Programmation Avancée sous Linux

Mark Mitchell Jeffrey Oldham Alex Samuel

Traduction : Sébastien Le Ray

Programmation avancée sous Linux

PREMIÈRE ÉDITION : Juin 2001

ISBN : 0-7357-1043-0

Numéro Catalogue de la Bibliothèque du Congrès : 00-105343

05 04 03 02 01 7 6 5 4 3 2 1

Interprétation du code : Le nombre à deux chiffres à l'extrême droite est l'année d'impression du livre ; le nombre à un chiffre à l'extrême droite est le rang d'impression du livre. Par exemple, le code 01-1 signifie que la première impression du livre a été réalisée en 2001.

Marques déposées

Non-responsabilité et avertissement

Traduction

Le plus grand soin a été apporté à la traduction en Français de ce livre, toutefois, des erreurs de traduction peuvent avoir été introduites en plus des éventuelles erreurs initialement présente. Les traducteurs ne peuvent être tenus pour responsables par quiconque de toute perte de données ou dommages occasionnés par l'utilisation des informations présentes dans ce livre.

La version originale de ce document est disponible sur le site des auteurs `http://www.advancedlinuxprogramming.com`

Table des matières

À propos des auteurs

Mark Mitchell a obtenu un *bachelor of arts* en informatique à Harvard en 1994 et un *master of science* à Standford en 1999. Ses recherches se concentrent sur la complexité de calcul et la sécurité informatique. Mark a participé activement au développement de la GNU Compiler Collection et s'applique à développer des logiciels de qualité.

Jeffrey Oldham a obtenu un *bachelor of arts* en informatique à l'Université Rice en 1991. Après avoir travaillé au Center for Research on Parallel Computation, il a obtenu un doctorat en philosophie à Standford en 2000. Ses recherches sont centrées sur la conception d'algorithmes en particulier les algorithmes de flux et combinatoires. Il travaille sur GCC et des logiciels de calcul scientifique.

Alex Samuel a été diplômé de Harvard en 1995 en physique. Il a travaillé en tant qu'ingénieur informatique à BBN avant de retourner étudier la physique à Caltech et au Standford Linear Accelerator Center. Alex administre le projet Software Carpentry et travaille sur divers autres projets comme les optimisations dans GCC.

Mark et Alex ont fondé ensemble **CodeSourcery LLC** en 1999. Jeffrey a rejoint l'entreprise en 2000. Les missions de CodeSourcery sont de fournir des outils de développement pour GNU/Linux et d'autres système d'exploitation ; faire de l'ensemble des outils GNU une suite de qualité commerciale respectant les standards ; et de fournir des services généraux de conseil et d'ingénierie. Le site Web de CodeSourcery est disponible à l'adresse `http://www.codesourcery.com`

À propos des relecteurs techniques

Ces relecteurs ont mis à disposition leur expertise tout au long du processus de création du livre *Programmation Avancée sous Linux*. Au fur et à mesure de l'écriture du livre, ces professionnels ont passé en revue tous les documents du point de vue de leur contenu technique, de leur organisation et de leur enchaînement. Leur avis a été critique afin de nous assurer que *Programmation Avancée sous Linux* corresponde aux besoins de nos lecteurs avec une information technique de grande qualité.

Glenn Becker a beaucoup de diplômes, tous dans le théâtre. Il travaille actuellement en tant que producteur en ligne pour SCIFI.COM, la composante en ligne de la chaîne SCI FI, à New York. Chez lui, il utilise Debian GNU/Linux et est obsédé par des sujets comme l'administration système, la sécurité, l'internationalisation de logiciels et XML.

John Dean a obtenu un *BSc(Hons)* à l'Université de Sheffield en 1974, en sciences théoriques. Lors de son premier cycle à Sheffield, John a éprouvé de l'intérêt pour l'informatique. En 1986, il a obtenu un MSc au Cranfield Institute of Science and Technology dans le domaine des commandes et systèmes commandés. Alors qu'il travaillait pour Rolls Royces et Associés, John s'est engagé dans le développement de logiciels de contrôle pour l'inspection assistée par ordinateur des unités de production de vapeur nucléaires. Depuis qu'il a quitté RR&A en 1978, il a travaillé pour l'industrie pétrochimique dans le développement et la maintenance de logiciels de contrôle de processus. John a travaillé en tant que développeur volontaire pour MySQL de 1996 à Mai 2000, il a alors rejoint MySQL comme employé à plein temps. Le champ de responsabilités de John concerne MySQL sous MS Windows et le développement de nouveaux clients graphiques pour MySQL utilisant le kit de développement Qt de Trolltech, sous Windows et sur les plateformes disposant de X-11..

Remerciements

Nous apprécions grandement le travail de pionniers de Richard Stallman, sans qui il n'y aurait jamais eu de projet GNU et de Linus Torvalds, sans qui il n'y aurait jamais eu de noyau Linux. Sans compter ceux qui ont travaillé sur le système d'exploitation GNU/Linux, nous les remercions tous.

Nous remercions les facultés de Harvard et Rice pour nos premiers cycles et Caltech et Standford pour notre préparation au diplôme. Sans tous ceux qui nous ont enseigné, nous n'aurions jamais osé enseigner aux autres !

W. Richard Stevens a écrit trois livres excellents sur la programmation sous UNIX et nous les avons consultés sans retenue. Roland McGrath, Ulrich Drepper et beaucoup d'autres ont écrit la bibliothèque GNU C et la documentation l'accompagnant.

Robert Brazile et Sam Kendall ont relu les premiers jets de ce livre et ont fait des suggestions pertinentes sur le ton et le contenu. Nos éditeurs techniques et relecteurs (en particulier Glenn Becker et John Dean) ont débusqué nos erreurs, fait des suggestions et nous ont continuellement encouragé. Bien sûr, s'il reste des erreurs, ce n'est pas de leur faute !

Merci à Ann Quinn, de New Riders, pour la gestion de tous les détails concernant la publication d'un livre ; Laura Loveall, également de New Riders, de ne pas nous avoir laissé dépasser de trop nos deadlines ; et Stéphanie Wall, elle aussi de New Riders, pour avoir été la première à nous encourager à écrire ce livre !

Dites nous ce que vous en pensez !

En tant que lecteur de ce livre, vous êtes les critiques et les commentateurs les plus importants. Nous tenons compte de votre opinion et voulons savoir ce que nous faisons bien et ce que nous pourrions mieux faire, dans quels domaines vous aimeriez que nous publiions et tout autre conseil que vous voudriez nous communiquer.

En tant qu'Éditeur Exécutif pour l'équipe de Développement Web au sein de New Riders Publishing, j'attends vos commentaires. Vous pouvez me contacter par fax, email ou m'écrire directement pour me faire savoir ce que vous aimez ou pas dans ce livre – mais également ce que nous pourrions faire pour rendre nos livres plus pertinents.

Notez s'il vous plaît que je ne peux pas vous aider sur des problèmes techniques liés au contenu de ce livre, et qu'en raison du volume important de courrier que je reçois, il ne m'est pas forcément possible de répondre à tous les messages.

Lorsque vous écrivez, assurez-vous s'il vous plaît d'inclure le titre et l'auteur de ce livre, ainsi que votre nom et votre numéro de téléphone ou de fax. Je lirai vos commentaires avec attention et les partagerai avec l'auteur et les éditeurs qui ont travaillé sur ce livre.

Fax :	317-581-4663
Courriel :	`Stephanie.Wall@newriders.com`
Adresse :	Stephanie Wall Executive Editor New Riders Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

Introduction

GNU/Linux a envahi le monde des ordinateurs comme une tempête. Il fut un temps où les utilisateurs d'ordinateurs étaient obligés de faire leur choix parmi des systèmes d'exploitation et des applications propriétaires. Ils n'avaient aucun moyen de corriger ou d'améliorer ces programmes, ne pouvaient pas regarder « sous le capot », et étaient souvent forcés d'accepter des licences restrictives. GNU/Linux et d'autres systèmes open source ont changé cet état de faits – désormais, les utilisateurs de PC, les administrateurs et les développeurs peuvent choisir un environnement complet gratuit avec des outils, des applications et l'accès au code source.

Une grande partie du succès de GNU/Linux est liée à sa nature open source. Comme le code source des programmes est disponible pour le public, tout un chacun peut prendre part au développement, que ce soit en corrigeant un petit bogue ou en développant et distribuant une application majeure complète. Cette opportunité a attiré des milliers de développeurs compétents de par le monde pour contribuer à de nouveaux composants et améliorations pour GNU/Linux, à un tel point que les systèmes GNU/Linux modernes peuvent rivaliser avec les fonctionnalités de n'importe quel système propriétaire et les distributions incluent des milliers de programmes et d'applications se répartissant sur plusieurs CD-ROM ou DVD.

Le succès de GNU/Linux a également avalisé la philosophie UNIX. La plupart des Interfaces de Programmation d'Application (Application Programming Interfaces, API) introduites dans les UNIX AT&T et BSD survivent dans Linux et constituent les fondations sur lesquelles sont construits les programmes. La philosophie UNIX de faire fonctionner de concert beaucoup de petits programmes orientés ligne de commande est le principe d'organisation qui rend GNU/Linux si puissant. Même lorsque ces programmes sont encapsulés dans des interfaces graphiques simples à utiliser, les commandes sous-jacentes sont toujours disponibles pour les utilisateurs confirmés et les scripts.

Une application GNU/Linux puissante exploite la puissance de ces API et de ces commandes au niveau de son fonctionnement interne. Les API GNU/Linux donnent accès à des fonctionnalités sophistiquées comme la communication interprocessus, le multithreading et la communication réseau hautes performances. Et beaucoup de problèmes peuvent être résolus simplement en assemblant des commandes et des programmes existants au moyen de scripts.

GNU et Linux

D'où vient le nom GNU/Linux ? Vous avez certainement déjà entendu parler de Linux auparavant et vous avez peut-être entendu parler du Projet GNU. Vous n'avez peut-être jamais

entendu le nom GNU/Linux, bien que vous soyez probablement familier avec le système auquel il se réfère.

Linux vient de Linus Torvalds, le créateur et l'auteur original du noyau[1] à la base du fonctionnement d'un système GNU/Linux. Le noyau est le programme qui effectue les opérations de base d'un système d'exploitation : il contrôle et fait l'interface avec le matériel, gère l'allocation de la mémoire et des autres ressources, permet à plusieurs programmes de s'exécuter en même temps, gère le système de fichiers *et caetera*.

Le noyau en lui-même n'offre pas de fonctionnalités utiles à l'utilisateur. Il ne peut même pas afficher une invite pour que celui-ci entre des commandes élémentaires. Il n'offre aucun moyen de gérer ou d'éditer les fichiers, de communiquer avec d'autres ordinateurs ou d'écrire des programmes. Ces tâches requièrent l'utilisation d'une large gamme d'autres produits comme les shells de commande, les utilitaires de fichiers, les éditeurs et les compilateurs. La plupart de ces programmes utilisent à leur tour des bibliothèques de fonctions comme la bibliothèque contenant les fonctions standards du C qui ne sont pas incluses dans le noyau.

Sur les systèmes GNU/Linux, la plupart de ces programmes sont des logiciels développés dans le cadre du Projet GNU[2]. Une grande partie de ces logiciels ont précédé le noyau Linux. Le but du projet GNU est de « développer un système d'exploitation de type UNIX qui soit un logiciel libre » (d'après le site Web du Projet GNU, `http://www.gnu.org`).

Le noyau Linux et les logiciels du Projet GNU ont prouvé qu'ils sont une combinaison puissante. Bien que cette association soit souvent appelée « Linux » par raccourci, le système complet ne pourrait fonctionner sans les logiciels GNU, pas plus qu'ils ne pourraient fonctionner sans le noyau. Pour cette raison, dans ce livre, nous ferons référence au système complet par le nom GNU/Linux, excepté lorsque nous parlerons spécifiquement du noyau Linux.

La licence publique générale de GNU

Les codes sources contenus dans ce livre son couverts par la *Licence Publique Générale GNU* (GNU General Public Licence, GPL), qui est reproduite dans l'Annexe F, « Licence Publique Générale GNU ». Un nombre important de logiciels libres, en particulier ceux pour GNU/Linux, sont couverts par celle-ci. Par exemple, le noyau Linux lui-même est sous GPL, comme beaucoup d'autres programmes et bibliothèques GNU que vous trouverez dans les distributions GNU/Linux. Si vous utilisez des sources issues de ce livre, soyez sûr d'avoir bien lu et bien compris les termes de la GPL.

Le site Web du Projet GNU comprend une discussion dans le détail de la GPL (`http://www.gnu.org/copyleft/copyleft.fr.html`) et des autres licences de logiciels libres. Vous trouverez plus d'informations sur les licences open source sur `http://opensource.org/licenses/`.

À qui est destiné ce livre ?

Ce livre est destiné à trois types de lecteurs :

[1] NdT : le terme original kernel est parfois employé. Nous conserverons noyau dans la suite de ce livre.
[2] GNU est un acronyme récursif, il signifie « GNU's Not UNIX » (GNU n'est pas UNIX).

- Les développeurs ayant déjà l'expérience de la programmation sous GNU/Linux et voulant en savoir plus sur certaines de ses fonctionnalités et possibilités avancées. Vous pouvez être intéressé par l'écriture de programmes plus sophistiqués avec des fonctionnalités comme le multiprocessing, le multithreading, la communication interprocessus et l'interaction avec les périphériques matériels. Vous pouvez vouloir améliorer vos programmes en les rendant plus rapides, plus fiables et plus sécurisés ou en les concevant de façon à ce qu'ils interagissent mieux avec le reste du système GNU/Linux.
- Les développeurs ayant de l'expérience avec un autre système de type UNIX intéressés par le développement de logiciels pour GNU/Linux. Vous pouvez être déjà familier avec des API standards comme celles de la spécification POSIX. Pour développer des logiciels GNU/-Linux, vous devez connaître les particularités du système, ses limitations, les possibilités supplémentaires qu'il offre et ses conventions.
- Les développeurs venant d'un système non-UNIX, comme la plate-forme Win32 de Microsoft. Vous devriez déjà connaître les principes permettant d'écrire des logiciels de bonne qualité, mais vous avez besoin d'apprendre les techniques spécifiques que les programmes GNU/Linux utilisent pour interagir avec le système et entre eux. Et vous voulez être sûr que vos programmes s'intègrent naturellement au sein du système GNU/Linux et se comportent selon les attentes des utilisateurs.

Ce livre n'a pas pour but d'être un guide ou une référence pour tous les aspects de la programmation GNU/Linux. Au lieu de cela, nous aurons une approche didactique, en introduisant les concepts et les techniques les plus importants et en donnant des exemples sur la façon de les utiliser. La Section 1.5, « Obtenir plus d'informations », dans le Chapitre 1, « Pour commencer », présente d'autres sources de documentation, où vous pourrez trouver des détails plus complets sur les différents aspects de la programmation GNU/Linux.

Comme ce livre traite de sujets avancés, nous supposerons que vous êtes déjà familier avec le langage C et que vous savez comment utiliser les fonctions de la bibliothèque standard du C dans vos programmes. Le langage C est le langage le plus utilisé dans le développement de logiciels GNU/Linux ; la plupart des commandes et des bibliothèques dont traite ce livre, et le noyau Linux lui-même, sont écrits en C.

Les informations présentées dans ce livre sont également applicables au C++ car il s'agit grossièrement d'un sur-ensemble du C. Même si vous programmez dans un autre langage, vous trouverez ces informations utiles car les API en langage C et leurs conventions constituent le sabir de GNU/Linux.

Si vous avez déjà programmé sur un autre système de type UNIX, il y a de bonnes chances que vous sachiez vous servir des fonctions d'E/S de bas niveau de Linux (open, read, stat, etc). Elles sont différentes des fonctions d'E/S de la bibliothèque standard du C (fopen, fprintf, fscanf, etc). Toutes sont utiles dans le cadre de la programmation GNU/Linux et nous utiliserons les deux jeux de fonctions d'E/S dans ce livre. Si vous n'êtes pas familiers avec les fonctions d'E/S de bas niveau, allez à la fin de ce livre et lisez l'Annexe B, « E/S de bas niveau », avant de commencer le Chapitre 2, « Écrire des logiciels GNU/Linux de qualité ».

Ce livre n'offre pas une introduction générale aux systèmes GNU/Linux. Nous supposons que vous avez déjà une connaissance basique de la façon d'interagir avec un système GNU/Linux et d'effectuer des opérations élémentaires dans des environnements en ligne de commande ou

graphiques. Si vous êtes nouveau sur GNU/Linux, commencez avec un des nombreux excellents livres d'introduction, comme *Inside Linux de Michael Tolber* (New Riders Publishing, 2001).

Conventions

Ce livre suit quelques conventions typographiques :

- Un nouveau terme est présenté en *italique* la première fois qu'il apparaît.
- Les listings de programme, les fonctions, les variables et autres « termes informatiques » sont présentés en police à chasse fixe – par exemple, `printf("Hello,␣World␣!\bksl␣\n")`.
- Les noms des commandes, des fichiers et des répertoires sont également en police à chasse fixe – par exemple, `cd /`.
- Lorsque nous montrons des interactions avec un shell de commandes, nous utiliserons `%` comme invite shell (votre shell est probablement configuré pour utiliser une invite différente). Tout ce qui se trouve après l'invite est ce que vous tapez, alors que les autres lignes de texte constituent la réponse du système. Par exemple, dans cette séquence :

  ```
  % uname
  Linux
  ```

  le système fournit une invite `%`. Vous entrez la commande `uname`. Le système répond en affichant `Linux`.
- Le titre de chaque listing inclut un nom de fichier entre parenthèses. Si vous saisissez le listing, sauvegardez-le dans un fichier portant ce nom. Vous pouvez également télécharger les codes sources depuis le site Web[3] de *Programmation Avancée Sous Linux* (`http://www.newriders.com` ou `http://www.advancedlinuxprogramming.com`).

Nous avons écrit ce livre et développé les programmes qu'il présente en utilisant la distribution Red Hat 6.2 de GNU/Linux. Cette distribution comprend la version 2.2.14 du noyau Linux, la version 2.1.3 de la bibliothèque GNU C et la version EGCS 1.1.2 du compilateur C GNU. Les informations et programmes de ce livre devraient être valables pour les autres versions et distibutions de GNU/Linux, y compris les séries 2.4 du noyau Linux et les séries 2.2 de la bibliothèque GNU C.

[3] En anglais.

Première partie

Programmation UNIX Avancée avec Linux

Table des Matières

Chapitre 1

Pour commencer

Ce chapitre présente les étapes de base nécessaires à la création d'un programme Linux en C ou C++. En particulier, il explique comment créer et modifier un code source C ou C++, compiler ce code et déboguer le résultat. Si vous êtes déjà familier avec la programmation sous Linux, vous pouvez aller directement au Chapitre 2, « Écrire des logiciels GNU/Linux de qualité » ; lisez attentivement la Section 2.3, « Écrire et utiliser des bibliothèques », pour plus d'informations sur la comparaison des éditions de liens statique et dynamique que vous ne connaissez peut-être pas.

Dans la suite de ce livre, nous supposerons que vous êtes familier avec le langage de programmation C ou C++ et avec les fonctions les plus courantes de la bibliothèque C standard. Les exemples de code source dans ce livre sont en C, excepté lorsqu'ils montrent une fonctionnalité ou une difficulté propre au C++. Nous supposerons également que vous savez comment effectuer les opérations élémentaires avec le shell de commande Linux, comme créer des répertoires et copier des fichiers. Comme beaucoup de programmeurs Linux ont commencé à programmer dans un environnement Windows, nous soulignerons occasionnellement les similitudes et les contrastes entre Windows et Linux.

1.1 L'éditeur Emacs

Un *éditeur* est le programme que vous utilisez pour éditer le code source. Beaucoup d'éditeurs différents sont disponibles sous Linux, mais le plus populaire et celui offrant le plus de fonctionnalités est certainement GNU Emacs.

Si vous êtes familier avec un autre éditeur, vous pouvez certainement l'utiliser à la place. Rien dans le reste du livre ne dépend de l'utilisation d'Emacs. Si vous n'avez pas déjà un éditeur favori sous Linux, vous pouvez continuer avec le mini-didacticiel fourni ici.

Si vous aimez Emacs et voulez en savoir plus sur ses fonctionnalités avancées, vous pouvez lire un des nombreux livres disponibles sur le sujet. Un excellent didacticiel, *Introduction à GNU Emacs*, a été écrit par Debra Cameron, Bill Rosenblatt et Eric Raymond (O'Reilly 1997).

À propos d'Emacs

Emacs est beaucoup plus qu'un simple éditeur. Il s'agit d'un programme incroyablement puissant, à tel point que chez CodeSourcery, il est appelé affectueusement le Seul Vrai Programme (*One True Program*), ou simplement OTP pour faire court. Vous pouvez écrire et lire vos e-mails depuis Emacs et vous pouvez le personnaliser et l'étendre de façon trop vaste pour que nous en parlions ici. Vous pouvez même surfer sur le Web depuis Emacs!

1.1.1 Ouvrir un fichier Source C ou C++

Vous pouvez lancer Emacs en saisissant `emacs` suivi de la touche Entrée dans votre terminal. Lorsque Emacs a démarré, vous pouvez utiliser les menus situés dans la partie supérieure pour créer un nouveau fichier source. Cliquez sur le menu **File**, sélectionnez **Open File** puis saisissez le nom du fichier que vous voulez ouvrir dans le *minibuffer* au bas de l'écran[1]. Si vous voulez créer un fichier source C, utilisez un nom de fichier se terminant par `.c` ou `.h`. Si vous désirez créer un fichier C++, utilisez un nom de fichier se terminant par `.cpp`, `.hpp`, `.cxx`, `.hxx`, `.C` ou `.H`. Lorsque le fichier est ouvert, vous pouvez taper comme vous le feriez dans un programme de traitement de texte. Pour sauvegarder le fichier, sélectionnez l'entrée **Save Buffer** dans le menu **File**. Lorsque vous avez terminé d'utiliser Emacs, vous pouvez choisir l'option **Exit Emacs** dans le menu **File**.

Si vous n'aimez pas cliquer, vous pouvez utiliser les raccourcis clavier pour ouvrir ou fermer un fichier et sortir d'Emacs. Pour ouvrir un fichier, saisissez `C-x C-f` (`C-x` signifie de maintenir la touche Control enfoncée tout en appuyant sur la touche x). Pour sauvegarder un fichier, saisissez `C-x C-s`. Pour sortir d'Emacs, saisissez simplement `C-x C-c`. Si vous voulez devenir un peu plus familier avec Emacs, sélectionnez l'entrée **Emacs Tutorial** dans le menu **Help**. Le didacticiel vous propose quantité d'astuces sur l'utilisation efficace d'Emacs.

1.1.2 Formatage automatique

Si vous êtes un habitué de la programmation dans un *Environnement de Développement Intégré* (Integrated Development Evironment, IDE), vous êtes habitué à l'assistance au formatage fourni par l'éditeur. Emacs peut vous offrir le même type de fonctionnalité. Si vous ouvrez un fichier C ou C++, Emacs devine qu'il contient du code, pas simplement du texte ordinaire. Si vous appuyez sur la touche Tab sur une ligne blanche, Emacs déplace le curseur au point d'indentation approprié. Si vous appuyez sur la touche Tab sur une ligne contenant déjà du texte, Emacs indente le texte. Donc, par exemple, supposons que vous ayez saisi ce qui suit :

```
int main()
{
printf("Hello, world\n");
}
```

Si vous pressez la touche Tab sur la ligne de l'appel à `printf`, Emacs reformatera votre code comme suit :

[1] Si vous n'utilisez pas un système X Window, vous devrez appuyer sur F10 pour accéder aux menus.

```
int main()
{
  printf("Hello, world\n");
}
```

Remarquez comment la ligne a été correctement indentée.

En utilisant Emacs, vous verrez comment il peut vous aider à effectuer toutes sortes de tâches de formatage compliquées. Si vous êtes ambitieux, vous pouvez programmer Emacs pour effectuer littéralement tout formatage que vous pourriez imaginer. Des gens ont utilisé ces fonctionnalités pour implémenter des modifications d'Emacs pour éditer à peu près n'importe quelle sorte de documents, implémenter des jeux[2] et des interfaces vers des bases de données.

1.1.3 Coloration syntaxique

En plus de formater votre code, Emacs peut faciliter la lecture du code C et C++ en colorant les différents éléments de sa syntaxe. Par exemple, Emacs peut colorer les mots clés d'une certaine façon, les types intégrés comme `int` d'une autre et les commentaires d'une autre encore. Utiliser des couleurs facilite la détection de certaines erreurs de syntaxe courantes

La façon la plus simple d'activer la coloration est d'éditer le fichier `~/.emacs` et d'y insérer la chaîne suivante :

```
(global-font-lock-mode t)
```

Sauvegardez le fichier, sortez d'Emacs et redémarrez-le. Ouvrez un fichier C ou C++ et admirez !

Vous pouvez avoir remarqué que la chaîne que vous avez inséré dans votre `.emacs` ressemble à du code LISP. C'est parce-qu'il s'*agit* de code LISP ! La plus grande partie d'Emacs est écrite en LISP. Vous pouvez ajouter des fonctionnalités à Emacs en écrivant en LISP.

1.2 Compiler avec GCC

Un *compilateur* transforme un code source lisible par un humain en code objet lisible par la machine qui peut être exécuté. Les compilateurs de choix disponibles sur les systèmes Linux font tous partie de la GNU Compiler Collection, plus communément appelée GCC[3] . GCC inclut également des compilateurs C, C++, Java, Objective-C, Fortran et Chill. Ce livre se concentre plus particulièrement sur la programmation C et C++.

Supposons que vous ayez un projet comme celui du Listing 1.2 avec un fichier source C++ (`reciprocal.cpp`) et un fichier source C (`main.c`) comme dans le Listing 1.1. Ces deux fichiers sont supposés être compilés puis liés entre eux pour produire un programme appelé `reciprocal`[4]. Ce programme calcule l'inverse d'un entier.

Listing 1.1 – (*main.c*) – **Fichier source C**

```
1 #include <stdio.h>
```

[2]Essayez la commande `M-x dunnet` si vous voulez jouer à un jeu d'aventures en mode texte à l'ancienne.

[3]Pour plus d'informations sur GCC, visitez `http://gcc.gnu.org/`.

[4]Sous Windows, les exécutables portent habituellement des noms se terminant en `.exe`. Les programmes Linux par contre, n'ont habituellement pas d'extension. Donc, l'équivalent Windows de ce programme s'appellerait probablement `reciprocal.exe` ; la version Linux est tout simplement `reciprocal`.

```
2  #include <stdlib.h>
3  #include "reciprocal.hpp"
4
5  int main (int argc, char **argv)
6  {
7    int i;
8
9    i = atoi (argv[1]);
10   printf ("L'inverse de %d est %g\n", i, reciprocal (i));
11   return 0;
12 }
```

Listing 1.2 – (*reciprocal.cpp*) – **Fichier source C++**

```
1  #include <cassert>
2  #include "reciprocal.hpp"
3
4  double reciprocal (int i) {
5    // i doit être différent de zéro
6    assert (i != 0);
7    return 1.0/i;
8  }
```

Il y a également un fichier d'entête appelé reciprocal.hpp (voir Listing 1.3).

Listing 1.3 – (*reciprocal.hpp*) – **Fichier d'entête**

```
1  #ifdef __cplusplus
2  extern "C" {
3  #endif
4
5  extern double reciprocal (int i);
6
7  #ifdef __cplusplus
8  }
9  #endif
```

La première étape est de traduire le code C et C++ en code objet.

1.2.1 Compiler un fichier source isolé

Le nom du compilateur C est `gcc`. Pour compiler un fichier source C, utilisez l'option `-c`. Donc par exemple, cette commande compile le fichier source `main.c` :

```
% gcc -c main.c
```

Le fichier objet résultant est appelé `main.o`. Le compilateur C++ s'appelle `g++`. Son mode opératoire est très similaire à `gcc` ; la compilation de `reciprocal.cpp` s'effectue via la commande suivante :

```
% g++ -c reciprocal.cpp
```

L'option `-c` indique à `g++` de ne compiler le fichier que sous forme d'un fichier objet ; sans cela, `g++` tenterait de lier le programme afin de produire un exécutable. Une fois cette commande saisie, vous obtenez un fichier objet appelé `reciprocal.o`.

Vous aurez probablement besoin de quelques autres options pour compiler un programme d'une taille raisonnable. L'option `-I` est utilisée pour indiquer à GCC où rechercher les fichiers

d'entête. Par défaut, GCC cherche dans le répertoire courant et dans les répertoires où les entêtes des bibliothèques standards sont installés. Si vous avez besoin d'inclure des fichiers d'entête situés à un autre endroit, vous aurez besoin de l'option `-I`. Par exemple, supposons que votre projet ait un répertoire appelé `src`, pour les fichiers source, et un autre appelé `include`. Vous compileriez `reciprocal.cpp` comme ceci pour indiquer à `g++` qu'il doit utiliser en plus le répertoire `include` pour trouver `reciprocal.hpp` :

```
% g++ -c -I ../include reciprocal.cpp
```

Parfois, vous pourriez vouloir définir des macros au niveau de la ligne de commande. Par exemple, dans du code de production, vous ne voudriez pas du surcoût de l'assertion présente dans `reciprocal.cpp` ; elle n'est là que pour vous aider à déboguer votre programme. Vous désactivez la vérification en définissant la macro `NDEBUG`. Vous pourriez ajouter un `#define` explicite dans `reciprocal.cpp`, mais cela nécessiterait de modifier la source elle-même. Il est plus simple de définir `NDEBUG` sur la ligne de commande, comme ceci :

```
% g++ -c -D NDEBUG reciprocal.cpp
```

Si vous aviez voulu donner une valeur particulière à `NDEBUG`, vous auriez pu saisir quelque chose de ce genre :

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

Si vous étiez réellement en train de compiler du code de production, vous voudriez probablement que GCC optimise le code afin qu'il s'exécute aussi rapidement que possible. Vous pouvez le faire en utilisant l'option en ligne de commande `-O2` (GCC a plusieurs niveaux d'optimisation ; le second niveau convient pour la plupart des programmes). Par exemple, ce qui suit compile `reciprocal.cpp` avec les optimisations activées :

```
% g++ -c -O2 reciprocal.cpp
```

Notez que le fait de compiler avec les optimisations peut rendre votre programme plus difficile à déboguer avec un débogueur (voyez la Section 1.4, « Déboguer avec le débogueur GNU (GDB) »). De plus, dans certaines circonstances, compiler avec les optimisations peut révéler des bogues qui n'apparaissaient pas auparavant.

Vous pouvez passer un certain nombre d'autres options à `gcc` et `g++`. Le meilleur moyen d'en obtenir une liste complète est de consulter la documentation en ligne. Vous pouvez le faire en saisissant ceci à l'invite de commandes :

```
% info gcc
```

1.2.2 Lier les fichiers objet

Maintenant que vous avez compilé `main.c` et `reciprocal.cpp`, vous devez les lier. Vous devriez toujours utiliser `g++` pour lier un programme qui contient du code C++, même s'il contient également du code C. Si votre programme ne contient que du code C, vous devriez utiliser `gcc` à la place. Comme ce programme contient à la fois du code C et du code C++, vous devriez utiliser `g++`, comme ceci :

```
% g++ -o reciprocal main.o reciprocal.o
```

L'option `-o` donne le nom du fichier à générer à l'issue de l'étape d'édition de liens. Vous pouvez maintenant lancer `reciprocal` comme ceci :

```
% ./reciprocal 7
L'inverse de 7 est 0.142857
```

Comme vous pouvez le voir, `g++` a automatiquement inclus les bibliothèques d'exécution C standards contenant l'implémentation de `printf`. Si vous aviez eu besoin de lier une autre bibliothèque (comme un kit de développement d'interfaces utilisateur), vous auriez indiqué la bibliothèque avec l'option `-l`. Sous Linux, les noms des bibliothèques commencent quasiment toujours par `lib`. Par exemple, la bibliothèque du Module d'Authentification Enfichable (Pluggable Authentication Module, PAM) est appelée `libpam.a`. Pour inclure `libpam.a` lors de l'édition de liens, vous utiliserez une commande de ce type :

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

Le compilateur ajoutera automatiquement le préfixe `lib` et le suffixe `.a`.

