr(STDIN_FILENO, TCSAFLUSH, & terminal);
    while ((i = fgetc(stdin)) != EOF)
        fprintf(stdout, "%02X ", i);
    return EXIT_SUCCESS;
}
```

Les caractères saisis pendant les cinq secondes d'attente du programme sont purgés lors du `tcsetattr()` suivant :

```
$ ./exemple-flush
FLUSH dans 5 secondes
ceci sera oublié (Entrée)
FLUSH !
azerty (Entrée)
61 7A 65 72 74 79 0A
(Contrôle-C)
$
```

Il existe une fonction `tcflush()` qui assure le même rôle, en permettant de vider au choix le buffer d'entrée ou celui de sortie. En Python, elle s'appelle `termios.tcflush()`.

```
| int tcflush (int terminal, int buffer);
```

Le second argument peut prendre l'une des valeurs suivantes :

Nom	Signification
TCIFLUSH	Purge du buffer d'entrée, comme nous l'avons fait avec <code>tcsetattr()</code> .
TCOFLUSH	Purge du buffer de sortie.
TCIOFLUSH	Purge des deux buffers.

Si on désire s'assurer que toutes les données écrites ont bien été traitées par le terminal, on peut utiliser la fonction `tcdrain()` en C ou `termios.tcdrain()` en Python. Elle est bloquante tant que le buffer de sortie n'est pas vide.

```
| int tcdrain (int descripteur);
```

Attention, si on se sert des fonctions de la bibliothèque `stdio`, il faut penser à employer `fflush(stdout)` avant d'utiliser `tcdrain()`, sinon des données pourraient encore se trouver dans la mémoire tampon interne à la bibliothèque.

On peut aussi vouloir bloquer temporairement l'entrée ou la sortie sur un terminal. Ceci est surtout utilisé dans des applications de communication, afin d'assurer un contrôle de flux entre les deux extrémités. La fonction `tcflow()` permet de gérer ainsi les deux canaux indépendamment :

```
| int tcflow (int terminal, int blocage);
```

Elle dispose du même fonctionnement en Python, sous le nom `termios.tcflow()`. Le second argument peut prendre l'une des valeurs suivantes :

Nom	Signification
TCIOFF	On envoie au terminal un caractère <code>STOP</code> (voir plus loin). Il va donc arrêter d'émettre des données afin que nous puissions traiter le contenu du buffer d'entrée.
TCION	On envoie un caractère <code>START</code> au terminal. Il peut reprendre l'envoi de données.
TCOOFF	On interdit temporairement l'émission de données (suite à la réception d'une requête du terminal). Les tentatives d'écriture seront bloquantes.
TCOON	On débloque l'émission de données. Les écritures pourront reprendre.

La routine `tcflow()` peut être employée dans un gestionnaire de signal. Ceci permet d'implémenter un contrôle de flux. Le gestionnaire peut être invoqué en réception de données, par exemple avec l'une des méthodes de traitement asynchrone étudiées dans le chapitre 25. Lors de l'arrivée de certaines conditions (caractères spéciaux), le gestionnaire peut ainsi bloquer le buffer de sortie. Si des tentatives d'écriture ont lieu dans le corps du programme, il restera bloqué, sinon il se déroulera normalement. Lorsque la condition de déblocage se présentera, le gestionnaire pourra libérer le buffer de sortie et relancer ainsi l'exécution normale du processus.

Notons l'existence d'une dernière fonction de cette famille, `tcsendbreak()`, (`termios.tcsendbreak()` en Python) qui permet d'envoyer un caractère *Break* sur la ligne série du terminal :

```
| int tcsendbreak (int terminal, int duree);
```

Un signal *Break* n'est pas vraiment un caractère mais plutôt une succession de bits à zéro pendant une durée supérieure au temps d'émission d'un caractère. Ce signal est donc reconnu par le périphérique connecté sur la liaison série comme une pseudo-erreur réclamant son attention. L'utilisation dépend ensuite du protocole de haut niveau choisi ; on peut l'employer pour synchroniser le dialogue par exemple. Le second argument représente la durée, mais l'unité n'est pas définie par SUSv4. Aussi est-il conseillé d'employer une valeur nulle, ce qui correspond, de manière portable, à un caractère *Break* qui dure entre un quart et une demi-seconde.

Nous allons à présent examiner les valeurs possibles pour les membres de la structure `termios`. Certaines constantes sont définies par SUSv4, d'autres sont des extensions BSD ou Système V. La liste des attributs est un peu fastidieuse, mais il est utile de bien voir les différents éléments entrant en jeu, principalement en ce qui concerne la table `c_cc[]`. En Python, les constantes seront préfixées par `termios`.

Membre `c_iflag` de la structure `termios`

Ce champ est composé d'un *OU* binaire entre différents arguments, permettant de définir la discipline de ligne pour les caractères provenant du terminal.

Nom	Signification
BRKINT IGNBRK	Lorsqu'un caractère <i>Break</i> arrive, il est ignoré si <code>IGNBRK</code> est présent. Sinon, si <code>BRKINT</code> est actif, il déclenchera le signal <code>SIGINT</code> sur le processus en avant-plan. Si aucun de ces attributs n'est présent, le caractère <i>Break</i> sera lu comme un zéro « <code>\0</code> » présentant une erreur de cadrage (voir <code>PARMRK</code> page suivante).

Nom	Signification
ICRNL IGNCR	Si IGNCR est activé, les caractères de retour chariot sont ignorés en entrée. Si ICRNL est présent, ils sont transformés en saut de ligne. Sinon, ils sont lus comme des caractères « \r ».
IGNPAR PARMRK	Si IGNPAR est actif, un caractère présentant une erreur de cadrage (trop long ou trop court) ou de parité est ignoré. Si PARMRK est présent, le caractère est préfixé du code d'erreur 0xFF, 0x00 (un caractère 0xFF valide est alors indiqué sous forme 0xFF0xFF). Sinon, le caractère est simplement lu comme un zéro « \0 ».
IMAXBEL	Extension BSD et Système V non définie par SUSv4. Avec cet attribut, le système déclenchera un bip sur le terminal lorsque le buffer d'entrée sera plein. La taille du buffer d'entrée est de MAX_INPUT caractères. Cette valeur (255 sous Linux) est définie dans <code><limits.h></code> .
INLCR	Avec cet attribut, un caractère « nouvelle ligne » reçu en entrée sera converti en retour chariot. Sinon, il est lu comme le caractère « \n » habituel.
INPCK	Cet attribut active la vérification de la parité des caractères reçus.
ISTRIP	Si ISTRIP est actif, les caractères reçus en entrée seront tronqués pour tenir sur 7 bits. Le huitième bit est purement et simplement supprimé.
IUCLC	Extension Système V non définie par SUSv4. Un caractère majuscule reçu en entrée est transformé en minuscule. Les caractères sur 8 bits sont ramenés sur 7 bits comme avec ISTRIP.
IXON	Lorsque cet attribut est actif, le terminal assure le contrôle de flux. Les caractères STOP et START définis dans la table <code>c_cc[]</code> que nous verrons plus bas servent à bloquer ou à débloquent l'envoi de données par le processus. Sinon, ces caractères parviennent indemnes au programme.
IXOFF	Avec cet attribut, l'ordinateur gère le contrôle de flux en entrée en envoyant les caractères STOP et START au terminal, en fonction du remplissage du buffer d'entrée.

Membre `c_oflag` de la structure `termios`

OPOST sert à autoriser l'action des autres attributs du champ `c_oflag`.

Noms	Signification
BSDLY BS0, BS1	BSDLY correspond à un masque de bits recouvrant les attributs de configuration du délai de retour en arrière <i>Backspace</i> . Ce masque doit donc être effacé du champ <code>c_oflag</code> avant d'y inscrire l'une des deux valeurs possibles de délai BS0 ou BS1.
CRDLY CR0, CR1, CR2, CR3	CRDLY est le masque de bits correspondant aux délais de retour chariot. Les quatre valeurs possibles vont de CR0 à CR3.
FFDLY FF0, FF1	FFDLY est le masque de délai de saut de page, avec deux valeurs possibles, FF0 et FF1.

Noms	Signification
NLDLY NLO, NL1	Ces attributs représentent le masque et les deux valeurs pour le changement de ligne.
OCRNL ONLCR	Avec OCRNL, les caractères de retour chariot « \r » seront remplacés par des caractères « \n » de nouvelle ligne. Avec ONLCR, la conversion inverse a lieu.
OFDEL	Extension non décrite par SUSv4. Le caractère de remplissage, utilisé pour temporiser les sorties, sera le caractère DEL0x7F si cet attribut est actif. Sinon, il s'agira du caractère nul.
OFILL	Cet attribut demande au pilote de périphérique d'utiliser le remplissage avec un caractère nul ou DEL pour gérer les délais de temporisation. Sinon, le pilote attendra simplement sans rien envoyer.
OLCUC	Extension non décrite par SUSv4. Les caractères minuscules seront transformés en majuscules en sortie avec cet attribut. Les caractères hors de la table Ascii sont tronqués à 7 bits.
ONLRET	Avec l'attribut ONLRET, on suppose que le caractère « \n » effectue également le travail du retour chariot « \r » en sortie.
ONOCR	Si cet attribut est actif, aucun caractère « \r » ne sera envoyé en première position d'une ligne.
TABDLY TAB0, TAB1 TAB2, TAB3 XTABS	Masque et valeurs pour le délai de tabulation horizontale. La valeur XTABS demande au pilote de périphérique de remplacer les tabulations par des espaces, en plaçant les taquets toutes les huit colonnes.
VTDLY VT0, VT1	Masque et valeurs de délai pour la tabulation verticale.

Membre c_cflag de la structure termios

Dans ce champ, on trouve des informations concernant la liaison série entre l'unité centrale et le terminal. Dans le cas où le terminal n'est pas réellement connecté par une liaison série (consoles virtuelles, pseudo-terminaux), ce champ est ignoré.

Il existe quelques extensions BSD sur d'autres systèmes, mais les attributs utilisés sous Linux sont tous définis par SUSv4, à l'exception du contrôle de flux CTS/RTS.

Noms	Signification
CLOCAL	Cet attribut sert à inhiber l'utilisation des lignes de contrôle du modem.
CREAD	Lorsque cet attribut est actif, la réception de données est possible.
CRTSCTS	Extension non définie par SUSv4. Le contrôle de flux avec les signaux RTS/CTS est activé par cet attribut.
CSIZE CS5, CS6, CS7, CS8	CSIZE représente le masque binaire recouvrant les bits utilisés pour définir la longueur des caractères transmis. On emploie les constantes CS5 à CS8 pour fixer la taille du caractère. Nous en verrons un exemple plus tard.

Noms	Signification
CSTOPB	Lorsque cet attribut est actif, les caractères seront délimités par deux bits d'arrêt. Sinon, ils n'auront qu'un seul bit d'arrêt.
HUPCL	Si cet attribut est activé lorsque le descripteur est refermé par le dernier processus qui le tenait ouvert, les signaux de contrôle sont abaissés et le modem est alors raccroché.
PARENB PARODD	L'attribut PARENB active l'utilisation de la parité. Lorsque PARODD est présent, il s'agit d'une parité impaire, sinon c'est une parité paire.

Membre `c_lflag` de la structure `termios`

Ce champ contient la configuration de la partie locale du pilote de périphérique, recouvrant surtout les notions d'écho des caractères saisis et la gestion des signaux provenant du terminal.

Nom	Signification
ECHO	Cet attribut représente l'écho des caractères saisis.
ECHOCTL	Extension BSD et Système V non définie par SUSv4 : écho des caractères de contrôle (inférieurs à 0x1F) sous forme de lettres préfixées du symbole ^.
ECHOE	En mode canonique, le caractère ERASE efface la lettre précédente, et WERASE efface le mot précédent.
ECHOK	En mode canonique, le caractère KILL efface toute la ligne.
ECHONL	En mode canonique, le caractère « \n » est renvoyé en écho même si l'attribut ECHO n'est pas activé.
FLUSHO	Extension BSD et Système V, cet attribut est basculé à la réception du caractère DISCARD. Il indique que le buffer de sortie doit être purgé.
ICANON	Attribut permettant de passer du mode brut au mode canonique.
IEXTEN	Activation du mode étendu dans le traitement des entrées depuis le terminal. Le comportement de certains caractères de contrôle que nous examinerons plus loin dépend de cet attribut.
ISIG	Activation des signaux dus aux caractères INTR, QUIT, SUSP ou DSUSP.
NOFLSH	Cet attribut indique que les buffers d'entrée et de sortie ne doivent pas être vidés lorsque les signaux SIGINT ou SIGQUIT sont déclenchés par le terminal. Ceci s'applique aussi au buffer d'entrée seul, avec le signal SIGSUSP.
PENDIN	Avec cet attribut, qui est une extension BSD et Système V, la frappe d'une touche oblige à réafficher tous les caractères qui ne sont pas encore lus.
TOSTOP	Cet attribut indique qu'un processus en arrière-plan qui tente d'écrire sur son terminal de contrôle recevra le signal SIGTTOU. Nous avons observé ce phénomène dans le chapitre 11.

Membre `c_cc[]` de la structure `termios`

Finalement, ce membre contient une table de NCCS valeurs, représentant pour la plupart des caractères ayant des fonctions particulières. Il existe des constantes pour accéder aux différents indices dans cette table.

Voyons tout d’abord les caractères spéciaux définis par `c_cc[]`.

Nom	Défaut	Signification
DISCARD	<code>^o</code>	Ce caractère peut être configuré dans <code>c_cc[VDISCARD]</code> . Si la saisie étendue <code>IEXTEN</code> est configurée dans <code>c_iflag</code> , ce caractère désactive l’écriture jusqu’à l’arrivée d’un second DISCARD.
EOF	<code>^d</code>	Ce caractère peut être configuré dans <code>c_cc[VEOF]</code> . En mode canonique, il indique la fin d’une saisie. L’appel système <code>read()</code> se termine en renvoyant zéro octet.
EOL		Ce caractère peut être configuré dans <code>c_cc[VEOL]</code> . Il s’agit d’un second caractère de saut de ligne en mode canonique.
EOL2		Ce caractère peut être configuré dans <code>c_cc[VEOL2]</code> . Il s’agit encore d’un caractère supplémentaire indiquant la fin de ligne.
ERASE	<code>^h</code>	Ce caractère peut être configuré dans <code>c_cc[VERASE]</code> . Il permet, en mode canonique, d’effacer le dernier caractère saisi. Si le début de ligne est atteint, il n’a pas d’effet.
INTR	<code>^c</code>	Ce caractère peut être configuré dans <code>c_cc[VINTR]</code> . Si l’attribut <code>ISIG</code> est présent dans <code>c_iflag</code> , le signal <code>SIGINT</code> est envoyé aux processus du groupe en avant-plan au moment de sa réception.
KILL	<code>^u</code>	Ce caractère peut être configuré dans <code>c_cc[VKILL]</code> . En mode canonique, il sert à effacer la ligne en cours.
LNEXT	<code>^v</code>	Ce caractère peut être configuré dans <code>c_cc[VLNEXT]</code> . En mode canonique, avec l’attribut <code>IEXTEN</code> dans <code>c_iflag</code> , il sert à lire la valeur du caractère suivant qui est appuyé sans l’interpréter. Par exemple, la séquence <code>^v ^u</code> affiche le caractère <code>^u</code> sans effacer la ligne.
QUIT	<code>^\</code>	Ce caractère peut être configuré dans <code>c_cc[VQUIT]</code> . Si l’attribut <code>ISIG</code> est actif, le signal <code>SIGQUIT</code> est envoyé à tous les processus du groupe en avant-plan, ce qui par défaut les termine avec un fichier <code>core</code> .
REPRINT	<code>^r</code>	Ce caractère peut être configuré dans <code>c_cc[VREPRINT]</code> . En mode canonique, si <code>IEXTEN</code> est activé dans <code>c_iflag</code> , ce caractère demande le réaffichage de la ligne complète.
START	<code>^q</code>	Ce caractère peut être configuré dans <code>c_cc[VSTART]</code> . Si l’attribut <code>IXON</code> est configuré dans <code>c_iflag</code> , ce caractère sert à redémarrer la lecture si elle était bloquée. Si l’attribut <code>IXOFF</code> est présent dans <code>c_iflag</code> , le pilote envoie automatiquement ce caractère vers le terminal lorsque le buffer d’entrée se trouve à nouveau disponible.

Nom	Défaut	Signification
STOP	^S	Ce caractère peut être configuré dans <code>c_cc[VSTOP]</code> . Il s'agit du caractère complémentaire de <code>START</code> , servant à bloquer les écritures s'il est reçu et si l'attribut <code>IXON</code> est actif. De même, il est envoyé automatiquement au terminal lorsque le buffer d'entrée est plein, si <code>IXOFF</code> est présent.
SUSP	^Z	Ce caractère peut être configuré dans <code>c_cc[VSUSP]</code> . Si l'attribut <code>ISIG</code> est présent dans <code>c_iflag</code> et si le contrôle des jobs est actif, le signal <code>SIGTSTP</code> est envoyé aux processus du groupe en avant-plan, ce qui suspend les processus sans les terminer.
WERASE	^W	Ce caractère peut être configuré dans <code>c_cc[VWERASE]</code> . En mode canonique, avec l'attribut <code>IEXTEN</code> , ce caractère sert à effacer le dernier mot en reculant jusqu'à rencontrer un caractère non blanc, puis en effaçant tous les caractères jusqu'à un caractère blanc.

Il existe deux caractères spéciaux qui ne peuvent pas être modifiés et qui n'ont donc pas réellement d'emplacement dans la table `c_cc[]` :

Nom	Défaut	Signification
CR	\r	Ce caractère peut être converti par le pilote de terminal en mode canonique, en fonction des attributs <code>ICRNL</code> ou <code>IGNCR</code> du membre <code>c_iflag</code> .
NL	\n	En mode canonique, ce caractère sert à délimiter les lignes. En fonction de l'attribut <code>INLCR</code> du champ <code>c_iflag</code> , il peut être converti en \r.

Les caractères `CR`, `EOL`, `EOL2` et `NL` sont les seuls qui peuvent être renvoyés par un appel `read()` dans le mode canonique. Tous les autres caractères de contrôle sont gérés entièrement par le pilote de terminal. Bien entendu, en mode brut, le pilote transmet tous les caractères au processus lecteur.

En plus de ces caractères, la table `c_cc []` contient deux valeurs numériques :

Indice	Signification
VTIME	En mode non canonique, durée maximale d'attente d'un appel <code>read()</code> . Nous détaillerons ceci plus bas.
VMIN	En mode non canonique, nombre minimal de caractères à renvoyer pour qu'un appel <code>read()</code> se termine avant le délai ci-dessus.

Pour tester facilement toutes les options de configuration offertes par la structure `termios`, on peut utiliser le programme `/bin/stty`. Celui-ci travaille sur le descripteur de son entrée standard et affiche l'état de la structure `termios`, ou nous permet de la modifier. Pour consulter tous les attributs `termios`, il faut utiliser `stty -a`.

Même lorsqu'il s'agit de transformer une option, `stty` travaille sur le descripteur de son entrée standard, ce qui nous conduit à écrire des choses comme