Comme avec les fichiers d'entête, l'éditeur de liens recherche les bibliothèques dans certains emplacements standards, ce qui inclut les répertoires `/lib` et `/usr/lib` qui contiennent les bibliothèques système standards. Si vous voulez que l'éditeur de liens cherche en plus dans d'autres répertoires, vous devez utiliser l'option `-L`, qui est l'équivalent de l'option `-I` dont nous avons parlé plus tôt. Vous pouvez utiliser cette commande pour indiquer à l'éditeur de liens de rechercher les bibliothèques dans le répertoire `/usr/local/lib/pam` avant de les rechercher dans les emplacements habituels :

```
% g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
```

Bien que vous n'ayez pas à utiliser l'option `-I` pour que le préprocesseur effectue ses recherches dans le répertoire courant, vous devez utiliser l'option `-L` pour que l'éditeur de liens le fasse. Par exemple, vous devrez utiliser ce qui suit pour indiquer à l'éditeur de liens de rechercher la bibliothèque test dans le répertoire courant :

```
% gcc -o app app.o -L. -ltest
```

1.3 Automatiser le processus avec GNU Make

Si vous êtes habitué à la programmation pour le système d'exploitation Windows, vous avez probablement l'habitude de travailler avec un Environnement de Développement Intégré (IDE). Vous ajoutez les fichiers à votre projet puis l'IDE compile ce projet automatiquement. Bien que des IDE soient disponibles pour Linux, ce livre n'en traite pas. Au lieu de cela, il vous montre comment vous servir de GNU Make pour recompiler votre code automatiquement, comme le font en fait la majorité des programmeurs Linux.

L'idée de base derrière `make` est simple. Vous indiquez à `make` quelles *cibles* vous désirez compiler puis donnez des *règles* expliquant comment les compiler. Vous pouvez également spécifier des *dépendances* qui indiquent quand une cible particulière doit être recompilée.

Dans notre projet exemple `reciprocal`, il y a trois cibles évidentes : `reciprocal.o`, `main.o` et `reciprocal` lui-même. Vous avez déjà à l'esprit les règles nécessaires à la compilation de ces cibles sous forme des lignes de commande données précédemment. Les dépendances nécessitent un minimum de réflexion. Il est clair que `reciprocal` dépend de `reciprocal.o` et `main.o` car vous ne pouvez pas passer à l'étape d'édition de liens avant d'avoir compilé chacun des fichiers objets. Les fichiers objets doivent être recompilés à chaque fois que le fichier source correspondant est modifié. Il y a encore une subtilité : une modification de `reciprocal.hpp` doit entraîner la recompilation des deux fichiers objets car les deux fichiers source incluent ce fichier d'entête.

En plus des cibles évidentes, il devrait toujours y avoir une cible `clean`. Cette cible supprime tous les fichiers objets générés afin de pouvoir recommencer sur des bases saines. La règle pour cette cible utilise la commande `rm` pour supprimer les fichiers.

Vous pouvez fournir toutes ces informations à `make` en les plaçant dans un fichier nommé `Makefile`. Voici ce qu'il contient :

```
reciprocal: main.o reciprocal.o
  g++ $(CFLAGS) -o reciprocal main.o reciprocal.o

main.o: main.c reciprocal.hpp
  gcc $(CFLAGS) -c main.c

reciprocal.o: reciprocal.cpp reciprocal.hpp
  g++ $(CFLAGS) -c reciprocal.cpp

clean:
  rm -f *.o reciprocal
```

Vous pouvez voir que les cibles sont listées sur la gauche, suivies de deux-points puis des dépendances. La règle pour la construction d'une cible est placée sur la ligne suivante (ignorez le `$(CFLAGS)` pour l'instant). La ligne décrivant la règle doit commencer par un caractère de tabulation ou `make` ne s'y retrouvera pas. Si vous éditez votre `Makefile` dans Emacs, Emacs vous assistera dans le formatage.

Si vous supprimez les fichiers objets que vous avez déjà créé et que vous tapez simplement :

```
% make
```

sur la ligne de commande, vous obtiendrez la sortie suivante :

```
% make
gcc -c main.c
g++ -c reciprocal.cpp
g++ -o reciprocal main.o reciprocal.o
```

Vous constatez que `make` a automatiquement compilé les fichiers objet puis les a liés. Si vous modifiez maintenant `main.c` d'une façon quelconque puis saisissez `make` de nouveau, vous obtiendrez la sortie suivante :

```
% make
gcc -c main.c
g++ -o reciprocal main.o reciprocal.o
```

Vous constatez que `make` recompile `main.o` et réédite les liens, il ne recompile pas `reciprocal.cpp` car aucune des dépendances de `reciprocal.o` n'a changé.

`$(CFLAGS)` est une variable de `make`. Vous pouvez définir cette variable soit dans le `Makefile` lui-même soit sur la ligne de commande. GNU `make` substituera la variable par sa valeur lorsqu'il exécutera la règle. Donc, par exemple, pour recompiler avec les optimisations activées, vous procéderiez de la façon suivante :

```
% make clean
rm -f *.o reciprocal
% make CFLAGS=-O2
gcc -O2 -c main.c
g++ -O2 -c reciprocal.cpp
g++ -O2 -o reciprocal main.o reciprocal.o
```

Notez que le drapeau `-O2` a été inséré à la place de `$(CFLAGS)` dans les règles.

Dans cette section, nous n'avons présenté que les capacités les plus basiques de `make`. Vous pourrez en apprendre plus grâce à la commande suivante :

```
% info make
```

Dans ce manuel, vous trouverez des informations sur la façon de rendre un `Makefile` plus simple à maintenir, comment réduire le nombre de règles à écrire et comment calculer automatiquement les dépendances. Vous pouvez également trouver plus d'informations dans *GNU, Autoconf, Automake et Libtool* de Gary V. Vaughan, Ben Ellitson, Tom Tromey et Ian Lance Taylor (New Riders Publishing, 2000).

1.4 Déboguer avec le débogueur GNU (GDB)

Le *débogueur* est le programme que vous utilisez pour trouver pourquoi votre programme ne se comporte pas comme vous pensez qu'il le devrait. Vous y aurez souvent recours[5]. Le débogueur GNU (*GNU debugger*, GDB) est le débogueur utilisé par la plupart des programmeurs Linux. Vous pouvez utiliser GDB pour exécuter votre code pas à pas, poser des points d'arrêt et examiner les valeurs des variables locales.

1.4.1 Compiler avec les informations de débogage

Pour utiliser GDB, vous devez compiler en activant les informations de débogage. Pour cela, ajoutez l'option `-g` sur la ligne de commande de compilation. Si vous utilisez un `Makefile` comme nous l'avons expliqué plus haut, vous pouvez vous contenter de positionner `CFLAGS` à `-g` lors de l'exécution de `make`, comme ceci :

```
% make CFLAGS=-g
gcc -g -c main.c
g++ -g -c reciprocal.cpp
g++ -g -o reciprocal main.o reciprocal.o
```

Lorsque vous compilez avec `-g`, le compilateur inclut des informations supplémentaires dans les fichiers objets et les exécutables. Le débogueur utilise ces informations pour savoir à quelle adresse correspond à quelle ligne et dans quel fichier source, afficher les valeurs des variables *et cætera*.

[5] . . . à moins que votre programme ne fonctionne toujours du premier coup.

1.4.2 Lancer GDB

Vous pouvez démarrer `gdb` en saisissant :

```
% gdb reciprocal
```

Lorsque GDB démarre, il affiche l'invite :

```
(gdb)
```

La première étape est de lancer votre programme au sein du débogueur. Entrez simplement la commande `run` et les arguments du programme. Essayez de lancer le programme sans aucun argument, comme ceci :

```
(gdb) run
Starting program: reciprocal

Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287     strtol.c: No such file or directory.
(gdb)
```

Le problème est qu'il n'y a aucun code de contrôle d'erreur dans main. Le programme attend un argument, mais dans ce cas, il n'en a reçu aucun. Le message `SIGSEGV` indique un plantage du programme. GDB sait que le plantage a eu lieu dans une fonction appelée `__strtol_internal`. Cette fonction fait partie de la bibliothèque standard, et les sources ne sont pas installées ce qui explique le message « No such file or directory » (Fichier ou répertoire inexistant). Vous pouvez observer la pile en utilisant la commande `where` :

```
(gdb) where
#0 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
    at strtol.c:287
#1 0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

Vous pouvez voir d'après cet extrait que `main` a appelé la fonction `atoi` avec un pointeur `NULL` ce qui est la source de l'erreur.

Vous pouvez remonter de deux niveaux dans la pile jusqu'à atteindre `main` en utilisant la commande `up` :

```
(gdb) up 2
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1]);
```

Notez que GDB est capable de trouver le fichier source `main.c` et qu'il affiche la ligne contenant l'appel de fonction erroné. Vous pouvez inspecter la valeurs des variables en utilisant la commande `print` :

```
(gdb) print argv[1]
$2 = 0x0
```

Cela confirme que le problème vient d'un pointeur `NULL` passé à `atoi`.

Vous pouvez placer un point d'arrêt en utilisant la commande `break` :

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

Cette commande place un point d'arrêt sur la première ligne de `main`[6]. Essayez maintenant de relancer le programme avec un argument, comme ceci :

```
(gdb) run 7
Starting program: reciprocal 7
Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8         i = atoi (argv[1])
```

Vous remarquez que le débogueur s'est arrêté au niveau du point d'arrêt.

Vous pouvez passer à l'instruction se trouvant après l'appel à `atoi` en utilisant la commande `next` :

```
 gdb) next
9         printf ("L'inverse de %d est %g\n", i, reciprocal (i));
```

Si vous voulez voir ce qui se passe à l'intérieur de la fonction `reciprocal`, utilisez la commande `step`, comme ceci :

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6         assert (i != 0);
```

Vous êtes maintenant au sein de la fonction `reciprocal`.

Vous pouvez trouver plus commode d'exécuter `gdb` au sein d'Emacs plutôt que de le lancer directement depuis la ligne de commande. Utilisez la commande `M-x gdb` pour démarrer `gdb` dans une fenêtre Emacs. Si vous stoppez au niveau d'un point d'arrêt, Emacs ouvre automatiquement le fichier source approprié. Il est plus facile de se rendre compte de ce qui se passe en visualisant le fichier dans son ensemble plutôt qu'une seule ligne.

1.5 Obtenir plus d'informations

Quasiment toutes les distributions Linux disposent d'une masse importante de documentation. Vous pourriez apprendre la plupart des choses dont nous allons parler dans ce livre simplement en lisant la documentation de votre distribution Linux (bien que cela vous prendrait probablement plus de temps). La documentation n'est cependant pas toujours très bien organisée, donc la partie la plus subtile est de trouver ce dont vous avez besoin. La documentation date quelquefois un peu, aussi, prenez tout ce que vous y trouvez avec un certain recul. Si le système ne se comporte pas comme le dit une *page de manuel*, c'est peut-être parce que celle-ci est obsolète.

Pour vous aider à naviguer, voici les sources d'information les plus utiles sur la programmation avancée sous Linux.

[6] Certaines personnes ont fait la remarque que `break main` (NdT. littéralement « casser main ») est amusant car vous ne vous en servez en fait uniquement lorsque main a déjà un problème.

1.5.1 Pages de manuel

Les distributions Linux incluent des pages de manuel pour les commandes les plus courantes, les appels système et les fonctions de la bibliothèque standard. Les pages de manuel sont divisées en sections numérotées ; pour les programmeurs, les plus importantes sont celles-ci :

(1) Commandes utilisateur

(2) Appels système

(3) Fonctions de la bibliothèque standard

(8) Commandes système/d'administration

Les numéros indiquent les sections des pages de manuel. Les pages de manuel de Linux sont installées sur votre système ; utilisez la commande man pour y accéder. Pour accéder à une page de manuel, invoquez simplement `man nom`, où `nom` est un nom de commande ou de fonction. Dans un petit nombre de cas, le même nom apparaît dans plusieurs sections ; vous pouvez indiquer explicitement la section en plaçant son numéro devant le nom. Par exemple, si vous saisissez la commande suivante, vous obtiendrez la page de manuel pour la commande `sleep` (dans la section 1 des pages de manuel Linux) :

```
% man sleep
```

Pour visualiser la page de manuel de la fonction `sleep` de la bibliothèque standard, utilisez cette commande :

```
% man 3 sleep
```

Chaque page de manuel comprend un résumé sur une ligne de la commande ou fonction. La commande `whatis` nom liste toutes les pages de manuel (de toutes les sections) pour une commande ou une fonction correspondant à nom. Si vous n'êtes pas sûr de la commande ou fonction à utiliser, vous pouvez effectuer une recherche par mot-clé sur les résumés via `man -k mot-clé`.

Les pages de manuel contiennent des informations très utiles et devraient être le premier endroit vers lequel vous orientez vos recherches. La page de manuel d'une commande décrit ses options en ligne de commande et leurs arguments, ses entrées et sorties, ses codes d'erreur, sa configuration et ce qu'elle fait. La page de manuel d'un appel système ou d'une fonction de bibliothèque décrit les paramètres et valeurs de retour, liste les codes d'erreur et les effets de bord et spécifie le fichier d'entête à inclure si vous utilisez la fonction.

1.5.2 Info

Le système de documentation Info contient des informations plus détaillées pour beaucoup de composants fondamentaux du système GNU/Linux et quelques autres programmes. Les pages Info sont des documents hypertextes, similaires aux pages Web. Pour lancer le navigateur texte Info, tapez simplement `info` à l'invite de commande. Vous obtiendrez un menu avec les documents Info présents sur votre système (appuyez sur `Ctrl+H` pour afficher les touches permettant de naviguer au sein d'un document Info).

Parmi les documents Info les plus utiles, on trouve :

gcc Le compilateur gcc

libc La bibliothèque C GNU, avec beaucoup d'appels système

gdb Le débogueur GNU

emacs L'éditeur de texte Emacs

info Le système Info lui-même

Presque tous les outils de programmation standards sous Linux (y compris `ld`, l'éditeur de liens ; `as`, l'assembleur et `gprof`, le profiler) sont accompagnés de pages Info très utiles. Vous pouvez accéder directement à un document Info en particulier en indiquant le nom de la page sur la ligne de commandes :

```
% info libc
```

Si vous programmez la plupart du temps sous Emacs, vous pouvez accéder au navigateur Info intégré en appuyant sur `M-x info` ou `C-h i`.

1.5.3 Fichiers d'entête

Vous pouvez en apprendre beaucoup sur les fonctions système disponibles et comment les utiliser en observant les fichiers d'entête. Ils sont placés dans `/usr/include` et `/usr/include/sys`. Si vous obtenez des erreurs de compilation lors de l'utilisation d'un appel système, par exemple, regardez le fichier d'entête correspondant pour vérifier que la signature de la fonction est la même que celle présentée sur la page de manuel.

Sur les systèmes Linux, beaucoup de détails obscurs sur le fonctionnement des appels systèmes apparaissent dans les fichiers d'entête situés dans les répertoires `/usr/include/bits`, `/usr/include/asm` et `/usr/include/linux`. Ainsi, le fichier `/usr/include/bits/signum.h` définit les valeurs numériques des signaux (décrits dans la Section 3.3, « Signaux » du Chapitre 3, « Processus »). Ces fichiers d'entête constituent une bonne lecture pour les esprits curieux. Ne les incluez pas directement dans vos programmes, cependant ; utilisez toujours les fichiers d'entête de `/usr/include` ou ceux mentionnés dans la page de manuel de la fonction que vous utilisez.

1.5.4 Code source

Nous sommes dans l'Open Source, non ? Le juge en dernier ressort de la façon dont doit fonctionner le système est le code source, et par chance pour les programmeurs Linux, ce code source est disponible librement. Il y a des chances pour que votre système Linux comprenne tout le code source du système et des programmes fournis ; si ce n'est pas le cas, vous avez le droit, selon les termes de la Licence Publique Générale GNU, de les demander au distributeur (le code source n'est toutefois pas forcément installé. Consultez la documentation de votre distribution pour savoir comment l'installer).

Le code source du noyau Linux lui-même est habituellement stocké sous `/usr/src/linux`. Si ce livre vous laisse sur votre faim concernant les détails sur le fonctionnement des processus, de la mémoire partagée et des périphériques système, vous pouvez toujours apprendre directement à partir du code. La plupart des fonctions décrites dans ce livre sont implémentées dans la bibliothèque C GNU ; consultez la documentation de votre distribution pour connaître l'emplacement du code source de la bibliothèque C.

Chapitre 2

Écrire des logiciels GNU/Linux de qualité

Ce chapitre présente quelques techniques de base utilisées par la plupart des programmeurs GNU/Linux. En suivant grossièrement les indications que nous allons présenter, vous serez à même d'écrire des programmes qui fonctionnent correctement au sein de l'environnement GNU/Linux et correspondent à ce qu'attendent les utilisateurs au niveau de leur façon de fonctionner.

2.1 Interaction avec l'environnement d'exécution

Lorsque vous avez étudié pour la première fois le langage C ou C++, vous avez appris que la fonction spéciale `main` est le point d'entrée principal pour un programme. Lorsque le système d'exploitation exécute votre programme, il offre un certain nombre de fonctionnalités qui aident le programme à communiquer avec le système d'exploitation et l'utilisateur. Vous avez probablement entendu parler des deux paramètres de `main`, habituellement appelés `argc` et `argv`, qui reçoivent les entrées de votre programme. Vous avez appris que `stdin` et `stdout` (ou les flux `cin` et `cout` en C++) fournissent une entrée et une sortie via la console. Ces fonctionnalités sont fournies par les langages C et C++, et elles interagissent avec le système d'une certaine façon. GNU/Linux fournit en plus d'autres moyens d'interagir avec l'environnement d'exécution.

2.1.1 La liste d'arguments

Vous lancez un programme depuis l'invite de commandes en saisissant le nom du programme. Éventuellement, vous pouvez passer plus d'informations au programme en ajoutant un ou plusieurs mots après le nom du programme, séparés par des espaces. Ce sont des *arguments de ligne de commande* (vous pouvez passer un argument contenant des espaces en le plaçant entre guillemets). Plus généralement, on appelle cela la *liste d'arguments* du programme car ils ne viennent pas nécessairement de la ligne de commande. Dans le Chapitre 3, « Processus », vous

verrez un autre moyen d'invoquer un programme, avec lequel un programme peut indiquer directement la liste d'arguments d'un autre programme.

Lorsqu'un programme est invoqué depuis la ligne de commande, la liste d'arguments contient toute la ligne de commande, y compris le nom du programme et tout argument qui aurait pu lui être passé. Supposons, par exemple, que vous invoquiez la commande `ls` depuis une invite de commandes pour afficher le contenu du répertoire racine et les tailles de fichiers correspondantes au moyen de cette commande :

```
% ls -s /
```

La liste d'arguments que le programme ls reçoit est composée de trois éléments. Le premier est le nom du programme lui-même, saisi sur la ligne de commande, à savoir `ls`. Les second et troisième élément sont les deux arguments de ligne de commande, `-s` et `/`.

La fonction `main` de votre programme peut accéder à la liste d'arguments *via* ses paramètres `argc` et `argv` (si vous ne les utilisez pas, vous pouvez simplement les omettre). Le premier paramètre, `argc`, est un entier qui indique le nombre d'éléments dans la liste. Le second paramètre, `argv`, est un tableau de pointeurs sur des caractères. La taille du tableau est `argc`, et les éléments du tableau pointent vers les éléments de la liste d'arguments, qui sont des chaînes terminées par zéro.

Utiliser des arguments de ligne de commande consiste à examiner le contenu de `argc` et `argv`. Si vous n'êtes pas intéressé par le nom du programme, n'oubliez pas d'ignorer le premier élément.

Le Listing 2.1 montre l'utilisation de `argv` et `argc`.

Listing 2.1 – (*arglist.c*) – **Utiliser *argc* et *argv***

```
#include <stdio.h>

int main (int argc, char* argv[])
{
  printf ("Le nom de ce programme est '%s'.\n", argv[0]);
  printf ("Ce programme a été invoqué avec %d arguments.\n", argc - 1);
  /* A-t-on spécifié des arguments sur la ligne de commande ? */
  if (argc > 1) {
    /* Oui, les afficher. */
    int i;
    printf ("Les arguments sont :\n");
    for (i = 1; i < argc; ++i)
      printf (" %s\n", argv[i]);
  }
  return 0;
}
```

2.1.2 Conventions de la ligne de commande GNU/Linux

Presque tous les programmes GNU/Linux obéissent à un ensemble de conventions concernant l'interprétation des arguments de la ligne de commande. Les arguments attendus par un programme sont classés en deux catégories : les *options* (ou *drapeaux*[1]) et les autres arguments. Les options modifient le comportement du programme, alors que les autres arguments fournissent des entrées (par exemple, les noms des fichiers d'entrée).

[1] NdT. *flags* en anglais

Les options peuvent prendre deux formes :

- Les *options courtes* sont formées d'un seul tiret et d'un caractère isolé (habituellement une lettre en majuscule ou en minuscule). Elles sont plus rapides à saisir.
- Les *options longues* sont formées de deux tirets suivis d'un nom composé de lettres majuscules, minuscules et de tirets. Les options longues sont plus faciles à retenir et à lire (dans les scripts shell par exemple).

Généralement, un programme propose une version courte et une version longue pour la plupart des options qu'il prend en charge, la première pour la brièveté et la seconde pour la lisibilité. Par exemple, la plupart des programmes acceptent les options `-h` et `-help` et les traitent de façon identique. Normalement, lorsqu'un programme est invoqué depuis la ligne de commande, les options suivent immédiatement le nom du programme. Certaines options attendent un argument immédiatement à leur suite. Beaucoup de programmes, par exemple, acceptent l'option `-output foo` pour indiquer que les sorties du programme doivent être redirigées vers un fichier appelé `foo`. Après les options, il peut y avoir d'autres arguments de ligne de commande, typiquement les fichiers ou les données d'entrée.

Par exemple, la commande `ls -s /` affiche le contenu du répertoire racine. L'option `-s` modifie le comportement par défaut de `ls` en lui demandant d'afficher la taille (en kilooctets) de chaque entrée. L'argument `/` indique à `ls` quel répertoire lister. L'option `-size` est synonyme de `-s`, donc la commande aurait pu être invoquée sous la forme `ls -size /`.

Les *Standards de Codage GNU* dressent la liste des noms d'options en ligne de commande couramment utilisés. Si vous avez l'intention de proposer des options identiques, il est conseillé d'utiliser les noms préconisés dans les standards de codage. Votre programme se comportera de façon similaire aux autres et sera donc plus simple à prendre en main pour les utilisateurs. Vous pouvez consulter les grandes lignes des Standards de Codage GNU à propos des options en ligne de commandes *via* la commande suivante depuis une invite de commande sur la plupart des systèmes GNU/Linux :

```
% info "(standards)User Interfaces"
```

2.1.3 Utiliser *getopt_long*

L'analyse des options de la ligne de commande est une corvée. Heureusement, la bibliothèque GNU C fournit une fonction que vous pouvez utiliser dans les programmes C et C++ pour vous faciliter la tâche (quoiqu'elle reste toujours quelque peu ennuyeuse). Cette fonction, `getopt_long`, interprète à la fois les options courtes et longues. Si vous utilisez cette fonction, incluez le fichier d'en-tête `<getopt.h>`.

Supposons par exemple que vous écriviez un programme acceptant les trois options du Tableau 2.1. Le programme doit par ailleurs accepter zéro ou plusieurs arguments supplémentaires, qui sont les noms de fichiers d'entrée.

Pour utiliser `getopt_long`, vous devez fournir deux structures de données. La première est une chaîne contenant les options courtes valables, chacune sur une lettre. Une option qui requiert un argument est suivie par deux-points. Pour notre programme, la chaîne ho :v indique que les options valides sont `-h`, `-o` et `-v`, la seconde devant être suivie d'un argument.

TAB. 2.1 – Exemple d'Options pour un Programme

Forme Courte	Forme Longue	Fonction
`-h`	`-help`	Affiche l'aide-mémoire et quitte
`-o nom fichier`	`-output nom fichier`	Indique le nom du fichier de sortie
`-v`	`-verbose`	Affiche des messages détaillés

Pour indiquer les options longues disponibles, vous devez construire un tableau d'éléments `struct option`. Chaque élément correspond à une option longue et dispose de quatre champs. Généralement, le premier champ est le nom de l'option longue (sous forme d'une chaîne de caractères, sans les deux tirets) ; le second est `1` si l'option prend un argument, `0` sinon ; le troisième est `NULL` et le quatrième est un caractère qui indique l'option courte synonyme de l'option longue. Tous les champs du dernier élément doivent être à zéro. Vous pouvez construire le tableau comme ceci :

```
const struct option long_options[] = {
   { "help",    0, NULL, 'h' },
   { "output",  1, NULL, 'o' },
   { "verbose", 0, NULL, 'v' },
   { NULL,      0, NULL, 0   }
};
```

Vous invoquez la fonction `getopt_long` en lui passant les arguments `argc` et `argv` de `main` , la chaîne de caractères décrivant les options courtes et le tableau d'éléments `struct option` décrivant les options longues.

- À chaque fois que vous appelez `getopt_long`, il n'analyse qu'une seule option et renvoie la lettre de l'option courte pour cette option ou `-1` s'il n'y a plus d'option à analyser.
- Typiquement, vous appelez `getopt_long` dans une boucle, pour traiter toutes les options que l'utilisateur a spécifié et en les gérant au sein d'une structure `switch`.
- Si `getopt_long` rencontre une option invalide (une option que vous n'avez pas indiquée comme étant une option courte ou longue valide), il affiche un message d'erreur et renvoie le caractère `?` (un point d'interrogation). La plupart des programmes s'interrompent dans ce cas, éventuellement après avoir affiché des informations sur l'utilisation de la commande.
- Lorsque le traitement d'une option requiert un argument, la variable globale `optarg` pointe vers le texte constituant cet argument.
- Une fois que `getopt_long` a fini d'analyser toutes les options, la variable globale `optind` contient l'index (dans `argv`) du premier argument qui n'est pas une option.

Le Listing 2.2 montre un exemple d'utilisation de `getopt_long` pour le traitement des arguments.

Listing 2.2 – (*getopt_long.c*) – **Utilisation de *getopt_long***

```
1  #include <getopt.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  /* Nom du programme.  */
5  const char* program_name;
6  /* Envoie les informations sur l'utilisation de la commande vers STREAM
7     (typiquement stdout ou stderr) et quitte le programme avec EXIT_CODE.
8     Ne retourne jamais. */
```

```
9  void print_usage (FILE* stream, int exit_code)
10 {
11   fprintf (stream, "Utilisation : %s options [fichierentrée ...]\n",
12                    program_name);
13   fprintf (stream,
14            " -h --help               Affiche ce message.\n"
15            " -o --output filename    Redirige la sortie vers un fichier.\n"
16            " -v --verbose            Affiche des messages détaillés.\n");
17   exit (exit_code);
18 }
19 /* Point d'entrée du programme. ARGC contient le nombre d'éléments de la liste
20    d'arguments ; ARGV est un tableau de pointeurs vers ceux-ci. */
21 int main (int argc, char* argv[])
22 {
23   int next_option;
24   /* Chaîne listant les lettres valides pour les options courtes. */
25   const char* const short_options = "ho:v";
26   /* Tableau décrivant les options longues valides. */
27   const struct option long_options[] = {
28     { "help",     0, NULL, 'h' },
29     { "output",   1, NULL, 'o' },
30     { "verbose", 0, NULL, 'v' },
31     { NULL,       0, NULL, 0   }   /* Requis à la fin du tableau.  */
32 };
33 /* Nom du fichier vers lequel rediriger les sorties, ou NULL pour
34     la sortie standard. */
35 const char* output_filename = NULL;
36 /* Indique si l'on doit afficher les messages détaillés. */
37 int verbose = 0;
38 /* Mémorise le nom du programme, afin de l'intégrer aux messages.
39     Le nom est contenu dans argv[0]. */
40 program_name = argv[0];
41 do {
42    next_option = getopt_long (argc, argv, short_options,
43                               long_options, NULL);
44    switch (next_option)
45    {
46    case 'h':   /* -h or --help */
47      /* L'utilisateur a demandé l'aide-mémoire. L'affiche sur la sortie
48         standard et quitte avec le code de sortie 0 (fin normale). */
49      print_usage (stdout, 0);
50    case 'o':   /* -o ou --output */
51      /* Cette option prend un argument, le nom du fichier de sortie.  */
52      output_filename = optarg;
53      break;
54    case 'v':   /* -v ou --verbose */
55      verbose = 1;
56      break;
57    case '?':   /* L'utilisateur a saisi une option invalide. */
58      /* Affiche l'aide-mémoire sur le flux d'erreur et sort avec le code
59         de sortie un (indiquant une fin anormale). */
60      print_usage (stderr, 1);
61    case -1:    /* Fin des options.  */
62      break;
63    default:    /* Quelque chose d'autre : inattendu.  */
64      abort ();
65    }
66 }
67 while (next_option != -1);
68 /* Fin des options. OPTIND pointe vers le premier argument qui n'est pas une
69     option. À des fins de démonstration, nous les affichons si l'option
70     verbose est spécifiée. */
```

```
71    if (verbose) {
72      int i;
73      for (i = optind; i < argc; ++i)
74        printf ("Argument : %s\n", argv[i]);
75    }
76    /* Le programme principal se place ici.  */
77    return 0;
78  }
```

L'utilisation de `getopt_long` peut sembler nécessiter beaucoup de travail, mais écrire le code nécessaire à l'analyse des options de la ligne de commandes vous-même vous prendrait encore plus longtemps. La fonction `getopt_long` est très sophistiquée et permet une grande flexibilité dans la spécification des types d'options acceptées. Cependant, il est bon de se tenir à l'écart des fonctionnalités les plus avancées et de conserver la structure d'options basiques décrite ici.

2.1.4 E/S standards

La bibliothèque standard du C fournit des flux d'entrée et de sortie standards (`stdin` et `stdout` respectivement). Il sont utilisés par `printf`, `scanf` et d'autres fonctions de la bibliothèque. Dans la tradition UNIX, l'utilisation de l'entrée et de la sortie standard est fréquente pour les programmes GNU/Linux. Cela permet l'enchaînement de plusieurs programmes au moyen des pipes[2] et de la redirection des entrées et sorties (consultez la page de manuel de votre shell pour savoir comment les utiliser).

La bibliothèque C fournit également `stderr`, le flux d'erreurs standard. Il est d'usage que les programmes envoient les messages d'erreur et d'avertissement vers la sortie des erreurs standard plutôt que vers la sortie standard. Cela permet aux utilisateurs de séparer les messages normaux des messages d'erreur, par exemple, en redirigeant la sortie standard vers un fichier tout en laissant les erreurs s'afficher sur la console. La fonction `fprintf` peut être utilisée pour écrire sur `stderr`, par exemple :

```
% fprintf (stderr, "Erreur : ...");
```

Ces trois flux sont également accessibles via les commandes d'E/S UNIX de bas niveau (`read`, `write`, *etc*), par le biais des descripteurs de fichiers. Les descripteurs de fichiers sont `0` pour `stdin`, `1` pour `stdout` et `2` pour `stderr`.

Lors de l'appel d'un programme, il peut être utile de rediriger à la fois la sortie standard et la sortie des erreurs vers un fichier ou un pipe. La syntaxe à utiliser diffère selon les shells ; la voici pour les shells de type Bourne (y compris `bash`, le shell par défaut sur la plupart des distributions GNU/Linux) :

```
% programme > fichier_sortie.txt 2>&1
% programme 2>&1 | filtre
```

La syntaxe `2>&1` indique que le descripteur de fichiers `2` (`stderr`) doit être fusionné avec le descripteur de fichiers `1` (`stdout`). Notez que `2>&1` doit être placé après une redirection vers un fichier (premier exemple) mais avant une redirection vers un pipe (second exemple).

Notez que `stdout` est bufferisée. Les données écrites sur stdout ne sont pas envoyées vers la console (ou un autre dispositif si l'on utilise la redirection) avant que le tampon ne soit plein,

[2] NdT. appelés aussi parfois tubes ou canaux.

que le programme ne se termine normalement ou que `stdout` soit fermé. Vous pouvez purger explicitement le tampon de la façon suivante :

```
fflush (stdout);
```

Par contre, `stderr` n'est pas bufferisée ; les données écrites sur `stderr` sont envoyées directement vers la console[3].