```
| $ stty -crtscts < /dev/ttyS0
```

pour supprimer le contrôle de flux matériel sur le premier port série.

En ce qui concerne le terminal « classique », `stty` permet de modifier pratiquement toutes les options `termios` en les faisant figurer sur la ligne de commande, éventuellement précédées d'un « - » pour les désactiver. Il existe une option `sane` qui permet de rétablir le terminal dans un état normal. Pour faire des expériences avec `stty`, il faut éviter que le shell n'interfère, aussi avons-nous deux possibilités :

- Utiliser un shell minimal, qui ne gère pas le clavier de manière trop complète, à la manière de `/bin/ash`.
- Écrire une petite application bouclant en lecture et en exécution de ce type :

```
exemple-system-shell.c :
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
    char ligne[MAX_CANON];
    while (1) {
        fprintf(stderr, "-> ");
        if (fgets(ligne, MAX_CANON, stdin) == NULL)
            break;
        system(ligne);
    }
    return 0;
}
```

On se reportera à la page de manuel de `stty` pour voir toutes ces options.

```
$ ./exemple-system-shell
-> stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = M-^?; eol2 = M-^?;
swtch = M-^?; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
```

```

-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -
ixoff
-iuclc ixany imaxbel iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0
ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echopr
echoctl echoke
-> stty istrip
-> Pression sur éêç :C)C(C'
sh: -c: line 0: Erreur de syntaxe près du symbole inattendu « ) »
sh: -c: line 0: `C)C(C''
-> stty -onlcr
-> ls
  client_pty      exemple_isatty      exemple_serie      Makefile
client_pty.c      exemple_isatty.c      exemple_serie.c      serie_2_stdout
                  exemple_flush      exemple_pty      exemple_socket
      serie_2_stdout.c
                  exemple_flush.c      exemple_pty.c      exemple_socket.c
stdin_2_serie
                                      exemple_getpass
      exemple_raw      exemple-system-shell      stdin_2_serie.c
                                      exemple_getpass.c
      exemple_raw.c      exemple-system-shell.c
                                      -> stty sane
-> ls
client_pty      exemple_isatty      exemple_serie      Makefile
client_pty.c      exemple_isatty.c      exemple_serie.c      serie_2_stdout
exemple_flush      exemple_pty      exemple_socket      serie_2_stdout.c
exemple_flush.c      exemple_pty.c      exemple_socket.c      stdin_2_serie
exemple_getpass      exemple_raw      exemple-system-shell      stdin_2_serie.c
exemple_getpass.c      exemple_raw.c      exemple-system-shell.c
-> (Contrôle-D)
$

```

Basculement du terminal en mode brut

Lorsqu'un terminal est initialisé, il se trouve par défaut en mode canonique. Il est possible de passer en mode brut en lisant la structure `termios` à l'aide de `tcgetattr()`, puis de modifier la structure avant de la réécrire dans le terminal avec `tcsetattr()`.

Le passage en mode brut correspond aux opérations suivantes :

- Effacement des attributs `BRKINT`, `IGNBRK`, `ICRNL`, `IGNCR`, `PARMRK`, `INLCR`, `ISTRIP`, `IXON` de `c_iflag`, `OPOST` de `c_oflag`, `ECHO`, `ECHONL`, `ICANON`, `IEXTEN`, `ISIG` de `c_lflag` et `PARENB` de `c_cflag`.

- Configuration d'une taille de caractère CS8 dans `c_cflag` (après en avoir effacé le masque CSIZE).
- Configuration de `c_cc[VTIME] = 0` et `c_cc[VMIN] = 1`.

Pour simplifier cette opération, on utilise donc une fonction de bibliothèque nommée `cfmakeraw()`, qui réalise ces étapes.

```
void cfmakeraw (struct termios * configuration);
```

Un programme qui passe un terminal en mode brut doit s'assurer de restituer le mode canonique avant de se terminer. Pour cela, il suffit de garder une copie de la structure `termios`.

Lorsqu'on demande une lecture sur un descripteur de terminal en mode brut, le comportement est dicté par les variables `c_cc[VMIN]` et `c_cc[VTIME]`. La première correspond au nombre minimal de caractères qui doivent être lus pour que l'appel `read()` revienne avant que son délai ne soit écoulé. Ce délai est indiqué en dixièmes de seconde dans la seconde variable. Les cas possibles sont donc les suivants :

<code>c_cc[VMIN]</code>	<code>c_cc[VTIME]</code>	Réaction à un <code>read()</code>
0	0	L'appel système <code>read()</code> revient immédiatement. Si des caractères sont disponibles, ils sont renvoyés tout de suite.
> 0	0	L'appel <code>read()</code> ne revient que lorsque <code>VMIN</code> caractères au moins sont disponibles. Sinon, il peut bloquer indéfiniment.
0	> 0	L'appel système ne durera qu'au plus <code>VTIME</code> dixièmes de seconde. Si des caractères arrivent entre-temps, il se termine, sinon il renvoie 0 à l'expiration du délai.
> 0	> 0	L'appel système <code>read()</code> ne reviendra que si <code>VMIN</code> caractères sont reçus ou si le délai de <code>VTIME</code> dixièmes de seconde entre deux caractères est écoulé. Le délai est réinitialisé à l'arrivée de chaque nouveau caractère. Attention, le délai n'est pris en compte qu'après l'arrivée du premier caractère.

Naturellement, dans toutes ces situations `read()` peut se terminer avant l'instant indiqué s'il a lu le nombre de caractères demandés en argument ou si un signal l'interrompt.

En général, on utilisera :

- Soit `c_cc[VMIN]=1` et `c_cc[VTIME]=0` pour avoir une lecture brute bloquante. Ceci permet par exemple d'implémenter un éditeur de texte, il s'agit de la configuration mise en place par `cfmakeraw()`.
- Soit `c_cc[VMIN]=0` et `c_cc[VTIME]=0` pour une lecture non bloquante, afin de laisser le programme se dérouler en parallèle, comme dans un jeu.

Dans le programme suivant, nous allons basculer en mode brut avec lecture non bloquante. En attendant que l'utilisateur saisisse un caractère, le programme fait tourner une barre sur la gauche de la ligne de saisie. À chaque pression sur une touche, le code hexadécimal du caractère correspondant est affiché. Le programme se termine en appuyant sur « q ».

```
exemple-raw.c :
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

struct termios sauvegarde;

int  initialisation_clavier (int fd);
int  restauration_clavier  (int fd);

int main (void)
{
    char c = 0;
    int i = 0;
    char * chaine = "-\\|/";

    initialisation_clavier(STDIN_FILENO);
    while (1) {
        if (read(STDIN_FILENO, & c, 1) == 1)
            if (c == 'q')
                break;
        fprintf(stdout, "\\r%c (%02X)", chaine[i], c);
        fflush(stdout);
        if (chaine[++ i] == '\\0')
            i = 0;
        usleep(100000);
    }
    restauration_clavier(STDIN_FILENO);
    return EXIT_SUCCESS;
}

int initialisation_clavier (int fd)
{
    struct termios configuration;
    if (tcgetattr(fd, & configuration) != 0)
        return -1;
    memcpy(& sauvegarde, & configuration, sizeof(struct termios));
    cfmakeraw(& configuration);
    configuration.c_cc[VMIN] = 0;
```

```
    if (tcsetattr(fd, TCSANOW, & configuration) != 0)
        return -1;
    return 0;
}

int restauration_clavier (int fd)
{
    tcsetattr(fd, TCSANOW, & sauvegarde);
    return 0;
}
```

Nous ne pouvons évidemment pas montrer d'exemple d'exécution ici, puisque l'intérêt de ce programme est son comportement dynamique.

On voit qu'avec ces fonctionnalités, il est possible de définir des routines de saisie dynamique à la volée. Les touches de fonction (*F1*, *F2*...), les flèches de déplacement, les caractères accentués ou les touches de contrôle (*Inser*, *Suppr*, etc.) renvoient des codes composés par plusieurs octets, généralement préfixés par 0x1B (*ESC*). La gestion de ces touches est spécifique au terminal et est peu portable.

Pour utiliser toutes les possibilités d'action d'un clavier de manière portable, il faudra se tourner vers les fonctionnalités *curses*, dont l'implémentation sous Linux est assurée par la bibliothèque *ncurses*. Cette bibliothèque donne accès à toutes les manipulations de texte en plein écran. Il existe de nombreuses applications utilisant *ncurses*, comme *gdb*, *mc*, la configuration de la compilation du noyau Linux, etc.

Avant d'essayer de modifier la configuration d'un descripteur, il convient de vérifier s'il s'agit bien d'un terminal. Pour cela on utilise la fonction `isatty()`, déclarée dans `<unistd.h>` :

```
| int isatty (int descripteur);
```

Cette fonction (ou `os.isatty()` en Python) renvoie une valeur non nulle si le descripteur est bien un terminal. Dans ce cas, il est possible d'employer la fonction `ttynam()` – `os.ttynam()` en Python – pour retrouver le nom du périphérique associé, par exemple `/dev/ttyS1`.

```
| char * ttynam (int descripteur);
```

La chaîne renvoyée se trouve dans une zone de mémoire statique. La bibliothèque Glibc propose également une version réentrante `ttynam_r()` :

```
| int ttynam_r (int descripteur, char * buffer, int longueur);
```

Enfin, il existe une routine nommée `ctermid()` – `os.ctermid()` – permettant de récupérer le nom du terminal de contrôle du processus. Il ne s'agit pas toujours du véritable nom du périphérique mais généralement de `/dev/tty`. Cette chaîne est toujours valide pour ouvrir un descripteur avec `open()`.

```
char * ctermid (char * chaine);
```

Si le pointeur passé en argument est `NULL`, `ctermid()` renvoie une chaîne de caractères allouée dans une zone de mémoire statique. Sinon, il stocke le nom dans le buffer transmis, lequel doit contenir au moins `L_ctermid` caractères.

Voyons quelques possibilités d'utilisation de ces routines.

```
exemple-isatty.c :
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (void)
{
    if (isatty(STDIN_FILENO))
        fprintf(stdout, "stdin : %s\n", ttyname(STDIN_FILENO));
    else
        fprintf(stdout, "stdin : Pas un terminal ! \n");
    fprintf(stdout, "Terminal de controle : %s\n", ctermid(NULL));
    return EXIT_SUCCESS;
}
```

Sur la console virtuelle 2 (accessible par *Ctrl-Alt-F2* depuis l'environnement graphique) :

```
$ ./exemple-isatty
stdin : /dev/tty2
Terminal de controle : /dev/tty
$
```

Dans un terminal X-term ou depuis une connexion distante par SSH :

```
$ ./exemple-isatty
stdin : /dev/pts/6
Terminal de controle : /dev/tty
$
```

L'intérêt essentiel de trouver le nom du terminal de contrôle est de pouvoir effectuer une saisie même si l'entrée standard a été redirigée dans un fichier. La fonction `getpass()`, proposée par la bibliothèque Glibc permet de lire un mot de passe depuis le terminal de contrôle du processus. Cette routine est par exemple employée par `/bin/su`.

```
| char * getpass (const char * invite);
```

Le message transmis en argument est affiché avant de lire le mot de passe. Naturellement, la bibliothèque C supprime l'écho sur le terminal durant la saisie. En Python, cette fonction existe sous le nom `getpass.getpass()`.

Pour connaître ou modifier l'identité du groupe de processus se trouvant en avant-plan sur un terminal, on peut utiliser `tcgetpgrp()` et `tcsetpgrp()` :

```
| pid_t tcgetpgrp (int descripteur);  
| int tcsetpgrp (int descripteur, pid_t groupe);
```

Nous avons examiné les notions de groupe de processus en avant-plan et de terminal de contrôle d'une session dans le chapitre 2.

Connexion à distance sur une socket

Lorsqu'on désire permettre à un utilisateur de se connecter à distance sur une application, il suffit de mettre en place un serveur TCP, comme nous l'avons vu dans un chapitre précédent. On peut écrire les messages à afficher sur le terminal de l'utilisateur dans la socket avec `write()` et lire ses réponses avec `read()`. L'utilisateur pourra se connecter avec `telnet`, par exemple pour administrer le logiciel à distance.

Le programme suivant va tenter d'implémenter tout ceci en utilisant `dup2()` pour rediriger l'entrée et la sortie standard vers la socket de communication, puis en appelant `system()` sur les chaînes de caractères saisies.