Cela peut conduire à des résultats quelque peu surprenants. Par exemple, cette boucle n'affiche pas un point toutes les secondes ; au lieu de cela, les points sont placés dans le tampon, et ils sont affichés par groupe lorsque le tampon est plein.

```
while (1) {
  printf (".");
  sleep (1);
}
```

Avec cette boucle, par contre, les points sont affichés au rythme d'un par seconde :

```
while (1) {
  fprintf (stderr, ".");
  sleep (1);
}
```

2.1.5 Codes de sortie de programme

Lorsqu'un programme se termine, il indique son état au moyen d'un code de sortie. Le code de sortie est un entier court ; par convention, un code de sortie à zéro indique une fin normale, tandis qu'un code différent de zéro signale qu'une erreur est survenue. Certains programmes utilisent des codes de sortie différents de zéro variés pour distinguer les différentes erreurs.

Avec la plupart des shells, il est possible d'obtenir le code de sortie du dernier programme exécuté en utilisant la variable spéciale `$?`. Voici un exemple dans lequel la commande ls est invoquée deux fois et son code de sortie est affiché après chaque invocation. Dans le premier cas, `ls` se termine correctement et renvoie le code de sortie `0`. Dans le second cas, `ls` rencontre une erreur (car le fichier spécifié sur la ligne de commande n'existe pas) et renvoie donc un code de sortie différent de `0`.

```
% ls /
bin   coda etc    lib         misc nfs proc
boot dev    home lost+found mnt     opt root
% echo $?
0
% ls fichierinexistant
ls: fichierinexistant: Aucun fichier ou répertoire de ce type
% echo $?
1
```

Un programme C ou C++ donne son code de sortie en le retournant depuis la fonction `main`. Il y a d'autres méthodes pour fournir un code de sortie et des codes de sortie spéciaux sont assignés aux programmes qui se terminent de façon anormale (sur un signal). Ils sont traités de manière plus approfondie dans le Chapitre 3.

[3] En C++, la même distinction s'applique à `cout` et `cerr`, respectivement. Notez que le token `endl` purge un flux en plus d'y envoyer un caractère de nouvelle ligne ; si vous ne voulez pas purger le flux (pour des raisons de performances par exemple), utilisez une constante de nouvelle ligne, `'n'`, à la place.

2.1.6 L'environnement

GNU/Linux fournit à tout programme s'exécutant un *environnement*. L'environnement est une collection de paires variable/valeur. Les variables d'environnement et leurs valeurs sont des chaînes de caractères. Par convention, les variables d'environnement sont en majuscules d'imprimerie.

Vous êtes probablement déjà familier avec quelques variables d'environnement courantes. Par exemple :

USER contient votre nom d'utilisateur.

HOME contient le chemin de votre répertoire personnel.

PATH contient une liste de répertoires séparés par deux-points dans lesquels Linux recherche les commandes que vous invoquez.

DISPLAY contient le nom et le numéro d'affichage du serveur X Window sur lequel apparaissent les fenêtres des programmes graphiques X Window.

Votre shell, comme n'importe quel autre programme, dispose d'un environnement. Les shells fournissent des méthodes pour examiner et modifier l'environnement directement. Pour afficher l'environnement courant de votre shell, invoquez le programme `printenv`. Tous les shells n'utilisent pas la même syntaxe pour la manipulation des variables d'environnement ; voici la syntaxe pour les shells de type Bourne :

- Le shell crée automatiquement une variable shell pour chaque variable d'environnement qu'il trouve, afin que vous puissiez accéder aux valeurs des variables d'environnement en utilisant la syntaxe `$nomvar`. Par exemple :

  ```
  % echo $USER
  samuel
  % echo $HOME
  /home/samuel
  ```

- Vous pouvez utiliser la commande export pour exporter une variable shell vers l'environnement. Par exemple, pour positionner la variable d'environnement `EDITOR`, vous utiliserez ceci :

  ```
  % EDITOR=emacs
  % export EDITOR
  ```

  Ou, pour faire plus court :

  ```
  % export EDITOR=emacs
  ```

Dans un programme, vous pouvez accéder à une variable d'environnement au moyen de la fonction `getenv` de `<stdlib.h>`. Cette fonction accepte le nom d'une variable et renvoie la valeur correspondante sous forme d'une chaîne de caractères ou `NULL` si cette variable n'est pas définie dans l'environnement. Pour positionner ou supprimer une variable d'environnement, utilisez les fonctions `setenv` et `unsetenv`, respectivement.

Énumérer toutes les variables de l'environnement est un petit peu plus subtil. Pour cela, vous devez accéder à une variable globale spéciale appelée environ, qui est définie dans la bibliothèque C GNU. Cette variable, de type `char**`, est un tableau terminé par `NULL` de pointeurs vers des chaînes de caractères. Chaque chaîne contient une variable d'environnement, sous la forme `VARIABLE=valeur`.

Le programme du Listing 2.3, par exemple, affiche tout l'environnement en bouclant sur le tableau `environ`.

Listing 2.3 – (*print-env.c*) – **Afficher l'Environnement d'Exécution**

```
#include <stdio.h>
/* La variable ENVIRON contient l'environnement. */
extern char** environ;
int main ()
{
  char** var;
  for (var = environ; *var != NULL; ++var)
    printf ("%s\n", *var);
  return 0;
}
```

Ne modifiez pas environ vous-même ; utilisez plutôt les fonctions `setenv` et `getenv`.

Lorsqu'un nouveau programme est lancé, il hérite d'une copie de l'environnement du programme qui l'a invoqué (le shell, s'il a été invoqué de façon interactive). Donc, par exemple, les programmes que vous lancez depuis le shell peuvent examiner les valeurs des variables d'environnement que vous positionnez dans le shell.

Les variables d'environnement sont couramment utilisées pour passer des paramètres de configuration aux programmes. Supposons, par exemple, que vous écriviez un programme qui se connecte à un serveur Internet pour obtenir des informations. Vous pourriez écrire le programme de façon à ce que le nom du serveur soit saisi sur la ligne de commande. Cependant, supposons que le nom du serveur ne soit pas quelque chose que les utilisateurs changent très souvent. Vous pouvez utiliser une variable d'environnement spéciale – disons `SERVER_NAME` – pour spécifier le nom du serveur ; si cette variable n'existe pas, une valeur par défaut est utilisée. Une partie de votre programme pourrait ressembler au Listing 2.4.

Listing 2.4 – (*client.c*) – **Extrait d'un Programme Client Réseau**

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  char* server_name = getenv ("SERVER_NAME");
  if (server_name == NULL)
    /* La variable SERVER_NAME n'est pas définie. Utilisation de la valeur
       par défaut. */
    server_name = "server.my-company.com";
  printf ("Accès au serveur %s\n", server_name);
  /* Accéder au serveur ici... */
  return 0;
}
```

Supposons que ce programme s'appelle `client`. En admettant que vous n'ayez pas défini la variable `SERVER_NAME`, la valeur par défaut pour le nom du serveur est utilisée :

```
% client
Accès au serveur server.my-company.com
```

Mais il est facile de spécifier un serveur différent :

```
% export SERVER_NAME=backup-server.elsewhere.net
% client
Accès au serveur backup-server.elsewhere.net
```

2.1.7 Utilisation de fichiers temporaires

Parfois, un programme a besoin de créer un fichier temporaire, pour stocker un gros volume de données temporairement ou passer des informations à un autre programme. Sur les systèmes GNU/Linux, les fichiers temporaires sont stockés dans le répertoire `/tmp`. Lors de l'utilisation de fichiers temporaires, vous devez éviter les pièges suivants :

- Plus d'une copie de votre programme peuvent être lancées simultanément (par le même utilisateur ou par des utilisateurs différents). Les copies devraient utiliser des noms de fichiers temporaires différents afin d'éviter les collisions.
- Les permissions du fichier devraient être définies de façon à éviter qu'un utilisateur non autorisé puisse altérer la manière dont s'exécute le programme en modifiant ou remplaçant le fichier temporaire.
- Les noms des fichiers temporaires devraient être générés de façon imprévisible de l'extérieur ; autrement, un attaquant pourrait exploiter le délai entre le test d'existence du nom de fichier et l'ouverture du nouveau fichier temporaire.

GNU/Linux fournit des fonctions, `mkstemp` et `tmpfile`, qui s'occupent de ces problèmes à votre place (en plus de fonctions qui ne le font pas). Le choix de la fonction dépend de l'utilisation que vous aurez du fichier, à savoir le passer à un autre programme ou utiliser les fonctions d'E/S UNIX (`open`, `write`, *etc.*) ou les fonctions de flux d'E/S de la bibliothèque C (`fopen`, `fprintf`, *etc.*).

Utilisation de mkstemp

La fonction `mkstemp` crée un nom de fichier temporaire à partir d'un modèle de nom de fichier, crée le fichier avec les permissions adéquates afin que seul l'utilisateur courant puisse y accéder, et ouvre le fichier en lecture/écriture. Le modèle de nom de fichier est une chaîne de caractères se terminant par `"XXXXXX"` (six X majuscules) ; `mkstemp` remplace les `X` par des caractères afin que le nom de fichier soit unique. La valeur de retour est un descripteur de fichier ; utilisez les fonctions de la famille de `write` pour écrire dans le fichier temporaire.

Les fichiers temporaires créés par `mkstemp` ne sont pas effacés automatiquement. C'est à vous de supprimer le fichier lorsque vous n'en avez plus besoin (les programmeurs devraient être attentifs à supprimer les fichiers temporaires ; dans le cas contraire, le système de fichiers `/tmp` pourrait se remplir, rendant le système inutilisable). Si le fichier temporaire n'est destiné qu'à être utilisé par le programme et ne sera pas transmis à un autre programme, c'est une bonne idée d'appeler `unlink` sur le fichier temporaire immédiatement. La fonction `unlink` supprime l'entrée de répertoire correspondant à un fichier, mais comme le système tient à jour un décompte du nombre de références sur chaque fichier, un fichier n'est pas effacé tant qu'il reste un descripteur de fichier ouvert pour ce fichier. Comme Linux ferme les descripteurs de fichiers quand un programme se termine, le fichier temporaire sera effacé même si votre programme se termine de manière anormale.

Les deux fonctions du Listing 2.5 présentent l'utilisation de `mkstemp`. Utilisées ensemble, ces fonctions facilitent l'écriture d'un tampon mémoire vers un fichier temporaire (afin que la mémoire puisse être libérée ou réutilisée) et sa relecture ultérieure.

Listing 2.5 – (*temp_file.c*) – **Utiliser *mkstemp***

```
#include <stdlib.h>
#include <unistd.h>
/* Handle sur un fichier temporaire créé avec write_temp_file. Avec
   cette implémentation, il s'agit d'un descripteur de fichier. */
typedef int temp_file_handle;
/* Écrit LENGTH octets de BUFFER dans un fichier temporaire. Unlink est
   appelé immédiatement sur le fichier temporaire. Renvoie un handle sur
   le fichier temporaire. */
temp_file_handle write_temp_file (char* buffer, size_t length)
{
  /* Crée le nom du fichier et le fichier. XXXXXX sera remplacé par des
     caractères donnant un nom de fichier unique. */
  char temp_filename[] = "/tmp/temp_file.XXXXXX";
  int fd = mkstemp (temp_filename);
  /* Appelle unlink immédiatement afin que le fichier soit supprimé dès que
     le descripteur sera fermé. */
  unlink (temp_filename);
  /* Écrit le nombre d'octets dans le fichier avant tout. */
  write (fd, &length, sizeof (length));
  /* Écrit des données proprement dites. */
  write (fd, buffer, length);
  /* Utilise le descripteur de fichier comme handle
     sur le fichier temporaire. */
  return fd;
}
/* Lit le contenu du fichier temporaire TEMP_FILE créé avec
   write_temp_file. La valeur de retour est un tampon nouvellement alloué avec
   ce contenu, que l'appelant doit libérer avec free.
   *LENGTH est renseigné avec la taille du contenu, en octets.
   Le fichier temporaire est supprimé. */
char* read_temp_file (temp_file_handle temp_file, size_t* length)
{
  char* buffer;
  /* Le handle sur TEMP_FILE est le descripteur du fichier temporaire. */
  int fd = temp_file;
  /* Se place au début du fichier. */
  lseek (fd, 0, SEEK_SET);
  /* Lit les données depuis le fichier temporaire. */
  read (fd, length, sizeof (*length));
  /* Alloue un buffer et lit les données. */
  buffer = (char*) malloc (*length);
  read (fd, buffer, *length);
  /* Ferme le descripteur de fichier ce qui provoque la suppression du
     fichier temporaire. */
  close (fd);
  return buffer;
}
```

Utilisation de tmpfile

Si vous utilisez les fonctions d'E/S de la bibliothèque C et n'avez pas besoin de passer le fichier temporaire à un autre programme, vous pouvez utiliser la fonction `tmpfile`. Elle crée et ouvre un fichier temporaire, et renvoie un pointeur de fichier. Le fichier temporaire a déjà été traité par `unlink`, comme dans l'exemple précédent, afin d'être supprimé automatiquement lorsque le pointeur sur le fichier est fermé (avec `fclose`) ou lorsque le programme se termine.

GNU/Linux propose diverses autres fonctions pour générer des fichiers temporaires et des noms de fichiers temporaires, par exemple, `mktemp`, `tmpnam` et `tempnam`. N'utilisez pas ces fonctions, cependant, car elles souffrent des problèmes de fiabilité et de sécurité mentionnés plus haut.

2.2 Créer du code robuste

Écrire des programmes s'exécutant correctement dans des conditions d'utilisation "normales" est dur ; écrire des programmes qui se comportent avec élégance dans des conditions d'erreur l'est encore plus. Cette section propose quelques techniques de codage pour trouver les bogues plus tôt et pour détecter et traiter les problèmes dans un programme en cours d'exécution.

Les exemples de code présentés plus loin dans ce livre n'incluent délibérément pas de code de vérification d'erreur ou de récupération sur erreur car cela risquerait d'alourdir le code et de masquer la fonctionnalité présentée. Cependant, l'exemple final du Chapitre 11, « Application GNU/Linux d'Illustration », est là pour montrer comment utiliser ces techniques pour produire des applications robustes.

2.2.1 Utiliser *assert*

Un bon objectif à conserver à l'esprit en permanence lorsque l'on code des programmes est que des bogues ou des erreurs inattendues devraient conduire à un crash du programme, dès que possible. Cela vous aidera à trouver les bogues plus tôt dans les cycles de développement et de tests. Il est difficile de repérer les dysfonctionnements qui ne se signalent pas d'eux-mêmes et n'apparaissent pas avant que le programme soit à la disposition de l'utilisateur.

Une des méthodes les plus simples pour détecter des conditions inattendues est la macro C standard `assert`. Elle prend comme argument une expression booléenne. Le programme s'arrête si l'expression est fausse, après avoir affiché un message d'erreur contenant le nom du fichier, le numéro de ligne et le texte de l'expression où l'erreur est survenue. La macro `assert` est très utile pour une large gamme de tests de cohérence internes à un programme. Par exemple, utilisez `assert` pour vérifier la validité des arguments passés à une fonction, pour tester des préconditions et postconditions lors d'appels de fonctions (ou de méthodes en C++) et pour tester des valeurs de retour inattendues.

Chaque utilisation de assert constitue non seulement une vérification de condition à l'exécution mais également une documentation sur le fonctionnement du programme au cœur du code source. Si votre programme contient une instruction `assert(condition)` cela indique à quelqu'un lisant le code source que condition devrait toujours être vraie à ce point du programme et si condition n'est pas vraie, il s'agit probablement d'un bogue dans le programme.

Pour du code dans lequel les performances sont essentielles, les vérifications à l'exécution comme celles induites par l'utilisation de assert peuvent avoir un coût significatif en termes de performances. Dans ce cas, vous pouvez compiler votre code en définissant la macro `NDEBUG`, en utilisant l'option `-DNDEBUG` sur la ligne de commande du compilateur. Lorsque `NDEBUG` est définie, le préprocesseur supprimera les occurrences de la macro `assert`. Il est conseillé de ne le faire que lorsque c'est nécessaire pour des raisons de performances et uniquement pour des fichiers sources concernés par ces questions de performances.

Comme il est possible que le préprocesseur supprime les occurrences de `assert`, soyez attentif à ce que les expressions que vous utilisez avec `assert` n'aient pas d'effet de bord. En particulier, vous ne devriez pas appeler de fonctions au sein d'expressions `assert`, y affecter des valeurs à des variables ou utiliser des opérateurs de modification comme `++`.

Supposons, par exemple, que vous appeliez une fonction, `do_something`, de façon répétitive dans une boucle. La fonction `do_something` renvoie zéro en cas de succès et une valeur différente de zéro en cas d'échec, mais vous ne vous attendez pas à ce qu'elle échoue dans votre programme. Vous pourriez être tenté d'écrire :

```
for (i = 0; i < 100; ++i)
  assert (do_something () == 0);
```

Cependant, vous pourriez trouver que cette vérification entraîne une perte de performances trop importante et décider plus tard de recompiler avec la macro `NDEBUG` définie. Cela supprimerait totalement l'appel à `assert`, l'expression ne serait donc jamais évaluée et `do_something` ne serait jamais appelée. Voici un extrait de code effectuant la même vérification, sans ce problème :

```
for (i = 0; i < 100; ++i) {
  int status = do_something ();
  assert (status == 0);
}
```

Un autre élément à conserver à l'esprit est que vous ne devez pas utiliser `assert` pour tester les entrées utilisateur. Les utilisateurs n'apprécient pas lorsque les applications plantent en affichant un message d'erreur obscur, même en réponse à une entrée invalide. Vous devriez cependant toujours vérifier les saisies de l'utilisateur et afficher des messages d'erreurs compréhensibles. N'utilisez `assert` que pour des tests internes lors de l'exécution.

Voici quelques exemples de bonne utilisation d'assert :

- Vérification de pointeurs nuls, par exemple, comme arguments de fonction invalides. Le message d'erreur généré par `{assert (pointer != NULL)}`,

  ```
  Assertion 'pointer != ((void *)0)' failed.
  ```

  est plus utile que le message d'erreur qui serait produit dans le cas du déréfencement d'un pointeur nul :

  ```
  Erreur de Segmentation
  ```

- Vérification de conditions concernant la validité des paramètres d'une fonction. Par exemple, si une fonction ne doit être appelée qu'avec une valeur positive pour le paramètre `foo`, utilisez cette expression au début de la fonction :

  ```
  assert (foo > 0);
  ```

  Cela vous aidera à détecter les mauvaises utilisations de la fonction, et montre clairement à quelqu'un lisant le code source de la fonction qu'il y a une restriction quant à la valeur du paramètre.

Ne vous retenez pas, utilisez assert librement partout dans vos programmes.

2.2.2 Problèmes lors d'appels système

La plupart d'entre nous a appris comment écrire des programmes qui s'exécutent selon un chemin bien défini. Nous divisons le programme en tâches et sous-tâches et chaque fonction

accomplit une tâche en invoquant d'autres fonctions pour effectuer les opérations correspondant aux sous-tâches. On attend d'une fonction qu'étant donné des entrées précises, elle produise une sortie et des effets de bord corrects.

Les réalités matérielles et logicielles s'imposent face à ce rêve. Les ordinateurs ont des ressources limitées ; le matériel subit des pannes ; beaucoup de programmes s'exécutent en même temps ; les utilisateurs et les programmeurs font des erreurs. C'est souvent à la frontière entre les applications et le système d'exploitation que ces réalités se manifestent. Aussi, lors de l'utilisation d'appels système pour accéder aux ressources, pour effectuer des E/S ou à d'autres fins, il est important de comprendre non seulement ce qui se passe lorsque l'appel fonctionne mais également comment et quand l'appel peut échouer.

Les appels systèmes peuvent échouer de plusieurs façons. Par exemple :

- Le système n'a plus de ressources (ou le programme dépasse la limite de ressources permises pour un seul programme). Par exemple, le programme peut tenter d'allouer trop de mémoire, d'écrire trop de données sur le disque ou d'ouvrir trop de fichiers en même temps.
- Linux peut bloquer un appel système lorsqu'un programme tente d'effectuer une opération non permise. Par exemple, un programme pourrait tenter d'écrire dans un fichier en lecture seule, d'accéder à la mémoire d'un autre processus ou de tuer un programme d'un autre utilisateur.
- Les arguments d'un appel système peuvent être invalides, soit parce-que l'utilisateur a fourni des entrées invalides, soit à cause d'un bogue dans le programme. Par exemple, le programme peut passer une adresse mémoire ou un descripteur de fichier invalide à un appel système ; ou un programme peut tenter d'ouvrir un répertoire comme un fichier régulier ou passer le nom d'un fichier régulier à un appel système qui attend un répertoire.
- Un appel système peut échouer pour des raisons externes à un programme. Cela arrive le plus souvent lorsqu'un appel système accède à un périphérique matériel. Ce dernier peut être défectueux, ne pas supporter une opération particulière ou un lecteur peut être vide.
- Un appel système peut parfois être interrompu par un événement extérieur, comme l'arrivée d'un signal. Il ne s'agit pas tout à fait d'un échec de l'appel, mais il est de la responsabilité du programme appelant de relancer l'appel système si nécessaire.

Dans un programme bien écrit qui utilise abondamment les appels système, il est courant qu'il y ait plus de code consacré à la détection et à la gestion d'erreurs et d'autres circonstances exceptionnelles qu'à la fonction principale du programme.

2.2.3 Codes d'erreur des appels système

Une majorité des appels système renvoie zéro si tout se passe bien ou une valeur différente de zéro si l'opération échoue (toutefois, beaucoup dérogent à la règle ; par exemple, `malloc` renvoie un pointeur nul pour indiquer une erreur. Lisez toujours la page de manuel attentivement lorsque vous utilisez un appel système). Bien que cette information puisse être suffisante pour déterminer si le programme doit continuer normalement, elle ne l'est certainement pas pour une récupération fiable des erreurs.

La plupart des appels système utilisent une variable spéciale appelée errno pour stocker des

informations additionnelles en cas d'échec[4]. Lorsqu'un appel échoue, le système positionne errno à une valeur indiquant ce qui s'est mal passé. Comme tous les appels système utilisent la même variable `errno` pour stocker des informations sur une erreur, vous devriez copier sa valeur dans une autre variable immédiatement après l'appel qui a échoué. La valeur de `errno` sera écrasée au prochain appel système.

Les valeurs d'erreur sont des entiers ; les valeurs possibles sont fournies par des macros préprocesseur, libellées en majuscules par convention et commençant par « E » ? par exemple, `EACCESS` ou `EINVAL`. Utilisez toujours ces macros lorsque vous faites référence à des valeurs de `errno` plutôt que les valeurs entières. Incluez le fichier d'entête `<errno.h>` si vous utilisez des valeurs de `errno`.

GNU/Linux propose une fonction utile, `strerror`, qui renvoie une chaîne de caractères contenant la description d'un code d'erreur de `errno`, utilisable dans les messages d'erreur. Incluez `<string.h>` si vous désirez utiliser `strerror`.

GNU/Linux fournit aussi `perror`, qui envoie la description de l'erreur directement vers le flux `stderr`. Passez à `perror` une chaîne de caractères à ajouter avant la description de l'erreur, qui contient habituellement le nom de la fonction qui a échoué. Incluez `<stdio.h>` si vous utilisez `perror`.

Cet extrait de code tente d'ouvrir un fichier ; si l'ouverture échoue, il affiche un message d'erreur et quitte le programme. Notez que l'appel de `open` renvoie un descripteur de fichier si l'appel se passe correctement ou `-1` dans le cas contraire.

```
fd = open ("inputfile.txt", O_RDONLY);
if (fd == -1) {
  /* L'ouverture a échoué, affiche un message d'erreur et quitte. */
  fprintf (stderr, "erreur lors de l'ouverture de : %s\n", strerror (errno));
  exit (1);
}
```

Selon votre programme et la nature de l'appel système, l'action appropriée lors d'un échec peut être d'afficher un message d'erreur, d'annuler une opération, de quitter le programme, de réessayer ou même d'ignorer l'erreur. Il est important cependant d'avoir du code qui gère toutes les raisons d'échec d'une façon ou d'une autre.

Un code d'erreur possible auquel vous devriez particulièrement vous attendre, en particulier avec les fonctions d'E/S, est `EINTR`. Certaines fonctions, comme `read`, `select` et `sleep`, peuvent mettre un certain temps à s'exécuter. Elles sont considérées comme étant bloquantes car l'exécution du programme est bloquée jusqu'à ce que l'appel se termine. Cependant, si le programme reçoit un signal alors qu'un tel appel est en cours, celui-ci se termine sans que l'opération soit achevée. Dans ce cas, `errno` est positionnée à `EINTR`. En général, vous devriez relancer l'appel système dans ce cas.

Voici un extrait de code qui utilise l'appel `chown` pour faire de l'utilisateur `user_id` le propriétaire d'un fichier désigné par `path`. Si l'appel échoue, le programme agit selon la valeur de `errno`. Notez que lorsque nous détectons ce qui semble être un bogue, nous utilisons `abort` ou `assert`, ce qui provoque la génération d'un fichier core. Cela peut être utile pour un débogage postmortem. Dans le cas d'erreurs irrécupérables, comme des conditions de manque de mémoire,

[4]En réalité, pour des raisons d'isolement de threads, `errno` est implémentée comme une macro, mais elle est utilisée comme une variable globale.

nous utilisons plutôt `exit` et une valeur de sortie différente de zéro car un fichier core ne serait pas vraiment utile.

```
rval = chown (path, user_id, -1);
if (rval != 0) {
  /* Sauvegarde errno car il sera écrasé par le prochain appel système */
  int error_code = errno;
  /* L'opération a échoué ; chown doit retourner -1 dans ce cas. */
  assert (rval == -1);
  /* Effectue l'action appropriée en fonction de la valeur de errno. */
  switch (error_code) {
  case EPERM:       /* Permission refusée. */
  case EROFS:       /* PATH est sur un système de fichiers en lecture seule */
  case ENAMETOOLONG: /* PATH est trop long. */
  case ENOENT:       /* PATH n'existe pas. */
  case ENOTDIR:      /* Une des composantes de PATH n'est pas un répertoire */
  case EACCES:       /* Une des composantes de PATH n'est pas accessible. */
    /* Quelque chose ne va pas. Affiche un message d'erreur. */
    fprintf (stderr, "erreur lors du changement de propriétaire de %s: %s\n",
             path, strerror (error_code));

    /* N'interrompt pas le programme ; possibilité de proposer à l'utilisateur
       de choisir un autre fichier... */
    break;
  case EFAULT:
    /* PATH contient une adresse mémoire invalide.
       Il s'agit sûrement d'un bogue */
    abort ();
  case ENOMEM:
    /* Plus de mémoire disponible. */
    fprintf (stderr, "%s\n", strerror (error_code));
    exit (1);
  default:
     /* Autre code d'erreur innatendu. Nous avons tenté de gérer tous les
        codes d'erreur possibles ; si nous en avons oublié un
        il s'agit d'un bogue */
    abort ();
  };
}
```

Vous pourriez vous contenter du code suivant qui se comporte de la même façon si l'appel se passe bien :

```
rval = chown (path, user_id, -1);
assert (rval == 0);
```

Mais en cas d'échec, cette alternative ne fait aucune tentative pour rapporter, gérer ou reprendre après l'erreur.

L'utilisation de la première ou de la seconde forme ou de quelque chose entre les deux dépend des besoins en détection et récupération d'erreur de votre programme.

2.2.4 Erreurs et allocation de ressources

Souvent, lorsqu'un appel système échoue, il est approprié d'annuler l'opération en cours mais de ne pas terminer le programme car il peut être possible de continuer l'exécution suite à cette

erreur. Une façon de le faire est de sortir de la fonction en cours en renvoyant un code de retour qui indique l'erreur.

Si vous décidez de quitter une fonction au milieu de son exécution, il est important de vous assurer que toutes les ressources allouées précédemment au sein de la fonction sont libérées. Ces ressources peuvent être de la mémoire, des descripteurs de fichier, des pointeurs sur des fichiers, des fichiers temporaires, des objets de synchronisation, etc. Sinon, si votre programme continue à s'exécuter, les ressources allouées préalablement à l'échec de la fonction seront perdues.

Considérons par exemple une fonction qui lit un fichier dans un tampon. La fonction pourrait passer par les étapes suivantes :

1. Allouer le tampon ;
2. Ouvrir le fichier ;
3. Lire le fichier dans le tampon ;
4. Fermer le fichier ;
5. Retourner le tampon.

Le Listing 2.6 montre une façon d'écrire cette fonction.

Si le fichier n'existe pas, l'Étape 2 échouera. Une réponse appropriée à cet événement serait que la fonction retourne `NULL`. Cependant, si le tampon a déjà été alloué à l'Étape 1, il y a un risque de perdre cette mémoire. Vous devez penser à libérer le tampon dans chaque bloc de la fonction qui en provoque la sortie. Si l'Étape 3 ne se déroule pas correctement, vous devez non seulement libérer le tampon mais également fermer le fichier.

Listing 2.6 – (*readfile.c*) – **Libérer les Ressources**

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
char* read_from_file (const char* filename, size_t length)
{
  char* buffer;
  int fd;
  ssize_t bytes_read;
  /* Alloue le tampon. */
  buffer = (char*) malloc (length);
  if (buffer == NULL)
    return NULL;
  /* Ouvre le fichier. */
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    /* L'ouverture a échoué. Libère le tampon avant de quitter. */
    free (buffer);
    return NULL;
  }
  /* Lit les données. */
  bytes_read = read (fd, buffer, length);
  if (bytes_read != length) {
    /* La lecture a échoué. Libère le tampon et ferme fd avant de quitter. */
    free (buffer);
    close (fd);
    return NULL;
  }
```

```
30    /* Tout va bien. Ferme le fichier et renvoie le tampon. */
31    close (fd);
32    return buffer;
33  }
```

Linux libère la mémoire, ferme les fichiers et la plupart des autres ressources automatiquement lorsqu'un programme se termine, il n'est donc pas nécessaire de libérer les tampons et de fermer les fichiers avant d'appeler `exit`. Vous pourriez néanmoins devoir libérer manuellement d'autres ressources partagées, comme les fichiers temporaires et la mémoire partagée, qui peuvent potentiellement survivre à un programme.

2.3 Écrire et utiliser des bibliothèques

Pratiquement tous les programmes sont liés à une ou plusieurs bibliothèques. Tout programme qui utilise une fonction C (comme `printf` ou `malloc`) sera lié à la bibliothèque d'exécution C. Si votre programme a une interface utilisateur graphique (Graphical User Interface, GUI), il sera lié aux bibliothèques de fenêtrage. Si votre programme utilise une base de données, le fournisseur de base de données vous fournira des bibliothèques permettant d'accéder à la base de donnée de façon pratique.

Dans chacun de ces cas, vous devez décider si la bibliothèque doit être liée de façon *statique* ou *dynamique*. Si vous choisissez la liaison statique, votre programme sera plus gros et plus difficile à mettre à jour, mais probablement plus simple à déployer. Si vous optez pour la liaison dynamique, votre programme sera petit, plus simple à mettre à jour mais plus compliqué à déployer. Cette section explique comment effectuer une liaison statique et dynamique, examine les deux options en détail et donne quelques règles empiriques pour décider quelle est la meilleure dans votre cas.

2.3.1 Archives

Une *archive* (ou bibliothèque statique) est tout simplement une collection de fichiers objets stockée dans un seul fichier objet (une archive est en gros équivalent à un fichier `.LIB` sous Windows). Lorsque vous fournissez une archive à l'éditeur de liens, il recherche au sein de cette archive les fichiers dont il a besoin, les extrait et les lie avec votre programme comme si vous aviez fourni ces fichiers objets directement.

Vous pouvez créer une archive en utilisant la commande `ar`. Les fichiers archives utilisent traditionnellement une extension `.a` plutôt que l'extension `.o` utilisée par les fichiers objets ordinaires. Voici comment combiner `test1.o` et `test2.o` dans une seule archive `libtest.a` :

```
% ar cr libtest.a test1.o test2.o
```

Le drapeau `cr` indique à `ar` de créer l'archive[5]. Vous pouvez maintenant lier votre programme avec cette archive en utilisant l'option `-ltest` avec `gcc` ou `g++`, comme le décrit la Section 1.2.2, « Lier les fichiers objet » du Chapitre 1, « Pour commencer ».

[5] Vous pouvez utiliser d'autres options pour supprimer un fichier d'une archive ou effectuer d'autres opérations sur l'archive. Ces opérations sont rarement utilisées mais sont documentées sur la page de manuel de `ar`.

Lorsque l'éditeur de liens détecte une archive sur la ligne de commande, il y recherche les définitions des symboles (fonctions ou variables) qui sont référencés dans les fichiers objets qui ont déjà été traités mais ne sont pas encore définis. Les fichiers objets qui définissent ces symboles sont extraits de l'archive et inclus dans l'exécutable final. Comme l'éditeur de liens effectue une recherche dans l'archive lorsqu'il la rencontre sur la ligne de commande, il est habituel de placer les archives à la fin de la ligne de commande. Par exemple, supposons que `test.c` contienne le code du Listing 2.7 et que `app.c` contienne celui du Listing 2.8.

Listing 2.7 – (*test.c*) – **Contenu de la bibliothèque**

```
1 int f ()
2 {
3     return 3;
4 }
```

Listing 2.8 – (*app.c*) – **Programme Utilisant la Bibliothèque**

```
1 int main()
2 {
3     return f ();
4 }
```

Supposons maintenant que `test.o` soit combiné à un autre fichier objet quelconque pour produire l'archive `libtest.a`. La ligne de commande suivante ne fonctionnerait pas :

```
% gcc -o app -L. -ltest app.o
app.o: In function "main":
app.o(.text+0x4): undefined reference to "f"
collect2: ld returned 1 exit status
```

Le message d'erreur indique que même si `libtest.a` contient la définition de `f`, l'éditeur de liens ne la trouve pas. C'est dû au fait que `libtest.a` a été inspectée lorsqu'elle a été détectée pour la première fois et à ce moment l'éditeur de liens n'avait pas rencontré de référence à `f`.

Par contre, si l'on utilise la ligne de commande suivante, aucun message d'erreur n'est émis :

```
% gcc -o app app.o -L. -ltest
```

La raison en est que la référence à `f` dans `app.o` oblige l'éditeur de liens à inclure le fichier objet `test.o` depuis l'archive `libtest.a`.

2.3.2 Bibliothèques partagées

Une *bibliothèque partagée* (également appelée objet partagé ou bibliothèque dynamique) est similaire à une archive en ceci qu'il s'agit d'un groupe de fichiers objets. Cependant, il y a beaucoup de différences importantes. La différence la plus fondamentale est que lorsqu'une bibliothèque partagée est liée à un programme, l'exécutable final ne contient pas vraiment le code présent dans la bibliothèque partagée. Au lieu de cela, l'exécutable contient simplement une référence à cette bibliothèque. Si plusieurs programmes sur le système sont liés à la même bibliothèque partagée, ils référenceront tous la bibliothèque, mais aucun ne l'inclura réellement. Donc, la bibliothèque est « partagée » entre tous les programmes auxquels elle est liée.

Une seconde différence importante est qu'une bibliothèque partagée n'est pas seulement une collection de fichiers objets, parmi lesquels l'éditeur de liens choisit ceux qui sont nécessaires pour satisfaire les références non définies. Au lieu de cela, les fichiers objets qui composent la bibliothèque sont combinés en un seul fichier objet afin qu'un programme lié à la bibliothèque partagée inclut toujours tout le code de la bibliothèque plutôt que de n'inclure que les portions nécessaires.

Pour créer une bibliothèque partagée, vous devez compiler les objets qui constitueront la bibliothèque en utilisant l'option `-fPIC` du compilateur, comme ceci :

```
% gcc -c -fPIC test1.c
```

L'option `-fPIC` indique au compilateur que vous allez utiliser `test1.o` en tant qu'élément d'un objet partagé.

Code Indépendant de la Position (Position-Independent Code, PIC)

PIC signifie code indépendant de la position. Les fonctions d'une bibliothèque partagée peuvent être chargées à différentes adresses dans différents programmes, le code de l'objet partagé ne doit donc pas dépendre de l'adresse (ou position) à laquelle il est chargé. Cette considération n'a pas d'impact à votre niveau, en tant que programmeur, excepté que vous devez vous souvenir d'utiliser l'option `-fPIC` lors de la compilation du code utilisé pour la bibliothèque partagée.

Puis, vous combinez les fichiers objets au sein d'une bibliothèque partagée, comme ceci :

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

L'option `-shared` indique à l'éditeur de liens de créer une bibliothèque partagée au lieu d'un exécutable ordinaire. Les bibliothèques partagées utilisent l'extension `.so`, ce qui signifie objet partagé (shared object). Comme pour les archives statiques, le nom commence toujours par `lib` pour indiquer que le fichier est une bibliothèque.

Lier un programme à une bibliothèque partagée se fait de la même manière que pour une archive statique. Par exemple, la ligne suivante liera le programme à `libtest.so` si elle est dans le répertoire courant ou dans un des répertoires de recherche de bibliothèques du système :

```
% gcc -o app app.o -L. -ltest
```

Supposons que `libtest.a` et `libtest.so` soient disponibles. L'éditeur de liens doit choisir une seule des deux bibliothèques. Il commence par rechercher dans chaque répertoire (tout d'abord ceux indiqués par l'option `-L`, puis dans les répertoires standards). Lorsque l'éditeur de liens trouve un répertoire qui contient soit `libtest.a` soit `libtest.so`, il interrompt ses recherches. Si une seule des deux variantes est présente dans le répertoire, l'éditeur de liens la sélectionne. Sinon, il choisit la version partagée à moins que vous ne lui spécifiiez explicitement le contraire. Vous pouvez utiliser l'option `-static` pour utiliser les archives statiques. Par exemple, la ligne de commande suivante utilisera l'archive `libtest.a`, même si la bibliothèque partagée `libtest.so` est disponible :

```
% gcc -static -o app app.o -L. -ltest
```

La commande `ldd` indique les bibliothèques partagées liées à un programme. Ces bibliothèques doivent être disponibles à l'exécution du programme. Notez que `ldd` indiquera une bibliothèque supplémentaire appelée `ld-linux.so` qui fait partie du mécanisme de liaison dynamique de GNU/Linux.

Utiliser *LD_LIBRARY_PATH*

Lorsque vous liez un programme à une bibliothèque partagée, l'éditeur de liens ne place pas le chemin d'accès complet à la bibliothèque dans l'exécutable. Il n'y place que le nom de la bibliothèque partagée. Lorsque le programme est exécuté, le système recherche la bibliothèque partagée et la charge. Par défaut, cette recherche n'est effectuée que dans `/lib` et `/usr/lib` par défaut. Si une bibliothèque partagée liée à votre programme est placée à un autre endroit que dans ces répertoires, elle ne sera pas trouvée et le système refusera de lancer le programme.

Une solution à ce problème est d'utiliser les options `-Wl`, `-rpath` lors de l'édition de liens du programme. Supposons que vous utilisiez ceci :

```
% gcc -o app app.o -L. -ltest -Wl,-rpath,/usr/local/lib
```

Dans ce cas, lorsqu'on lance le programme `app`, recherche les bibliothèques nécessaires dans `/usr/local/lib`.

Une autre solution à ce problème est de donner une valeur à la variable d'environnement `LD_LIBRARY_PATH` au lancement du programme. Tout comme la variable d'environnement `PATH`, `LD_LIBRARY_PATH` est une liste de répertoires séparés par deux-points. Par exemple, si vous positionnez `LD_LIBRARY_PATH` à `/usr/local/lib:/opt/lib` alors les recherches seront effectuées au sein de `/usr/local/lib` et `/opt/lib` avant les répertoires standards `/lib` et `/usr/lib`. Vous devez être conscient que si `LD_LIBRARY_PATH` est renseigné, l'éditeur de liens recherchera les bibliothèques au sein des répertoires qui y sont spécifiés avant ceux passés via l'option `-L` lorsqu'il crée un exécutable[6].

2.3.3 Bibliothèques standards

Même si vous ne spécifiez aucune bibliothèque lorsque vous compilez votre programme, il est quasiment certain qu'il utilise une bibliothèque partagée. Car GCC lie automatiquement la bibliothèque standard du C, `libc`, au programme. Les fonctions mathématiques de la bibliothèque standard du C ne sont pas inclues dans `libc` ; elles sont placées dans une bibliothèque séparée, `libm`, que vous devez spécifier explicitement. Par exemple, pour compiler et lier un programme `compute.c` qui utilise des fonctions trigonométriques comme `sin` et `cos`, vous devez utiliser ce code :

```
% gcc -o compute compute.c -lm
```

Si vous écrivez un programme C++ et le liez en utilisant les commandes `c++` ou `g++`, vous aurez également la bibliothèque standard du C++, `libstdc++`.

[6] Vous pourriez voir des références à `LD_RUN_PATH` dans certaines documentations en ligne. Ne croyez pas ce que vous lisez, cette variable n'a en fait aucun effet sous GNU/Linux.

2.3.4 Dépendances entre bibliothèques

Les bibliothèques dépendent souvent les unes des autres. Par exemple, beaucoup de systèmes GNU/Linux proposent `libtiff`, une bibliothèque contenant des fonctions pour lire et écrire des fichiers images au format TIFF. Cette bibliothèque utilise à son tour les bibliothèques `libjpeg` (routines de manipulation d'images JPEG) et `libz` (routines de compression).

Le Listing 2.9 présente un très petit programme qui utilise `libtiff` pour ouvrir une image TIFF.

Listing 2.9 – (*tifftest.c*) – **Utilisation de libtiff**

```
1 #include <stdio.h>
2 #include <tiffio.h>
3 int main (int argc, char** argv)
4 {
5   TIFF* tiff;
6   tiff = TIFFOpen (argv[1], "r");
7   TIFFClose (tiff);
8   return 0;
9 }
```

Sauvegardez ce fichier sous le nom de `tifftest.c`. Pour le compiler et le lier à `libtiff`, spécifiez `-ltiff` sur votre ligne de commande :

```
% gcc -o tifftest tifftest.c -ltiff
```

Par défaut, c'est la version partagée de la bibliothèque `libtiff` qui sera utilisée, il s'agit du fichier `/usr/lib/libtiff.so`. Comme `libtiff` utilise `libjpeg` et `libz`, les versions partagées de ces bibliothèque sont également liées (une bibliothèque partagée peut pointer vers d'autres bibliothèques partagées dont elle dépend). Pour vérifier cela, utilisez la commande `ldd` :

```
% ldd tifftest
        libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
        libc.so.6 => /lib/libc.so.6 (0x40060000)
        libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
        libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
        ...
```

Pour lier ce programme de façon statique, vous devez spécifier les deux autres bibliothèques :

```
% gcc -static -o tifftest tifftest.c -ltiff -ljpeg -lz -lm
```

De temps en temps, deux bibliothèques dépendent mutuellement l'une de l'autre. En d'autres termes, la première archive fait référence à des symboles définis dans la seconde et *vice versa*. Cette situation résulte généralement d'une mauvaise conception mais existe. Dans ce cas, vous pouvez fournir une même bibliothèque plusieurs fois sur la ligne de commande. L'éditeur de liens recherchera les symboles manquants à chaque apparition de la bibliothèque. Par exemple, cette ligne provoque deux recherches de symboles au sein de `libfoo` :

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

Donc, même si `libfoo.a` fait référence à des symboles de `libbar.a` et *vice versa*, le programme sera compilé avec succès.

2.3.5 Avantages et inconvénients

Maintenant que vous savez tout à propos des archives statiques et des bibliothèques partagées, vous vous demandez probablement lesquelles utiliser. Il y a quelques éléments à garder à l'esprit.

Un avantage majeur des bibliothèques partagées est qu'elles permettent d'économiser de la place sur le système où le programme est installé. Si vous installez dix programmes et qu'ils utilisent tous la même bibliothèque partagée, vous économiserez beaucoup d'espace à utiliser une bibliothèque partagée. Si vous utilisez une archive statique à la place, l'archive est incluse dans les dix programmes. Donc, utiliser des bibliothèques partagées permet d'économiser de l'espace disque. Cela réduit également le temps de téléchargement si votre programme est distribué via le Web.

Un avantage lié aux bibliothèques partagées est que les utilisateurs peuvent mettre à niveau les bibliothèques sans mettre à niveau tous les programmes qui en dépendent. Par exemple, supposons que vous créiez une bibliothèque partagée qui gère les connexions HTTP. Beaucoup de programmes peuvent dépendre de cette bibliothèque. Si vous découvrez un bogue dans celle-ci, vous pouvez mettre à niveau la bibliothèque. Instantanément, tous les programmes qui en dépendent bénéficieront de la correction ; vous n'avez pas à repasser par une étape d'édition de liens pour tous les programmes, comme vous le feriez avec une archive statique.

Ces avantages pourraient vous faire penser que vous devez toujours utiliser les bibliothèques partagées. Cependant, il existe des raisons valables pour utiliser les archives statiques. Le fait qu'une mise à jour de la bibliothèque affecte tous les programmes qui en dépendent peut représenter un inconvénient. Par exemple, si vous développez un logiciel critique, il est préférable de le lier avec une archive statique afin qu'une mise à jour des bibliothèques sur le système hôte n'affecte pas votre programme (autrement, les utilisateurs pourraient mettre à jour les bibliothèques, empêchant votre programme de fonctionner, puis appeler votre service client en disant que c'est de votre faute !).

Si vous n'êtes pas sûr de pouvoir installer vos bibliothèques dans `/lib` ou `/usr/lib`, vous devriez définitivement réfléchir à deux fois avant d'utiliser une bibliothèque partagée (vous ne pourrez pas installer vos bibliothèques dans ces répertoires si votre logiciel est destiné à pouvoir être installé par des utilisateurs ne disposant pas des droits d'administrateur). De plus, l'astuce de l'option `-Wl`, `-rpath` ne fonctionnera pas si vous ne savez pas où seront placées les bibliothèques en définitive. Et demander à vos utilisateurs de positionner `LD_LIBRARY_PATH` leur impose une étape de configuration supplémentaire. Comme chaque utilisateur doit le faire individuellement, il s'agit d'une contrainte additionnelle non négligeable.

Il faut bien peser ces avantages et inconvénients pour chaque programme que vous distribuez.

2.3.6 Chargement et déchargement dynamiques

Il est parfois utile de pouvoir charger du code au moment de l'exécution sans lier ce code explicitement. Par exemple, considérons une application qui supporte des « plugins », comme un navigateur Web. Le navigateur permet à des développeurs tiers de créer des plugins pour fournir des fonctionnalités supplémentaires. Ces développeurs créent des bibliothèques partagées et les placent à un endroit prédéfini. Le navigateur charge alors automatiquement le code de ces bibliothèques.

Cette fonctionnalité est disponible sous Linux en utilisant la fonction `dlopen`. Vous ouvrez une bibliothèque appelée `libtest.so` en appelant `dlopen` comme ceci :

```
dlopen ("libtest.so", RTLD_LAZY)
```

(Le second paramètre est un drapeau qui indique comment lier les symboles de la bibliothèque partagée. Vous pouvez consulter les pages de manuel de `dlopen` pour plus d'informations, mais `RTLD_LAZY` est généralement l'option conseillée). Pour utiliser les fonctions de chargement dynamique, incluez l'entête `<dlfcn.h>` et liez avec l'option `-ldl` pour inclure la bibliothèque `libdl`.

La valeur de retour de cette fonction est un `void *` utilisé comme référence sur la bibliothèque partagée. Vous pouvez passer cette valeur à la fonction `dlsym` pour obtenir l'adresse d'une fonction chargée avec la bibliothèque partagée. Par exemple, si `libtest.so` définit une fonction appelée `my_function`, vous pourriez l'appeler comme ceci :

```
void* handle = dlopen ("libtest.so", RTLD_LAZY);
void (*test)() = dlsym (handle, "my_function");
(*test)();
dlclose (handle);
```

L'appel système `dlsym` peut également être utilisé pour obtenir un pointeur vers une variable `static` de la bibliothèque partagée.

`dlopen` et `dlsym` renvoient toutes deux `NULL` si l'appel échoue. Dans ce cas, vous pouvez appeler `dlerror` (sans paramètre) pour obtenir un message d'erreur lisible décrivant le problème.

La fonction `dlclose` décharge la bibliothèque. Techniquement, `dlopen` ne charge la bibliothèque que si elle n'est pas déjà chargée. Si elle l'est, `dlopen` se contente d'incrémenter le compteur de références pour la bibliothèque. De même, `dlclose` décrémente ce compteur et ne décharge la bibliothèque que s'il a atteint zéro.

Si vous écrivez du code en C++ dans votre bibliothèque partagée, vous devriez déclarer les fonctions et variables que vous voulez rendre accessible de l'extérieur avec le modificateur `extern "C"`. Par exemple, si la fonction C++ `my_function` est dans une bibliothèque partagée et que vous voulez la rendre accessible par `dlsym`, vous devriez la déclarer comme suit :

```
extern "C" void my_function ();
```

Cela évite que le compilateur C++ ne modifie le nom de la fonction ce qui résulterait en un nom différent à l'aspect sympathique qui encode des informations supplémentaires sur la fonction. Un compilateur C ne modifie pas les noms ; il utilise toujours les noms que vous donnez à votre fonction ou variable.

Chapitre 3

Processus

Une instance d'un programme en cours d'exécution est appelée un *processus*. Si vous avez deux fenêtres de terminal sur votre écran, vous exécutez probablement deux fois le même programme de terminal – vous avez deux processus de terminal. Chaque fenêtre de terminal exécute probablement un shell ; chaque shell en cours d'exécution est un processus indépendant. Lorsque vous invoquez un programme depuis le shell, le programme correspondant est exécuté dans un nouveau processus ; le processus shell reprend lorsque ce processus ce termine.

Les programmeurs expérimentés utilisent souvent plusieurs processus coopérant entre eux au sein d'une même application afin de lui permettre de faire plus d'une chose à la fois, pour améliorer sa robustesse et utiliser des programmes déjà existants.

La plupart des fonctions de manipulation de processus décrites dans ce chapitre sont similaires à celles des autres systèmes UNIX. La majorité d'entre elles est déclarée au sein de l'entête `<unistd.h>` ; vérifiez la page de manuel de chaque fonction pour vous en assurer.

3.1 Introduction aux processus

Même lorsque tout ce que vous faites est être assis devant votre ordinateur, des processus sont en cours d'exécution. Chaque programme utilise un ou plusieurs processus. Commençons par observer les processus déjà présents sur votre ordinateur.

3.1.1 Identifiants de processus

Chaque processus d'un système Linux est identifié par son *identifiant de processus* unique, quelquefois appelé *pid* (*process ID*). Les identifiants de processus sont des nombres de 16 bits assignés de façon séquentielle par Linux aux processus nouvellement créés.

Chaque processus a également un processus parent (sauf le processus spécial `init`, décrit dans la Section 3.4.3, « Processus zombies »). Vous pouvez donc vous représenter les processus d'un système Linux comme un arbre, le processus `init` étant la racine. *L'identifiant de processus parent* (*parent process ID*), ou *ppid*, est simplement l'identifiant du parent du processus.

Lorsque vous faites référence aux identifiants de processus au sein d'un programme C ou C++, utilisez toujours le `typedef pid_t`, défini dans `<sys/types.h>`. Un programme peut obtenir

l'identifiant du processus au sein duquel il s'exécute en utilisant l'appel système `getpid()` et l'identifiant de son processus parent avec l'appel système `getppid()`. Par exemple, le programme du Listing 3.1 affiche son identifiant de processus ainsi que celui de son parent.

Listing 3.1 – (*print-pid.c*) – **Afficher l'Identifiant de Processus**

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main ()
5  {
6    printf ("L'identifiant du processus est %d\n", (int) getpid ());
7    printf ("L'identifiant du processus parent est %d\n", (int) getppid ());
8    return 0;
9  }
```

Notez que si vous invoquez ce programme plusieurs fois, un identifiant de processus différent est indiqué à chaque fois car chaque invocation crée un nouveau processus. Cependant, si vous l'invoquez toujours depuis le même shell, l'identifiant du processus parent (c'est-à-dire l'identifiant de processus du shell) est le même.

3.1.2 Voir les processus actifs

La commande `ps` affiche les processus en cours d'exécution sur votre système. La version GNU/Linux de `ps` dispose d'un grand nombre d'options car elle tente d'être compatible avec les versions de `ps` de plusieurs autres variantes d'UNIX. Ces options contrôlent les processus listés et les informations les concernant qui sont affichées.

Par défaut, invoquer `ps` affiche les processus contrôlés par le terminal ou la fenêtre de terminal la commande est invoquée. Par exemple :

```
% ps
  PID TTY      TIME CMD
21693 pts/8 00:00:00 bash
21694 pts/8 00:00:00 ps
```

Cette invocation de `ps` montre deux processus. Le premier, `bash`, est le shell s'exécutant au sein du terminal. Le second est l'instance de `ps` en cours d'exécution. La première colonne, `PID`, indique l'identifiant de chacun des processus.

Pour un aperçu plus détaillé des programmes en cours d'exécution sur votre système GNU/-Linux, invoquez cette commande :

```
% ps -e -o pid,ppid,command
```

L'option `-e` demande à `ps` d'afficher tous processus en cours d'exécution sur le système. L'option `-o pid,ppid,command` indique à ps les informations à afficher sur chaque processus – dans ce cas, l'identifiant de processus, l'identifiant du processus parent et la commande correspondant au processus.

Voici les premières et dernières lignes de la sortie de cette commande sur mon système. Elles peuvent différer des vôtres, selon les programmes en cours d'exécution sur votre système :

Formats de Sortie ps

Avec l'option `-o` de la commande `ps`, vous indiquez les informations sur les processus que vous voulez voir s'afficher sous forme d'une liste de valeurs séparées par des virgules. Par exemple, `ps -o pid,user,start_time,command` affiche l'identifiant de processus, le nom de l'utilisateur propriétaire du processus, l'heure de démarrage du processus et la commande s'exécutant au sein du processus. Consultez la page de manuel de `ps` pour une liste complète des codes. Vous pouvez utiliser les options `-f` (*full listing*, listing complet), `-l` (long listing) ou `-j` (jobs listing) pour obtenir trois formats de listing prédéfinis.

```
% ps -e -o pid,ppid,command
  PID PPID COMMAND
    1     0 init [5]
    2     1 [kflushd]
    3     1 [kupdate]
...
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid,ppid,command
```

Notez que l'identifiant du processus parent de la commande `ps`, `21727`, est l'identifiant du processus de `bash`, le shell depuis lequel j'ai invoqué `ps`. L'identifiant du processus parent de `bash` est `21725`, l'identifiant du processus du programme `xterm` dans lequel le shell s'exécute.

3.1.3 Tuer un processus

Vous pouvez tuer un processus en cours d'exécution grâce à la commande `kill`. Spécifiez simplement sur la ligne de commande l'identifiant du processus à tuer.

La commande kill envoie un signal[1] `SIGTERM`, ou de terminaison au processus. Cela termine le processus, à moins que le programme en cours d'exécution ne gère ou ne masque le signal `SIGTERM`. Les signaux sont décrits dans la Section 3.3, « Signaux ».

3.2 Créer des processus

Deux techniques courantes sont utilisées pour créer de nouveaux processus. La première est relativement simple mais doit être utilisée avec modération car elle est peu performante et présente des risques de sécurité considérables. La seconde technique est plus complexe mais offre une flexibilité, une rapidité et une sécurité plus grandes.

3.2.1 Utiliser *system*

La fonction `system` de la bibliothèque standard propose une manière simple d'exécuter une commande depuis un programme, comme si la commande avait été tapée dans un shell. En fait,

[1] Vous pouvez également utiliser la commande `kill` pour envoyer d'autres signaux à un processus. Reportez-vous à la Section 3.4, « Fin de processus ».

`system` crée un sous-processus dans lequel s'exécute le shell Bourne standard (`/bin/sh`) et passe la commande à ce shell pour qu'il l'exécute. Par exemple, le programme du Listing 3.2 invoque la commande `ls` pour afficher le contenu du répertoire racine, comme si vous aviez saisi `ls -l /` dans un shell.

Listing 3.2 – (*system.c*) – **Utiliser la Fonction system**

```
1 int main ()
2 {
3   int return_value;
4   return_value = system ("ls -l /");
5   return return_value;
6 }
```

La fonction `system` renvoie le code de sortie de la commande shell. Si le shell lui-même ne peut pas être lancé, `system` renvoie `127` ; si une autre erreur survient, `system` renvoie `-1`.

Comme la fonction `system` utilise un shell pour invoquer votre commande elle est soumises aux fonctionnalités, limitations et failles de sécurité du shell système. Vous ne pouvez pas vous reposer sur la disponibilité d'une version spécifique du shell Bourne. Sur beaucoup de systèmes UNIX, `/bin/sh` est en fait un lien symbolique vers un autre shell. Par exemple, sur la plupart des systèmes GNU/Linux, `/bin/sh` pointe vers `bash` (le Bourne-Again SHell) et des distributions différentes de GNU/Linux utilisent différentes version de `bash`. Invoquer un programme avec les privilèges root via la fonction `system` peut donner des résultats différents selon le système GNU/Linux. Il est donc préférable d'utiliser la méthode de `fork` et `exec` pour créer des processus.

3.2.2 Utiliser *fork* et *exec*

Les API DOS et Windows proposent la famille de fonctions `spawn`. Ces fonctions prennent en argument le nom du programme à exécuter et créent une nouvelle instance de processus pour ce programme. Linux ne dispose pas de fonction effectuant tout cela en une seule fois. Au lieu de cela, Linux offre une fonction, `fork`, qui produit un processus fils qui est l'exacte copie de son processus parent. Linux fournit un autre jeu de fonctions, la famille `exec`, qui fait en sorte qu'un processus cesse d'être une instance d'un certain programme et devienne une instance d'un autre. Pour créer un nouveau processus, vous utilisez tout d'abord `fork` pour créer une copie du processus courant. Puis vous utilisez `exec` pour transformer un de ces processus en une instance du programme que vous voulez créer.

Appeler *fork* et *exec*

Lorsqu'un programme appelle `fork`, une copie du processus, appelée *processus fils*, est créée. Le processus parent continue d'exécuter le programme à partir de l'endroit où `fork` a été appelé. Le processus fils exécute lui aussi le même programme à partir du même endroit.

Qu'est-ce qui différencie les deux processus alors ? Tout d'abord, le processus fils est un nouveau processus et dispose donc d'un nouvel identifiant de processus, distinct de celui de l'identifiant de son processus parent. Un moyen pour un programme de savoir s'il fait partie du processus père ou du processus fils est d'appeler la méthode `getpid`. Cependant, la fonction `fork` renvoie des valeurs différentes aux processus parent et enfant – un processus « rentre » dans le

fork et deux en « ressortent » avec des valeurs de retour différentes. La valeur de retour dans le processus père est l'identifiant du processus fils. La valeur de retour dans le processus fils est zéro. Comme aucun processus n'a l'identifiant zéro, il est facile pour le programme de déterminer s'il s'exécute au sein du processus père ou du processus fils.

Le Listing 3.3 est un exemple d'utilisation de `fork` pour dupliquer un processus. Notez que le premier bloc de la structure `if` n'est exécuté que dans le processus parent alors que la clause `else` est exécutée dans le processus fils.

Listing 3.3 – (*fork.c*) – **Utiliser *fork* pour Dupliquer un Processus**

```
1  int main ()
2  {
3    pid_t child_pid;
4    printf ("ID de processus du programme principal : %d\n",(int)getpid ());
5    child_pid = fork ();
6    if (child_pid != 0) {
7      printf ("je suis le processus parent, ID : %d\n", (int) getpid ());
8      printf ("Identifiant du processus fils : %d\n", (int) child_pid);
9    }
10   else
11     printf ("je suis le processus fils, ID : %d\n", (int) getpid ());
12   return 0;
13 }
```

Utiliser la famille de exec

Les fonctions `exec` remplacent le programme en cours d'exécution dans un processus par un autre programme. Lorsqu'un programme appelle la fonction `exec`, le processus cesse immédiatement d'exécuter ce programme et commence l'exécution d'un autre depuis le début, en supposant que l'appel à `exec` se déroule correctement.

Au sein de la famille de `exec` existent plusieurs fonctions qui varient légèrement quant aux possibilités qu'elles proposent et à la façon de les appeler.

- Les fonctions qui contiennent la lettre `p` dans leur nom (`execvp` et `execlp`) reçoivent un nom de programme qu'elles recherchent dans le path courant ; il est nécessaire de passer le chemin d'accès complet du programme aux fonctions qui ne contiennent pas de `p`.
- Les fonctions contenant la lettre `v` dans leur nom (`execv`, `execvp` et `execve`) reçoivent une liste d'arguments à passer au nouveau programme sous forme d'un tableau de pointeurs vers des chaînes terminé par `NULL`. Les fonctions contenant la lettre `l` (`execl`, `execlp` et `execle`) reçoivent la liste d'arguments via le mécanisme du nombre d'arguments variables du langage C.
- Les fonctions qui contiennent la lettre `e` dans leur nom (`execve` et `execle`) prennent un argument supplémentaire, un tableau de variables d'environnement. L'argument doit être un tableau de pointeurs vers des chaînes terminé par `NULL`. Chaque chaîne doit être de la forme `"VARIABLE=valeur"`.

Dans la mesure où `exec` remplace le programme appelant par un autre, on n'en sort jamais à moins que quelque chose ne se déroule mal.

Les arguments passés à un programme sont analogues aux arguments de ligne de commande que vous transmettez à un programme lorsque vous le lancez depuis un shell. Ils sont accessibles

par le biais des paramètres `argc` et `argv` de `main`. Souvenez-vous que lorsqu'un programme est invoqué depuis le shell, celui-ci place dans le premier élément de la liste d'arguments (`argv[0]`) le nom du programme, dans le second (`argv[1]`) le premier paramètre en ligne de commande, *etc.* Lorsque vous utilisez une fonction `exec` dans votre programme, vous devriez, vous aussi, passer le nom du programme comme premier élément de la liste d'arguments.

Utiliser fork et exec

Un idiome courant pour l'exécution de sous-programme au sein d'un programme est d'effectuer un `fork` puis d'appeler `exec` pour le sous-programme. Cela permet au programme appelant de continuer à s'exécuter au sein du processus parent alors qu'il est remplacé par le sous-programme dans le processus fils.

Le programme du Listing 3.4, comme celui du Listing 3.2, liste le contenu du répertoire racine en utilisant la commande `ls`. Contrairement à l'exemple précédent, cependant, il utilise la commande `ls` directement, en lui passant les arguments `-l` et `/` plutôt que de l'invoquer depuis un shell.

Listing 3.4 – (*fork-exec.c*) – **Utiliser *fork* et *exec***

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <unistd.h>
/* Crée un processus fils exécutant un nouveau programme. PROGRAM est le nom
   du programme à exécuter ; le programme est recherché dans le path.
   ARG_LIST est une liste terminée par NULL de chaînes de caractères à passer
   au programme comme liste d'arguments. Renvoie l'identifiant du processus
   nouvellement créé. */
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  /* Duplique ce processus. */
  child_pid = fork ();
  if (child_pid != 0)
    /* Nous sommes dans le processus parent. */
    return child_pid;
  else {
    /* Exécute PROGRAM en le recherchant dans le path. */
    execvp (program, arg_list);
    /* On ne sort de la fonction execvp uniquement si une erreur survient. */
    fprintf (stderr, "une erreur est survenue au sein de execvp\n");
    abort ();
  }
}
int main ()
{
  /* Liste d'arguments à passer à la commande "ls". */
  char* arg_list[] = {
    "ls",     /* argv[0], le nom du programme. */
    "-l",
    "/",
    NULL      /* La liste d'arguments doit se terminer par NULL.  */
  };
  /* Crée un nouveau processus fils exécutant la commande "ls". Ignore
      l'identifiant du processus fils renvoyé. */
  spawn ("ls", arg_list);
```

```
38    printf ("Fin du programme principal\n");
39    return 0;
40  }
```

3.2.3 Ordonnancement de processus

Linux ordonnance le processus père indépendamment du processus fils ; il n'y a aucune garantie sur celui qui sera exécuté en premier ou sur la durée pendant laquelle le premier s'exécutera avant que Linux ne l'interrompe et ne passe la main à l'autre processus (ou à unprocessus quelconque s'exécutant sur le système). Dans notre cas, lorsque le processus parent se termine, la commande `ls` peut s'être exécutée entièrement, partiellement ou pas du tout[2]. Linux garantit que chaque processus finira par s'exécuter – aucun processus ne sera à cours de ressources.