```
| exemple-socket.c :  
| #include <limits.h>  
| #include <signal.h>  
| #include <stdio.h>  
| #include <stdlib.h>  
| #include <string.h>  
| #include <unistd.h>  
  
| #include <arpa/inet.h>  
| #include <netdb.h>
```

```
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

void gestionnaire (int numero)
{
    exit(EXIT_SUCCESS);
}

void traite_connexion (int fd)
{
    char chaine[MAX_CANON];
    char * fin;
    FILE * fp;

    if ((fp = fdopen(fd, "r+")) == NULL) {
        perror("fdopen");
        exit(EXIT_FAILURE);
    }

    if (! isatty(fd)) {
        strcpy(chaine, "Vous n'etes pas connecte sur un terminal ! \n");
        write(fd, chaine, strlen(chaine));
    }
    dup2(fd, STDIN_FILENO);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);

    while (1) {
        fprintf(stderr, "--> ");
        if (fgets(chaine, MAX_CANON, fp) == NULL)
            break;
        if ((fin = strpbrk(chaine, "\n\r")) != NULL)
            fin[0] = '\0';
        if (strcasecmp(chaine, "fin") == 0) {
            kill(getppid(), SIGINT);
            exit(EXIT_SUCCESS);
        }
        system(chaine);
    }
    exit(EXIT_SUCCESS);
}

int main (void)
{
    int sock;
    int sock_2;
    struct sockaddr_in adresse;
    socklen_t longueur;
```



```
if (signal(SIGINT, gestionnaire) != 0) {
    perror("signal");
    exit(EXIT_FAILURE);
}

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket");
    exit(EXIT_FAILURE);
}

memset(& adresse, 0, sizeof(struct sockaddr));
adresse.sin_family = AF_INET;
adresse.sin_addr . s_addr = htonl(INADDR_ANY);
adresse.sin_port = 0;

if (bind(sock, (struct sockaddr *) & adresse,
        sizeof(adresse)) < 0) {
    perror("bind");
    exit(EXIT_FAILURE);
}

longueur = sizeof(struct sockaddr_in);
if (getsockname(sock, (struct sockaddr *) & adresse,
        & longueur) < 0) {
    perror("getsockname");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "Mon adresse : IP = %s, Port = %u \n",
        inet_ntoa(adresse.sin_addr),
        ntohs(adresse.sin_port));

listen(sock, 5);
while (1) {
    longueur = sizeof(struct sockaddr_in);
    sock_2 = accept(sock, (struct sockaddr *) &adresse,
        &longueur);

    if (sock_2 < 0)
        continue;
    switch (fork()) {
        case 0 : /* Fils */
            close(sock);
            traite_connexion(sock_2);
            exit(EXIT_SUCCESS);
        default :
            close(sock_2);
            break;
    }
}
```

```
close(sock);  
return EXIT_SUCCESS;  
}
```

Au début, ce système semble fonctionner correctement, mais on s'aperçoit assez vite de ses limites, notamment si on essaye d'invoquer une commande gérant l'écran entier, comme l'éditeur `vi`, ou si on veut saisir un mot de passe caché :

```
$ ./exemple-socket  
Mon adresse : IP = 0.0.0.0, Port = 43646  
$ telnet 192.168.3.3 43646  
Trying 192.168.3.3...  
Connected to 192.168.3.3.  
Escape character is '^['.  
Vous n'etes pas connecte sur un terminal !  
--> ls  
Makefile  
exemple-flush  
exemple-flush.c  
[...]  
exemple_socket  
exemple_socket.c  
-> echo "un message"  
un message  
-> su  
standard in must be a tty  
-> fin  
Connection closed by foreign host.  
$
```

Nous voyons bien que le problème qui se pose est que la liaison par réseau entre `telnet` – qui peut gérer un terminal correctement – et notre serveur n'est pas suffisante pour autoriser le fonctionnement normal d'applications manipulant l'écran en mode brut.

Pour résoudre ce problème, nous devons faire appel à un pseudo-terminal.

Utilisation d'un pseudo-terminal

La notion de pseudo-terminal est souvent un peu confuse. Il s'agit en fait d'un périphérique virtuel géré par le noyau, offrant deux moitiés distinctes :

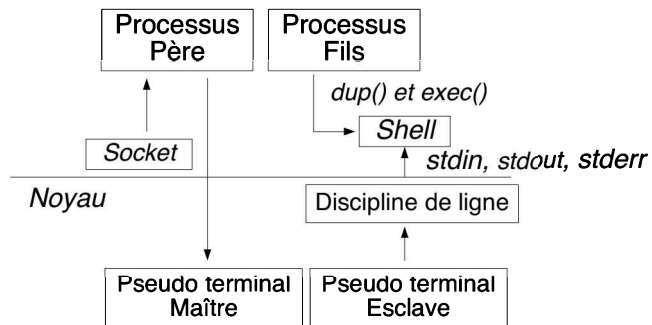
- Le pseudo-terminal maître est un périphérique géré comme un descripteur de fichier habituel, sur lequel on utilise `open()`, `read()`, `write()`, `close()`.

- Le pseudo-terminal esclave est vu par les processus exactement comme un terminal normal. Tout ce qu'on écrit sur le pseudo-terminal maître est lisible sur l'esclave et, inversement, ce qui est écrit sur le pseudo-terminal esclave est immédiatement accessible du côté maître. De plus, le noyau gère une discipline de ligne sur le pseudo-terminal esclave.

Nous allons donc pouvoir mettre en place le mécanisme décrit sur la figure 32-2 pour autoriser des connexions réseau complètes.

Figure 32-2

Utilisation d'un pseudo-terminal pour des connexions externes



Historiquement, l'accès aux pseudo-terminals se faisait en ouvrant directement des descripteurs particuliers : `/dev/ptyXN` pour les terminaux maîtres, et `/dev/ttyXN` pour les pseudo-terminals esclaves, où *X* correspondait à une lettre dans l'ensemble « pqrstuvwxyzabcde » et *N* à un chiffre hexadécimal compris entre 0 et F.

À partir du noyau Linux 2.2, les pseudo-terminals dits devpts ont fait leur apparition, conformément aux spécifications Unix 98. Les pseudo-terminals sont maintenant situés dans `/dev/pts`, à condition qu'une pseudo-partition de type devpts soit montée sur ce répertoire.

Pour ouvrir un pseudo-terminal, on commence par invoquer la routine `getpt()`, une extension Gnu déclarée dans `<stdlib.h>` :

```
| int getpt (void);
```

Cette routine renvoie un descripteur de fichier correspondant au pseudo-terminal maître. Pour l'obtenir, elle demande au noyau d'ouvrir le fichier spécial `/dev/ptmx`. Voyant cela, le noyau lui attribue un descripteur d'un nouveau pseudo-terminal maître. Une fois l'ouverture effectuée, il est nécessaire de modifier les droits d'accès au pseudo-terminal esclave – le seul qui sera vraiment visible dans l'arborescence du sys-

tème de fichiers. Pour cela on appelle la fonction `grantpt()` en lui transmettant le descripteur du côté maître. Cette fonction est définie par SUSv4.

```
| int grantpt (int descripteur);
```

Pour être sûr d'être portable, un programme doit ensuite appeler la routine `unlockpt()`, qui permet de déverrouiller le pseudo-terminal esclave associé au descripteur maître.

```
| int unlockpt (int descripteur);
```

Finalement, pour pouvoir ouvrir le pseudo-terminal esclave, il faut obtenir son nom dans le système de fichiers. Ceci est assuré par la fonction `ptsname()`, qui renvoie un pointeur sur une chaîne de caractères en mémoire statique contenant le nom du pseudo-terminal esclave associé au descripteur maître passé en argument.

```
| char * ptsname (int descripteur);
```

Il existe une extension Gnu réentrante nommée `ptsname_r()` :

```
| int ptsname_r (int descripteur, char * buffer, size_t longueur);
```

Il ne faut pas être effrayé par la multitude de routines à invoquer successivement. L'utilisation des pseudo-terminaux répond à un schéma bien défini, qu'on réutilise directement dans chaque application.

Le principe de notre programme est le suivant :

- Le processus père ouvre un pseudo-terminal maître disponible. Ensuite il passe son pseudo-terminal en mode brut afin de ne pas interférer avec les échanges de données.
- Le fils ouvre le pseudo-terminal esclave associé au maître. Il en fait son terminal de contrôle et le duplique sur l'entrée et la sortie standard. Enfin, il exécute un shell.
- Le processus père copie directement ce qu'il reçoit sur sa socket de communication vers le pseudo-terminal maître, et inversement.

Il faut attirer l'attention sur le fait que ce programme offre, sans vérification d'identité, un accès immédiat à la machine. Ne l'utilisez donc que sur une station se trouvant sur un réseau local sûr ou sur un compte utilisateur expérimental et sans privilèges.

Dans la description ci-dessus, le processus père considéré est déjà obtenu par un `fork()` après l'établissement de la connexion réseau. Nous avons donc deux niveaux

de relations père-fils, mais nous ne nous intéressons ici qu'à la seconde, qui concerne les deux moitiés du pseudo-terminal.

```
exemple-pty.c :
#define _GNU_SOURCE 500

#include <fcntl.h>
#include <limits.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>

void copie_entrees_sorties (int fd, int sock)
{
    int    max;
    fd_set set;
    char   buffer[4096];
    int    nb_lus;

    max = sock < fd ? fd : sock;
    while (1) {
        FD_ZERO(& set);
        FD_SET(sock, & set);
        FD_SET(fd, & set);
        if (select(max + 1, & set, NULL, NULL, NULL) < 0)
            break;
        if (FD_ISSET(sock, &set)) {
            if ((nb_lus = read(sock, buffer, 4096)) >= 0)
                write(fd, buffer, nb_lus);
            else
                break;
        }
        if (FD_ISSET(fd, &set)) {
            if ((nb_lus = read(fd, buffer, 4096)) >= 0)
                write(sock, buffer, nb_lus);
            else
                break;
        }
    }
}
```

```
void traite_connexion (int sock)
{
    int    fd_maitre;
    int    fd_esclave;
    struct termios termios_stdin;
    struct termios termios_maitre;
    char * args [2] = { "/bin/sh", NULL };
    char * nom_esclave;

    if ((fd_maitre = getpt()) < 0) {
        perror("pas de Pseudo TTY Unix 98 disponibles \n");
        exit(EXIT_FAILURE);
    }
    grantpt(fd_maitre);
    unlockpt(fd_maitre);
    nom_esclave = ptsname(fd_maitre);

    tcgetattr(STDIN_FILENO, & termios_stdin);
    switch (fork()) {
        case -1 :
            perror("fork");
            exit(EXIT_FAILURE);
        case 0 : /* fils */
            close(fd_maitre);
            // Détachement du terminal de contrôle précédent
            setsid();
            // Ouverture du pseudo-terminal esclave qui devient
            // alors le terminal de contrôle de ce processus.
            if ((fd_esclave = open(nom_esclave, O_RDWR)) < 0) {
                perror("open");
                exit(EXIT_FAILURE);
            }
            tcsetattr(fd_esclave, TCSANOW, & termios_stdin);
            dup2(fd_esclave, STDIN_FILENO);
            dup2(fd_esclave, STDOUT_FILENO);
            dup2(fd_esclave, STDERR_FILENO);
            execv(args[0], args);
            break;
        default :
            tcgetattr(fd_maitre, & termios_maitre);
            cfmakeraw(& termios_maitre);
            tcsetattr(fd_maitre, TCSANOW, & termios_maitre);
            copie_entrees_sorties(fd_maitre, sock);
            exit(EXIT_SUCCESS);
    }
}
```

```
int main (void)
{
    int          sock;
    int          sock_2;
    struct sockaddr_in adresse;
    socklen_t     longueur;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    signal(SIGCHLD, SIG_IGN);
    memset(& adresse, 0, sizeof(struct sockaddr));
    adresse.sin_family = AF_INET;
    adresse.sin_addr.s_addr = htonl(INADDR_ANY);
    adresse.sin_port = 0;

    if (bind(sock, (struct sockaddr *) & adresse, sizeof(adresse)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }

    longueur = sizeof(struct sockaddr_in);
    if (getsockname(sock, (struct sockaddr *) & adresse, & longueur) < 0) {
        perror("getsockname");
        exit(EXIT_FAILURE);
    }
    printf("Mon adresse : IP = %s, Port = %u \n",
           inet_ntoa(adresse.sin_addr), ntohs(adresse.sin_port));
    listen(sock, 5);
    while (1) {
        longueur = sizeof(struct sockaddr_in);
        sock_2 = accept(sock, & adresse, & longueur);
        if (sock_2 < 0)
            continue;
        switch (fork()) {
            case 0 : /* Fils */
                close(sock);
                traite_connexion(sock_2);
                exit(EXIT_SUCCESS);
            default :
                close(sock_2);
                break;
        }
    }
    close(sock);
    return EXIT_SUCCESS;
}
```

La fonction `main()` reste inchangée par rapport au programme précédent. L'exécution du programme et la connexion réseau se déroulent déjà mieux qu'auparavant :

```
$ ./exemple-pty
Mon adresse : IP = 0.0.0.0, Port = 38040
$ telnet 192.168.3.3 38040
Trying 192.168.3.3...
Connected to 192.168.3.3 (192.168.3.3).
Escape character is '^['.
sh-4.1$ su
su
Mot de passe : (Mot de passe visible !)
# whoami
whoami
root
# exit
exit
sh-4.1$ tty
tty
/dev/pts/2
sh-4.1$
sh-4.1$ exit
exit
$

(Contrôle-C)
$
```

Le terminal de contrôle du shell lancé est bien dirigé vers un pseudo-terminal esclave, `/dev/pts/7` en l'occurrence. Nous remarquons toutefois un problème : les commandes saisies sont répétées avant d'être exécutées. Un second problème se présente avec la saisie du mot de passe qui continue à être visible. Il y a donc un réglage de l'écho local qui n'est pas correct. Quant à l'utilisation d'un programme interactif comme `vi`, elle est très fortement perturbée (même si le processus a l'impression de travailler correctement sur un terminal).

Le véritable problème vient en fait de l'application `telnet`. Celle-ci est conçue pour dialoguer avec le démon `telnetd` et pas avec n'importe quel processus de connexion. Notre programme ne respecte pas le protocole TELNET (décrit entre autres dans la *RFC 854*).

Nous pouvons toutefois obtenir un résultat satisfaisant en écrivant notre propre processus client, qui copiera son entrée standard vers une socket réseau, et inversement copiera le contenu de la socket sur sa sortie standard. Il nous suffit de reprendre le

programme tcp-2-stdout.c du chapitre 27, dont nous modifions quelque peu la fonction main().

```
client-pty.c :
[...]
```

```
int main (int argc, char * argv[])
{
    int                sock;
    struct sockaddr_in adresse;
    char               buffer[LG_BUFFER];
    int                nb_lus;
    struct termios      termios_stdin, termios_raw;

    fd_set set;

    if (lecture_arguments(argc, argv, & adresse, "tcp") < 0)
        exit(EXIT_FAILURE);
    adresse.sin_family = AF_INET;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
    if (connect(sock, (struct sockaddr *) & adresse,
                sizeof(struct sockaddr_in)) < 0) {
        perror("connect");
        exit(EXIT_FAILURE);
    }
    tcgetattr(STDIN_FILENO, & termios_stdin);
    tcgetattr(STDIN_FILENO, & termios_raw);
    cfmakeraw(& termios_raw);
    tcsetattr(STDIN_FILENO, TCSANOW, & termios_raw);

    while (1) {
        FD_ZERO(& set);
        FD_SET(sock, & set);
        FD_SET(STDIN_FILENO, & set);
        if (select(sock + 1, & set, NULL, NULL, NULL) < 0)
            break;
        if (FD_ISSET(sock, & set)) {
            if ((nb_lus = read(sock, buffer, LG_BUFFER)) == 0)
                break;
            write(STDOUT_FILENO, buffer, nb_lus);
        }
        if (FD_ISSET(STDIN_FILENO, & set)) {
            if ((nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER)) == 0)
                break;
            write(sock, buffer, nb_lus);
        }
    }
}
```

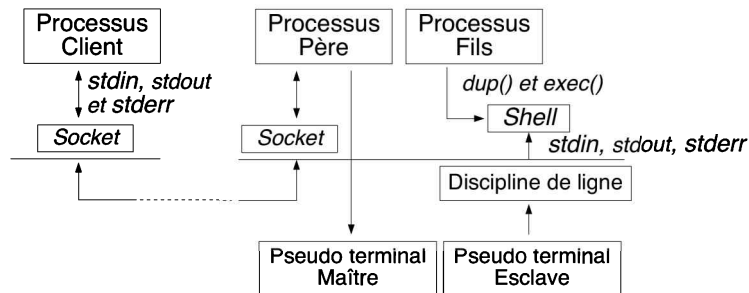
```

    tcsetattr(STDIN_FILENO, TCSANOW, & termios_stdin);
    return EXIT_SUCCESS;
}
[...]
```

Nous avons pris soin de basculer le terminal sur lequel s'exécute le client en mode non canonique, afin de laisser le shell se trouvant à l'autre extrémité de la chaîne responsable de la gestion de l'écran.

Nous avons donc réalisé un arrangement correspondant à la figure 32-3.

Figure 32-3
Organisation de notre
connexion à distance



L'exécution cette fois-ci est parfaitement concluante, tous les programmes fonctionnent normalement, y compris les éditeurs plein écran comme `vi` ou des saisies de mot de passe caché. Le lecteur est encouragé à expérimenter lui-même cette situation, car le comportement est essentiellement dynamique.

On peut obtenir le même comportement avec le script Python suivant. On remarquera l'utilisation de `pty.openpty()`, plus simple que les appels en C pour obtenir les deux terminaux.