Vous pouvez indiquer qu'un processus est moins important – et devrait avoir une priorité inférieure – en lui assignant une valeur de *priorité d'ordonnancement*[3] plus grande. Par défaut, chaque processus a une priorité d'ordonnancement de zéro. Une valeur plus élevée signifie que le processus a une priorité d'exécution plus faible ; inversement un processus avec une priorité d'ordonnancement plus faible (c'est-à-dire, négative) obtient plus de temps d'exécution.

Pour lancer un programme avec une priorité d'ordonnancement différente de zéro, utilisez la commande `nice`, en spécifiant la valeur de la priorité d'ordonnancement avec l'option `-n`. Par exemple, voici comment vous pourriez invoquer la commande `sort input.txt > output.txt`, une opération de tri longue, avec une priorité réduite afin qu'elle ne ralentisse pas trop le système :

```
% nice -n 10 sort input.txt > output.txt
```

Vous pouvez utiliser la commande `renice` pour changer la priorité d'ordonnancement d'un processus en cours d'exécution depuis la ligne de commande.

Pour changer la priorité d'ordonnancement d'un processus en cours d'exécution par programmation, utilisez la fonction `nice`. Son argument est une valeur d'ajustement qui est ajoutée à la valeur de la priorité d'ordonnancement du processus qui l'appelle. Souvenez-vous qu'une valeur positive augmente la priorité d'ordonnancement du processus et donc réduit la priorité d'exécution du processus.

Notez que seuls les processus disposant des privilèges root peuvent lancer un processus avec une priorité d'ordonnancement négative ou réduire la valeur de la priorité d'ordonnancement d'un processus en cours d'exécution. Cela signifie que vous ne pouvez passer des valeurs négatives aux commandes `nice` et `renice` que lorsque vous êtes connecté en tant que root et seul un processus s'exécutant avec les privilèges root peut passer une valeur négative à la fonction `nice`. Cela évite que des utilisateurs ordinaires puissent s'approprier tout le temps d'exécution.

[2] Une méthode visant à exécuter les deux processus de manière séquentielle est présentée dans la Section 3.4.1, « Attendre la fin d'un processus ».

[3] NdT. le terme original est *niceness* (gentillesse), ce qui explique que plus la valeur est grande, plus le processus est « gentil » et donc moins il est prioritaire.

3.3 Signaux

Les *signaux* sont des mécanismes permettant de manipuler et de communiquer avec des processus sous Linux. Le sujet des signaux est vaste ; nous traiterons ici quelques uns des signaux et techniques utilisés pour contrôler les processus.

Un signal est un message spécial envoyé à un processus. Les signaux sont asynchrones ; lorsqu'un processus reçoit un signal, il le traite immédiatement, sans même terminer la fonction ou la ligne de code en cours. Il y a plusieurs douzaines de signaux différents, chacun ayant une signification différente. Chaque type de signal est caractérisé par son numéro de signal, mais au sein des programmes, on y fait souvent référence par un nom. Sous Linux, ils sont définis dans `/usr/include/bits/signum.h` (vous ne devriez pas inclure ce fichier directement dans vos programmes, utilisez plutôt `<signal.h>`).

Lorsqu'un processus reçoit un signal, il peut agir de différentes façons, selon l'action enregistrée pour le signal. Pour chaque signal, il existe une action par défaut, qui détermine ce qui arrive au processus si le programme ne spécifie pas d'autre comportement. Pour la plupart des signaux, le programme peut indiquer un autre comportement – soit ignorer le signal, soit appeler un gestionnaire de signal, fonction chargée de traiter le signal. Si un gestionnaire de signal est utilisé, le programme en cours d'exécution est suspendu, le gestionnaire est exécuté, puis, une fois celui-ci terminé, le programme reprend.

Le système Linux envoie des signaux aux processus en réponse à des conditions spécifiques. Par exemple, `SIGBUS` (erreur de bus), `SIGSEGV` (erreur de segmentation) et `SIGFPE` (exception de virgule flottante) peuvent être envoyés à un programme essayant d'effectuer une action non autorisée. L'action par défaut pour ces trois signaux est de terminer le processus et de produire un ficher core.

Un processus peut également envoyer des signaux à un autre processus. Une utilisation courante de ce mécanisme est de terminer un autre processus en lui envoyant un signal `SIGTERM` ou `SIGKILL`[4]. Une autre utilisation courante est d'envoyer une commande à un programme en cours d'exécution. Deux signaux "définis par l'utilisateur" sont réservés à cet effet : `SIGUSR1` et `SIGUSR2`. Le signal `SIGHUP` est également parfois utilisé dans ce but, habituellement pour réveiller un programme inactif ou provoquer une relecture du fichier de configuration.

La fonction `sigaction` peut être utilisée pour paramétrer l'action à effectuer en réponse à un signal. Le premier paramètre est le numéro du signal. Les deux suivants sont des pointeurs vers des structures `sigaction` ; la première contenant l'action à effectuer pour ce numéro de signal, alors que la seconde est renseignée avec l'action précédente. Le champ le plus important, que ce soit dans la première ou la seconde structure sigaction est `sa_handler`. Il peut prendre une des trois valeurs suivantes :

- `SIG_DFL`, qui correspond à l'action par défaut pour le signal ;
- SIG_IGN , qui indique que le signal doit être ignoré ;
- Un pointeur vers une fonction de gestion de signal. La fonction doit prendre un paramètre, le numéro du signal et être de type `void`.

[4]Quelle est la différence ? Le signal `SIGTERM` demande au processus de se terminer ; le processus peut ignorer la requête en masquant ou ignorant le signal. Le signal `SIGKILL` tue toujours le processus immédiatement car il est impossible de masquer ou ignorer `SIGKILL`.

Comme les signaux sont asynchrones, le programme principal peut être dans un état très fragile lorsque le signal est traité et donc pendant l'exécution du gestionnaire de signal. C'est pourquoi vous devriez éviter d'effectuer des opérations d'entrées/sorties ou d'appeler la plupart des fonctions système ou de la bibliothèque C depuis un gestionnaire de signal.

Un gestionnaire de signal doit effectuer le minimum nécessaire au traitement du signal, puis repasser le contrôle au programme principal (ou terminer le programme). Dans la plupart des cas, cela consiste simplement à enregistrer que le signal est survenu. Le programme principal vérifie alors périodiquement si un signal a été reçu et réagit en conséquence.

Il est possible qu'un gestionnaire de signal soit interrompu par l'arrivée d'un autre signal. Bien que cela puisse sembler être un cas rare, si cela arrive, il peut être très difficile de diagnostiquer et résoudre le problème (c'est un exemple de conditions de concurrence critique, traité dans le Chapitre 4, « Threads », Section 4.4, « Synchronisation et sections critiques »). C'est pourquoi vous devez être très attentif à ce que votre programme fait dans un gestionnaire de signal.

Même l'affectation d'une valeur à une variable globale peut être dangereuse car elle peut en fait être effectuée en deux instructions machine ou plus, et un second signal peut survenir, laissant la variable dans un état corrompu. Si vous utilisez une variable globale pour indiquer la réception d'un signal depuis un gestionnaire de signal, elle doit être du type spécial `sig_atomic_t`. Linux garantit que les affectations de valeur à des variables de ce type sont effectuées en une seule instruction et ne peuvent donc pas être interrompues. Sous Linux, `sig_atomic_t` est un `int` ordinaire ; en fait, les affectations de valeur à des types de la taille d'un `int` ou plus petits ou à des pointeurs, sont atomiques. Néanmoins, si vous voulez écrire un programme portable vers n'importe quel système UNIX standard, utilisez le type `sig_atomic_t`.

Le squelette de programme du Listing 3.5, par exemple, utilise une fonction de gestion de signal pour compter le nombre de fois où le programme reçoit le signal `SIGUSR1`, un des signaux utilisables par les applications.

Listing 3.5 – (*sigusr1.c*) – **Utiliser un Gestionnaire de Signal**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
sig_atomic_t sigusr1_count = 0;
void handler (int signal_number)
{
  ++sigusr1_count;
}
int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);
  /* Faire quelque chose de long ici.  */
  /* ... */
  printf ("SIGUSR1 a été reçu %d fois\n", sigusr1_count);
  return 0;
}
```

3.4 Fin de processus

Dans des conditions normales, un processus peut se terminer de deux façons : soit le programme appelle la fonction `exit`, soit la fonction `main` du programme se termine. Chaque processus a un code de sortie : un nombre que le processus renvoie à son Le code de sortie est l'argument passé à la fonction `exit` ou la valeur retournée depuis `main`.

Un processus peut également se terminer de façon anormale, en réponse à un signal. Par exemple, les signaux `SIGBUS`, `SIGSEGV` et `SIGFPE` évoqués précédemment provoquent la fin du processus. D'autres signaux sont utilisés pour terminer un processus explicitement. Le signal `SIGINT` est envoyé à un processus lorsque l'utilisateur tente d'y mettre fin en saisissant `Ctrl+C` dans son terminal. Le signal `SIGTERM` est envoyé par la commande `kill`. L'action par défaut pour ces deux signaux est de mettre fin au processus. En appelant la fonction `abort`, un processus s'envoie à lui-même le signal `SIGABRT` ce qui termine le processus et produit un fichier core. Le signal de terminaison le plus puissant est `SIGKILL` qui met fin à un processus immédiatement et ne peut pas être bloqué ou géré par un programme.

Chacun de ces signaux peut être envoyé en utilisant la commande `kill` en passant une option supplémentaire sur la ligne de commande ; par exemple, pour terminer un programme fonctionnant mal en lui envoyant un `SIGKILL`, invoquez la commande suivante, où `pid` est un identifiant de processus :

```
% kill -KILL pid}
```

Pour envoyer un signal depuis un programme, utilisez la fonction `kill`. Le premier paramètre est l'identifiant du processus cible. Le second est le numéro du signal ; utilisez `SIGTERM` pour simuler le comportement par défaut de la commande `kill`. Par exemple, si `child_pid` contient l'identifiant du processus fils, vous pouvez utiliser la fonction `kill` pour terminer un processus fils depuis le père en l'appelant comme ceci :

```
kill (child_pid, SIGTERM);
```

Incluez les entêtes `<sys/types.h>` et `<signal.h>` si vous utilisez la fonction `kill`.

Par convention, le code de sortie est utilisé pour indiquer si le programme s'est exécuté correctement. Un code de sortie à zéro indique une exécution correcte, alors qu'un code différent de zéro indique qu'une erreur est survenue. Dans ce dernier cas, la valeur renvoyée peut donner une indication sur la nature de l'erreur. C'est une bonne idée d'adopter cette convention dans vos programmes car certains composants du système GNU/Linux sont basés dessus. Par exemple, les shells supposent l'utilisation de cette convention lorsque vous connectez plusieurs programmes avec les opérateurs `&&` (et logique) et `||` (ou logique). C'est pour cela que vous devriez renvoyer zéro explicitement depuis votre fonction `main`, à moins qu'une erreur ne survienne.

Avec la plupart des shells, il est possible d'obtenir le code de sortie du dernier programme exécuté en utilisant la variable spéciale `$?` Voici un exemple dans lequel la commande ls est invoquée deux fois et son code de sortie est affiché après chaque invocation. Dans le premier cas, `ls` se termine correctement et renvoie le code de sortie `0`. Dans le second cas, ls rencontre une erreur (car le fichier spécifié sur la ligne de commande n'existe pas) et renvoie donc un code de sortie différent de `0`.

```
% ls /
bin   coda etc    lib         misc nfs proc
```

```
boot dev    home lost+found mnt     opt root
% echo $?
0
% ls fichierinexistant
ls: fichierinexistant: Aucun fichier ou répertoire de ce type
% echo $?
1
```

Notez que même si le paramètre de la fonction `exit` est de type `int` et que la fonction `main` renvoie un `int`, Linux ne conserve pas les 32 bits du code de sortie. En fait, vous ne devriez utiliser que des codes de sortie entre 0 et 127. Les codes de sortie au dessus de 128 ont une signification spéciale – lorsqu'un processus se termine à cause d'un signal, son code de sortie est zéro plus le numéro du signal.

3.4.1 Attendre la fin d'un processus

Si vous avez saisi et exécuté l'exemple de `fork` et `exec` du Listing 3.4, vous pouvez avoir remarqué que la sortie du programme `ls` apparaît souvent après la fin du programme principal. Cela est dû au fait que le processus fils, au sein duquel s'exécute `ls`, est ordonnancé indépendamment du processus père. Comme Linux est un système d'exploitation multitâche, les deux processus semblent s'exécuter simultanément et vous ne pas prédire si le programme `ls` va s'exécuter avant ou après le processus père.

Dans certaines situations, cependant, il est souhaitable que le processus père attende la fin d'un ou plusieurs processus fils. Pour cela, il est possible d'utiliser les appels systèmes de la famille de `wait`. Ces fonctions vous permettent d'attendre la fin d'un processus et permettent au processus parent d'obtenir des informations sur la façon dont s'est terminé son fils. Il y a quatre appels système différents dans la famille de `wait` ; vous pouvez choisir de récupérer peu ou beaucoup d'informations sur le processus qui s'est terminé et vous pouvez indiquer si vous voulez savoir quel processus fils s'est terminé.

3.4.2 Les appels système *wait*

La fonction la plus simple s'appelle simplement `wait`. Elle bloque le processus appelant jusqu'à ce qu'un de ses processus fils se termine (ou qu'une erreur survienne). Elle retourne un code de statut via un pointeur sur un entier, à partir duquel vous pouvez extraire des informations sur la façon dont s'est terminé le processus fils. Par exemple, la macro `WEXITSTATUS` extrait le code de sortie du processus fils.

Vous pouvez utiliser la macro `WIFEXITED` pour déterminer si un processus s'est terminé correctement à partir de son code de statut (via la fonction `exit` ou la sortie de `main`) ou est mort à cause d'un signal non intercepté. Dans ce dernier cas, utilisez la macro `WTERMSIG` pour extraire le numéro du signal ayant causé la mort du processus à partir du code de statut.

Voici une autre version de la fonction `main` de l'exemple de `fork` et `exec`. Cette fois, le processus parent appelle `wait` pour attendre que le processus fils, dans lequel s'exécute la commande `ls`, se termine.

```
int main ()
{
  int child_status;
```

```
  /* Liste d'arguments à passer à la commande "ls". */
  char* arg_list[] = {
     "ls",     /* argv[0], le nom du programme. */
     "-l",
     "/",
     NULL      /* La liste d'arguments doit se terminer par NULL.  */
  };
  /* Crée un nouveau processus fils exécutant la commande "ls". Ignore
      l'identifiant du processus fils renvoyé. */
  spawn ("ls", arg_list);
  /* Attend la fin du processus fils. */
  wait (&child_status);
  if (WIFEXITED (child_status))
    printf ("processus fils terminé normalement, le code de sortie %d\n",
            WEXITSTATUS (child_status));
  else
    printf ("processus fils terminé anormalement\n");
  return 0;
}
```

Plusieurs appels système similaires sont disponibles sous Linux, ils sont plus flexibles ou apportent plus d'informations sur le processus fils se terminant. La fonction `waitpid` peut être utilisée pour attendre la fin d'un processus fils spécifique au lieu d'attendre n'importe quel processus fils. La fonction `wait3` renvoie des statistiques sur l'utilisation du processeur par le processus fils se terminant et la fonction `wait4` vous permet de spécifier des options supplémentaires quant au processus dont on attend la fin.

3.4.3 Processus zombies

Si un processus fils se termine alors que son père appelle la fonction `wait`, le processus fils disparaît et son statut de terminaison est transmis à son père via l'appel à `wait`. Mais que se passe-t-il lorsqu'un processus se termine et que son père n'appelle pas `wait` ? Disparaît-il tout simplement ? Non, car dans ce cas, les informations concernant la façon dont il s'est terminé – comme le fait qu'il se soit terminé normalement ou son code de sortie – seraient perdues. Au lieu de cela, lorsqu'un processus fils se termine, il devient un processus zombie.

Un processus zombie est un processus qui s'est terminé mais dont les ressources n'ont pas encore été libérées. Il est de la responsabilité du processus père de libérer les ressources occupées par ses fils zombies. La fonction `wait` le fait, il n'est donc pas nécessaire de savoir si votre processus fils est toujours en cours d'exécution avant de l'attendre. Supposons, par exemple, qu'un programme crée un processus fils, fasse un certain nombre d'autres opérations puis appelle `wait`. Si le processus fils n'est pas encore terminé à ce moment, le processus parent sera bloqué dans l'appel de `wait` jusqu'à ce que le processus fils se termine. Si le processus fils se termine avant que le père n'appelle `wait`, il devient un processus zombie. Lorsque le processus parent appelle `wait`, le statut de sortie du processus fils est extrait, le processus fils est supprimé et l'appel de `wait` se termine immédiatement.

Que se passe-t-il si le père ne libère pas les ressources de ses fils ? Ils restent dans le système, sous la forme de processus zombies. Le programme du Listing 3.6 crée un processus fils qui se termine immédiatement, puis s'interrompt pendant une minute, sans jamais libérer les ressources du processus fils.

Listing 3.6 – (*zombie.c*) – **Créer un Processus Zombie**

```
1  #include <stdlib.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  int main ()
5  {
6    pid_t child_pid;
7    /* Crée un processus fils. */
8    child_pid = fork ();
9    if (child_pid > 0) {
10     /* Nous sommes dans le processus parent. Attente d'une minute. */
11     sleep (60);
12   }
13   else {
14     /* Nous sommes dans le processus fils. Sortie immédiate. */
15     exit (0);
16   }
17   return 0;
18 }
```

Compilez ce fichier en un exécutable appelé `make-zombie`. Exécutez-le, et pendant ce temps, listez les processus en cours d'exécution sur le système par le biais de la commande suivante dans une autre fenêtre :

```
% ps -e -o pid,ppid,stat,cmd
```

Elle dresse la liste des identifiants de processus, de processus pères, de leur statut et de la ligne de commande du processus. Observez qu'en plus du processus père make-zombie, un autre processus `make-zombie` est affiché. Il s'agit du processus fils ; notez que l'identifiant de son père est celui du processus `make-zombie` principal. Le processus fils est marqué comme `<defunct>` et son code de statut est `Z` pour zombie.

Que se passe-t-il lorsque le programme principal de `make-zombie` se termine sans appeler `wait` ? Le processus zombie est-il toujours présent ? Non – essayez de relancer `ps` et notez que les deux processus `make-zombie` ont disparu. Lorsqu'un programme se termine, un processus spécial hérite de ses fils, le programme `init`, qui s'exécute toujours avec un identifiant de processus valant `1` (il s'agit du premier processus lancé lorsque Linux démarre). Le processus `init` libère automatiquement les ressources de tout processus zombie dont il hérite.

3.4.4 Libérer les ressources des fils de façon asynchrone

Si vous utilisez un processus fils uniquement pour appeler `exec`, il est suffisant d'appeler `wait` immédiatement dans le processus parent, ce qui le bloquera jusqu'à ce que le processus fils se termine. Mais souvent, vous voudrez que le processus père continue de s'exécuter alors qu'un ou plusieurs fils s'exécutent en parallèle. Comment être sûr de libérer les ressources occupées par tous les processus fils qui se sont terminés afin de ne pas laisser de processus zombie dans la nature, ce qui consomme des ressources ?

Une approche envisageable serait que le processus père appelle périodiquement `wait3` et `wait4` pour libérer les ressources des fils zombies. Appeler `wait` de cette façon ne fonctionne pas de manière optimale car si aucun fils ne s'est terminé, l'appelant sera bloqué jusqu'à ce que ce soit le cas. Cependant, `wait3` et `wait4` prennent un paramètre d'option supplémentaire auquel vous

pouvez passer le drapeau `WNOHANG`. Avec ce drapeau, la fonction s'exécute en mode non bloquant – elle libérera les ressources d'un processus fils terminé s'il y en a un ou se terminera s'il n'y en a pas. La valeur de retour de l'appel est l'identifiant du processus fils s'étant terminé dans le premier cas, zéro dans le second.

Une solution plus élégante est d'indiquer au processus père quand un processus fils se termine. Il y a plusieurs façons de le faire en utilisant les méthodes présentées dans le Chapitre 5, « Communication interprocessus », mais heureusement, Linux le fait pour vous, en utilisant les signaux. Lorsqu'un processus fils se termine, Linux envoie au processus père le signal `SIGCHLD`. L'action par défaut pour ce signal est de ne rien faire, c'est la raison pour laquelle vous pouvez ne jamais l'avoir remarqué auparavant.

Donc, une façon élégante de libérer les ressources des processus fils est de gérer `SIGCHLD`. Bien sûr, lorsque vous libérez les ressources du processus fils, il est important de stocker son statut de sortie si cette information est nécessaire, car une fois que les ressources du processus fils ont été libérées par `wait`, cette information n'est plus disponible. Le Listing 3.7 montre à quoi ressemble un programme utilisant un gestionnaire pour `SIGCHLD` afin de libérer les ressources de ses processus fils.

Listing 3.7 – (*sigchld.c*) – **Libérer les Ressources des Fils via SIGCHLD**

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
void clean_up_child_process (int signal_number)
{
  /* Nettoie le ou les processus fils. */
  int status;
  while(waitpid (-1, &status, WNOHANG));
  /* Stocke la statut de sortie du dernier dans une variable globale.  */
  child_exit_status = status;
}
int main ()
{
  /* Gère SIGCHLD en appelent clean_up_child_process. */
  struct sigaction sigchld_action;
  memset (&sigchld_action, 0, sizeof (sigchld_action));
  sigchld_action.sa_handler = &clean_up_child_process;
  sigaction (SIGCHLD, &sigchld_action, NULL);
  /* Faire diverses choses, entre autres créer un processus fils.  */
  /* ... */
  return 0;
}
```

Notez comment le gestionnaire de signal stocke le statut de sortie du processus fils dans une variable globale, à laquelle le programme principal peut accéder. Comme la variable reçoit une valeur dans un gestionnaire de signal, elle est de type `sig_atomic_t`.

Chapitre 4

Threads

Les threads[1], comme les processus,sont un mécanisme permettant à un programme de faire plus d'une chose à la fois. Comme les processus, les threads semblent s'exécuter en parallèle ; le noyau Linux les ordonnance de façon asynchrone, interrompant chaque thread de temps en temps pour donner aux autres une chance de s'exécuter.

Conceptuellement, un thread existe au sein d'un processus. Les threads sont une unité d'exécution plus fine que les processus. Lorsque vous invoquez un programme, Linux crée un nouveau processus et, dans ce processus, crée un thread, qui exécute le processus de façon séquentielle. Ce thread peut en créer d'autres ; tous ces threads exécutent alors le même programme au sein du même processus, mais chaque thread peut exécuter une partie différente du programme à un instant donné.

Nous avons vu comment un programme peut créer un processus fils. Celui-ci exécute immédiatement le programme de son père, la mémoire virtuelle, les descripteurs de fichiers, *etc.* de son père étant copiés. Le processus fils peut modifier sa mémoire, fermer les descripteurs de fichiers sans que cela affecte son père, et *vice versa*. Lorsqu'un programme crée un nouveau thread, par contre, rien n'est copié. Le thread créateur et le thread créé partagent tous deux le même espace mémoire, les mêmes descripteurs de fichiers et autres ressources. Si un thread modifie la valeur d'une variable, par exemple, l'autre thread verra la valeur modifiée. De même, si un thread ferme un descripteur de fichier, les autres threads ne peuvent plus lire ou écrire dans ce fichier. Comme un processus et tous ses threads ne peuvent exécuter qu'un seul programme à la fois, si un thread au sein d'un processus appelle une des fonctions `exec`, tous les autres threads se terminent (le nouveau programme peut, bien sûr, créer de nouveaux threads).

GNU/Linux implémente l'API de threading standard POSIX (appelée aussi *pthreads*). Toutes les fonctions et types de données relatifs aux threads sont déclarés dans le fichier d'entête `< pthread.h>`. Les fonctions de `pthread` ne font pas partie de la bibliothèque standard du C. Elles se trouvent dans `libpthread`, vous devez donc ajouter `-lpthread` sur la ligne de commande lors de l'édition de liens de votre programme.

[1] NdT. Appelés aussi « processus légers ».

4.1 Création de threads

Chaque thread d'un processus est caractérisé par un *identifiant de thread*. Lorsque vous manipulez les identifiants de threads dans des programmes C ou C++, veillez à utiliser le type `pthread_t`.

Lors de sa création, chaque thread exécute une *fonction de thread*. Il s'agit d'une fonction ordinaire contenant le code que doit exécuter le thread. Lorsque la fonction se termine, le thread se termine également. Sous GNU/Linux, les fonctions de thread ne prennent qu'un seul paramètre de type `void*` et ont un type de retour `void*`. Ce paramètre est l'*argument de thread* : GNU/Linux passe sa valeur au thread sans y toucher. Votre programme peut utiliser ce paramètre pour passer des données à un nouveau thread. De même, il peut utiliser la valeur de retour pour faire en sorte que le thread renvoie des données à son créateur lorsqu'il se termine.

La fonction `pthread_create` crée un nouveau thread. Voici les paramètres dont elle a besoin :

1. Un pointeur vers une variable `pthread_t`, dans laquelle l'identifiant du nouveau thread sera stocké ;
2. Un pointeur vers un objet d'attribut de thread. Cet objet contrôle les détails de l'interaction du thread avec le reste du programme. Si vous passez `NULL` comme argument de thread, le thread est créé avec les attributs par défaut. Ceux-ci sont traités dans la Section 4.1.5, « Attributs de thread » ;
3. Un pointeur vers la fonction de thread. Il s'agit d'un pointeur de fonction ordinaire de type : `void* (*)(void*)` ;
4. item Une valeur d'argument de thread de type `void*`. Quoi que vous passiez, l'argument est simplement transmis à la fonction de thread lorsque celui-ci commence à s'exécuter.

Un appel à `pthread_create` se termine immédiatement et le thread original continue à exécuter l'instruction suivant l'appel. Pendant ce temps, le nouveau thread débute l'exécution de la fonction de thread. Linux ordonnance les deux threads de manière asynchrone et votre programme ne doit pas faire d'hypothèse sur l'ordre d'exécution relatif des instructions dans les deux threads.

Le programme du Listing 4.1 crée un thread qui affiche `x` de façon continue sur la sortie des erreurs. Après l'appel de `pthread_create`, le thread principal affiche des `o` indéfiniment sur la sortie des erreurs.

Listing 4.1 – (*thread-create.c*) – **Créer un thread**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  /* Affiche des x sur stderr.  Paramètre inutilisé. Ne finit jamais. */
4  void* print_xs (void* unused)
5  {
6    while (1)
7      fputc ('x', stderr);
8    return NULL;
9  }
10 /* Le programme principal.  */
11 int main ()
12 {
13   pthread_t thread_id;
14   /* Crée un nouveau thread. Le nouveau thread exécutera la fonction
15      print_xs. */
```

```
16    pthread_create (&thread_id, NULL, &print_xs, NULL);
17    /* Affiche des o en continue sur stderr. */
18    while (1)
19      fputc ('o', stderr);
20    return 0;
21  }
```

Compilez ce programme en utilisant la commande suivante :

```
% cc -o thread-create thread-create.c -lpthread
```

Essayez de le lancer pour voir ce qui se passe. Notez que le motif formé par les `x` et les `o` est imprévisible car Linux passe la main alternativement aux deux threads.

Dans des circonstances normales, un thread peut se terminer de deux façons. La première, illustrée précédemment, est de terminer la fonction de thread. La valeur de retour de la fonction de thread est considérée comme la valeur de retour du thread. Un thread peut également se terminer explicitement en appelant `pthread_exit`. Cette fonction peut être appelée depuis la fonction de thread ou depuis une autre fonction appelée directement ou indirectement par la fonction de thread. L'argument de `pthread_exit` est la valeur de retour du thread.

4.1.1 Transmettre des données à un thread

L'argument de thread est une méthode pratique pour passer des données à un thread. Comme son type est `void*`, cependant, vous ne pouvez pas passer beaucoup de données directement en l'utilisant. Au lieu de cela, utilisez l'argument de thread pour passer un pointeur vers une structure ou un tableau de données. Une technique couramment utilisée est de définir une structure de données pour chaque argument de thread, qui contient les paramètres attendus par la fonction de thread.

En utilisant l'argument de thread, il est facile de réutiliser la même fonction pour différents threads. Ils exécutent alors tous les mêmes traitements mais sur des données différentes.

Le programme du Listing 4.2 est similaire à l'exemple précédent. Celui-ci crée deux nouveaux threads, l'un affiche des `x` et l'autre des `o`. Au lieu de les afficher indéfiniment, cependant, chaque thread affiche un nombre prédéterminé de caractères puis se termine en sortant de la fonction de thread. La même fonction de thread, `char_print`, est utilisée par les deux threads mais chacun est configuré différemment en utilisant une `struct char_print_parms`.

Listing 4.2 – (*thread-create2.c*) – **Créer Deux Threads**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  /* Paramètres de la fonction print.   */
4  struct char_print_parms
5  {
6     /* Caractère à afficher. */
7     char character;
8     /* Nombre de fois où il doit être affiché. */
9     int count;
10 };
11 /* Affiche un certain nombre de caractères sur stderr, selon le contenu de
12      PARAMETERS, qui est un pointeur vers une struct char_print_parms. */
13 void* char_print (void* parameters)
14 {
```

```
15      /* Effectue un transtypage du pointeur void vers le bon type. */
16      struct char_print_parms* p = (struct char_print_parms*) parameters;
17      int i;
18      for (i = 0; i < p->count; ++i)
19        fputc (p->character, stderr);
20      return NULL;
21    }
22    /* Programme principal.   */
23    int main ()
24    {
25      pthread_t thread1_id;
26      pthread_t thread2_id;
27      struct char_print_parms thread1_args;
28      struct char_print_parms thread2_args;
29      /* Crée un nouveau thread affichant 30 000 x. */
30      thread1_args.character = 'x';
31      thread1_args.count = 30000;
32      pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
33      /* Crée un nouveau thread affichant 20 000 'o'. */
34      thread2_args.character = ?o?;
35      thread2_args.count = 20000;
36      pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
37      return 0;
38    }
```

Mais attendez ! Le programme du Listing 4.2 est sérieusement bogué. Le thread principal (qui exécute la fonction `main`) crée les structures passées en paramètre aux threads (`thread1_args` et `thread2_args`) comme des variables locales puis transmet des pointeurs sur ces structures aux threads qu'il crée. Qu'est-ce qui empêche Linux d'ordonnancer les trois threads de façon à ce que `main` termine son exécution avant que l'un des deux autres threads s'exécutent ? Rien ! Mais si cela se produit, la mémoire contenant les structures de paramètres des threads sera libérée alors que les deux autres threads tentent d'y accéder.

4.1.2 Synchroniser des threads

Une solution possible est de forcer `main` à attendre la fin des deux autres threads. Ce dont nous avons besoin est une fonction similaire à `wait` qui attende la fin d'un thread au lieu de celle d'un processus. Cette fonction est `pthread_join`, qui prend deux arguments : l'identifiant du thread à attendre et un pointeur vers une variable `void*` qui recevra la valeur de retour du thread s'étant terminé. Si la valeur de retour du thread ne vous est pas utile, passez `NULL` comme second argument.

Le Listing 4.3 montre la fonction `main` sans le bogue du Listing 4.2. Dans cette version, `main` ne se termine pas jusqu'à ce que les deux threads affichant des `o` et des `x` se soient eux-même terminés, afin qu'ils n'utilisent plus les structures contenant les arguments.