```

exemple-pty.py :
#!/usr/bin/python
from __future__ import print_function
import pty
import os
import select
import socket
import sys
import termios

def connection_handler(newsock):
    try:
        (master_fd, slave_fd) = pty.openpty()
```

```
    if os.fork() == 0:
        os.setsid()
        os.close(master_fd)
        termios.tcsetattr(slave_fd, termios.TCSANOW, termios.tcgetattr(0))
        os.dup2(slave_fd, 0)
        os.dup2(slave_fd, 1)
        os.dup2(slave_fd, 2)
        os.execlp("/bin/sh", "/bin/sh")
        sys.exit(1)
except Exception as e:
    print("Error in child:", str(e))

os.close(slave_fd)
while True:
    try:
        (r,w,x) = select.select([master_fd, newsock], [], [])
        if master_fd in r:
            buffer = os.read(master_fd, 4096)
            newsock.send(buffer)
        if newsock in r:
            buffer = newsock.recv(4096)
            os.write(master_fd, buffer)
    except:
        sys.exit(0)

try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('', 0))
    (addr, port) = sock.getsockname()
    print("My port number is", port)
    sock.listen(5)
    while True:
        (newsock, addr) = sock.accept()
        if os.fork() == 0:
            connection_handler(newsock)
            sys.exit(0)
        newsock.close()
except Exception as e:
    print("Error in parent:", str(e))
```

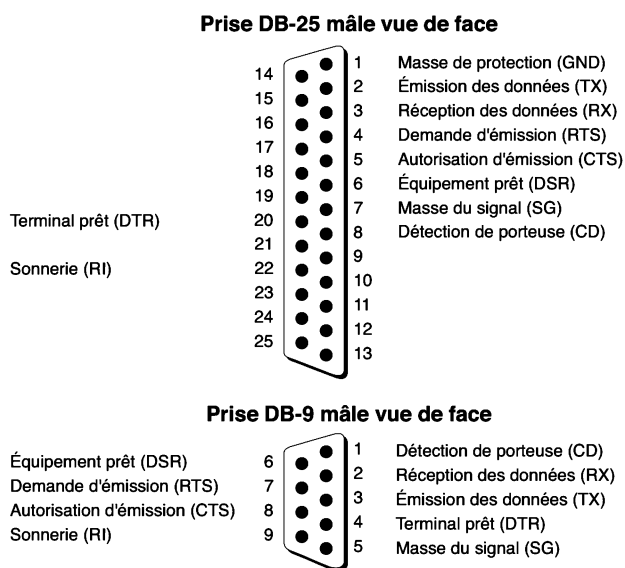
Nous avons pu obtenir un système assez intéressant en utilisant des pseudo-termi-
naux pour offrir une possibilité de connexion distante. Nous allons examiner mainte-
nant le dernier type principal d'utilisation des terminaux Unix : les liaisons série.

Configuration d'un port série RS-232

L'utilisation des ports série sur un PC sous Linux a diminué sensiblement depuis quelques années, car les périphériques « grand public » qui employaient traditionnellement ce port de communication (modems, imprimantes, souris, appareils photographiques numériques, etc.) sont aujourd'hui tous connectés sur les ports USB. Toutefois, le port série est encore largement utilisé pour la connexion de matériel industriel par exemple (cartes d'acquisition, interface avec des automates, des capteurs, des actionneurs...) aussi allons examiner rapidement la configuration de cette liaison. La plupart des connexions se font, du côté du PC, par l'intermédiaire d'un petit adaptateur USB-Série que l'on trouve très aisément dans le commerce.

Les ports série se présentent sur l'ordinateur sous forme de connecteurs mâles à 9 ou 25 broches, dont les noms sont indiqués sur la figure 32-4.

Figure 32-4
Brochage des prises RS-232



L'accès aux ports est possible en utilisant les fichiers spéciaux en mode caractère `/dev/ttyS0`, `/dev/ttyS1...` pour les « vrais » ports série, et `/dev/ttyUSB0`, `/dev/ttyUSB1`, etc... pour les ports accessibles via un adaptateur.

Il existait autrefois des ports `/dev/cua0`, `/dev/cua1`, etc. qui permettait une ouverture non bloquante du port. Ils sont aujourd'hui obsolètes et ont disparu sur la plupart des distributions Linux.

Le premier problème est de trouver quel fichier spécial correspond à tel ou tel connecteur. Cela est évident lorsque les ports sont numérotés sur le panneau arrière du PC, mais lorsqu'une dizaine de machines sont installées en *rack* et reliées à des prolongateurs série de plusieurs mètres passant dans un faux plancher, l'étiquetage devient plus problématique. Pour cela, on peut toutefois utiliser les signaux de contrôle de la ligne. Lors de l'ouverture d'un port série, l'ordinateur élève les signaux RTS et DTR. Nous pouvons donc observer la modification électrique sur le connecteur au moyen d'un voltmètre ou d'un testeur à LED. Les tensions utilisées sur une prise RS-232 sont considérées comme des 0 logiques si elles sont inférieures à -3 V, et comme des 1 logiques si elles sont supérieures à +3 V, ceci par rapport à la borne SG de masse des signaux.

Notre premier programme va donc ouvrir le port indiqué en argument de manière non bloquante, puis le refermer en indiquant au fur et à mesure les tensions qu'on doit mesurer sur le connecteur.

```
exemple-serie.c :
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char * argv[])
{
    int fd;
    char chaine[2];

    if (argc != 2) {
        fprintf(stderr, "%s <fichier special>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    printf("Nous allons verifier la tension entre les broches :\n"
           " 7 (-) et 20 (+) pour un connecteur DB-25 \n"
           " 5 (-) et 4 (+) pour un connecteur DB-9 \n \n ");

    printf("La tension doit etre inferieure a -3 V \n");
    printf("Pressez Entree pour continuer\n");
    fgets(chaine, 2, stdin);

    fd = open(argv[1], O_RDONLY | O_NONBLOCK);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }
}
```

```
printf("La tension doit etre superieure a +3 V\n");
printf("Pressez Entree pour continuer\n");
fgets(chaine, 2, stdin);

printf("La tension doit etre a nouveau < -3 V\n");
if (close(fd) < 0) {
    perror ("close");
    exit(EXIT_FAILURE);
}
return EXIT_SUCCESS;
}
```

Pour mesurer la tension sur un connecteur série à l'aide d'un voltmètre, il est souvent plus facile d'y enficher un connecteur à souder de l'autre genre, sans câblage. Les bornes à souder se trouvant au dos de ce connecteur peuvent accueillir les pointes de mesure du voltmètre en évitant les dérapages constants.

Il faut noter que les fichiers spéciaux de périphériques comme `/dev/ttyS0` disposent d'autorisations d'accès souvent restrictives. Pour continuer nos expériences, il faut modifier les permissions pour donner l'accès à tous les utilisateurs (à éviter sur un système public), ou créer un groupe particulier ayant les droits de lecture et écriture et inscrire dans ce groupe les utilisateurs habilités à manipuler le port série.

À présent que nous avons trouvé le connecteur correspondant à notre fichier spécial, nous allons essayer de transférer des fichiers d'un ordinateur à l'autre. Pour cela il faut configurer les divers éléments de la liaison série :

- La parité est configurée par l'association des options `PARENB` et `PARODD` du membre `c_cflag` de la structure `termios`.
- Le nombre de bits de données est défini par les options `CS5`, `CS6`, `CS7` ou `CS8` du champ `c_cflag`. Avant de fixer une valeur, on efface tous les bits correspondant à ces options à l'aide du masque `CSIZE`.
- Le nombre de bits d'arrêt est fourni par l'option `CSTOPB` du membre `c_cflag`.
- La vitesse de transmission est en réalité contenue dans le champ `c_cflag`, mais on utilise des routines spécialisées pour la lire ou la configurer.

Les fonctions `cfsetispeed()` et `cfsetospeed()` permettent de configurer la vitesse d'entrée ou de sortie dans la structure `termios` passée en argument.

```
int cfsetispeed (struct termios * configuration, speed_t vitesse);
int cfsetospeed (struct termios * configuration, speed_t vitesse);
```

Le type `speed_t` représente la vitesse et peut prendre l'une des valeurs suivantes : `B0`, `B50`, `B75`, `B110`, `B134`, `B150`, `B200`, `B300`, `B600`, `B1200`, `B1800`, `B2400`, `B4800`, `B9600`, `B19200`, `B38400`, `B57600` ou `B115200`. Naturellement, chaque constante représente la

vitesse correspondante mesurée en bits par seconde. La vitesse B0 sert à forcer le raccrochage d'un modem.

Pour lire la vitesse configurée dans une structure `termios`, on peut employer `cfgetispeed()` ou `cfgetospeed()` :

```
speed_t cfgetispeed (struct termios * configuration);
speed_t cfgetospeed (struct termios * configuration);
```

En Python, les vitesses sont directement accessibles dans la liste d'attributs utilisée par ces `termios.tcsetattr()` et `termios.tcgetattr()`. Pour faire dialoguer deux ordinateurs, nous utiliserons le même principe que ce que nous avons élaboré avec les sockets UDP, en transférant sur une liaison série le contenu de l'entrée standard, et inversement depuis la liaison vers la sortie standard.

Le programme suivant va recopier son entrée standard vers un port série indiqué en argument. On commence par ouvrir le fichier spécial de manière non bloquante pour supprimer l'attribut local du port, puis on bascule le descripteur en mode bloquant pour la suite des opérations.

```
stdin-2-serie.c :
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <termios.h>
#include <unistd.h>

#define LG_BUFFER    1024

void setspeed (struct termios * config, speed_t vitesse)
{
    cfsetispeed(config, vitesse);
    cfsetospeed(config, vitesse);
}

int main (int argc, char * argv[])
{
    char * nom_tty          = "/dev/ttyS0";
    int vitesse             = 9600;
    int type_parity         = 'n';
    int nb_bits_donnees     = 8;
    int nb_bits_arret       = 1;
    int fd_tty              = -1;
    struct termios configuration;
    struct termios sauvegarde;
```

```
char    buffer[LG_BUFFER];
int     nb_lus;

int     option;

opterr = 0;
while ((option = getopt(argc, argv, "hv:p:d:a:t:")) != -1) {
    switch (option) {
        case 'v' :
            if ((sscanf(optarg, "%d", & vitesse) != 1)
                || (vitesse < 50) || (vitesse > 115200)) {
                fprintf(stderr, "Vitesse %s invalide \n", optarg);
                exit(EXIT_FAILURE);
            }
            break;
        case 'p' :
            type_parity = optarg[0];
            if ((type_parity != 'n') && (type_parity != 'p')
                && (type_parity != 'i')) {
                fprintf(stderr, "Parite %c invalide \n", type_parity);
                exit(EXIT_FAILURE);
            }
            break;
        case 'd' :
            if ((sscanf(optarg, "%d", & nb_bits_donnees) != 1)
                || (nb_bits_donnees < 5) || (nb_bits_donnees > 8)) {
                fprintf(stderr, "Nb bits donnees %d invalide \n",
                    nb_bits_donnees);
                exit(EXIT_FAILURE);
            }
            break;
        case 'a' :
            if ((sscanf(optarg, "%d", & nb_bits_arret) != 1)
                || (nb_bits_arret < 1) || (nb_bits_arret > 2)) {
                fprintf(stderr, "Nb bits arret %d invalide \n",
                    nb_bits_arret);
                exit(EXIT_FAILURE);
            }
            break;
        case 't' :
            nom_tty = optarg;
            break;
        case 'h' :
            printf("Syntaxe %s [options]... \n", argv [0]);
            printf("Options : \n");
            printf("    -v <vitesse en bits/seconde> \n");
            printf("    -p <parite> (n)ulle (p)aire (i)mpaire \n");
            printf("    -d <bits de donnees> (5 a 8) \n");
            printf("    -a <bits d'arret> (1 ou 2) \n");
            printf("    -t <nom du peripherique> \n");
```



```
        exit(EXIT_SUCCESS);
    default :
        fprintf (stderr, "Option -h pour avoir de l'aide \n");
        exit(EXIT_FAILURE);
    }
}
// Ouverture non bloquante pour basculer en mode non local
fd_tty = open(nom_tty, O_RDWR | O_NONBLOCK);
if (fd_tty < 0) {
    perror(nom_tty);
    exit(EXIT_FAILURE);
}
if (tcgetattr(fd_tty, & configuration) != 0) {
    perror("tcgetattr");
    exit(EXIT_FAILURE);
}
configuration.c_cflag &= ~CLOCAL;
tcsetattr(fd_tty, TCSANOW, & configuration);
// Passage du descripteur en mode bloquant
fcntl(fd_tty, F_SETFL, fcntl(fd_tty, F_GETFL) & ~O_NONBLOCK);
fprintf(stderr, "Port serie ouvert \n");
tcgetattr(fd_tty, & configuration);
memcpy(& sauvegarde, & configuration, sizeof(struct termios));
cfmakeraw(& configuration);
if (vitesse < 50)
    setspeed(& configuration, B50);
else if (vitesse < 75)
    setspeed(& configuration, B75);
else if (vitesse < 110)
    setspeed(& configuration, B110);
else if (vitesse < 134)
    setspeed(& configuration, B134);
else if (vitesse < 150)
    setspeed(& configuration, B150);
else if (vitesse < 200)
    setspeed(& configuration, B200);
else if (vitesse < 300)
    setspeed(& configuration, B300);
else if (vitesse < 600)
    setspeed(& configuration, B600);
else if (vitesse < 1200)
    setspeed(& configuration, B1200);
else if (vitesse < 1800)
    setspeed(& configuration, B1800);
else if (vitesse < 2400)
    setspeed(& configuration, B2400);
else if (vitesse < 4800)
    setspeed(& configuration, B4800);
```

```
else if (vitesse < 9600)
    setspeed(& configuration, B9600);
else if (vitesse < 19200)
    setspeed(& configuration, B19200);
else if (vitesse < 34000)
    setspeed(& configuration, B38400);
else if (vitesse < 57600)
    setspeed(& configuration, B57600);
else
    setspeed(& configuration, B115200);
switch (type_parity) {
    case 'n' :
        configuration . c_cflag &= ~ PARENB;
        break;
    case 'p' :
        configuration . c_cflag |=  PARENB;
        configuration . c_cflag &= ~ PARODD;
        break;
    case 'i':
        configuration . c_cflag |=  PARENB;
        configuration . c_cflag |=  PARODD;
        break;
}
configuration.c_cflag &= ~ CSIZE;
if (nb_bits_donnees == 5)
    configuration.c_cflag |= CS5;
else if (nb_bits_donnees == 6)
    configuration.c_cflag |= CS6;
else if (nb_bits_donnees == 7)
    configuration.c_cflag |= CS7;
else if (nb_bits_donnees == 8)
    configuration.c_cflag |= CS8;
if (nb_bits_arret == 1)
    configuration.c_cflag &= ~ CSTOPB;
else
    configuration.c_cflag |=  CSTOPB;

configuration.c_cflag &= ~ CLOCAL;
configuration.c_cflag |=  HUPCL;