Listing 4.3 – (*thread-create2a.c*) – **Fonction *main* de *thread-create2.c* corrigée**

```
1   int main ()
2   {
3     pthread_t thread1_id;
4     pthread_t thread2_id;
5     struct char_print_parms thread1_args;
6     struct char_print_parms thread2_args;
7     /* Crée un nouveau thread affichant 30 000 x. */
```

```
8     thread1_args.character = ?x?;
9     thread1_args.count = 30000;
10    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
11    /* Crée un nouveau thread affichant 20 000 o. */
12    thread2_args.character = ?o?;
13    thread2_args.count = 20000;
14    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
15    /* S'assure que le premier thread est terminé. */
16    pthread_join (thread1_id, NULL);
17    /* S'assure que le second thread est terminé. */
18    pthread_join (thread2_id, NULL);
19    /* Nous pouvons maintenant quitter en toute sécurité.  */
20    return 0;
21  }
```

Morale de l'histoire : assurez vous que toute donnée que vous passez à un thread par référence n'est pas libérée, *même par un thread différent*, à moins que vous ne soyez sûr que le thread a fini de l'utiliser. Cela s'applique aux variables locales, qui sont libérées lorsqu'elles sont hors de portée, ainsi qu'aux variables allouées dans le tas, que vous libérez en appelant `free` (ou en utilisant `delete` en C++).

4.1.3 Valeurs de retour des threads

Si le second argument que vous passez à `pthread_join` n'est pas `NULL`, la valeur de retour du thread sera stockée à l'emplacement pointé par cet argument. La valeur de retour du thread, comme l'argument de thread, est de type `void*`. Si vous voulez renvoyer un simple `int` ou un autre petit nombre, vous pouvez le faire facilement est convertissant la valeur en `void*` puis en le reconvertissant vers le type adéquat après l'appel de `pthread_join`[2].

Le programme du Listing 4.4 calcule le n$^{\text{ième}}$ nombre premier dans un thread distinct. Ce thread renvoie le numéro du nombre premier demandé *via* sa valeur de retour de thread. Le thread principal, pendant ce temps, est libre d'exécuter d'autres traitements. Notez que l'algorithme de divisions successives utilisé dans `compute_prime` est quelque peu inefficace ; consultez un livre sur les algorithmes numériques si vous avez besoin de calculer beaucoup de nombres premiers dans vos programmes.

Listing 4.4 – (*primes.c*) – **Calcule des Nombres Premiers dans un Thread**

```
1   #include <pthread.h>
2   #include <stdio.h>
3   /* Calcules des nombres premiers successifs (très inefficace). Renvoie
4      le Nième nombre premier où N est la valeur pointée par *ARG. */
5   void* compute_prime (void* arg)
6   {
7     int candidate = 2;
8     int n = *((int*) arg);
9     while (1) {
10      int factor;
11      int is_prime = 1;
12      /* Teste si le nombre est premier par divisions successives. */
13      for (factor = 2; factor < candidate; ++factor)
14        if (candidate % factor == 0) {
```

[2]Notez que cette façon de faire n'est pas portable et qu'il est de votre responsabilité de vous assurer que la valeur peut être convertie en toute sécurité vers et depuis `void*` sans perte de bit.

```
15            is_prime = 0;
16            break;
17          }
18        /* Est-ce le nombre premier que nous cherchons ? */
19        if (is_prime) {
20          if (--n == 0)
21        /* Renvoie le nombre premier désiré via la valeur de retour du thread. */
22            return (void*) candidate;
23        }
24        ++candidate;
25      }
26      return NULL;
27    }
28    int main ()
29    {
30      pthread_t thread;
31      int which_prime = 5000;
32      int prime;
33      /* Démarre le thread de calcul jusqu'au 5 000ème nombre premier. */
34      pthread_create (&thread, NULL, &compute_prime, &which_prime);
35      /* Faire autre chose ici... */
36      /* Attend la fin du thread de calcul et récupère le résultat. */
37      pthread_join (thread, (void*) &prime);
38      /* Affiche le nombre premier calculé. */
39      printf("Le %dème nombre premier est %d.\n", which_prime, prime);
40      return 0;
41    }
```

4.1.4 Plus d'informations sur les identifiants de thread

De temps à autre, il peut être utile pour une portion de code de savoir quel thread l'exécute. La fonction `pthread_self` renvoie l'identifiant du thread depuis lequel elle a été appelée. Cet identifiant de thread peut être comparé à un autre en utilisant la fonction `pthread_equal`.

Ces fonctions peuvent être utiles pour déterminer si un identifiant de thread particulier correspond au thread courant. Par exemple, un thread ne doit jamais appeler `pthread_join` pour s'attendre lui-même (dans ce cas, `pthread_join` renverrait le code d'erreur `EDEADLK`). Pour le vérifier avant l'appel, utilisez ce type de code :

```
if (!pthread_equal (pthread_self (), other_thread))
  pthread_join (other_thread, NULL);
```

4.1.5 Attributs de thread

Les attributs de thread proposent un mécanisme pour contrôler finement le comportement de threads individuels. Souvenez-vous que `pthread_create` accepte un argument qui est un pointeur vers un objet d'attributs de thread. Si vous passez un pointeur nul, les attributs par défaut sont utilisés pour configurer le nouveau thread. Cependant, vous pouvez créer et personnaliser un objet d'attributs de threads pour spécifier vos propres valeurs.

Pour indiquer des attributs de thread personnalisés, vous devez suivre ces étapes :

1. Créez un objet `pthread_attr_t`. La façon la plus simple de le faire est de déclarer une variable automatique de ce type.

2. Appelez `pthread_attr_init` en lui passant un pointeur vers cet objet. Cela initialise les attributs à leur valeur par défaut.
3. Modifiez l'objet d'attributs pour qu'ils contiennent les valeurs désirées.
4. Passez un pointeur vers l'objet créé lors de l'appel à `pthread_create`.
5. Appelez `pthread_attr_destroy` pour libérer l'objet d'attributs. La variable `pthread_attr_t` n'est pas libérée en elle-même ; elle peut être réinitialisée avec `pthread_attr_init`.

Un même objet d'attributs de thread peut être utilisé pour démarrer plusieurs threads. Il n'est pas nécessaire de conserver l'objet d'attributs de thread une fois qu'ils ont été créés.

Pour la plupart des applications GNU/Linux, un seul attribut de thread a généralement de l'intérêt (les autres attributs disponibles sont principalement destinés à la programmation en temps réel). Cet attribut est l'*état de détachement* (*detach state*) du thread. Un thread peut être un *thread joignable* (par défaut) ou un *thread détaché*. Les ressources d'un thread joignable, comme pour un processus, ne sont pas automatiquement libérées par GNU/Linux lorsqu'il se termine. Au lieu de cela, l'état de sortie du thread reste dans le système (un peu comme pour un processus zombie) jusqu'à ce qu'un autre thread appelle `pthread_join` pour récupérer cette valeur de retour. Alors seulement, les ressources sont libérées. Les ressources d'un thread détaché, au contraire, sont automatiquement libérées lorsqu'il se termine. Cela impliquer qu'un autre thread ne peut attendre qu'il se termine avec `pthread_join` ou obtenir sa valeur de retour.

La fonction `pthread_attr_setdetachstate` permet de définir l'état de détachement dans un objet d'attributs de thread. Le premier argument est un pointeur vers l'objet d'attributs de thread et le second est l'état désiré. Comme l'état joignable est la valeur par défaut, il n'est nécessaire d'effectuer cet appel que pour créer des threads détachés ; passez `PTHREAD_CREATE_DETACHED` comme second argument.

Le code du Listing 4.5 crée un thread détaché en activant l'attribut de détachement de thread :

Listing 4.5 – (***detached.c***) – **Programme Squelette Créant un Thread Détaché**

```
#include <pthread.h>
void* thread_function (void* thread_arg)
{
  /* Effectuer les traitements ici... */
}
int main ()
{
  pthread_attr_t attr;
  pthread_t thread;
  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);
  /* Effectuer les traitements ici...  */
  /* Pas besoin d'attendre le deuxième thread.  */
  return 0;
}
```

Même si un thread est créé dans un état joignable, il peut être transformé plus tard en un thread détaché. Pour cela, appelez `pthread_detach`. Une fois qu'un thread est détaché, il ne peut plus être rendu joignable.

4.2 Annulation de thread

Dans des circonstances normales, un thread se termine soit en arrivant à la fin de la fonction de thread, soit en appelant la fonction `pthread_exit`. Cependant, il est possible pour un thread de demander la fin d'un autre thread. Cela s'appelle annuler un thread.

Pour annuler un thread, appelez `pthread_cancel`, en lui passant l'identifiant du thread à annuler. Un thread annulé peut être joint ultérieurement ; en fait, vous devriez joindre un thread annulé pour libérer ses ressources, à moins que le thread ne soit détaché (consultez la Section 4.1.5, « Attributs de thread »). La valeur de retour d'un thread annulé est la valeur spéciale `PTHREAD_CANCELED`.

Souvent un thread peut être en train d'exécuter un code qui doit être exécuté totalement ou pas du tout. Par exemple, il peut allouer des ressources, les utiliser, puis les libérer. Si le thread est annulé au milieu de ce code, il peut ne pas avoir l'opportunité de libérer les ressources et elles seront perdues. Pour éviter ce cas de figure, un thread peut contrôler si et quand il peut être annulé.

Un thread peut se comporter de trois façons face à l'annulation.

- Il peut être *annulable de façon asynchrone*. C'est-à-dire qu'il peut être annulé à n'importe quel moment de son exécution.
- Il peut être *annulable de façon synchrone*. Le thread peut être annulé, mais pas n'importe quand pendant son exécution. Au lieu de cela, les requêtes d'annulation sont mises en file d'attente et le thread n'est annulé que lorsqu'il atteint un certain point de son exécution.
- Il peut être *impossible à annuler*. Les tentatives d'annulation sont ignorées silencieusement.

Lors de sa création, un thread est annulable de façon synchrone.

4.2.1 Threads synchrones et asynchrones

Un thread annulable de façon asynchrone peut être annulé à n'importe quel point de son exécution. Un thread annulable de façon synchrone, par contre, ne peut être annulé qu'à des endroits précis de son exécution. Ces endroits sont appelés *points d'annulation*. Le thread mettra les requêtes d'annulation en attente jusqu'à ce qu'il atteigne le point d'annulation suivant.

Pour rendre un thread annulable de façon asynchrone, utilisez `pthread_setcanceltype`. Cela n'affecte que le thread effectuant l'appel. Si vous souhaitez rendre le thread annulable de façon asynchrone le premier argument doit être `PTHREAD_CANCEL_ASYNCHRONOUS`, si, vous préférez revenir immédiatement en état d'annulation synchrone passez `PTHREAD_CANCEL_DEFERRED`. Le second argument, s'il n'est pas `NULL`, est un pointeur vers une variable qui recevra le type d'annulation précédemment supportée par le thread. Cet appel, par exemple, rend le thread appelant annulable de façon asynchrone.

```
pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

A quoi ressemble un point d'annulation et où doit-il être placé ? La façon la plus directe de créer un point d'annulation est d'appeler `pthread_testcancel`. Cette fonction ne fait rien à part traiter une annulation en attente dans un thread annulable de façon synchrone. Vous devriez appeler `pthread_testcancel` périodiquement durant des calculs longs au sein d'une fonction de thread, aux endroits où le thread peut être annulé sans perte de ressources ou autres effets de bord.

Un certain nombre d'autres fonctions sont également des points d'annulation implicites. Elles sont listées sur la page de manuel de `pthread_cancel`. Notez que d'autres fonctions peuvent y faire appel en interne et donc constituer des points d'annulation implicites.

4.2.2 Sections critiques non-annulables

Un thread peut désactiver son annulation avec la fonction `pthread_setcancelstate`. Comme `pthread_setcanceltype`, elle affecte le thread appelant. La désactivation de l'annulation se fait en passant `PTHREAD_CANCEL_DISABLE` comme premier argument ; `PTHREAD_CANCEL_ENABLE` permet de la réactiver. Le second argument, s'il n'est pas `NULL`, pointe vers une variable qui recevra l'état précédent de l'annulation. Cet appel, par exemple, désactive l'annulation du thread appelant.

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, NULL);
```

L'utilisation de `pthread_setcancelstate` vous permet d'implémenter des sections critiques. Une section critique est une séquence de code qui doit être exécutée entièrement ou pas du tout ; en d'autres termes, si un thread commence l'exécution d'une section critique, il doit atteindre la fin de la section critique sans être annulé.

Par exemple, supposons que vous écriviez une routine pour un programme bancaire qui transfère de l'argent d'un compte à un autre. Pour cela, vous devez ajouter la valeur au solde d'un compte et la soustraire du solde de l'autre. Si le thread exécutant votre routine est annulé entre les deux opérations, le programme aura augmenté de façon incorrecte l'argent détenu par la banque en ne terminant pas la transaction. Pour éviter cela, placez les deux opérations au sein d'une section critique.

Vous pourriez implémenter le transfert avec une fonction comme `process_transaction` présentée dans le Listing 4.6. Cette fonction désactive l'annulation de thread pour démarrer une section critique avant toute modification de solde.

Listing 4.6 – (*critical-section.c*) – **Transaction avec Section Critique**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <string.h>
4  /* Tableau des soldes de comptes, indexé par numéro de compte.  */
5  float* account_balances;
6  /* Transfère DOLLARS depuis le compte FROM_ACCT vers TO_ACCT. Renvoie
7     0 si la transaction s'est bien déroulée ou 1 si le solde de FROM_ACCT
8     est trop faible. */
9  int process_transaction (int from_acct, int to_acct, float dollars)
10 {
11   int old_cancel_state;
12   /* Vérifie le solde de FROM_ACCT. */
13   if (account_balances[from_acct] < dollars)
14     return 1;
15   /* Début de la section critique. */
16   pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &old_cancel_state);
17   /* Transfère l'argent. */
18   account_balances[to_acct] += dollars;
19   account_balances[from_acct] -= dollars;
20   /* Fin de la section critique. */
21   pthread_setcancelstate (old_cancel_state, NULL);
22   return 0;
23 }
```

Notez qu'il est important de restaurer l'ancien état de l'annulation à la fin de la section critique plutôt que de le positionner systématiquement à `PTHREAD_CANCEL_ENABLE`. Cela vous permet d'appeler la fonction `process_transaction` en toute sécurité depuis une autre section critique – dans ce cas, votre fonction remettrait l'état de l'annulation à la même valeur que lorsqu'elle a débuté.

4.2.3 Quand utiliser l'annulation de thread ?

En général, ce n'est pas une bonne idée d'utiliser l'annulation de thread pour en terminer l'exécution, excepté dans des circonstances anormales. En temps normal, il est mieux d'indiquer au thread qu'il doit se terminer, puis d'attendre qu'il se termine de lui-même. Nous reparlerons des techniques de communication avec les threads plus loin dans ce chapitre et dans le Chapitre 5, « Communication interprocessus ».

4.3 Données propres à un thread

Contrairement aux processus, tous les threads d'un même programme partagent le même espace d'adressage. Cela signifie que si un thread modifie une valeur en mémoire (par exemple, une variable globale), cette modification est visible dans tous les autres threads. Cela permet à plusieurs threads de manipuler les mêmes données sans utiliser les mécanismes de communication interprocessus (décrits au Chapitre 5).

Néanmoins, chaque thread a sa propre pile d'appel. Cela permet à chacun d'exécuter un code différent et d'utiliser des sous-routines la façon classique. Comme dans un programme monothreadé, chaque invocation de sous-routine dans chaque thread a son propre jeu de variables locales, qui sont stockées dans la pile de ce thread.

Cependant, il peut parfois être souhaitable de dupliquer certaines variables afin que chaque thread dispose de sa propre copie. GNU/Linux supporte cette fonctionnalité avec une zone de *données propres au thread*. Les variables stockées dans cette zone sont dupliquées pour chaque thread et chacun peut modifier sa copie sans affecter les autres. Comme tous les threads partagent le même espace mémoire, il n'est pas possible d'accéder aux données propres à un thread *via* des références classiques. GNU/Linux fournit des fonction dédiées pour placer des valeurs dans la zone de données propres au thread et les récupérer.

Vous pouvez créer autant d'objets de données propres au thread que vous le désirez, chacune étant de type `void*`. Chaque objet est référencé par une clé. Pour créer une nouvelle clé, et donc un nouvel objet de données pour chaque thread, utilisez `pthread_key_create`. Le premier argument est un pointeur vers une variable `pthread_key_t`. Cette clé peut être utilisée par chaque thread pour accéder à sa propre copie de l'objet de données correspondant. Le second argument de `pthread_key_create` est une fonction de libération des ressources. Si vous passez un pointeur sur une fonction, GNU/Linux appelle celle-ci automatiquement lorsque chaque thread se termine en lui passant la valeur de la donnée propre au thread correspondant à la clé. Ce mécanisme est très pratique car la fonction de libération des ressources est appelée même si le thread est annulé à un moment quelconque de son exécution. Si la valeur propre au thread est nulle, la fonction

de libération de ressources n'est pas appelée. Si vous n'avez pas besoin de fonction de libération de ressources, vous pouvez passer `NULL` au lieu d'un pointeur de fonction.

Une fois que vous avez créé une clé, chaque thread peut positionner la valeur correspondant à cette clé en appelant `pthread_setspecific`. Le premier argument est la clé, et le second est la valeur propre au thread à stocker sous forme de `void*`. Pour obtenir un objet de données propre au thread, appelez `pthread_getspecific`, en lui passant la clé comme argument.

Supposons, par exemple, que votre application partage une tâche entre plusieurs threads. À des fins d'audit, chaque thread doit avoir un fichier journal séparé dans lequel des messages d'avancement pour ce thread sont enregistrés. La zone de données propres au thread est un endroit pratique pour stocker le pointeur sur le fichier journal dans chaque thread.

Le Listing 4.7 présente une implémentation d'un tel mécanisme. La fonction `main` de cet exemple crée une clé pour stocker le pointeur vers le fichier propre à chaque thread puis la stocke au sein de la variable globale `thread_lock_key`. Comme il s'agit d'une variable globale, elle est partagée par tous les threads. Lorsqu'un thread commence à exécuter sa fonction, il ouvre un fichier journal et stocke le pointeur de fichier sous cette clé. Plus tard, n'importe lequel de ces thread peut appeler `write_to_thread_log` pour écrire un message dans le fichier journal propre au thread. Cette fonction obtient le pointeur de fichier du fichier log du thread depuis les données propres au thread et y écrit le message.

Listing 4.7 – (*tsd.c*) – **Fichiers Logs par Thread avec Données Propres au Thread**

```
#include <malloc.h>
#include <pthread.h>
#include <stdio.h>
/* Clé utilisée pour associer un pointeur de fichier à chaque thread. */
static pthread_key_t thread_log_key;
/* Écrit MESSAGE vers le fichier journal du thread courant. */
void write_to_thread_log (const char* message)
{
  FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);
  fprintf (thread_log, "%s\n", message);
}
/* Ferme le pointeur vers le fichier journal THREAD_LOG.  */
void close_thread_log (void* thread_log){
  fclose ((FILE*) thread_log);
}
void* thread_function (void* args)
{
  char thread_log_filename[20];
  FILE* thread_log;
  /* Génère le nom de fichier pour le fichier journal de ce thread. */
  sprintf (thread_log_filename, "thread%d.log", (int) pthread_self ());
  /* Ouvre le fichier journal. */
  thread_log = fopen (thread_log_filename, "w");
  /* Stocke le pointeur de fichier dans les données propres au thread sous
     la clé thread_log_key. */
  pthread_setspecific (thread_log_key, thread_log);
  write_to_thread_log ("Démarrage du Thread.");
  /* Placer les traitements ici... */
  return NULL;
}
int main ()
{
  int i;
  pthread_t threads[5];
```

```
35    /* Crée une clé pour associer les pointeurs de fichier journal dans les
36       données propres au thread. Utilise close_thread_log pour libérer
37       les pointeurs de fichiers. */
38    pthread_key_create (&thread_log_key, close_thread_log);
39    /* Crée des threads pour effectuer les traitements. */
40    for (i = 0; i < 5; ++i)
41      pthread_create (&(threads[i]), NULL, thread_function, NULL);
42    /* Attend la fin de tous les threads. */
43    for (i = 0; i < 5; ++i)
44      pthread_join (threads[i], NULL);
45    return 0;
46  }
```

Remarquez que `thread_function` n'a pas besoin de fermer le fichier journal. En effet, lorsque la clé du fichier a été créée, `close_thread_log` a été spécifiée comme fonction de libération de ressources pour cette clé. Lorsqu'un thread se termine, GNU/Linux appelle cette fonction, en lui passant la valeur propre au thread correspondant à la clé du fichier journal. Cette fonction prend soin de fermer le fichier journal.

`Gestionnaires de Libération de Ressources`

Les fonctions de libération de ressources associées aux clés des données spécifiques au thread peuvent être très pratiques pour s'assurer que les ressources ne sont pas perdues lorsqu'un thread se termine ou est annulé. Quelquefois, cependant, il est utile de pouvoir spécifier des fonctions de libération de ressources sans avoir à créer un nouvel objet de données propre au thread qui sera dupliqué pour chaque thread. Pour ce faire, GNU/Linux propose des *gestionnaires de libération de ressources*.

Un gestionnaire de libération de ressources est tout simplement une fonction qui doit être appelée lorsqu'un thread se termine. Le gestionnaire prend un seul paramètre `void*` et la valeur de l'argument est spécifiée lorsque le gestionnaire est enregistré – cela facilite l'utilisation du même gestionnaire pour libérer plusieurs instances de ressources.

Un gestionnaire de libération de ressources est une mesure temporaire, utilisé pour libérer une ressource uniquement si le thread se termine ou est annulé au lieu de terminer l'exécution d'une certaine portion de code. Dans des circonstances normales, lorsqu'un thread ne se termine pas et n'est pas annulé, la ressource doit être libérée explicitement et le gestionnaire de ressources supprimé.

Pour enregistrer un gestionnaire de libération de ressources, appelez `pthread_cleanup_push`, en lui passant un pointeur vers la fonction de libération de ressources et la valeur de son argument `void*`. L'enregistrement du gestionnaire doit être équilibré par un appel à `pthread_cleanup_pop` qui le supprime. Dans une optique de simplification, `pthread_cleanup_pop` prend un indicateur `int` en argument ; s'il est différent de zéro, l'action de libération de ressources est exécuté lors de la suppression.

L'extrait de programme du Listing 4.8 montre comment vous devriez utiliser un gestionnaire de libération de ressources pour vous assurer qu'un tampon alloué dynamiquement est libéré si le thread se termine.

Listing 4.8 – (***cleanup.c***) – **Extrait de Programme Montrant l'Utilisation d'un Gestionnaire de Libération de Ressources**

```
1  #include <malloc.h>
2  #include <pthread.h>
```

```
3  /* Alloue un tampon temporaire.  */
4  void* allocate_buffer (size_t size)
5  {
6    return malloc (size);
7  }
8  /* Libère un tampon temporaire.  */
9  void deallocate_buffer (void* buffer)
10 {
11   free (buffer);
12 }
13 void do_some_work ()
14 {
15   /* Alloue un tampon temporaire.  */
16   void* temp_buffer = allocate_buffer (1024);
17   /* Enregistre un gestionnaire de libération de ressources pour ce tampon
18      pour le libérer si le thread se termine ou est annulé. */
19   pthread_cleanup_push (deallocate_buffer, temp_buffer);
20   /* Placer ici des traitements qui pourraient appeler pthread_exit ou être
21      annulés... */
22   /* Supprime le gestionnaire de libération de ressources. Comme nous passons
23      une valeur différente de zéro, la libération est effectuée
24      par l'appel de deallocate_buffer. */
25   pthread_cleanup_pop (1);
26 }
```

Comme l'argument de `pthread_cleanup_pop` est différent de zéro dans ce cas, la fonction de libération de ressources `deallocate_buffer` est appelée automatiquement et il n'est pas nécessaire de le faire explicitement. Dans ce cas simple, nous aurions pu utiliser la fonction de la bibliothèque standard `free` comme fonction de libération de ressources au lieu de `deallocate_buffer`.

4.3.1 Libération de ressources de thread en C++

Les programmeurs C++ sont habitués à avoir des fonctions de libération de ressources «gratuitement» en plaçant les actions adéquates au sein de destructeurs. Lorsque les objets sont hors de portée, soit à cause de la fin d'un bloc, soit parce qu'une exception est lancée, le C++ s'assure que les destructeurs soient appelés pour les variables automatiques qui en ont un. Cela offre un mécanisme pratique pour s'assurer que le code de libération de ressources est appelé quelle que soit la façon dont le bloc se termine.

Si un thread appelle `pthread_exit`, cependant, le C++ ne garantit pas que les destructeurs soient appelés pour chacune des variables automatiques situées sur la pile du thread. Une façon intelligente d'obtenir le même comportement est d'invoquer `pthread_exit` au niveau de la fonction de thread en lançant une exception spéciale.

Le programme du Listing 4.9 démontre cela. Avec cette technique, une fonction indique son envie de quitter le thread en lançant une `ThreadExitException` au lieu d'appeler `pthread_exit` directement. Comme l'exception est interceptée au niveau de la fonction de thread, toutes les variables locales situées sur la pile du thread seront détruites proprement lorsque l'exception est remontée.

Listing 4.9 – (*cxx-exit.cpp*) – **Implémenter une Sortie de Thread en C++**

```
1  #include <pthread.h>
2  class ThreadExitException
3  {
```

```
4  public:
5    /* Crée une exception signalant la fin d'un thread avec RETURN_VALUE. */
6    ThreadExitException (void* return_value)
7      : thread_return_value_ (return_value)   {
8     }
9     /* Quitte le thread en utilisant la valeur de retour fournie dans le
10       constructeur. */
11    void* DoThreadExit ()
12    {
13      pthread_exit (thread_return_value_);
14    }
15 private:
16    /* Valeur de retour utilisée lors de la sortie du thread. */
17    void* thread_return_value_;
18 };
19 void do_some_work ()
20 {
21    while (1) {
22      /* Placer le code utile ici... */
23      if (should_exit_thread_immediately ())
24        throw ThreadExitException (/* thread?s return value = */ NULL);
25    }
26 }
27 void* thread_function (void*)
28 {
29   try {
30     do_some_work ();
31   }
32   catch (ThreadExitException ex) {
33     /* Une fonction a signalé que l'on devait quitter le thread. */
34     ex.DoThreadExit ();
35   }
36   return NULL;
37 }
```

4.4 Synchronisation et sections critiques

Programmer en utilisant les threads demande beaucoup de rigueur car la plupart des programmes utilisant les threads sont des programmes concurrents. En particulier, il n'y a aucun moyen de connaître la façon dont seront ordonnancés les threads les uns par rapport aux autres. Un thread peut s'exécuter pendant longtemps ou le système peut basculer d'un thread à un autre très rapidement. Sur un système avec plusieurs processeurs, le système peut même ordonnancer plusieurs threads afin qu'ils s'exécutent physiquement en même temps.

Déboguer un programme qui utilise les threads est compliqué car vous ne pouvez pas toujours reproduire facilement le comportement ayant causé le problème. Vous pouvez lancer le programme une première fois sans qu'aucun problème ne survienne et la fois suivante, il plante. Il n'y a aucun moyen de faire en sorte que le système ordonnance les threads de la même façon d'une fois sur l'autre.

La cause la plus vicieuse de plantage des programmes utilisant les threads est lorsque ceux-ci tentent d'accéder aux mêmes données. Comme nous l'avons dit précédemment, il s'agit d'un des aspects les plus puissants des threads mais cela peut également être dangereux. Si un thread n'a pas terminé la mise à jour d'une structure de données lorsqu'un autre thread y accède, il s'en suit

une situation imprévisible. Souvent, les programmes bogués qui utilisent les threads contiennent une portion de code qui ne fonctionne que si un thread a la main plus souvent – ou plus tôt – qu'un autre. Ces bogues sont appelés *conditions de concurrence critique* ; les threads sont en concurrence pour modifier la même structure de données.

4.4.1 Conditions de concurrence critique

Supposons que votre programme ait une série de tâches en attente traitées par plusieurs threads concurrents. La file d'attente des tâches est représentée par une liste chaînée d'objets `struct job`.

Après que chaque thread a fini une opération, il vérifie la file pour voir si une nouvelle tâche est disponible. Si `job_queue` n'est pas `NULL`, le thread supprime la tête de la liste chaînée et fait pointer `job_queue` vers la prochaine tâche de la liste.

La fonction de thread qui traite les tâches de la liste pourrait ressembler au Listing 4.10.

Listing 4.10 – (***job-queue1.c***) – **Fonction de Thread Traitant une File de Tâches**

```
#include <malloc.h>
struct job {
   /* Champ de chaînage.  */
   struct job* next;
   /* Autres champs décrivant la tâche... */
};
/* Liste chaînée de tâches en attente. */
struct job* job_queue;
/* Traite les tâches jusqu'à ce que la file soit vide. */
void* thread_function (void* arg)
{
   while (job_queue != NULL) {
     /* Récupère la tâche suivante. */
     struct job* next_job = job_queue;
     /* Supprime cette tâche de la liste. */
     job_queue = job_queue->next;
     /* Traite la tâche. */
     process_job (next_job);
     /* Libération des ressources. */
     free (next_job);
   }
   return NULL;
}
```

Supposons maintenant que deux threads finissent une tâche à peu près au même moment, mais qu'il ne reste qu'une seule tâche dans la liste. Le premier thread regarde si `job_queue` est nul ; comme ce n'est pas le cas, le thread entre dans la boucle et stocke le pointeur vers la tâche dans `next_job`. À ce moment, Linux interrompt le thread et passe la main au second. Le second thread vérifie également `job_queue`, comme elle n'est pas `NULL`, affecte la même valeur que le premier à `next_job`. Par une malheureuse coïncidence, nous avons deux threads exécutant la même tâche.

Pire, un des thread va positionner `job_queue` à `NULL` pour supprimer l'objet de la liste. Lorsque l'autre évaluera `job_queue->next`, il en résultera une erreur de segmentation.

C'est un exemple de condition de concurrence critique. En d'« heureuses » circonstances, cet ordonnancement particulier des threads ne surviendra jamais et la condition de concurrence

critique ne sera jamais révélée. Dans des circonstances différentes, par exemple dans le cas d'un système très chargé (ou sur le nouveau serveur multiprocesseur d'un client important !) le bogue peut apparaître.

Pour éliminer les conditions de concurrence critique, vous devez trouver un moyen de rendre les opérations atomiques. Une opération atomique est indivisible et impossible à interrompre ; une fois qu'elle a débuté, elle ne sera pas suspendue ou interrompue avant d'être terminée, et aucune autre opération ne sera accomplie pendant ce temps. Dans notre exemple, nous voulons vérifier la valeur de `job_queue` et si elle n'est pas `NULL`, supprimer la première tâche, tout cela en une seule opération atomique.

4.4.2 Mutexes

La solution pour notre problème de concurrence critique au niveau de la file de tâches est de n'autoriser qu'un seul thread à accéder à la file. Une fois que le thread commence à observer la file d'attente, aucun autre thread ne doit pouvoir y accéder jusqu'à ce que le premier ait décidé s'il doit traiter une tâche, et si c'est le cas, avant qu'il n'ait supprimé la tâche de la liste.

L'implantation de ce mécanisme requiert l'aide du système d'exploitation. GNU/Linux propose des *mutexes*, raccourcis de *MUTual EXclusion locks* (verrous d'exclusion mutuelle). Un mutex est un verrouillage spécial qu'un seul thread peut utiliser à la fois. Si un thread verrouille un mutex puis qu'un second tente de verrouiller le même mutex, ce dernier est *bloqué* ou suspendu. Le second thread est *débloqué* uniquement lorsque le premier déverrouille le mutex – ce qui permet la reprise de l'exécution. GNU/Linux assure qu'il n'y aura pas de condition de concurrence critique entre deux threads tentant de verrouiller un mutex ; seul un thread obtiendra le verrouillage et tous les autres seront bloqués.

On peut faire l'analogie entre un mutex et une porte de toilettes. Lorsque quelqu'un entre dans les toilettes et verrouille la porte, si une personne veut entrer dans les toilettes alors qu'ils sont occupés, elle sera obligée d'attendre dehors jusqu'à ce que l'occupant sorte.

Pour créer un mutex, créez une variable `pthread_mutex_t` et passez à `pthread_mutex_init` un pointeur sur cette variable. Le second argument de `pthread_mutex_init` est un pointeur vers un objet d'attributs de mutex. Comme pour `pthread_create`, si le pointeur est nul, ce sont les valeurs par défaut des attributs qui sont utilisées. La variable mutex ne doit être initialisée qu'une seule fois. Cet extrait de code illustre la déclaration et l'initialisation d'une variable mutex :

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Une façon plus simple de créer un mutex avec les attributs par défaut est de l'initialiser avec la valeur spéciale `PTHREAD_MUTEX_INITIALIZER`. Aucun appel à `pthread_mutex_init` n'est alors nécessaire. Cette méthode est particulièrement pratique pour les variables globales (et les membres `static` en C++). L'extrait de code précédent pourrait être écrit comme suit :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un thread peut essayer de verrouiller un mutex en appelant `pthread_mutex_lock` dessus. Si le mutex était déverrouillé, il devient verrouillé et la fonction se termine immédiatement. Si le mutex était verrouillé par un autre thread, `pthread_mutex_lock` bloque l'exécution et ne se termine que lorsque le mutex est déverrouillé par l'autre thread. Lorsque le mutex est déverrouillé, seul un

des threads suspendus (choisi aléatoirement) est débloqué et autorisé à accéder au mutex ; les autres threads restent bloqués.

Un appel à `pthread_mutex_unlock` déverrouille un mutex. Cette fonction doit toujours être appelée par le thread qui a verrouillé le mutex.

Listing 4.11 montre une autre version de la file de tâches. Désormais, celle-ci est protégée par un mutex. Avant d'accéder à la file (que ce soit pour une lecture ou une écriture), chaque thread commence par verrouiller le mutex. Ce n'est que lorsque la séquence de vérification de la file et de suppression de la tâche est terminée que le mutex est débloqué. Cela évite l'apparition des conditions de concurrence ciritique citées précédemment.

Listing 4.11 – (***job-queue.c***) – **Fonction de Thread de File de Tâches, Protégée par un Mutex**