// Contrôle de flux CTS/RTS spécifique Linux
configuration.c_cflag |=  CRTSCTS;

if (tcsetattr(fd_tty, TCSANOW, & configuration) < 0) {
    perror("tcsetattr");
    exit(EXIT_FAILURE);
}
fprintf(stderr, "Port serie configure \n");

fprintf(stderr, "Debut de l'envoi des donnees \n");
```

```
while (1) {
    nb_lus = read(STDIN_FILENO, buffer, LG_BUFFER);
    if (nb_lus < 0) {
        perror("read");
        exit(EXIT_FAILURE);
    }
    if (nb_lus == 0)
        break;
    write(fd_tty, buffer, nb_lus);
}
fprintf(stderr, "Fin de l'envoi des donnees \n");
sleep(2);
close(fd_tty);
// Restauration de la configuration originale
fd_tty = open(nom_tty, O_RDWR | O_NONBLOCK);
sauvegarde.c_cflag |= CLOCAL;
tcsetattr(fd_tty, TCSANOW, & sauvegarde);
close(fd_tty);
return EXIT_SUCCESS;
}
```

Dans le programme `serie-2-stdout.c`, seule la partie centrale de la routine `main()` a été modifiée :

```
serie-2-stdout.c :
[...]
int main (int argc, char * argv [])
{
    [...]
    fprintf(stderr, "Port serie configure \n");

    fprintf(stderr, "Debut de la reception des donnees \n");
    while (1) {
        nb_lus = read(fd_tty, buffer, LG_BUFFER);
        if (nb_lus < 0) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        if (nb_lus == 0)
            break;
        write(STDOUT_FILENO, buffer, nb_lus);
    }
    fprintf(stderr, "Fin de la reception des donnees \n");
    close(fd_tty);
    // Restauration de la configuration originale
    fd_tty = open(nom_tty, O_RDWR | O_NONBLOCK);
    sauvegarde.c_cflag |= CLOCAL;
    tcsetattr(fd_tty, TCSANOW, & sauvegarde);
}
```

```

    close(fd_tty);
    return EXIT_SUCCESS;
}

```

Nous avons également ajouté un délai maximal de cinq secondes entre deux `read()`, pour détecter la fin de transmission lorsque les lignes RTS et CTS ne sont pas correctement câblées. Ceci est réalisé avec :

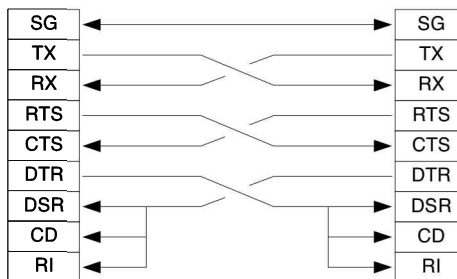
```

configuration.c_cc[VMIN] = 0;
configuration.c_cc[VTIME] = 50;

```

Pour se servir de ces utilitaires, il faut disposer d'un câble de liaison, dit *Null-Modem*, qui croise les lignes de données et de contrôle, et simule la détection de porteuse lorsque l'autre ordinateur est prêt. La figure 32-5 montre un exemple de ce câble, qu'on trouve parfois sous le nom de connexion DTE-DTE complète.

Figure 32-5
Câble Null-Modem complet



Voici un exemple d'exécution sur deux ordinateurs reliés par un câble *Null-Modem* entre deux adaptateurs USB-Série :

```

$ ./stdin-2-serie -t /dev/ttyUSB0 < stdin-2-serie.c
Port serie ouvert
$ ./serie-2-stdout -t /dev/ttyUSB0 > fichier-recu
Port serie ouvert
Port serie configure
Debut de la reception des donnees
Port serie configure
Debut de l'envoi des donnees
  (quelques secondes s'écoulent)
Fin de l'envoi des donnees
  Fin de la reception des donnees
$ ls fichier*

```

```
fichier-recu
$ md5sum fichier-recu
9dd3519f4927a1ac62052ba685dedf37 fichier-recu
$
$ md5sum stdin-2-serie.c
9dd3519f4927a1ac62052ba685dedf37 stdin-2-serie.c
$
```

L'utilitaire `md5sum` invoqué ici calcule une somme de contrôle représentant une sorte de signature du fichier indiqué. On vérifie ainsi aisément que les données ont bien été transmises d'un ordinateur à l'autre. Nos deux programmes ne représentent que des exemples très simplifiés pour le transfert de fichiers. Si on voulait en faire de véritables outils sérieux, il faudrait vérifier les conditions d'erreur à la réception et utiliser un protocole permettant de demander à l'émetteur de renvoyer à nouveau un paquet de données erroné.

Conclusion

Nous avons examiné dans ce dernier chapitre les aspects les plus utiles de la gestion des terminaux. En ce qui concerne l'utilisation de fonctionnalités étendues pour le terminal (édition plein écran, etc.), on emploiera de préférence la bibliothèque `ncurses`, qui offre de nombreuses possibilités et sait gérer l'essentiel des terminaux courants.

Les lecteurs désireux d'approfondir le sujet sur les pseudo-terminaux pourront se tourner vers [Stevens 1993] *Advanced Programming in the UNIX Environment*.

Pour les liaisons série, on trouvera des renseignements dans de nombreux ouvrages, en particulier dans [Nelson 1994] *Communications série, guide du développeur C++*, même s'il est plutôt orienté vers le monde Dos. La configuration de ces liaisons pour des terminaux Unix est abordée dans [Frisch 2003] *Les bases de l'administration système*.

Bibliographie

Standards

On trouvera la norme SUSv4 (*Single Unix Specifications version 4*) sur le site de l'association *Open Group* :

<http://www.unix.org/version4/>

Les RFC (*request for comments*) sont disponibles sur de nombreux sites. Entre autres, celui de l'IETF :

<http://www.ietf.org/rfc.html>

Livres et articles

[Bach 1989] Maurice J. Bach – *Conception du système Unix* – Masson, Prentice-Hall, 1989 (titre original *The Design of the UNIX Operating System*).

[Bentley 1989] Jon Bentley – *Programming Pearls* – Addison-Wesley, 1989.

[Blaess 2007] Christophe Blaess – *Scripts Shell Unix et Linux par la pratique* – Eyrolles, 2007.

[Blaess 2001a] Christophe Blaess, Christophe Grenier, Frédéric Raynal – *Éliminer les failles de sécurité dès le développement d'une application* – Linux Magazine France numéros 23 à 28, décembre 2000 à mai 2001. (disponible sur <http://www.blaess.fr/christophe/>).

[Blaess 2015] Christophe Blaess – *Solutions temps réel sous Linux* – Eyrolles, 2015.

[Butenhof 1997] David Butenhof – *Programming with POSIX Threads* – Addison Wesley, 1997.

- [Cassagne 1998] Bernard Cassagne – *Introduction au langage C* – http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html.
- [Cheswick 1991] Bill Cheswick – *An Evening With Berferd, in Which a Hacker Is Lured, Endured, and Studied* – <http://cm.bell-labs.com/who/ches/papers/>.
- [Ferrero 1993] Alexis Ferrero – *Les réseaux Ethernet* – Addison-Wesley, 1993.
- [Frisch 2003] AËlen Frisch – *Les bases de l'administration système* – O'Reilly & Associates, troisième édition 2003 (titre original *Essential System Administration*).
- [Gallmeister 1995] Bill O. Gallmeister – *Posix.4 Programming for the Real World* – O'Reilly & Associates, 1995.
- [Kernighan 1994] Brian W. Kernighan et Denis M. Ritchie – *Le langage C* – Masson, Prentice-Hall, 1994. Traduction française par Jean-François Groff et Éric Mottier (titre original *The C Programming Language*).
- [Kerrisk 2010] Michael Kerrisk – *The Linux Programming Interface* – No Starch Press 2010.
- [Knuth 1973a] Donald E. Knuth – *The Art of Computer Programming – Fundamental Algorithms* – volume 1, Addison-Wesley Publishing Company, 1973.
- [Knuth 1973b] Donald E. Knuth – *The Art of Computer Programming – Seminumerical Algorithms* – volume 2, Addison-Wesley Publishing Company, 1973.
- [Knuth 1973c] Donald E. Knuth – *The Art of Computer Programming – Sorting and Searching* – volume 3, Addison-Wesley Publishing Company, 1973.
- [Kœnig 1992] Andrew Kœnig – *Pièges du langage C* – Addison-Wesley, 1992 (titre original *C Traps and Pitfalls*).
- [Levine 1994] John R. Levine, Tony Mason et Doug Brown – *lex & yacc* – O'Reilly & Associates, 1994 (titre original *lex & yacc*).
- [Lewine 1994] Donald Lewine – *Posix Programmer's Guide* – O'Reilly & Associates, 1994.
- [Love 2003] Robert Love – *Linux Kernel Development* – Developer's Library, 2003.
- [McConnell 1994] Steve McConnell – *Programmation professionnelle* – Microsoft Press, 1994 (titre original *Code Complete*).
- [Maguire 1995] Steve Maguire – *L'art du code* – Microsoft Press, 1995 (titre original *Writing Solid Code*).
- [Minoux 1986] Michel Minoux et Georges Bartnik – *Graphes, algorithmes, logiciels* – Dunod informatique, 1986.
- [Nelson 1994] Mark Nelson – *Communications série, guide du développeur C++* – Dunod, 1994 (titre original *Serial Communication, A C++ Developer's Guide*).

- [Newham 1995] Cameron Newham et Bill Rosenblatt – *Le shell Bash* – O'Reilly & Associates, 1995 (titre original *Learning the Bash Shell*).
- [Nichols 1996] Bradford Nichols, Dick Buttlar et Jacqueline Proulx Farrell – *Pthreads Programming* – O'Reilly & Associates, 1996.
- [Press 1993] William H. Press et al. – *Numerical Recipes in C, the Art of Scientific Computing* – Cambridge University Press, 1993.
- [Raymond 2003] Eric S. Raymond – *The Art of UNIX Programming* – Addison-Wesley, 2003.
- [Stevens 1990] W. Richard Stevens – *Unix Network Programming* – Prentice-Hall, 1990.
- [Stevens 1993] W. Richard Stevens – *Advanced Programming in the Unix Environment* – Addison-Wesley, 1993.
- [Tanenbaum 2008] Andrew S. Tanenbaum et Albert S. Woodhull – *Operating Systems, Design and Implementation* – Prentice-Hall, troisième édition, 2008.
- [Torvalds 2001] Linux Torvalds, David Diamond – *Il était une fois Linux* – Osman Eyrolles Multimédia, 2001 (titre original *Just For Fun*).

Index

/bin/chrt 296
/bin/kill 310, 699
/bin/ln 896, 897
/bin/mknod 904
/bin/mount 934
/bin/nohup 308
/bin/renice 280
/bin/stty 309, 585, 992
/bin/sync 618, 776
/bin/taskset 287, 297
/dev/pts 1003
/dev/shm 724
/etc/fstab 479, 684, 925
/etc/group 48, 912
/etc/mstab 928
/etc/passwd 915
/etc/services 793
/etc/shells 918
/proc 849
/proc/self/ 849
/sbin/ifconfig 82, 783
/sbin/iptables 784
/sbin/route 784
/usr/bin/gencat 955
/usr/bin/ipcrm 731
/usr/bin/ipcs 731
/usr/bin/mkfifo 703
/usr/bin/mtrace 408
/usr/bin/top 923
/usr/bin/uptime 923
__free_hook 413
__malloc_hook 413
__realloc_hook 413
_BSD_SOURCE 50
_edata 386
_end 386, 406
_etext 386
_exit() 123, 130

_IOFBF 637
_IOLBF 637
_IONBF 637
_NSIG 302
_POSIX_ASYNCHRONOUS_IO
765
_POSIX_MEMLOCK 425
_POSIX_MEMLOCK_RANGE
425
_POSIX_PRIORITY_SCHEDUL
ING 288
_POSIX_REALTIME_SIGNALS
303, 314
_POSIX_SYNCHRONIZED_IO
777
_REENTRANT 156
_SC_CLK_TCK 254, 255
_SC_PAGESIZE 426
_XOPEN_SOURCE 333

A

abort() 124, 304
accept() 809
access() 889, 901
addmntent() 930
adjtime() 215
adjtimex() 214, 215
AES 529
AF_INET 800, 802
AF_INET6 800, 802
AF_UNIX 800
affinité 284
AI_CANONNAME 788
AI_NUMERICHOST 787
AI_NUMERICSERV 787
AI_PASSIVE 788
aio_cancel() 775
aio_error() 768, 769, 771

aio_fsync() 778
aio_read() 766, 770
aio_return() 768, 769, 771
aio_suspend() 773, 775
aio_write() 766
alarm() 237, 305, 321, 360, 361
alarme 230
alloca() 399, 458
alphasort() 845
année 216, 221, 222
annulation 165, 183, 194
point d'~ 186
Ansi VI, 535
appel-système 2
arbre binaire 489, 499, 500, 502,
513
argc 101, 103
argument 101
Ascii 452, 463, 464
asctime() 219, 220, 226
asctime_r() 219
assert() 124, 125, 146, 152, 397
atexit() 122, 127, 129
atoff() 544
atoi() 544
atol() 544
atoll() 544
attach (commande Gdb) 273
attributs de thread 174, 200

B

basename() 474
Bash 29, 94, 264, 343, 699
bcmp() 452
bcopy() 450
bibliothèque 127
big endian 795, 796, 797
bind() 803, 819, 1001

- bindtextdomain() 960
- Bison 478, 608
- blocage de signal 348, 354
- brk() 386
- broadcast 831
- BSD 799
- bsearch() 492
- bss 386
- btowc() 466
- BUFSIZ 636, 638
- Busybox 896
- bzero() 447
- C**
- c_flag 989
- c_iflag 987
- c_lflag 990
- c_oflag 988
- cabs() 553
- calloc() 391, 392, 393, 404
- CAP_CHOWN 60, 890
- CAP_DAC_OVERRIDE 60
- CAP_DAC_READ_SEARCH 61
- CAP_FOWNER 61
- CAP_FSETID 61
- CAP_IPC_LOCK 61, 423
- CAP_IPC_OWNER 61, 727
- CAP_KILL 61, 318
- CAP_LEASE 61
- CAP_LINUX_IMMUTABLE 61
- CAP_MKNOD 61
- CAP_NET_ADMIN 61
- CAP_NET_BIND_SERVICE 61, 793
- CAP_NET_BROADCAST 61
- CAP_NET_RAW 61, 801
- CAP_SETGID 61, 915
- CAP_SETPCAP 61
- CAP_SETUID 61
- CAP_SYS_ADMIN 61, 920, 934
- CAP_SYS_BOOT 61
- CAP_SYS_CHROOT 61, 854
- CAP_SYS_MODULE 61
- CAP_SYS_NICE 61, 277, 282, 296
- CAP_SYS_PACCT 61
- CAP_SYS_PTRACE 61
- CAP_SYS_RAWIO 61
- CAP_SYS_RESOURCE 62, 260
- CAP_SYS_TIME 62, 214
- CAP_SYS_TTY_CONFIG 62
- capacités 60
- capget() 62
- capset() 62
- caractère accentué 464, 468, 541, 587
- caractère large 465, 466, 467, 470
- carg() 553
- catclose() 955, 956
- catgets() 955, 956
- catopen() 953, 956
- cbc_crypt() 531
- ccos() 554
- ccosf() 554
- ccosl() 554
- ceil() 555, 556
- ceilf() 555
- ceill() 555
- cfgetispeed() 1015
- cfgetospeed() 1015
- cfmakeraw() 995, 996, 1006, 1009, 1017
- cfsetispeed() 1014, 1015
- cfsetospeed() 1014, 1015
- chaîne de caractères 445, 452, 455, 456, 458, 460, 461, 463, 478, 543
- chaîne de caractères larges 467
- chdir() 847, 848, 854, 901
- chmod 42
- chmod() 884, 888, 901
- chown 42
- chown() 890, 901
- chroot() 854
- cimag() 553
- classe de réseau 785
- clavier 309, 311
- CLD_CONTINUED 143
- CLD_EXITED 143
- CLD_KILLED 143
- CLD_STOPPED 142, 143
- clé 203, 204, 726
- clearenv() 96
- clearerr() 641
- client TCP 814
- CLOCK 249
- clock() 252, 253
- CLOCK_BOOTTIME 212
- clock_gettime() 211
- clock_getres() 212
- CLOCK_MONOTONIC 212, 247
- CLOCK_MONOTONIC_COARSE 212
- CLOCK_MONOTONIC_RAW 212
- CLOCK_PROCESS_CPUTIME_ID 212, 247
- CLOCK_REALTIME 212, 247
- CLOCK_REALTIME_COARSE 212
- clock_settime() 212
- clock_t 253, 254, 255
- CLOCK_THREAD_CPUTIME_ID 212, 247
- clockid_t 247
- CLOCKS_PER_SEC 253, 254
- clone() 155
- close() 647, 648, 692, 694, 819
- closedir() 842, 843
- closelog() 942, 944
- close-on-exec 73, 671, 673
- commande
 - ligne de ~ 101, 110
- communication 688, 743
- comparison_fn_t 486
- compilateur 4, 10
- complex 552, 553, 554
- condition (variable) 191
- conj() 553
- connect() 814, 818, 823
- core 15, 124, 136, 264, 303, 304, 306
- cos() 554
- cosf() 554
- cosl() 554

CPU 269, 274, 283, 287
 CPU_CLR() 285
 CPU_ISSET() 285
 CPU_SET() 285
 cpu_set_t 285
 CPU_ZERO() 285
 creal() 553
 creat() 644
 crypt() 525, 526, 527, 528
 crypt_r() 528
 cryptage 524, 525, 526, 527, 529, 530
 ctermid() 998
 ctime() 219, 220
 ctime_r() 219
 cuserid() 917
D
 datagram 786
 date 208, 216, 221, 779
 DATEMSK 227
 dates d'un fichier 894
 daytime 818
 deadlock 181, 197
 débogueur 5, 13, 312
 débordement de buffer 414, 416
 débordement de pile 308
 démon 56, 308, 835, 837
 DES 526, 528, 529
 DES_DECRYPT 530
 DES_ENCRYPT 530
 des_setparity() 531, 533
 descripteur 320, 563, 611, 619, 642, 643, 658, 659, 666, 669, 672, 689, 750, 763, 799, 847
 devpts 1003
 difftime() 231
 Dijkstra 720
 DIR 842, 843, 844
 dirent 842, 843
 discipline de ligne 982
 dnotify 906
 données privées 202
 dup() 669, 670
 dup2() 669, 671, 694, 695, 696

E
 E2BIG 74, 144
 EACCES 74, 144, 801
 EACCESS 804
 EADDRINUSE 804, 819, 830
 EAGAIN 38, 144, 655, 662, 663, 745
 EBADF 144
 EBUSY 144, 178, 191, 857, 862
 ecb_crypt() 530, 533
 ECHILD 136, 144
 Eclipse 5, 14, 18
 ecvt() 549, 550
 ecvt_r() 552
 EDEADLK 144, 680
 EDOM 144, 557, 560
 EEXIST 144, 857
 EFAULT 74, 144
 EFBIG 144, 261, 661
 EINPROGRESS 768, 769
 EINTR 144, 331, 352, 662, 663
 EINVAL 74, 144
 EIO 74, 144, 312, 656, 660
 EISDIR 74, 144, 862
 ELIBBAD 74
 ELOOP 74, 144
 Emacs 8
 EMFILE 74, 144
 EMLINK 144
 ENAMETOOLONG 74, 144, 919
 endfsent() 926
 endgrent() 914
 endmntent() 928, 929, 931
 endpwent() 916
 endusershell() 918
 endutent() 936
 endutxent() 939
 ENFILE 74, 144
 ENODEV 144
 ENOENT 74, 144
 ENOEXEC 74, 144
 ENOLCK 145
 ENOMEM 38, 74, 145
 ENOSPC 145, 661, 857

ENOSYS 145, 709
 ENOTBLK 145
 ENOTDIR 74, 145
 ENOTEMPTY 145, 857, 862
 ENOTSOCK 804
 ENOTTY 145
 entrée standard 564
 environ 66, 67, 90, 92
 environnement 89, 110
 variable d'~ 89, 92, 98, 110, 386
 ENXIO 145, 703
 EOF 537, 584, 597, 698
 EOPNOTSUPP 526
 EPERM 74, 145, 318
 EPIPE 145, 310, 662, 699, 823
 epoll 757
 EPOLL_CLOEXEC 758
 epoll_create() 757
 epoll_create1() 758
 epoll_ctl() 758
 epoll_pwait() 758
 epoll_wait() 758
 EPROTONOSUPPORT 801
 ERANGE 145, 560
 EROFS 145
 errno 38, 143, 156, 356, 526, 557
 espace d'adressage 154, 155, 163, 172, 200, 276, 386, 387, 426, 732
 ESPIPE 145
 ESRCH 145, 184, 318
 Ethernet 782
 ETIMEDOUT 194
 ETXTBSY 74, 75, 76, 145, 428
 EWOULDBLOCK 145
 EXDEV 145, 862
 exec() 56, 60, 65, 423
 execl() 66
 execlp() 66, 73
 execlp() 66, 71, 72, 673
 execv() 66, 69, 76, 78
 execve() 66, 67, 68, 758, 901
 execvp() 66, 70
 exit() 74, 121, 122, 127, 132, 166

EXIT_FAILURE 121
 EXIT_SUCCESS 121, 122
 exp() 559
 export (commande shell) 94
 expression rationnelle 515, 516
 ext4 842

F

F_DUPFD 672
 F_GETFD 672, 673
 F_GETFL 675
 F_GETLK 677, 679
 F_GETOWN 676, 764
 F_GETSIG 676, 764
 F_RDLCK 677, 681
 F_SETFD 672, 673
 F_SETFL 675, 744, 747
 F_SETLK 677, 678, 684
 F_SETLKW 677, 678, 679, 681
 F_SETOWN 676, 763
 F_SETSIG 676, 764
 F_UNLCK 677, 679
 F_WRLCK 677, 681
 fchdir() 848, 850, 851
 fchmod() 888
 fchown() 890
 fclose() 615, 616, 617, 621, 625
 fcloseall() 617
 fcntl() 671, 673, 675, 677, 678, 679, 744, 747, 763, 764
 fcvt() 550
 fcvt_r() 552
 FD_CLOEXEC 672, 673
 FD_CLR() 752
 FD_ISSET() 752, 753, 762
 fd_set 752
 FD_SET() 752, 761
 FD_SETSIZE 751, 753
 FD_ZERO() 752, 761
 fdatsync() 777
 fdopen() 619, 620, 651, 653, 703, 706, 1000