```
#include <malloc.h>
#include <pthread.h>
struct job {
   /* Pointeur vers la tâche suivante.  */
   struct job* next;
};
/* Liste chaînée des tâches en attente.   */
struct job* job_queue;
/* Mutex protégeant la file de tâches. */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
/* Traite les tâches en attente jusqu'à ce que la file soit vide.  */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    /* Verrouille le mutex de la file de tâches. */
    pthread_mutex_lock (&job_queue_mutex);
    /* Il est maintenant sans danger de vérifier si la file est vide. */
    if (job_queue == NULL)
      next_job = NULL;
    else {
      /* Récupère la tâche suivante. */
      next_job = job_queue;
      /* Supprime cette tâche de la liste. */
      job_queue = job_queue->next;
    }
    /* Déverrouille le mutex de la file de tâches car nous en avons fini avec
       la file pour l'instant. */
    pthread_mutex_unlock (&job_queue_mutex);
    /* La file était vide ? Si c'est la cas, le thread se termine.  */
    if (next_job == NULL)
      break;
    /* Traite la tâche. */
    process_job (next_job);
    /* Libération des ressources. */
    free (next_job);
  }
  return NULL;
}
```

Tous les accès à `job_queue`, le pointeur de données partagé, ont lieu dans la section comprise entre l'appel à `pthread_mutex_lock` et celui à `pthread_mutex_unlock`. L'accès à un objet décrivant une tâche, stocké dans `next_job`, a lieu en dehors de cette zone uniquement une fois que l'objet a été supprimé de la liste et a donc été rendu inaccessible aux autres threads.

Notez que si la file est vide (c'est-à-dire que `job_queue` est `NULL`), nous ne quittons pas la boucle immédiatement car dans ce cas le mutex resterait verrouillé empêchant l'accès à la file de tâche par un quelconque autre thread. Au lieu de cela, nous notons que c'est le cas en mettant `next_job` à `NULL` et en ne quittant la boucle qu'après avoir déverrouillé le mutex.

L'utilisation de mutex pour verrouiller `job_queue` n'est pas automatique ; c'est à vous d'ajouter le code pour verrouiller le mutex avant d'accéder à la variable et le déverrouiller ensuite. Par exemple, une fonction d'ajout de tâche à la file pourrait ressembler à ce qui suit :

```
void enqueue_job (struct job* new_job)
{
  pthread_mutex_lock (&job_queue_mutex);
  new_job->next = job_queue;
  job_queue = new_job;
  pthread_mutex_unlock (&job_queue_mutex);
}
```

4.4.3 Interblocage de mutexes

Les mutexes offrent un mécanisme permettant à un thread de bloquer l'exécution d'un autre. Cela ouvre la possibilité à l'apparition d'une nouvelle classe de bogues appelés *interblocages* (deadlocks). Un interblocage survient lorsque un ou plusieurs threads sont bloqués en attendant quelque chose qui n'aura jamais lieu.

Un type simple d'interblocage peut survenir lorsqu'un même thread tente de verrouiller un mutex deux fois d'affilé. Ce qui se passe dans un tel cas dépend du type de mutex utilisé. Trois types de mutexes existent :

- Le verrouillage d'un *mutex rapide* (le type par défaut) provoquera un interblocage. Une tentative de verrouillage sur le mutex est bloquante jusqu'à ce que le mutex soit déverrouillé. Mais comme le thread est bloqué sur un mutex qu'il a lui-même verrouillé, le verrou ne pourra jamais être supprimé.
- Le verrouillage d'un *mutex récursif* ne cause pas d'interblocage. Un mutex récursif peut être verrouillé plusieurs fois par le même thread en toute sécurité. Le mutex se souvient combien de fois `pthread_mutex_lock` a été appelé par le thread qui détient le verrou ; ce thread doit effectuer autant d'appels à `pthread_mutex_unlock` pour que le mutex soit effectivement déverrouillé et qu'un autre thread puisse y accéder.
- GNU/Linux détectera et signalera un double verrouillage sur un *mutex à vérification d'erreur* qui causerait en temps normal un interblocage. Le second appel à `pthread_mutex_lock` renverra le code d'erreur `EDEADLK`.

Par défaut, un mutex GNU/Linux est de type rapide. Pour créer un mutex d'un autre type, commencez par créer un objet d'attributs de mutex en déclarant une variable de type `pthread_mutexattr_t` et appelez `pthread_mutexattr_init` en lui passant un pointeur dessus. Puis définissez le type de mutex en appelant `pthread_mutexattr_setkind_np` ; son premier argument est un pointeur vers l'objet d'attributs de mutex et le second est `PTHREAD_MUTEX_RECURSIVE_NP` pour un mutex récursif ou `PTHREAD_MUTEX_ERRORCHECK_NP` pour un mutex à vérification d'erreurs. Passez un pointeur vers l'objet d'attributs de mutex à `pthread_mutex_init` pour créer un mutex du type désiré, puis détruisez l'objet d'attributs avec `pthread_mutexattr_destroy`.

Le code suivant illustre la création d'un mutex à vérification d'erreurs, par exemple :

```
pthread_mutexattr_t attr;
pthread_mutex_t mutex;
pthread_mutexattr_init (&attr);
pthread_mutexattr_setkind_np (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
pthread_mutexattr_destroy (&attr);
```

Comme le suggère le suffixe «`np`», les mutexes récursifs et à vérification d'erreurs sont spécifiques à GNU/Linux et ne sont pas portables. Ainsi, il est généralement déconseillé de les utiliser dans vos programmes (les mutexes à vérification d'erreurs peuvent cependant être utiles lors du débogage).

4.4.4 Vérification de mutex non bloquante

De temps en temps, il peut être utile de tester si un mutex est verrouillé sans être bloqué s'il l'est. Par exemple, un thread peut avoir besoin de verrouiller un mutex mais devoir faire autre chose au lieu de se bloquer si le mutex est déjà verrouillé. Comme `pthread_mutex_lock` ne se termine pas avant le déverrouillage du mutex, une autre fonction est nécessaire.

GNU/Linux fournit `pthread_mutex_trylock` pour ce genre de choses. Appelé sur un mutex déverrouillé, `pthread_mutex_trylock` verrouille le mutex comme le ferait `pthread_mutex_lock` et renvoie zéro. Par contre, si le mutex est déjà verrouillé par un autre mutex, `pthread_mutex_trylock` ne sera pas bloquante. Au lieu de cela, elle se terminera en renvoyant le code d'erreur `EBUSY`. Le verrou sur le mutex détenu par l'autre thread n'est pas affecté. Vous pouvez réessayer plus tard d'obtenir un verrou.

4.4.5 Sémaphores pour les threads

Dans l'exemple précédent, dans lequel plusieurs threads traitent les tâches d'une file, la fonction principale des threads traite la tâche suivante jusqu'à ce qu'il n'en reste plus, à ce moment, le thread se termine. Ce schéma fonctionne si toutes les tâches sont mises dans la file au préalable ou si de nouvelles tâches sont ajoutées au moins aussi vite que la vitesse de traitement des threads. Cependant, si les threads fonctionnent trop rapidement, la file de tâches se videra et les threads se termineront. Si de nouvelles tâches sont mises dans la file plus tard, il ne reste plus de thread pour les traiter. Ce que nous devrions faire est de mettre en place un mécanisme permettant de bloquer les threads lorsque la file se vide et ce jusqu'à ce que de nouvelles tâches soient disponibles.

L'utilisation de *sémaphores* permet de faire ce genre de choses. Un sémaphore est un compteur qui peut être utilisé pour synchroniser plusieurs threads. Comme avec les mutexes, GNU/Linux garantit que la vérification ou la modification de la valeur d'un sémaphore peut être accomplie en toute sécurité, sans risque de concurrence critique.

Chaque sémaphore dispose d'une valeur de compteur, qui est un entier positif ou nul. Un sémaphore supporte deux opérations de base :

- Une opération d'*attente* (wait), qui décrémente la valeur du sémaphore d'une unité. Si la valeur est déjà à zéro, l'opération est bloquante jusqu'à ce que la valeur du sémaphore redevienne positive (en raison d'une opération de la part d'un autre thread). Lorsque

la valeur du sémaphore devient positive, elle est décrémentée d'une unité et l'opération d'attente se termine.

– Une opération de *réveil* (post) qui incrémente la valeur du sémaphore d'une unité. Si le sémaphore était précédemment à zéro et que d'autres threads étaient en attente sur ce même sémaphore, un de ces threads est débloqué et son attente se termine (ce qui ramène la valeur du sémaphore à zéro).

Notez que GNU/Linux fournit deux implémentations légèrement différentes des sémaphores. Celle que nous décrivons ici est l'implantation POSIX standard. Utilisez cette implémentation lors de la communication entre threads. L'autre implémentation, utilisée pour la communication entre les processus, est décrite dans la Section 5.2, « Sémaphores de processus ». Si vous utilisez les sémaphores, incluez `<semaphore.h>`.

Un sémaphore est représenté par une variable `sem_t`. Avant de l'utiliser, vous devez l'initialiser par le biais de la fonction `sem_init`, à laquelle vous passez un pointeur sur la variable `sem_t`. Le second paramètre doit être à zéro[3], et le troisième paramètre est la valeur initiale du sémaphore. Si vous n'avez plus besoin d'un sémaphore, il est conseillé de libérer les ressources qu'il occupe avec `sem_destroy`.

Pour vous mettre en attente sur un sémaphore, utilisez `sem_wait`. Pour effectuer une opération de réveil, utilisez `sem_post`. Une fonction permettant une mise en attente non bloquante, `sem_trywait`, est également fournie. Elle est similaire à `pthread_mutex_trylock` – si l'attente devait bloquer en raison d'un sémaphore à zéro, la fonction se termine immédiatement, avec le code d'erreur `EAGAIN`, au lieu de bloquer le thread.

GNU/Linux fournit également une fonction permettant d'obtenir la valeur courante d'un sémaphore, `sem_getvalue`, qui place la valeur dans la variable `int` pointée par son second argument. Vous ne devriez cependant pas utiliser la valeur du sémaphore que vous obtenez à partir de cette fonction pour décider d'une opération d'attente ou de réveil sur un sémaphore. Utiliser ce genre de méthode peut conduire à des conditions de concurrence critique : un autre thread peut changer la valeur du sémaphore entre l'appel de `sem_getvalue` et l'appel à une autre fonction de manipulation des sémaphores. Utilisez plutôt les fonctions d'opérations d'attente et de réveil atomiques.

Pour revenir à notre exemple de file d'attente de tâches, nous pouvons utiliser un sémaphore pour compter le nombre de tâches en attente dans la file. Le Listing 4.12 contrôle la file avec un sémaphore. La fonction `enqueue_job` ajoute une nouvelle tâche à la file d'attente.

Listing 4.12 – (*job-queue3.c*) – **File de Tâches Contrôlée par un Sémaphore**

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct job {
   /* Champ de chaînage.  */
   struct job* next;
   /* Autres champs décrivant la tâche... */
};
/* Liste chaînée des tâches en attente. */
struct job* job_queue;
```

[3] Une valeur différente de zéro indiquerait que le sémaphore peut être partagé entre les processus, ce qui n'est pas supporté sous GNU/Linux pour ce type de sémaphore.

```
11 /* Mutex protégeant job_queue. */
12 pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
13 /* Sémaphore comptant le nombre de tâches dans la file. */
14 sem_t job_queue_count;
15 /* Initialisation de la file de tâches. */
16 void initialize_job_queue ()
17 {
18   /* La file est initialement vide. */
19   job_queue = NULL;
20   /* Initialise le sémaphore avec le nombre de tâches dans la file. Sa valeur
21      initiale est zéro. */
22   sem_init (&job_queue_count, 0, 0);
23 }
24 /* Traite les tâches en attente jusqu'à ce que la file soit vide. */
25 void* thread_function (void* arg)
26 {
27   while (1) {
28     struct job* next_job;
29     /* Se met en attente sur le sémaphore de la file de tâches. Si sa valeur
30        est positive, indiquant que la file n'est pas vide, décrémente
31        le total d'une unité. Si la file est vide, bloque jusqu'à ce
32        qu'une nouvelle tâche soit mise en attente. */
33     sem_wait (&job_queue_count);
34     /* Verrouille le mutex sur la file de tâches. */
35     pthread_mutex_lock (&job_queue_mutex);
36     /* À cause du sémaphore, nous savons que la file n'est pas vide. Récupère
37     donc la prochaine tâche disponible. */
38     next_job = job_queue;
39     /* Supprime la tâche de la liste. */
40     job_queue = job_queue->next;
41     /* Déverrouille le mutex de la file d'attente car nous en avons fini avec
42        celle-ci pour le moment. */
43     pthread_mutex_unlock (&job_queue_mutex);
44     /* Traite la tâche. */
45     process_job (next_job);
46     /* Libération des ressources. */
47     free (next_job);
48   }
49   return NULL;
50 }
51 /* Ajoute une nouvelle tâche en tête de la file.  */
52 void enqueue_job (/* Passez les données sur la tâche ici */)
53 {
54   struct job* new_job;
55   /* Alloue un nouvel objet job. */
56   new_job = (struct job*) malloc (sizeof (struct job));
57   /* Positionner les autres champs de la struct job ici... */
58   /* Verrouille le mutex de la file de tâches avant d'y accéder. */
59   pthread_mutex_lock (&job_queue_mutex);
60   /* Ajoute la nouvelle tâche en tête de file. */
61   new_job->next = job_queue;
62   job_queue = new_job;
63   /* Envoie un signal de réveil sémaphore pour indiquer qu'une nouvelle tâche
64      est disponible. Si des threads sont bloqués en attente sur le sémaphore,
65      l'un d'eux sera débloqué et pourra donc traiter la tâche. */
66   sem_post (&job_queue_count);
67   /* Déverrouille le mutex de la file de tâches. */
68   pthread_mutex_unlock (&job_queue_mutex);
69 }
```

Avant de prélever une tâche en tête de file, chaque thread se mettra en attente sur le

sémaphore. Si la valeur de celui-ci est zéro, indiquant que la file est vide, le thread sera tout simplement bloqué jusqu'à ce que le sémaphore devienne positif, indiquant que la tâche a été ajoutée à la file.

La fonction `enqueue_job` ajoute une tâche à la file. Comme `thread_function`, elle doit verrouiller le mutex de la file avant de la modifier. Après avoir ajouté la tâche à la file, elle envoie un signal de réveil au sémaphore, indiquant qu'une nouvelle tâche est disponible. Dans la version du Listing 4.12, les threads qui traitent les tâches ne se terminent jamais ; si aucune tâche n'est disponible pendant un certain temps, ils sont simplement bloqués dans `sem_wait`.

4.4.6 Variables de condition

Nous avons montré comment utiliser un mutex pour protéger une variable contre les accès simultanés de deux threads et comment utiliser les sémaphores pour implémenter un compteur partagé. Une *variable de condition* est un troisième dispositif de synchronisation que fournit GNU/Linux ; avec ce genre de mécanisme, vous pouvez implémenter des conditions d'exécution plus complexes pour le thread.

Supposons que vous écriviez une fonction de thread qui exécute une boucle infinie, accomplissant une tâche à chaque itération. La boucle du thread, cependant, a besoin d'être contrôlée par un indicateur : la boucle ne s'exécute que lorsqu'il est actif ; dans le cas contraire, la boucle est mise en pause.

Le Listing 4.13 montre comment vous pourriez l'implémenter au moyen d'une simple boucle. À chaque itération, la fonction de thread vérifie que l'indicateur est actif. Comme plusieurs threads accèdent à l'indicateur, il est protégé par un mutex. Cette implémentation peut être correcte mais n'est pas efficace. La fonction de thread utilisera du temps processeur, que l'indicateur soit actif ou non, à vérifier et revérifier cet indicateur, verrouillant et déverrouillant le mutex à chaque fois. Ce dont vous avez réellement besoin, est un moyen de mettre le thread en pause lorsque l'indicateur n'est pas actif, jusqu'à ce qu'un certain changement survienne qui pourrait provoquer l'activation de l'indicateur.

Listing 4.13 – (*spin-condvar.c*) – **Implémentation Simple de Variable de Condition**

```
1  #include <pthread.h>
2  int thread_flag;
3  pthread_mutex_t thread_flag_mutex;
4  void initialize_flag ()
5  {
6    pthread_mutex_init (&thread_flag_mutex, NULL);
7    thread_flag = 0;
8  }
9  /* Appelle do_work de façon répétée tant que l'indicateur est actif ; sinon,
10    tourne dans la boucle. */
11 void* thread_function (void* thread_arg)
12 {
13   while (1) {
14     int flag_is_set;
15     /* Protège l'indicateur avec un mutex. */
16     pthread_mutex_lock (&thread_flag_mutex);
17     flag_is_set = thread_flag;
18     pthread_mutex_unlock (&thread_flag_mutex);
19     if (flag_is_set)
20       do_work ();
```

```
21      /* Rien à faire sinon, à part boucler.  */
22    }
23    return NULL;
24  }
25  /* Positionne la valeur de l'indicateur de thread à FLAG_VALUE.  */
26  void set_thread_flag (int flag_value)
27  {
28    /* Protège l'indicateur avec un verrouillage de mutex. */
29    pthread_mutex_lock (&thread_flag_mutex);
30    thread_flag = flag_value;
31    pthread_mutex_unlock (&thread_flag_mutex);
32  }
```

Une variable de condition vous permet de spécifier une condition qui lorsqu'elle est remplie autorise l'exécution du thread et inversement, une condition qui lorsqu'elle est remplie bloque le thread. Du moment que tous les threads susceptibles de modifier la condition utilisent la variable de condition correctement, Linux garantit que les threads bloqués à cause de la condition seront débloqués lorsque la condition change.

Comme pour les sémaphores, un thread peut se mettre en *attente* sur une variable de condition. Si le thread A est en attente sur une variable de condition, il est bloqué jusqu'à ce qu'un autre thread, le thread B, valide la même variable de condition. Contrairement à un sémaphore, une variable de condition ne dispose pas de compteur ou de mémoire ; le thread A doit se mettre en attente sur la variable de condition avant que le thread B ne la valide. Si le thread B valide la condition avant que le thread A ne fasse un wait dessus, la validation est perdue, et le thread A reste bloqué jusqu'à ce qu'un autre thread ne valide la variable de condition à son tour.

Voici comment vous utiliseriez une variable de condition pour rendre l'exemple précédent plus efficace :

- La boucle dans `thread_function` vérifie l'indicateur. S'il n'est pas actif, le thread se met en attente sur la variable de condition.
- La fonction `set_thread_flag` valide la variable de condition après avoir changé la valeur de l'indicateur. De cette façon, si `thread_function` est bloquée sur la variable de condition, elle sera débloquée et vérifiera la condition à nouveau.

Il y a un problème avec cette façon de faire : il y a une concurrence critique entre la vérification de la valeur de l'indicateur et la validation ou l'attente sur la variable de condition. Supposons que `thread_function` ait vérifié l'indicateur et ait déterminé qu'il n'était pas actif. À ce moment, l'ordonnancer de Linux suspend ce thread et relance le principal. Par hasard, le thread principal est dans `set_thread_flag`. Il active l'indicateur puis valide la variable de condition. Comme aucun thread n'est en attente sur la variable de condition à ce moment (souvenez-vous que `thread_function` a été suspendue avant de se mettre en attente sur la variable de condition), la validation est perdue. Maintenant, lorsque Linux relance l'autre thread, il commence à attendre la variable de condition et peut être bloqué pour toujours.

Pour résoudre ce problème, nous avons besoin de verrouiller l'indicateur et la variable de condition avec un seul mutex. Heureusement, GNU/Linux dispose exactement de ce mécanisme. Chaque variable de condition doit être utilisée en conjugaison avec un mutex, pour éviter ce type de concurrence critique. Avec ce principe, la fonction de thread suit les étapes suivantes :

1. La boucle dans `thread_function` verrouille le mutex et lit la valeur de l'indicateur.

2. Si l'indicateur est actif, elle déverrouille le mutex et exécute la fonction de traitement.
3. Si l'indicateur n'est pas actif, elle déverrouille le mutex de façon atomique et se met en attente sur la variable de condition.

La fonctionnalité critique utilisée se trouve dans l'étape 3, GNU/Linux vous permet de déverrouiller le mutex et de vous mettre en attente sur la variable de condition de façon atomique, sans qu'un autre thread puisse intervenir. Cela élimine le risque qu'un autre thread ne change la valeur de l'indicateur et ne valide la variable de condition entre le test de l'indicateur et la mise en attente sur la variable de condition de `thread_function`.

Une variable de condition est représentée par une instance de `pthread_cond_t`. Souvenez-vous que chaque variable de condition doit être accompagnée d'un mutex. Voici les fonctions qui manipulent les variables de condition :

- `pthread_cond_init` initialise une variable de condition. Le premier argument est un pointeur vers une variable `pthead_cond_t`. Le second argument, un pointeur vers un objet d'attributs de variable de condition, est ignoré par GNU/Linux. Le mutex doit être initialisé à part, comme indiqué dans la Section 4.4.2, « Mutexes » ;
- `pthread_cond_signal` valide une variable de condition. Un seul des threads bloqués sur la variable de condition est débloqué. Si aucun thread n'est bloqué sur la variable de condition, le signal est ignoré. L'argument est un pointeur vers la variable `pthread_cond_t`.
  Un appel similaire, `pthread_cond_broadcast`, débloque *tous* les threads bloqués sur une variable de condition, au lieu d'un seul
- `pthread_cond_wait` bloque l'appelant jusqu'à ce que la variable de condition soit validée. L'argument est un pointeur vers la variable `pthread_cond_t`. Le second argument est un pointeur vers la variable `pthread_mutex_t`.
  Lorsque `pthread_cond_wait` est appelée, le mutex doit déjà être verrouillé par le thread appelant. Cette fonction déverrouille automatiquement le mutex et se met en attente sur la variable de condition. Lorsque la variable de condition est validée et que le thread appelant est débloqué, `pthread_cond_wait` réacquiert automatiquement un verrou sur le mutex.

Lorsque votre programme effectue une action qui pourrait modifier la condition que vous protégez avec la variable de condition, il doit suivre les étapes suivante (dans notre exemple, la condition est l'état de l'indicateur du thread, donc ces étapes doivent être suivies à chaque fois que l'indicateur est modifié) :

1. Verrouiller le mutex accompagnant la variable de condition.
2. Effectuer l'action qui pourrait modifier la condition (dans notre exemple, activer l'indicateur).
3. Valider ou effectuer un broadcast sur la variable de condition, selon le comportement désiré.
4. Déverrouiller le mutex accompagnant la variable de condition.

Le Listing 4.14 reprend l'exemple précédent, en utilisant une variable de condition pour protéger l'indicateur. Notez qu'au sein de `thread_function`, un verrou est posé sur le mutex avant de vérifier la valeur de `thread_flag`. Ce verrou est automatiquement libéré par `pthread_cond_wait` avant qu'il ne se bloque et automatiquement réacquis ensuite. Notez également que `set_thread_flag` verrouille le mutex avant de définir la valeur de `thread_flag` et de valider le mutex.

Listing 4.14 – (*condvar.c*) – **Contrôler un Thread avec une Variable de Condition**

```
#include <pthread.h>
int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;
void initialize_flag ()
{
  /* Initialise le mutex et la variable de condition. */
  pthread_mutex_init (&thread_flag_mutex, NULL);
  pthread_cond_init (&thread_flag_cv, NULL);
  /* Initialise la valeur de l'indicateur. */
  thread_flag = 0;
}
/* Appelle do_work de façon répétée tant que l'indicateur est actif ; bloque
   si l'indicateur n'est pas actif. */
void* thread_function (void* thread_arg)
{
  /* Boucle infinie. */
  while (1) {
    /* Verrouille le mutex avant d'accéder à la valeur de l'indicateur. */
    pthread_mutex_lock (&thread_flag_mutex);
    while (!thread_flag)
      /* L'indicateur est inactif. Attend la validation de la variable de
         condition, indiquant que la valeur de l'indicateur a changé. Lorsque
         la validation a lieu et que le thread se débloque, boucle et teste à
         nouveau l'indicateur. */
      pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
    /* Lorsque nous arrivons ici, nous savons que l'indicateur est actif.
       Déverrouille le mutex. */
    pthread_mutex_unlock (&thread_flag_mutex);
    /* Actions utiles. */
    do_work ();
  }
  return NULL;
}
/* Définit la valeur de l'indicateur à FLAG_VALUE.  */
void set_thread_flag (int flag_value)
{
  /* Verrouille le mutex avant d'accéder à la valeur de l'indicateur. */
  pthread_mutex_lock (&thread_flag_mutex);
  /* Définit la valeur de l'indicateur, puis valide la condition, si jamais
     thread_function est bloquée en attente de l'activation de l'indicateur.
     Cependant, thread_function ne peut pas réellement tester la valeur
     de l'indicateur tant que le mutex n'est pas déverrouillé. */
  thread_flag = flag_value;
  pthread_cond_signal (&thread_flag_cv);
  /* Déverrouille le mutex. */
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

La condition protégée par une variable de condition peut être d'une complexité quelconque. Cependant, avant d'effectuer une opération qui pourrait modifier la condition, un verrouillage du mutex doit être demandé, après quoi la variable de condition doit être validée.

Une variable de condition peut aussi être utilisée sans condition, simplement pour bloquer un thread jusqu'à ce qu'un autre thread le « réveille ». Un sémaphore peut également être utilisé pour ce faire. La principale différence est qu'un sémaphore se « souvient » de l'appel de réveil, même si aucun thread n'était bloqué en attente à ce moment, alors qu'une variable de condition ignore les appels de réveil, à moins qu'un thread ne soit bloqué en attente à ce moment. Qui plus

est, un sémaphore ne réactive qu'un thread par signal de réveil ; avec `pthread_cond_broadcast`, un nombre quelconque et inconnu de threads bloqués peuvent être relancés en une fois.

4.4.7 Interblocage avec deux threads ou plus

Des interblocages peuvent survenir lorsque deux threads (ou plus) sont bloqués, chacun attendant une validation de condition que seul l'autre peut effectuer. Par exemple, si le thread A est bloqué sur une variable de condition attendant que le thread B la valide et le thread B est bloqué sur une variable de condition attendant que le thread A la valide, un interblocage survient car aucun thread ne pourra jamais valider la variable attendue par l'autre. Vous devez être attentif afin d'éviter l'apparition de telles situations car elles sont assez difficiles à détecter.

Une situation courante menant à un interblocage survient lorsque plusieurs threads tentent de verrouiller le même ensemble d'objets. Par exemple, considérons un programme dans lequel deux threads différents, exécutant deux fonctions de thread distinctes, ont besoin de verrouiller les deux mêmes mutexes. Supposons que le thread A verrouille le mutex 1, puis le mutex 2 et que le thread B verrouille le mutex 2 avant le mutex 1. Avec un scénario d'ordonnancement pessimiste, Linux pourrait donner la main au thread A suffisamment longtemps pour qu'il verrouille le mutex 1 puis donne la main au thread B qui verrouille immédiatement le mutex 2. Désormais, aucun thread ne peut plus avancer cas chacun est bloqué sur un mutex que l'autre maintient verrouillé.

C'est un exemple de problème d'interblocage général qui peut impliquer non seulement des objets de synchronisation, comme les mutex, mais également d'autres ressources, comme des verrous sur des fichiers ou des périphériques. Le problème survient lorsque plusieurs threads tentent de verrouiller le même ensemble de ressources dans des ordres différents. La solution est de s'assurer que tous les threads qui verrouillent plus d'une ressource le font dans le même ordre.

4.5 Implémentation des threads sous GNU/Linux

L'implantation des threads POSIX sous GNU/Linux diffère de celle utilisée sur un certain nombre de systèmes de type UNIX sur un point important : sous GNU/Linux, les threads sont implémentés comme des processus. Lorsque vous appelez `pthread_create` pour créer un nouveau thread, Linux crée un nouveau processus qui exécute ce thread. Cependant, ce processus n'est pas identique à ceux créés au moyen de fork ; en particulier, il partage son espace d'adressage et ses ressources avec le processus original au lieu d'en recevoir des copies.

Le programme `thread-pid` du Listing 4.15 le démontre. Le programme crée un thread ; le thread original et le nouveau appellent tous deux la fonction getpid et affichent leurs identifiants de processus respectifs, puis bouclent indéfiniment.

Listing 4.15 – (*thread-pid.c*) – **Affiche les Identifiants de Processus des Threads**

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  void* thread_function (void* arg)
5  {
6    fprintf (stderr, "L'identifiant du thread fils est %d\n", (int) getpid ());
7    /* Boucle indéfiniment. */
8    while (1);
```

```
 9    return NULL;
10  }
11  int main ()
12  {
13    pthread_t thread;
14    fprintf(stderr, "L'identifiant du thread principal est %d\n",(int)getpid());
15    pthread_create (&thread, NULL, &thread_function, NULL);
16    /* Boucle indéfiniment. */
17    while (1);
18    return 0;
19  }
```

Lancez le programme en arrière-plan puis invoquez `ps x` pour afficher vos processus en cours d'exécution. N'oubliez pas de tuer le programme `thread-pid` ensuite – il consomme beaucoup de temps processeur pour rien. Voici à quoi la sortie pourrait ressembler :