 feof() 640, 641
 ferror() 640, 641
 fflush() 617, 634, 635
 fgetc() 584

fgetgrent() 914
 fgetgrent_r() 914
 fgetpos() 633
 fgetpwent() 917
 fgetpwent_r() 917
 fgets() 594, 596, 603, 605
 fgetwc() 586
 fgetws() 595
 fichier 425, 563, 611, 644, 654, 677, 841, 842, 883
 fichier temporaire 863, 864
 Fifo 702, 708, 902
 FIFO (ordonnancement) 288, 291, 292
 FILE 563, 612, 615
 file de messages 710, 728
 fileno() 642
 Find 23
 finite() 558
 Flex 478, 608
 flock() 677, 685
 floor() 555, 556
 floorf() 555
 floorl() 555
 flux 81, 563, 564, 611, 612, 613
 FNM_CASEFOLD 866
 FNM_FILE_NAME 866
 FNM_LEADING_DIR 866
 FNM_NOESCAPE 866
 FNM_NOMATCH 865
 FNM_PATHNAME 866
 FNM_PERIOD 866
 fnmatch() 865, 866
 fopen() 613, 614, 615, 621, 625, 651, 703
 fork() 36, 38, 56, 65, 78, 132, 150, 201, 202, 693
 fpos_t 633
 fprintf() 568, 570, 571, 572, 574, 576, 967
 fputc() 580, 581
 fputs() 582, 583
 fputwc() 580
 fputws() 582
 fread() 622, 625

free() 392, 395, 396, 397, 398, 402
 freeaddrinfo() 787
 freopen() 620, 621
 fscanf() 598, 601
 fseek() 627, 641
 fseeko() 629, 630, 631
 fsetpos() 633, 641
 FSIZE 663
 fstat() 430, 431, 674, 701, 885, 887
 fstatfs() 931
 fsync() 660, 776, 779
 ftell() 627, 628
 ftello() 629, 630, 631
 ftok() 726, 740
 ftruncate() 715, 716, 893, 894
 FTW 878, 880
 ftw() 878
 fuseau horaire 217, 222, 231, 232
 fwrite() 622, 625

G

Gcc 10
 gcvt() 550
 Gdb 13, 273, 312
 Gdbserver 15
 gestionnaire de signal 301, 309, 322, 323, 324, 335, 338, 344, 354, 355, 363, 369, 370, 378, 764
 get_current_working_dir_name() 848
 getaddrinfo() 787, 788, 794
 getc() 590
 getchar() 586
 getcwd() 848, 851
 getdate() 227, 228
 getdate_err 227, 228
 getdate_r() 227
 getdomainname() 920
 getegid() 48
 getenv() 92, 93, 97, 113
 geteuid() 41, 42, 45, 46
 getfsent() 927
 getfsfile() 927

- getfsspec() 927
- getgid() 48
- getgrent() 913
- getgrent_r() 913
- getgrgid() 913
- getgrgid_r() 913
- getgrnam() 913
- getgrnam_r() 913
- getgroups() 48, 49
- gethostid() 920
- gethostname() 919
- getitimer() 243
- getline() 596, 597, 607
- getlogin 86
- getlogin() 85, 917
- getlogin_r() 917
- getmntent() 928, 931
- getmntent_r() 928, 929
- getnameinfo() 789, 794
- getopt() 103, 104, 114, 520
- getopt_long() 106, 107, 108, 114
- getopt_long_only() 109
- getpass() 999
- getpeername() 806
- getpgid() 52
- getpgrp() 54, 55
- getpid() 36, 38, 55, 159
- getppid() 37, 38
- getpriority() 280, 281, 298
- getpt() 1003, 1006
- getpwent() 916
- getpwent_r() 916
- getpwnam() 916
- getpwnam_r() 916
- getpwuid() 916
- getpwuid_r() 916
- getresgid() 52
- getresuid() 47
- getrlimit() 149, 260, 664
- getrusage() 257, 258
- gets() 591, 592
- getsid() 57
- getsockname() 805, 1001
- getsockopt() 829
- getsubopt() 109, 115
- GetText 958
- gettext() 958, 960
- gettimeofday 756
- gettimeofday() 209, 210, 211, 241
- getuid() 41, 42, 45, 46, 854
- getusershell() 918
- getutent() 936, 938, 940
- getutent_r() 936
- getutid() 936
- getutid_r() 936
- getutline() 937
- getutline_r() 937
- getutxent() 939
- getutxid() 939
- getutxline() 939
- getw() 626
- getwchar() 586, 588
- getwd() 848
- GID 48, 73
- gid_t 48, 49
- Glibc VI, 3
- glob() 868, 871, 872
- glob_t 868
- globfree() 868, 872
- gmtime() 217, 218
- gmtime_r() 217
- Gnome 9, 30
- Gnu 478
- Gprof 19
- grantpt() 1004, 1006
- Grep 23, 516
- groupe
 - d'utilisateurs 48
 - de processus 52, 280
- GTK 31
- H**
- hard realtime 293
- hasmntopt() 930, 931
- hcreate() 507, 509
- hcreate_r() 507, 511
- hdestroy() 507, 510
- hdestroy_r() 508, 511
- heure 206, 208, 216, 221
 - d'été 232
- HOME 99, 100
- horaire 216
- horloge 206, 215, 231, 247, 253
- horodatage 205, 206
- hsearch() 508, 509
- hsearch_r() 508, 511
- htonl() 797
- htons() 797
- HUGE_VAL 560
- I**
- ICMP 785
- id 41
- identifiant
 - d'utilisateur 40
 - de groupe (d'utilisateurs) 48
 - de groupe (de processus) 52
 - de processus 36
 - de session 57
- IEEE 754 560
- IFS 80, 101
- Indent 21
- index() 474
- infini 558
- infnan() 560
- init 36, 37, 99, 123, 134
- initgroups() 914
- inode 659
- i-nœud 859, 884, 895
- inotify 906
- inotify_add_watch() 907, 909
- inotify_event 908
- inotify_init() 907, 909
- inotify_init1() 907
- inotify_rm_watch() 907
- INT_MAX 536, 570
- INT_MIN 536
- internationalisation 947
- interruption 297
- intervalle 230, 231
- ioctl() 982
- IP 782, 783, 785, 786, 793, 834
- IP_ADD_MEMBERSHIP 832
- IP_DROP_MEMBERSHIP 834
- ip_mreq 832
- IP_MULTICAST_IF 834
- IP_MULTICAST_LOOP 834

IP_MULTICAST_TTL 834
 IPC 709, 725, 726
 IPC_CREAT 728, 740
 IPC_EXCL 728
 IPC_NOWAIT 730
 IPC_PRIVATE 726, 727
 IPC_RMID 730
 IPC_SET 730
 IPC_STAT 730
 IPPROTO_ICMP 800
 IPPROTO_IP 829, 832, 834
 IPPROTO_RAW 800
 IPPROTO_TCP 800, 829, 834
 IPPROTO_UDP 800
 isalnum() 537, 539
 isalpha() 537, 539
 isascii() 538, 539
 isatty() 997, 998, 1000
 isblank() 538
 iscntrl() 538, 539
 isdigit() 538, 539, 590
 isgraph() 538, 539
 isinf() 558, 559
 islower() 538, 539
 isnan() 557, 558
 isprint() 538, 539
 ispunct() 538, 539
 isspace() 538, 539
 isupper() 538, 539
 iswalnum() 540
 iswalpha() 540
 iswcntrl() 540, 541
 iswdigit() 540, 541
 iswgraph() 540, 541
 iswlower() 540, 541, 543
 iswprint() 540, 541, 588
 iswpunct() 540, 541
 iswspace() 540, 541
 iswupper() 540, 541
 iswxdigit() 540, 541
 isxdigit() 538, 539
 ITIMER 244
 ITIMER_PROF 242
 ITIMER_REAL 242
 ITIMER_VIRTUAL 242

J

jiffies 254
 jour 216, 221

K

KDE 9, 30
 key_t 726
 kill() 317, 318, 329, 372
 killpg() 319
 Ksh 94

L

LANG 100, 949
 LC_ALL 949, 964
 LC_COLLATE 949
 LC_CTYPE 466, 949, 964
 LC_MESSAGES 949
 LC_MONETARY 949
 LC_NUMERIC 949, 964
 lchown() 890, 901
 lconv 972
 leader 54, 56, 307
 Lex 478
 lfind() 487, 491
 LFS 630
 libcap 62
 lien physique 860, 884, 895
 lien symbolique 897
 ligne de commande 101, 110
 limite 314
 link() 895
 LinuxThreads 155
 lio_listio() 772, 773, 774
 LIO_NOP 772
 LIO_NOWAIT 772, 774
 LIO_READ 772
 LIO_WAIT 772
 LIO_WRITE 772
 listen() 808, 809, 1001
 little endian 795
 ln 860
 localeconv() 972, 973
 localisation 148, 220, 463, 464, 548, 576, 947
 localtime() 217, 218, 223
 localtime_r() 217

LOCK_EX 685
 LOCK_SH 685
 LOCK_UN 685
 log() 559
 LOGNAME 100
 LONG_MAX 536, 546
 LONG_MIN 536, 546
 longjmp() 358, 360
 lsearch() 487, 488, 489
 lseek() 666, 667
 lstat() 885, 891, 901

M

M_MAP_THRESHOLD 405
 M_MMAP_MAX 406
 M_MMAP_THRESHOLD 406
 M_TOP_PAD 406
 M_TRIM_THRESHOLD 407
 MAC 782
 main thread 156, 158, 164, 166
 main() 92, 119
 Make 25
 Makefile 26
 malloc() 384, 385, 386, 388, 391, 397, 402, 404, 405, 428, 458
 MALLOC_CHECK_ 410, 412, 416
 MALLOC_MMAP_MAX 407
 MALLOC_MMAP_THRESHO
 LD 407
 MALLOC_TOP_PAD 407
 MALLOC_TRACE 408
 MALLOC_TRIM_THRESHOL
 D 407
 mallopt() 406
 mand 684
 mandatory lock 684
 MAP_ANON 428
 MAP_ANONYMOUS 428
 MAP_DENYWRITE 428
 MAP_FAILED 426
 MAP_FIXED 427
 MAP_GROWSDOWN 428
 MAP_PRIVATE 427, 428
 MAP_SHARED 427, 428, 431

- MAXFLOAT 559
 - MAXPATHLEN 852
 - MB_CUR_MAX 466
 - mbrtowc() 465
 - mbsrtowcs() 466
 - mbstate_t 465
 - mbstowcs() 466, 467, 470, 543
 - mbtowc() 465
 - MCL_CURRENT 424
 - MCL_FUTURE 424
 - MD5 526, 528
 - memcpy() 448
 - Memcheck 414
 - memchr() 454, 471, 472
 - memcmp() 450, 451
 - memcpy() 447, 449
 - memfrob() 525
 - memmem() 472
 - memmove() 447, 449, 450, 458
 - mémoire 383, 421, 446
 - alignement ~ 390
 - allocation ~ 385, 394, 396, 403, 417
 - faute de page ~ 257, 428
 - fuite ~ 402, 406, 407, 414
 - libération ~ 396
 - page ~ 74, 426
 - partagée 435, 715, 720, 922
 - physique 404
 - protection ~ 440
 - segment ~ 386
 - sur-réservation ~ 387
 - verrouillage en ~ 423, 424, 425
 - virtuelle 74, 387, 404, 421
 - memcpy() 448
 - memset() 389, 446, 447
 - Métaphone 485
 - microseconde 209, 238, 241
 - migration 283, 287
 - milliseconde 240, 251
 - MINSIGSTKSZ 338
 - minute 216, 222
 - mkdir() 857, 858
 - mkdtemp() 864
 - mkfifo() 702, 706, 707
 - mknod() 703, 902
 - mkstemp() 863, 864
 - mktemp() 863
 - mktime() 219
 - mlock() 423
 - mlockall() 423, 424
 - mmap() 390, 405, 426, 428, 430, 431, 437, 441, 715, 716
 - MMU (Memory Management Unit) 34
 - mode_t 645, 648, 857
 - mois 216, 221
 - mot de passe 525, 529
 - mount() 933
 - mprotect() 439, 441
 - mq_close() 711
 - mq_getattr() 713
 - mq_notify() 714
 - mq_open() 710, 711, 712, 713
 - mq_receive() 712, 714
 - mq_send() 711, 712
 - mq_setattr() 713
 - mq_timedreceive() 712
 - mq_timedsend() 711
 - mq_unlink() 711
 - mqd_t 710, 711
 - mqueue 724
 - mremap() 439
 - MREMAP_MAYMOVE 439
 - MS_ASYNC 438
 - MS_INVALIDATE 439
 - MS_SYNC 439
 - MSG_DONTROUTE 823
 - MSG_EXCEPT 730
 - MSG_NOERROR 730
 - MSG_OOB 823
 - MSG_PEEK 823
 - msgctl() 730, 731
 - msgget() 726, 727, 728
 - msgrcv() 729
 - msgsnd() 726, 729
 - msqid_ds 727, 731
 - msync() 438
 - mtrace() 407, 408
 - multicast 785, 831, 832
 - multicœur 282
 - multiplexage 743, 763
 - multiprocesseur 155, 184, 274, 276, 282
 - multitâche 275
 - munlock() 423
 - munlockall() 423, 424
 - munmap() 430, 431
 - muntrace() 407
 - mutex 177, 191, 195, 197
- N**
- NAME_MAX 844
 - named pipe 688, 702
 - NaN 557, 560
 - nanoseconde 240, 248
 - nanosleep() 239
 - Ncurses 29
 - NDEBUG 126
 - nearbyint() 555
 - nearbyintf() 555
 - nearbyintl() 555
 - Nedit 10
 - NetBeans 6
 - nftw() 878, 880
 - nice() 277, 278
 - nl_catd 953, 956
 - nl_langinfo() 976, 978
 - NLSPATH 954
 - nohup.out 308
 - noyau 2, 240, 257, 282, 291, 371, 422, 709, 922
 - NPTL 155, 182
 - NR_OPEN 268
 - NSIG 302, 318
 - ntohl() 797
 - ntohs() 797
 - NTP 215
- O**
- O_ACCMODE 675
 - O_APPEND 645, 658, 659
 - O_ASYNC 764
 - O_CREAT 644, 647, 648
 - O_DIRECTORY 850

- O_DSYNC 777
- O_EXCL 644, 648, 651, 652
- O_NOCTTY 645
- O_NONBLOCK 645, 703, 743, 744, 747
- O_RDONLY 644, 648, 675
- O_RDWR 644, 648, 675
- O_RSYNC 777
- O_SYNC 645, 660, 777, 778
- O_TRUNC 644
- O_WRONLY 644, 675
- off_t 630
- on_exit() 122, 127, 130, 131
- OOM Killer 387, 389, 393
- Open Group VII
- open() 76, 644, 648, 650, 652, 705, 778, 851, 901
- OPEN_MAX 616, 643, 835
- opendir() 842, 843
- openlog() 942, 944
- optarg 103, 104, 105
- opterr 103, 104
- optind 103, 104
- option 103
- optopt 103, 104
- ordonnancement 34, 246, 269, 276
- ordonnanceur 176, 274, 275
- OSI 782
- OTHER (ordonnancement) 288, 290, 292
- overcommit memory 387
- P**
- P_ALL 142
- P_GID 141
- P_PID 141
- page mémoire 74
 - faute de ~ 74
- PATH 66, 71, 79, 99, 100
- pause() 352, 353
- pclose() 81, 82, 83, 86
- PCthreads 155
- périphérique 884, 885, 902
- perror() 148, 149
- personality() 756
- PGID 73, 319
- phonétisation 485
- pthread_mutexattr_t 177
- pthread_setcancelstate() 184
- pthread_setcanceltype() 186
- PID 36, 73, 317
- pile 171, 200, 261, 308, 338, 345, 384, 386, 399
- ping 785, 801
- pipe 688, 693, 708
- pipe() 643, 688, 690, 691, 694, 696, 698, 744
- PIPE_BUF 701, 704
- point d'annulation 186, 187
- pointeur 306, 370, 384, 389, 395, 397, 402, 426, 446, 447, 606
- poll() 750, 756
- POLLERR 757
- pollfd 756
- POLLHUP 757
- POLLIN 757
- POLLNVAL 757
- POLLOUT 757
- POLLPRI 757
- popen() 81, 83, 86
- Posix VI, 153, 365, 709, 720
- POSIXLY_CORRECT 100
- PPID 37, 39, 73
- pread() 657, 771
- préemptible (noyau) 294
- préemptif (multitâche) 275, 292
- printf() 466, 567, 576
- PRIO_MAX 280
- PRIO_MIN 280
- PRIO_PGRP 280, 281
- PRIO_PROCESS 280, 281
- PRIO_USER 280, 281
- priorité 275, 276, 289, 292
- processus 33, 36, 65, 90, 166, 201, 252, 257, 310, 314, 317, 383, 422, 687, 688, 693, 923
 - groupe de ~ 52
- profileur 18, 19
- projection 425, 426, 427
- PROT_EXEC 427, 440
- PROT_NONE 440
- PROT_READ 427, 440
- PROT_WRITE 427, 440
- ps 35, 37, 132, 158, 269, 275, 282, 326, 403, 404
- pselect() 756
- pseudo-terminal 1002
- psignal() 315
- Pthread 153
- pthread 192
- pthread_atfork() 201
- pthread_attr_destroy() 175
- pthread_attr_getdetachstate() 174
- pthread_attr_getstackaddr() 200
- pthread_attr_getstacksize() 201
- pthread_attr_init() 174, 175
- pthread_attr_setdetachstate() 174, 175
- pthread_attr_setstackaddr() 200
- pthread_attr_setstacksize() 201
- pthread_attr_t 174, 175
- pthread_cancel() 183, 185, 190
- PTHREAD_CANCEL_ASYNC_HRONOUS 186, 187
- PTHREAD_CANCEL_DEFERRED 186, 187
- PTHREAD_CANCEL_DISABLE 184, 185, 187
- PTHREAD_CANCEL_ENABLE 184, 187
- PTHREAD_CANCELED 184
- pthread_cleanup_pop() 188, 189
- pthread_cleanup_push() 188, 189
- pthread_cond_broadcast() 194
- pthread_cond_destroy() 191
- pthread_cond_init() 191
- PTHREAD_COND_INITIALIZER 191
- pthread_cond_signal() 192, 193
- pthread_cond_t 191, 193
- pthread_cond_timedwait() 186, 194

pthread_cond_wait() 186, 191,
 192, 193, 194
 pthread_condattr_destroy() 195
 pthread_condattr_init() 195
 pthread_condattr_t 191
 pthread_create() 156, 157, 160,
 174, 175, 284
 PTHREAD_CREATE_DETAC
 HED 174, 175
 PTHREAD_CREATE_JOINAB
 LE 174
 pthread_detach() 172, 173
 pthread_equal() 156
 pthread_exit() 165, 166, 167
 pthread_getaffinity_np() 285
 pthread_getschedparam() 296
 pthread_getspecific() 203
 pthread_join() 169, 170, 171, 186
 pthread_key_create() 203
 pthread_key_delete() 203
 pthread_key_t 203
 PTHREAD_MUTEX_DEFAULT
 T 198
 pthread_mutex_destroy() 177
 PTHREAD_MUTEX_ERRORC
 HECK 198, 199
 pthread_mutex_init() 177, 199
 PTHREAD_MUTEX_INITIALIZER
 ZER 177, 179, 195
 pthread_mutex_lock() 178, 179,
 192, 195, 196, 197
 PTHREAD_MUTEX_NORMA
 L 198
 PTHREAD_MUTEX_RECURSI
 VE 198
 pthread_mutex_t 177, 179
 pthread_mutex_trylock() 178
 pthread_mutex_unlock() 178, 179,
 196, 197
 pthread_mutexattr_destroy() 198
 pthread_mutexattr_gettype() 198
 pthread_mutexattr_init() 198, 199
 pthread_mutexattr_settype() 198,
 199
 pthread_mutexattr_t 199

pthread_once() 204
 PTHREAD_ONCE_INIT 204
 pthread_once_t 204
 pthread_rwlock_destroy() 182
 pthread_rwlock_init() 181
 PTHREAD_RWLOCK_INITIA
 LIZER 181
 pthread_rwlock_rdlock() 182
 pthread_rwlock_t 181
 pthread_rwlock_tryrdlock() 182
 pthread_rwlock_trywrlock() 182
 pthread_rwlock_unlock() 182
 pthread_rwlock_wrlock() 182
 pthread_self() 172
 pthread_setaffinity_np() 285, 286
 pthread_setcancelstate() 185
 pthread_setschedparam() 295
 pthread_setspecific() 203, 204
 PTHREAD_STACK_MIN 200
 pthread_t 156, 157
 pthread_testcancel() 186
 PTP 215
 ptsname() 1004, 1006
 ptsname_r() 1004
 putc() 582
 putchar() 582
 putenv() 95, 96
 putpwent() 917
 puts() 583
 pututline() 937
 pututxline() 939
 putw() 626
 putwc() 582
 putwchar() 582
 PWD 100
 pwrite() 658, 666, 771

Q

qecvt() 551
 qecvt_r() 552
 qfcvt() 551
 qfcvt_r() 552
 qgcvt() 551
 qsort() 463, 494, 495, 845
 quad 551
 Quicksort 493

R

r/w locks 181
 raise() 319, 356
 rand() 524
 rawmemchr() 472
 re_comp() 524
 re_exec() 524
 read() 312, 654, 656
 readdir() 842, 843, 844
 readdir_r() 844
 readlink() 899, 900, 901
 readv() 656, 657
 realloc() 394, 395
 realpath() 852, 853
 realtime 293
 recherche dichotomique 492, 513
 recherche linéaire 486, 512
 recherche séquentielle 486, 488,
 490
 recv() 807, 822
 recvfrom() 807, 822
 REG_NOMATCH 518
 regcomp() 516, 517, 521, 846
 regerror() 517, 521
 regex_t 516
 regexec() 518, 521, 846
 regfree() 519, 522
 regmatch_t 518
 regular expression 515
 remove() 862
 rename() 862, 901
 répertoire 841, 842, 847
 réseau 781
 revents 757
 rewind() 628, 641
 rewinddir() 844
 rindex() 474
 rint() 554, 556
 rintf() 554
 rintl() 554
 RLIM_INFINITY 261, 266
 rlim_t 266
 RLIMIT_CORE 261, 423
 RLIMIT_CPU 261
 RLIMIT_DATA 261, 387

- RLIMIT_FSIZE 261, 664
- RLIMIT_MEMLOCK 261
- RLIMIT_NOFILE 261, 268
- RLIMIT_NPROC 149, 261
- RLIMIT_RSS 261
- RLIMIT_STACK 261, 400
- rmdir() 857, 858, 859
- routage 782
- route 833
- Rpm 28
- RR (ordonnancement) 288, 292
- RS-232 1012
- RSS 404
- RTAI 268, 300
- RUSAGE 258
- RUSAGE_CHILDREN 257, 258
- RUSAGE_SELF 257
- S**
- S_ISBLK() 885, 886
- S_ISCHR() 885, 886
- S_ISDIR() 885, 886
- S_ISFIFO() 701, 885, 886
- S_ISLNK() 885, 886
- S_ISREG() 885, 886
- S_ISSOCK() 885, 886
- SA_INTERRUPT 337
- SA_NOCLDSTOP 337
- SA_NODEFER 337
- SA_ONESHOT 337
- SA_ONSTACK 337, 338, 346
- SA_RESETHAND 337, 341
- SA_RESTART 337, 341, 361
- SA_SIGINFO 337, 369
- sbrk() 386, 406
- scandir() 845, 846, 867
- scanf() 598, 599, 602, 604
- SCHAR_MAX 536
- SCHAR_MIN 536
- SCHED_FIFO 289, 294, 296
- sched_get_priority_max() 289
- sched_get_priority_min() 289
- sched_getaffinity() 285
- sched_getcpu() 283, 284
- sched_getparam() 296, 298
- sched_getscheduler() 294, 298
- SCHED_OTHER 289, 294, 296
- sched_priority 296
- SCHED_RR 289, 294, 296
- sched_rr_get_interval() 297, 298
- sched_rt_period_us 291
- sched_rt_runtime_us 291
- sched_setaffinity() 285
- sched_setparam() 296
- sched_setscheduler() 295
- sched_yield() 290
- scheduling 34
- seconde 209, 216, 222, 240
 - de rattrapage 216
- SEEK_CUR 627, 666, 667
- SEEK_END 627, 631, 666
- SEEK_SET 627, 631, 666, 667
- seekdir() 844
- select() 750, 751, 752, 753, 762, 823, 1005
- SeLinux 63
- sem_destroy() 721
- SEM_FAILED 722
- sem_init() 721
- sem_open() 721, 722
- sem_post() 722, 723
- sem_t 721, 722
- sem_timedwait() 722
- sem_trywait() 722
- SEM_UNDO 735
- sem_unlink() 721, 724
- sem_wait() 722, 723
- semaine 217, 221
- sémaphore 720, 733
- sembuf 735
- semctl() 734, 736, 738
- semget() 726, 727, 734, 738, 740
- semid_ds 727
- semop() 726, 734, 735, 739, 740
- send() 807, 822
- sendto() 807, 822
- serveur 170
- serveur TCP 808
- session 56, 307
- set (commande shell) 94
- setbuf() 639
- setbuffer() 639
- setdomainname() 920
- setegid() 48
- setenv (commande shell) 94
- setenv() 95, 97
- seteuid() 44, 854
- setfsent() 926
- Set-GID 48, 682
- setgid() 48
- setgrent() 914
- setgroups() 50, 51
- sethostid() 920
- sethostname() 919
- setitimer() 242, 244, 305
- setjmp() 358, 360
- setlinebuf() 639
- setlocale() 463, 588, 948, 956, 963, 964, 965, 967, 969
- setmntent() 928, 929, 931
- setpgid() 55, 56, 307
- setpgrp() 319
- setpriority() 280, 282
- setpwent() 916
- setregid() 48
- setresgid() 52
- setresuid() 47
- setreuid() 44, 46
- setrlimit() 149, 268, 423, 664
- setsid() 57, 835
- setsockopt() 829, 831
- settimeofday() 214
- Set-UID 42, 44, 60, 78, 357, 801, 890
- setuid() 44, 45
- setusershell() 918
- setutent() 936
- setutxent() 939
- setvbuf() 637, 638
- SH4 796
- shell 39, 56, 90, 94, 343, 565, 675, 693, 725, 779, 865, 872, 918
- shift state 465
- SHM_LOCK 733
- shm_open() 715, 716

- SHM_RDONLY 733
- SHM_RND 732
- shm_unlink() 715
- SHM_UNLOCK 733
- shmat() 726, 732, 740
- shmctl() 732
- shmdt() 726, 732
- shmget() 726, 727, 732, 740
- shmids 727, 733
- SHMMAX 732
- SHMMIN 732
- SHRT_MAX 536
- SHRT_MIN 536
- shutdown() 821, 822
- SI_ASYNCIO 371
- SI_KERNEL 371
- SI_MESGQ 371
- SI_QUEUE 371
- SI_SIGIO 371
- SI_TIMER 371
- SI_USER 371
- SID 73
- si cle 221
- sig_atomic_t 354
- SIG_BLOCK 348, 350, 352, 377, 380
- SIG_DFL 323, 326, 343
- SIG_ERR 323
- SIG_IGN 323, 336, 343
- SIG_SETMASK 348, 351
- SIG_UNBLOCK 348, 353, 377
- SIGABRT 124, 304
- sigaction() 322, 324, 335, 336, 341, 343, 344, 369, 375
- sigaddset() 339
- SIGALRM 242, 305, 321, 335, 360
- sigaltstack() 338, 339, 346
- sigandset() 340
- sigblock() 348
- SIGBUS 306, 371
- SIGCHLD 123, 306, 371
- SIGCLD 306
- SIGCONT 270, 307, 311
- sigdelset() 339, 350, 380
- sigemptyset() 339, 350
- SIGEV_NONE 767
- SIGEV_SIGNAL 247, 767
- SIGEV_THREAD 247, 768
- sigfillset() 339, 350, 377, 380
- SIGFPE 307, 371
- siggetmask() 348
- SIGHUP 307
- SIGILL 308, 372
- siginfo_t 142
- SIGINT 121, 309, 340, 585
- siginterrupt() 331, 332
- SIGIO 309, 310, 763, 764
- SIGIOT 304
- sigisemptyset() 340
- sigismember() 339, 350
- sigjmp_buf 358, 359, 362
- SIGKILL 261, 310, 325, 389
- siglongjmp() 357, 358, 359, 362, 401, 402
- sigmask() 348
- signal 121, 124, 235, 242, 301, 302, 303, 308, 317, 322, 365, 656, 699, 763
- signal() 322, 324, 329, 332, 344, 664
- sigorset() 340
- sigpause() 354
- sigpending() 349, 350
- SIGPIPE 310, 662, 698, 699, 703, 823
- SIGPOLL 309, 372
- sigprocmask() 335, 347, 350, 353, 377, 380
- SIGPROF 242, 305
- sigqueue() 367, 369, 370, 371, 374, 376
- SIGQUIT 311, 340
- SIGRTMAX 302, 314, 366, 368
- SIGRTMIN 302, 314, 366, 368, 369
- SIGSEGV 306, 372, 441, 447, 453
- sigset_t 336, 339, 340, 350
- sigsetjmp() 358, 359, 362, 401
- sigsetmask() 348
- SIGSTKFLT 307
- SIGSTKSZ 338, 346
- SIGSTOP 270, 311, 325
- sigsuspend() 353
- SIGTERM 311
- sigtimedwait() 379, 382
- SIGTRAP 312, 372
- SIGTSTP 311
- SIGTTIN 312, 656
- SIGTTOU 312
- SIGURG 313
- SIGUSR1 313
- SIGUSR2 313
- sigval_t 767
- sigvec() 322
- SIGVTALRM 242, 305
- sigwaitinfo() 379, 380
- SIGWINCH 313
- SIGXCPU 261, 314
- SIGXFSZ 261, 314, 661, 663, 664
- size_t 384
- sizeof() 385, 397, 535, 623
- sleep() 235, 236, 238
- snprintf() 577
- SO 830
- SO_BROADCAST 830, 831
- SO_BSDCOMPAT 830
- SO_DEBUG 830
- SO_DONTROUTE 830
- SO_ERROR 830
- SO_KEEPALIVE 830
- SO_LINGER 830
- SO_OOINLINE 830
- SO_REUSEADDR 830
- SO_SNDBUF 830
- SO_SNDLOWAT 830
- SO_SNDTIMEO 830
- SO_TYPE 830
- SOCK_DGRAM 800, 807
- SOCK_RAW 800
- SOCK_STREAM 800
- sockaddr 801
- sockaddr_in 802, 805

- sockaddr_in6 802
 - socket 310, 313, 320, 763, 799, 999
 - socket() 643, 800, 1001
 - socketpair() 802
 - soft realtime 294
 - SOL_SOCKET 829
 - sommeil 235, 270
 - sortie
 - d'erreur 564
 - standard 564
 - Soundex 485
 - Splint 20, 576
 - sprintf() 577
 - sscanf() 598, 603, 606, 607
 - SSIZE_MAX 655
 - ssize_t 655
 - stack_t 338, 345
 - stackaddr 200
 - stacksize 200
 - standard
 - entrée ~ 565
 - sortie ~ 565
 - stat() 701, 885, 887, 901
 - statfs() 931, 932
 - statistiques 257
 - stderr 564, 616, 620, 634
 - STDERR_FILENO 643, 675
 - stdin 564, 616, 618, 620, 694
 - STDIN_FILENO 643, 675, 695
 - stdout 564, 616, 620, 634, 694
 - STDOUT_FILENO 643, 675, 695
 - time() 214, 215
 - stpcpy() 456, 460
 - stpncpy() 456, 457, 460
 - Strace 19
 - strcasecmp() 462, 484
 - strcasestr() 476
 - strcat() 459
 - strchr() 473
 - strcmp() 460
 - strcoll() 462, 484
 - strcpy() 453, 455, 459
 - strcspn() 477
 - strdup() 458
 - strdupa() 458
 - strerror() 146, 147, 569
 - strerror_r() 146
 - strfmon() 968, 969, 971
 - strfry() 524, 525, 705
 - strftime() 221, 223, 225
 - strlen() 452, 453
 - strncasecmp() 462
 - strncat() 459
 - strncmp() 461
 - strncpy() 448, 455, 456
 - strndup() 458
 - strndupa() 458
 - strnlen() 454, 456
 - strpbrk() 477, 478
 - strptime() 224, 226, 227
 - strrchr() 474
 - strsep() 482
 - strsignal() 315
 - strspn() 476, 477
 - strstr() 474, 475
 - strtod() 547
 - strtof() 547, 548
 - strtok() 478, 479, 480
 - strtok_r() 481
 - strtol() 545, 546
 - strtold() 547
 - strtoll() 545
 - strtoul() 545
 - strtoull() 545
 - struct addrinfo 787, 788
 - struct aiocb 766, 769
 - struct files_struct 658
 - struct flock 677, 678
 - struct fstab 927
 - struct group 912
 - struct hsearch_data 507
 - struct inode 658, 659
 - struct iovec 656, 657
 - struct itimerspec 248
 - struct itimerval 243, 244
 - struct mntent 928
 - struct mq_attr 711, 713
 - struct option 106
 - struct passwd 915
 - struct rlimit 149, 260, 265, 266
 - struct rusage 141, 257, 258
 - struct sched_param 296
 - struct sembuf 735
 - struct sigaction 336, 341, 342, 374, 769
 - struct sigevent 767, 772
 - struct siginfo 338, 374, 764
 - struct sockaddr 789
 - struct stat 431, 638, 884, 891, 894, 895, 901
 - struct statfs 932
 - struct task_struct 658
 - struct timespec 194, 240, 297, 379
 - struct timeval 209, 210, 211, 215, 230, 751, 755
 - struct timezone 209
 - struct tm 206, 216, 217, 223, 225
 - struct tms 255, 256
 - struct utimbuf 894
 - struct utmp 935, 938
 - struct utsname 921
 - strxfrm() 463
 - SUSv4 VII, 2
 - swap 74, 257, 261, 387, 422, 423, 430, 733, 923
 - symlink() 897
 - sync() 618, 660, 776
 - sys_nerr 150, 151
 - sysconf() 255, 426
 - sysctl 291
 - sysinfo() 922, 923
 - syslog 941
 - syslog() 835, 942, 944
 - syslogd 941
 - system() 77, 79, 693
 - Système V 709, 756
- T**
- table auto-organisatrice 491
 - table de hachage 505, 506, 513
 - table ordonnée 489
 - tâche 269
 - Tar 27
 - tcdrain() 986

- tcflow() 986
 - tcflush() 986
 - tcgetattr() 984, 985, 996, 1006, 1009, 1017
 - tcgetpgrp() 59, 999
 - TCP 781, 782, 785, 786, 792, 793, 794, 800
 - TCP_MAXSEG 834
 - TCP_NODELAY 834
 - tcsendbreak() 987
 - tcsetattr() 984, 985, 997, 1006, 1009, 1010, 1017
 - tcsetpgrp() 59, 999
 - Tcsh 94, 262
 - tdelete() 501
 - tdestroy() 501
 - telldir() 844
 - telnet 792, 794, 812
 - tempnam() 863
 - temporisation 242, 247, 305
 - temps-partagé 274, 288
 - ordonnancement ~ 288
 - temps-réel 248, 268, 288, 764
 - ordonnancement ~ 288, 293
 - sémaphores ~ 720
 - signal ~ 302, 314, 365, 372
 - TERM 99, 100
 - terminaison 119, 169
 - terminal 307, 311, 312, 313, 585, 636, 656, 981
 - ~ de contrôle 56
 - de contrôle 999
 - termios 982, 983, 985, 987, 994, 996
 - textdomain() 959, 960
 - tfind() 501, 503
 - thread 33, 153, 154, 183, 184, 201, 269, 285, 291, 687
 - principal 156, 158, 164, 166, 167
 - tick 240
 - time() 206, 208, 218, 238
 - time_t 206, 208, 215, 217, 231
 - TIME_WAIT 821
 - timer 240, 242, 243, 247
 - timer_create() 247, 249
 - timer_delete() 247
 - timer_settime() 248, 249
 - timer_t 247, 249
 - timeradd() 230
 - timerclear() 230
 - timerfd_create 251
 - timerfd_gettime 251
 - timerfd_settime 251
 - timerisset() 231
 - timersub() 231, 241
 - times 245
 - times() 254, 255, 256
 - TMPDIR 100
 - tmpfile() 122, 865
 - tmpfs 724
 - tmpnam() 863
 - toascii() 542
 - token 478
 - tolower() 542
 - toupper() 542
 - towlower() 542
 - towupper() 542, 543
 - tri 463, 483, 484, 486
 - truncate() 893, 901
 - tsearch() 500, 503
 - ttyname() 997, 998
 - ttyname_r() 997
 - tube 310, 320, 688, 744, 763
 - tube nommé 702, 885
 - twalk() 501, 503
 - type 535
 - TZ 232
 - tzset() 232, 233
- U**
- UCHAR_MAX 536
 - uClibc 3
 - UCS-2 464
 - UDP 782, 785, 786, 792, 793, 794, 800, 818
 - UID 40, 73
 - uid_t 41
 - UINT_MAX 536, 570
 - ulimit 172, 264, 303, 400
 - ULONG_MAX 536
 - umask 647, 650, 858
 - umask() 859, 905, 906
 - umount() 933
 - uname() 921
 - ungetc() 589, 590
 - ungetwc() 589
 - Unicode 464
 - union sigval 369, 373, 767
 - Unix 98 VI
 - unlink() 437, 703, 706, 860, 861, 865, 901
 - unlockpt() 1004, 1006
 - unsetenv() 95, 97
 - updwtmp() 941
 - USHRT_MAX 536
 - usleep() 238, 239, 240, 241
 - UTF-8 464, 540, 587
 - utilisateur
 - groupe d'~ 48
 - identifiant d'~ 40
 - utime() 894, 901
 - utimes() 894, 901
 - utmpname() 940
- V**
- va_end() 579
 - va_list 578, 579
 - va_start() 579
 - Valgrind 19, 414, 417, 418, 419, 420
 - variable 383
 - variable d'environnement 89, 92, 98, 110, 464
 - verrouillage 177, 184, 423, 677, 685
 - vfprintf() 578
 - vfprintf() 598
 - Vi 8
 - volatile 354
 - vprintf() 578
 - vscanf() 598
 - vsprintf() 578, 579
 - vsprintf() 578
 - vsscanf() 598
 - VSZ 404

W

waitpid() 136
wait() 40, 136, 137, 306
wait3() 136, 141
wait4() 136, 141
waitid() 141, 142
waitpid() 56, 78, 139, 140
wchar_t 465, 467, 469
wchart_t 540
WCOREDUMP() 136, 137
wctomb() 465
wcscasecmp() 469, 484
wscat() 468
wcschr() 473
wcscmp() 469
wscoll() 469, 470
wcscpy() 468
wcscspn() 477
wcslen() 468
wcsncasecmp() 469
wcsncat() 469
wcsncmp() 469
wcsncpy() 468
wcsnlen() 468
wcpbrk() 477
wcsrchr() 474
wcsrtombs() 466
wcsspncpy() 476
wcsstr() 474
wcstod() 547
wcstof() 547
wcstok() 481
wcstol() 545
wcstold() 547
wcstoll() 545
wcstombs() 466
wcstoul() 545
wcstoull() 545
wcsxfrm() 469
wctob() 466
wctomb() 465
WEOF 540, 588
WEXITSTATUS() 136, 137, 140
wide char 465
WIFEXITED() 136, 137, 140
WIFSIGNALED() 136, 137, 140
WIFSTOPPED() 136, 137, 140
wint_t 540, 541
wmemchr() 471
wmemcmp() 468
wmemcpy() 469
wmemmove() 469
wmemset() 468
WNOHANG 139
wordexp() 874, 876
wordexp_t 874
wordfree() 874, 877
write() 310, 314, 618, 658, 663, 664, 698, 699, 761
writev() 658, 665
WSTOPPED 142
WSTOPSIG() 136, 137, 140
WTERMSIG() 136, 137, 140
wtmp 940
WUNTRACED 139, 141

X

Xargs 23
Xenomai 268, 293, 300
xinetd 671, 837
Xlib 30
Xt 30
XtAppMainLoop() 763
X-Window 30

Y

Yacc 478

Z

zombie 123, 132, 271, 306, 310, 810