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
  PID TTY      STAT   TIME COMMAND
14042 pts/9    S      0:00 bash
14608 pts/9    R      0:01 ./thread-pid
14609 pts/9   S 0:00 ./thread-pid
14610 pts/9   R 0:01 ./thread-pid
14611 pts/9   R 0:00 ps x
% kill 14608
[1]+ Terminated         ./thread-pid
```

Notifications de Contrôle des Tâches dans le Shell

Les lignes débutant par [1] viennent du shell. Lorsque vous exécutez un programme en arrière-plan, le shell lui assigne un numéro de tâche – dans ce cas, 1 – et affiche l'identifiant de processus du programme. Si une tâche en arrière-plan se termine, le shell le signale lorsque vous invoquez une nouvelle commande.

Remarquez qu'il y a trois processus exécutant le programme `thread-pid`. Le premier, avec le pid 14608, est le thread principal du programme ; le troisième, avec le pid 14610, est le thread que nous avons créé pour exécuter `thread_function`.

Qu'en est-il du second thread, avec le pid 14609 ? Il s'agit du « thread de gestion » (manager thread) qui fait partie de l'implémentation interne des threads sous GNU/Linux. Le thread de contrôle est créé la première fois qu'un programme appelle `pthread_create` pour créer un nouveau thread.

4.5.1 Gestion de signaux

Supposons qu'un programme multithreadé reçoive un signal. Dans quel thread est invoqué le gestionnaire de signal ? Le comportement de l'interaction entre les threads et les signaux varie

d'un type d'UNIX à l'autre. Sous GNU/Linux, ce comportement est dicté par le fait que les threads sont implémentés comme des processus.

Comme chaque thread est un processus distinct, et comme un signal est délivré à un processus particulier, il n'y a pas d'ambiguïté au niveau du thread qui va recevoir le signal. Typiquement, les signaux envoyés de l'extérieur du programme sont envoyés au processus correspondant au thread principal du programme. Par exemple, si un processus se divise et que le processus fils exécute un programme multithreadé, le processus père conservera l'identifiant de processus du thread principal du programme du processus fils et l'utilisera pour envoyer des signaux à son fils. Il s'agit généralement d'un bonne convention que vous devriez suivre lorsque vous envoyez des signaux à un programme multithreadé.

Notez que cet aspect de l'implémentation de pthreads sous GNU/Linux va à l'encontre du standard de threads POSIX. Ne vous reposez pas sur ce comportement au sein de programmes destinés à être portables.

Au sein d'un programme multithreadé, il est possible pour un thread d'envoyer un signal à un autre thread bien défini. Utilisez la fonction `pthread_kill` pour cela. Son premier paramètre est l'identifiant du thread et le second, le numéro du signal.

4.5.2 L'appel système *clone*

Bien que les threads GNU/Linux créés dans le même programme soient implémentés comme des processus séparés, ils partagent leur espace mémoire virtuel et leurs autres ressources. Un processus fils créé avec `fork`, cependant, copie ces objets. Comment est créé le premier type de processus ?

L'appel système `clone` de Linux est une forme hybride entre `fork` et `pthread_create` qui permet à l'appelant de spécifier les ressources partagées entre lui et le nouveau processus. `clone` nécessite également que vous spécifiiez la région mémoire utilisée pour la pile d'exécution du nouveau processus. Bien que nous mentionnions `clone` pour satisfaire la curiosité du lecteur, cet appel système ne doit pas être utilisé au sein de programmes. Utilisez `fork` pour créer de nouveau processus ou `pthread_create` pour créer des threads.

4.6 Comparaison processus/threads

Pour certains programmes tirant partie du parallélisme, le choix entre processus et threads peut être difficile. Voici quelques pistes pour vous aider à déterminer le modèle de parallélisme qui convient le mieux à votre programme :

- Tous les threads d'un programme doivent exécuter le même code. Un processus fils, au contraire, peut exécuter un programme différent en utilisant une fonction `exec`.
- Un thread peut endommager les données d'autres threads du même processus car les threads partagent le même espace mémoire et leurs ressources. Par exemple, une écriture sauvage en mémoire *via* un pointeur non initialisé au sein d'un thread peut corrompre la mémoire d'un autre thread.

  Un processus corrompu par contre, ne peut pas agir de cette façon car chaque processus dispose de sa propre copie de l'espace mémoire du programme.

- Copier le contenu de la mémoire pour un nouveau processus a un coût en performances par rapport à la création d'un nouveau thread. Cependant, la copie n'est effectuée que lorsque la mémoire est modifiée, donc ce coût est minime si le processus fils ne fait que lire la mémoire.
- Les threads devraient être utilisés pour les programmes qui ont besoin d'un parallélisme finement contrôlé. Par exemple, si un problème peut être décomposé en plusieurs tâches presque identiques, les threads peuvent être un bon choix. Les processus devraient être utilisés pour des programmes ayant besoin d'un parallélisme plus grossier.
- Le partage de données entre des threads est trivial car ceux-ci partagent le même espace mémoire (cependant, il faut faire très attention à éviter les conditions de concurrence critique, comme expliqué plus haut). Le partage de données entre des processus nécessite l'utilisation de mécanismes IPC, comme expliqué dans le Chapitre 5. Cela peut être plus complexe mais diminue les risques que les processus souffrent de bugs liés au parallélisme.

Chapitre 5

Communication interprocessus

Le Chapitre 3, « Processus » traitait de la création de processus et montrait comment il est possible d'obtenir le code de sortie d'un processus fils. Il s'agit de la forme la plus simple de communication entre deux processus, mais en aucun cas de la plus puissante. Les mécanismes présentés au Chapitre 3 ne fournissent aucun moyen au processus parent pour communiquer avec le fils excepté *via* les arguments de ligne de commande et les variables d'environnement, ni aucun moyen pour le processus fils de communiquer avec son père, excepté par le biais de son code de sortie. Aucun de ces mécanismes ne permet de communiquer avec le processus fils pendant son exécution, ni n'autorise une communication entre les processus en dehors de la relation père-fils.

Ce chapitre présente des moyens de communication interprocessus qui dépassent ces limitations. Nous présenterons différentes façons de communiquer entre père et fils, entre des processus « sans liens » et même entre des processus s'exécutant sur des machines distinctes.

La *communication interprocessus* (interprocess communication, IPC) consiste à transférer des données entre les processus. Par exemple, un navigateur Internet peut demander une page à un serveur, qui envoie alors les données HTML. Ce transfert utilise des sockets dans une connexion similaire à celle du téléphone. Dans un autre exemple, vous pourriez vouloir imprimer les noms des fichiers d'un répertoire en utilisant une commande du type `ls | lpr`. Le shell crée un processus `ls` et un processus `lpr` distincts et connecte les deux au moyen d'un *tube* (ou pipe) représenté par le symbole "`|`". Un tube permet une communication à sens unique entre deux processus. Le processus `ls` écrit les données dans le tube et le processus `lpr` les lit à partir du tube.

Dans ce chapitre, nous traiterons de cinq types de communication interprocessus :

- La mémoire partagée permet aux processus de communiquer simplement en lisant ou écrivant dans un emplacement mémoire prédéfini.
- La mémoire mappée est similaire à la mémoire partagée, excepté qu'elle est associée à un fichier.
- Les tubes permettent une communication séquentiel d'un processus à l'autre.
- Les files FIFO sont similaires aux tubes excepté que des processus sans lien peuvent communiquer car le tube reçoit un nom dans le système de fichiers.

- Les sockets permettent la communication entre des processus sans lien, pouvant se trouver sur des machines distinctes.

Ces types d'IPC diffèrent selon les critères suivants :

- Ils restreignent ou non la communication à des processus liés (processus ayant un ancêtre commun), à des processus partageant le même système de fichiers ou à tout ordinateur connecté à un réseau.
- Un processus communiquant n'est limité qu'à la lecture ou qu'à l'écriture de données.
- Le nombre de processus pouvant communiquer.
- Les processus qui communiquent sont synchronisés par l'IPC – par exemple, un processus lecteur s'interrompt jusqu'à ce qu'il y ait des données à lire.

Dans ce chapitre, nous ne traiteront pas des IPC ne permettant qu'une communication limitée à un certain nombre de fois, comme communiquer en utilisant la valeur de sortie du fils.

5.1 Mémoire partagée

Une des méthodes de communication interprocessus les plus simples est d'utiliser la mémoire partagée. La mémoire partagée permet à deux processus ou plus d'accéder à la même zone mémoire comme s'ils avaient appelé `malloc` et avaient obtenu des pointeurs vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, tous les autres processus voient la modification.

5.1.1 Communication locale rapide

La mémoire partagée est la forme de communication interprocessus la plus rapide car tous les processus partagent la même mémoire. L'accès à cette mémoire partagée est aussi rapide que l'accès à la mémoire non partagée du processus et ne nécessite pas d'appel système ni d'entrée dans le noyau. Elle évite également les copies de données inutiles.

Comme le noyau ne coordonne pas les accès à la mémoire partagée, vous devez mettre en place votre propre synchronisation. Par exemple, un processus ne doit pas effectuer de lecture avant que des données aient été écrites et deux processus ne doivent pas écrire au même emplacement en même temps. Une stratégie courante pour éviter ces conditions de concurrence est d'utiliser des sémaphores, ce dont nous parlerons dans la prochaine section. Nos programmes d'illustration, cependant, ne montrent qu'un seul processus accédant à la mémoire, afin de se concentrer sur les mécanismes de la mémoire partagée et éviter d'obscurcir le code avec la logique de synchronisation.

5.1.2 Le modèle mémoire

Pour utiliser un segment de mémoire partagée, un processus doit allouer le segment. Puis, chaque processus désirant accéder au segment doit l'attacher. Après avoir fini d'utiliser le segment, chaque processus le détache. À un moment ou à un autre, un processus doit libérer le segment.

La compréhension du modèle mémoire de Linux aide à expliquer le processus d'allocation et d'attachement. Sous Linux, la mémoire virtuelle de chaque processus est divisée en pages. Chaque

processus conserve une correspondance entre ses adresses mémoire et ces pages de mémoire virtuelle, qui contiennent réellement les données. Même si chaque processus dispose de ses propres adresses, plusieurs tables de correspondance peuvent pointer vers la même page, permettant le partage de mémoire. Les pages mémoires sont examinées de plus près dans la Section 8.8, « La famille *mlock*: verrouillage de la mémoire physique », du Chapitre 8, « Appels système Linux ».

L'allocation d'un nouveau segment de mémoire partagée provoque la création de nouvelles pages de mémoire virtuelle. Comme tous les processus désirent accéder au même segment de mémoire partagée, seul un processus doit allouer un nouveau segment de mémoire partagée. Allouer un segment existant ne crée pas de nouvelles pages mais renvoie l'identifiant des pages existantes. Pour qu'un processus puisse utiliser un segment de mémoire partagée, il doit l'attacher, ce qui ajoute les correspondances entre sa mémoire virtuelle et les pages partagées du segment. Lorsqu'il en a terminé avec le segment, ces correspondances sont supprimées. Lorsque plus aucun processus n'a besoin d'accéder à ces segments de mémoire partagée, un processus exactement doit libérer les pages de mémoire virtuelle.

Tous les segments de mémoire partagée sont alloués sous forme de multiples entiers de la *taille de page* du système, qui est le nombre d'octets dans une page mémoire. Sur les systèmes Linux, la taille de page est de 4 Ko mais vous devriez vous baser sur la valeur renvoyée par `getpagesize`.

5.1.3 Allocation

Un processus alloue un segment de mémoire partagée en utilisant `shmget` (« SHared Memory GET », obtention de mémoire partagée). Son premier paramètre est une clé entière qui indique le segment à créer. Des processus sans lien peuvent accéder au même segment partagé en spécifiant la même valeur de clé. Malheureusement, d'autres processus pourraient avoir choisi la même valeur de clé fixé, ce qui provoquerait un conflit. Utiliser la constante spéciale `IPC_PRIVATE` comme valeur de clé garantit qu'un nouveau segment mémoire est créé.

Le second paramètre indique le nombre d'octets du segment. Comme les segments sont alloués en utilisant des pages, le nombre d'octets effectivement alloués est arrondi au multiple de la taille de page supérieur.

Le troisième paramètre est un *ou* binaire entre des indicateurs décrivant les options demandées à `shmget`. Voici ces indicateurs :

- `IPC_CREAT` – Cet indicateur demande la création d'un nouveau segment. Cela permet la création d'un nouveau segment tout en spécifiant une valeur de clé.
- `IPC_EXCL` – Cet indicateur, toujours utilisé avec `IPC_CREAT`, provoque l'échec de `shmget` si la clé de segment spécifiée existe déjà. Donc, cela permet au processus appelant d'avoir un segment « exclusif ». Si cette option n'est pas précisée et que la clé d'un segment existant est utilisé, `shmget` renvoie le segment existant au lieu d'en créer un nouveau.
- Indicateurs de mode – Cette valeur est constituée de 9 bits indiquant les permissions du propriétaire, du groupe et des autres utilisateurs pour contrôler l'accès au segment. Les bits d'exécution sont ignorés. Une façon simple de spécifier les permissions est d'utiliser les constantes définies dans `<sys/stat.h>` et documentées dans la page de manuel de section 2

de stat[1]. Par exemple, `S_IRUSR` et `S_IWUSR` spécifient des permissions de lecture et écriture pour le propriétaire du segment de mémoire partagée et `S_IROTH` et `S_IWOTH` spécifient des permissions de lecture et écriture pour les autres utilisateurs.

L'appel suivant à `shmget` crée un nouveau segment de mémoire partagée (ou accède à un segment existant, si `shm_key` est déjà utilisé) qui peut être lu et écrit par son propriétaire mais pas par les autres utilisateurs.

```
int segment_id = shmget (shm_key, getpagesize (),
                         IPC_CREAT | S_IRUSR | S_IWUSR);
```

Si l'appel se passe bien, `shmget` renvoie un identifiant de segment. Si le segment de mémoire partagée existe déjà, les permissions d'accès sont vérifiées et le système s'assure que le segment n'est pas destiné à être détruit.

5.1.4 Attachement et détachement

Pour rendre le segment de mémoire partagée disponible, un procesus doit utiliser `shmat` (« SHared Memory ATtach », attachement de mémoire partagée) en lui passant l'identifiant du segment de mémoire partagée `SHMID` renvoyé par shmget. Le second argument est un pointeur qui indique où vous voulez que le segment soit mis en correspondance dans l'espace d'adressage de votre processus ; si vous passez `NULL`, Linux sélectionnera une adresse disponible. Le troisième argument est un indicateur, qui peut prendre une des valeurs suivantes :

- `SHM_RND` indique que l'adresse spécifiée par le second paramètre doit être arrondie à un multiple inférieur de la taille de page. Si vous n'utilisez pas cet indicateur, vous devez aligner le second argument de `shmat` sur un multiple de page vous-même.
- `SHM_RDONLY` indique que le segment sera uniquement lu, pas écrit.

Si l'appel se déroule correctement, il renvoie l'adresse du segment partagé attaché. Les processus fils créés par des appels à `fork` héritent des segments partagés attachés ; il peuvent les détacher s'ils le souhaitent.

Lorsque vous en avez fini avec un segment de mémoire partagée, le segment doit être détaché en utilisant `shmdt` (« SHared Memory DeTach », Détachement de Mémoire Partagée) et lui passant l'adresse renvoyée par `shmat`. Si le segment n'a pas été libéré et qu'il s'agissait du dernier processus l'utilisant, il est supprimé. Les appels à `exit` et toute fonction de la famille d'`exec` détachent automatiquement les segments.

5.1.5 Contrôler et libérer la mémoire partagée

L'appel `shmctl` (« SHared Memory ConTroL », contrôle de la mémoire partagée) renvoie des informations sur un segment de mémoire partagée et peut le modifier. Le premier paramètre est l'identifiant d'un segment de mémoire partagée.

Pour obtenir des informations sur un segment de mémoire partagée, passez `IPC_STAT` comme second argument et un pointeur vers une `struct shmid_ds`.

Pour supprimer un segment, passez `IPC_RMID` comme second argument et `NULL` comme troisième argument. Le segment est supprimé lorsque le dernier processus qui l'a attaché le détache.

[1] Ces bits de permissions sont les mêmes que ceux utilisés pour les fichiers. Ils sont décrits dans la Section 10.3, « Permissions du système de fichiers ».

Chaque segment de mémoire partagée devrait être explicitement libéré en utilisant `shmctl` lorsque vous en avez terminé avec lui, afin d'éviter de dépasser la limite du nombre total de segments de mémoire partagée définie par le système. L'invocation de `exit` et `exec` détache les segments mémoire mais ne les libère pas.

Consultez la page de manuel de `shmctl` pour une description des autres opérations que vous pouvez effectuer sur les segments de mémoire partagée.

5.1.6 Programme exemple

Le programme du Listing 5.1 illustre l'utilisation de la mémoire partagée.

Listing 5.1 – (*shm.c*) – **Utilisation de la Mémoire Partagée**

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;
  /* Alloue le segment de mémoire partagée. */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
                       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
  /* Attache le segment de mémoire partagée. */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("mémoire partagée attachée à l'adresse %p\n", shared_memory);
  /* Détermine la taille du segment. */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("taille du segment : %d\n", segment_size);
  /* Écrit une chaîne dans le segment de mémoire partagée. */
  sprintf (shared_memory, "Hello, world.");
  /* Détache le segment de mémoire partagée. */
  shmdt (shared_memory);
  /* Réattache le segment de mémoire partagée à une adresse différente. */
  shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("mémoire partagée réattachée à l'adresse %p\n", shared_memory);
  /* Affiche la chaîne de la mémoire partagée. */
  printf ("%s\n", shared_memory);
  /* Détache le segment de mémoire partagée. */
  shmdt (shared_memory);

  /* Libère le segment de mémoire partagée.  */
  shmctl (segment_id, IPC_RMID, 0);

  return 0;
}
```

5.1.7 Débogage

La commande `ipcs` donne des informations sur les possibilités de communication inter-processus, y compris les segments de mémoire partagée. Utilisez l'option `-m` pour obtenir des

informations sur la mémoire partagée. Par exemple, ce code illustre le fait qu'un segment de mémoire partagée, numéroté `1627649`, est utilisé :

```
% ipcs -m

------ Segments de mémoire partagée --------
clé        shmid     propriétaire perms     octets   nattch état
0x00000000 1627649   user      640       25600     0
```

Si ce segment de mémoire avait été oublié par erreur par un programme, vous pouvez utiliser la commande `ipcrm` pour le supprimer.

```
% ipcrm shm 1627649
```

5.1.8 Avantages et inconvénients

Les segments de mémoire partagée permettent une communication bidirectionnelle rapide entre n'importe quel nombre de processus. Chaque utilisateur peut à la fois lire et écrire, mais un programme doit définir et suivre un protocole pour éviter les conditions de concurrence critique comme écraser des informations avant qu'elles ne soient lues. Malheureusement, Linux ne garantit pas strictement l'accès exclusif même si vous créez un nouveau segment partagé avec `IPC_PRIVATE`.

De plus, lorsque plusieurs processus utilisent un segment de mémoire partagée, ils doivent s'arranger pour utiliser la même clé.

5.2 Sémaphores de processus

Nous l'avons évoqué dans la section précédente, les processus doivent se coordonner pour accéder à la mémoire partagée. Comme nous l'avons dit dans la Section 4.4.5, « Sémaphores pour les threads », du Chapitre 4, « Threads », les sémaphores sont des compteurs qui permettent la synchronisation de plusieurs threads. Linux fournit une implémentation alternative distincte de sémaphores qui peuvent être utilisés pour synchroniser les processus (appelés sémaphores de processus ou parfois sémaphores System V). Les sémaphores de processus sont instanciés, utilisés et libérés comme les segments de mémoire partagée. Bien qu'un seul sémaphore soit suffisant pour quasiment toutes les utilisations, les sémaphores de processus sont regroupés en ensembles.

Tout au long de cette section, nous présentons les appels système relatifs aux sémaphores de processus, en montrant comment implémenter des sémaphores binaires isolés les utilisant.

5.2.1 Instanciation et libération

Les appels `semget` et `semctl` instancient et libèrent des sémaphores, ils sont analogues à shmget et shmctl pour la mémoire partagée. Invoquez `semget` avec une clé correspondant à un ensemble de sémaphores, le nombre de sémaphores dans l'ensemble et des indicateurs de permissions, comme pour `shmget` ; la valeur de retour est un identifiant d'ensemble de sémaphores. Vous pouvez obtenir l'identifiant d'un ensemble de sémaphores existant en passant la bonne valeur de clé ; dans ce cas, le nombre de sémaphores peut être à zéro.

Les sémaphores continuent à exister même après que tous les processus les utilisant sont terminés. Le dernier processus à utiliser un ensemble de sémaphores doit le supprimer explicitement afin de s'assurer que le système d'exploitation ne tombe pas à court de sémaphores. Pour cela, invoquez `semctl` avec l'identifiant de l'ensemble de sémaphores, le nombre de sémaphores qu'il contient, `IPC_RMID` en troisième argument et n'importe quelle valeur d'`union semun` comme quatrième argument (qui est ignoré). L'identifiant d'utilisateur effectif du processus appelant doit correspondre à celui de l'instanciateur du sémaphore (ou l'appelant doit avoir les droits root). Contrairement aux segments de mémoire partagée, la suppression d'un jeu de sémaphores provoque sa libération immédiate par Linux.

Le Listing 5.2 présente les fonctions d'allocation et de libération d'un sémaphore binaire.

Listing 5.2 – (*sem_all_deall.c*) – **Allouer et Libérer un Sémaphore Binaire**

```
1  #include <sys/ipc.h>
2  #include <sys/sem.h>
3  #include <sys/types.h>
4
5  /* Nous devons définir l'union semun nous-mêmes. */
6
7  union semun {
8     int val;
9     struct semid_ds *buf;
10    unsigned short int *array;
11    struct seminfo *__buf;
12 };
13
14 /* Obtient l'identifiant d'un sémaphore binaire, l'alloue si nécessaire. */
15
16 int binary_semaphore_allocation (key_t key, int sem_flags)
17 {
18    return semget (key, 1, sem_flags);
19 }
20
21
22 /* Libère un sémaphore binaire. Tous les utilisateurs doivent avoir fini de
23     s'en servir. Renvoie -1 en cas d'échec. */
24
25 int binary_semaphore_deallocate (int semid)
26 {
27    union semun ignored_argument;
28    return semctl (semid, 1, IPC_RMID, ignored_argument);
29 }
```

5.2.2 Initialisation des sémaphores

L'instanciation et l'initialisation des sémaphores sont deux opérations distinctes. Pour initialiser un sémaphore, utilisez `semctl` en lui passant zéro comme second argument et `SETALL` comme troisième paramètre. Pour le quatrième argument, vous devez créer un objet `union semun` et faire pointer son champ `array` vers un tableau de valeurs `unsigned short`. Chaque valeur est utilisée pour initialiser un sémaphore dans l'ensemble.

Le Listing 5.3 présente une fonction initialisant un sémaphore binaire.

Listing 5.3 – (*sem_init.c*) – **Initialiser un Sémaphore Binaire**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Nous devons définir l'union semun nous-mêmes. */

union semun {
   int val;
   struct semid_ds *buf;
   unsigned short int *array;
   struct seminfo *__buf;
};

/* Initialise un sémaphore binaire avec une valeur de 1. */

int binary_semaphore_initialize (int semid)
{
   union semun argument;
   unsigned short values[1];
   values[0] = 1;
   argument.array = values;
   return semctl (semid, 0, SETALL, argument);
}
```

5.2.3 Opérations d'attente et de réveil

Chaque sémaphore contient une valeur positive ou nulle et supporte des opérations d'attente et de réveil. L'appel système semop implémente les deux opérations. Son premier paramètre est l'identifiant d'un ensemble de sémaphores. Son second paramètre est un tableau d'éléments `struct sembuf` qui définit les opérations que vous voulez accomplir. Le troisième paramètre est la taille de ce tableau.

Les champs de la `struct sembuf` sont les suivants :

- `sem_num` est le numéro du sémaphore dans l'ensemble sur lequel est effectuée l'opération.
- `sem_op` est un entier spécifiant l'opération à accomplir.
  Si `sem_op` est un entier positif, ce chiffre est ajouté à la valeur du sémaphore immédiatement. Si `sem_op` est un nombre négatif, la valeur absolue de ce chiffre est soustraite de la valeur du sémaphore. Si cela devait rendre la valeur du sémaphore négative, l'appel est bloquant jusqu'à ce que la valeur du sémaphore atteigne la valeur absolue de `sem_op` (par le biais d'incrémentations effectuées par d'autres processus).
  Si `sem_op` est à zéro, l'opération est bloquante jusqu'à ce que la valeur atteigne zéro.
- `sem_flg` est un indicateur. Positionnez-le à `IPC_NOWAIT` pour éviter que l'opération ne soit bloquante ; au lieu de cela, l'appel à semop échoue si elle devait l'être. Si vous le positionnez à `SEM_UNDO`, Linux annule automatiquement l'opération sur le sémaphore lorsque le processus se termine.

Le Listing 5.4 illustre les opérations d'attente et de réveil pour un sémaphore binaire.

Listing 5.4 – (*sem_pv.c*) – **Attente et Réveil pour un Sémaphore Binaire**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

```

```
5     /* Se met en attente sur un sémaphore binaire. Bloque jusqu'à ce que la
6        valeur du sémaphore soit positive, puis le décrémente d'une unité. */
7  int binary_semaphore_wait (int semid)
8  {
9    struct sembuf operations[1];
10   /* Utilise le premier (et unique) sémaphore.  */
11   operations[0].sem_num = 0;
12   /* Décrémente d'une unité. */
13   operations[0].sem_op = -1;
14   /* Autorise l'annulation. */
15   operations[0].sem_flg = SEM_UNDO;
16
17   return semop (semid, operations, 1);
18 }
19
20 /* Envoie un signal de réveil à un sémaphore binaire : incrémente sa valeur
21    d'une unité. Sort de la fonction immédiatement. */
22 int binary_semaphore_post (int semid)
23 {
24   struct sembuf operations[1];
25   /* Utilise le premier (et unique) sémaphore. */
26   operations[0].sem_num = 0;
27   /* Incrémente d'une unité. */
28   operations[0].sem_op = 1;
29   /* Autorise l'annulation. */
30   operations[0].sem_flg = SEM_UNDO;
31
32   return semop (semid, operations, 1);
33 }
```

Passer l'indicateur `SEM_UNDO` permet de traiter le problème de la fin d'un processus alors qu'il dispose de ressources allouées *via* un sémaphore. Lorsqu'un processus se termine, volontairement ou non, la valeur du sémaphore est automatiquement ajustée pour « annuler » les actions du processus sur le sémaphore. Par exemple, si un processus qui a décrémenté le sémaphore est tué, la valeur du sémaphore est incrémentée.

5.2.4 Débogage des sémaphores

Utilisez la commande `ipcs -s` pour afficher des informations sur les ensembles de sémaphores existants. Utilisez la commande `ipcrm sem` pour supprimer un ensemble de sémaphores depuis la ligne de commande. Par exemple, pour supprimer l'ensemble de sémaphores ayant l'identifiant `5790517`, utilisez cette commande :

```
% ipcrm sem 5790517
```

5.3 Mémoire mappée

La mémoire mappée permet à différents processus de communiquer *via* un fichier partagé. Bien que vous puissiez concevoir l'utilisation de mémoire mappée comme étant à celle d'un segment de mémoire partagée avec un nom, vous devez être conscient qu'il existe des différences techniques. La mémoire mappée peut être utilisée pour la communication interprocessus ou comme un moyen pratique d'accéder au contenu d'un fichier.

La mémoire mappée crée une correspondance entre un fichier et la mémoire d'un processus. Linux divise le fichier en fragments de la taille d'une page puis les copie dans des pages de mémoire virtuelle afin qu'elles puissent être disponibles au sein de l'espace d'adressage d'un processus. Donc le processus peut lire le contenu du fichier par le biais d'accès mémoire classiques. Cela permet un accès rapide aux fichiers.

Vous pouvez vous représenter la mémoire mappée comme l'allocation d'un tampon contenant la totalité d'un fichier, la lecture du fichier dans le tampon, puis (si le tampon est modifié) l'écriture de celui-ci dans le fichier. Linux gère les opérations de lecture et d'écriture à votre place.

Il existe d'autres utilisations des fichiers de mémoire mappée que la communication interprocessus. Quelques unes d'entre elles sont traitées dans la Section 5.3.5, « Autres utilisations de *mmap* ».

5.3.1 Mapper un fichier ordinaire

Pour mettre en correspondance un fichier ordinaire avec la mémoire d'un processus, utilisez l'appel `mmap` (« Memory MAPped », Mémoire mappée, prononcez « em-map »). Le premier argument est l'adresse à laquelle vous désirez que Linux mette le fichier en correspondance au sein de l'espace d'adressage de votre processus ; la valeur `NULL` permet à Linux de choisir une adresse de départ disponible. Le second argument est la longueur de l'espace de correspondance en octets. Le troisième argument définit la protection de l'intervalle d'adresses mis en correspondance. La protection consiste en un *ou* binaire entre `PROT_READ`, `PROT_WRITE` et `PROT_EXEC`, correspondant aux permissions de lecture, d'écriture et d'exécution, respectivement. Le quatrième argument est un drapeau spécifiant des options supplémentaires. Le cinquième argument est un descripteur de fichier pointant vers le fichier à mettre en correspondance, ouvert en lecture. Le dernier argument est le déplacement, par rapport au début du fichier, à partir duquel commencer la mise en correspondance. Vous pouvez mapper tout ou partie du fichier en mémoire en choisissant le déplacement et la longueur de façon appropriée.

Le drapeau est un *ou* binaire entre ces contraintes :

- `MAP_FIXED` – Si vous utilisez ce drapeau, Linux utilise l'adresse que vous demandez pour mapper le fichier plutôt que de la considérer comme une indication. Cette adresse doit être alignée sur une page.
- `MAP_PRIVATE` – Les écritures en mémoire ne doivent pas être répercutées sur le fichier mis en correspondance, mais sur une copie privée du fichier. Aucun autre processus ne voit ces écritures. Ce mode ne doit pas être utilisé avec `MAP_SHARED`.
- `MAP_SHARED` – Les écritures sont immédiatement répercutées sur le fichier mis en correspondance. Utilisez ce mode lorsque vous utilisez la mémoire mappée pour l'IPC. Ce mode ne doit pas être utilisé avec `MAP_PRIVATE`.

Si l'appel se déroule avec succès, la fonction renvoie un pointeur vers le début de la mémoire. S'il échoue, la fonction renvoie `MAP_FAILED`.

5.3.2 Programmes exemples

Examinons deux programmes pour illustrer l'utilisation des régions de mémoire mappée pour lire et écrire dans des fichiers. Le premier programme, le Listing 5.5, génère un nombre aléatoire et l'écrit dans un fichier mappé en mémoire. Le second programme, le Listing 5.6, lit le nombre, l'affiche et le remplace par le double de sa valeur. Tous deux prennent en argument de ligne de commande le nom du fichier à mettre en correspondance avec la mémoire.

Listing 5.5 – (*mmap-write.c*) – **Écrit un Nombre Aléatoire dans un Fichier Mappé**

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <fcntl.h>
4  #include <sys/mman.h>
5  #include <sys/stat.h>
6  #include <time.h>
7  #include <unistd.h>
8  #define FILE_LENGTH 0x100
9
10 /* Renvoie un nombre aléatoire compris dans l'intervalle [low,high].  */
11
12 int random_range (unsigned const low, unsigned const high)
13 {
14   unsigned const range = high - low + 1;
15   return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
16 }
17
18 int main (int argc, char* const argv[])
19 {
20   int fd;
21   void* file_memory;
22   /* Initialise le générateur de nombres aléatoires.  */
23   srand (time (NULL));
24
25   /* Prépare un fichier suffisamment long pour contenir le nombre. */
26   fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
27   lseek (fd, FILE_LENGTH+1, SEEK_SET);
28
29   write (fd, "", 1);
30   lseek (fd, 0, SEEK_SET);
31
32  /* Met en correspondance le fichier et la mémoire. */
33   file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
34   close (fd);
35   /* Ecrit un entier aléatoire dans la zone mise en correspondance. */
36   sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
37   /* Libère la mémoire (facultatif car le programme se termine). */
38   munmap (file_memory, FILE_LENGTH);
39   return 0;
40 }
```

Le programme `mmap-write` ouvre le fichier, le créant s'il n'existe pas. Le second argument de `open` indique que le fichier est ouvert en lecture et écriture. Comme nous ne connaissons pas la taille du fichier, nous utilisons `lseek` pour nous assurer qu'il est suffisamment grand pour stocker un entier puis nous nous replaçons au début du fichier.

Le programme met en correspondance le fichier et la mémoire puis ferme le fichier car il n'est plus utile. Il écrit ensuite un entier aléatoire dans la mémoire mappée, et donc dans le fichier, puis

libère la mémoire. L'appel `munmap` n'est pas nécessaire car Linux supprimerait automatiquement la mise en correspondance à la fin du programme.

Listing 5.6 – (*mmap-read.c*) – **Lit un Entier Depuis un Fichier mis en correspondance avec la Mémoire et le Multiplie par Deux**

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;

  /* Ouvre le fichier. */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Met en correspondance le fichier et la mémoire. */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                      MAP_SHARED, fd, 0);
  close (fd);

  /* Lit l'entier, l'affiche et le multiplie par deux. */
  sscanf (file_memory, "%d", &integer);
  printf ("valeur : %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Libère la mémoire (facultatif car le programme se termine). */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

Le programme `mmap-read` lit le nombre à partir du fichier puis y écrit son double. Tout d'abord, il ouvre le fichier et le met en correspondance en lecture/écriture. Comme nous pouvons supposer que le fichier est suffisamment grand pour stocker un entier non signé, nous n'avons pas besoin d'utiliser `lseek`, comme dans le programme précédent. Le programme lit la valeur à partir de la mémoire en utilisant `sscanf` puis formate et écrit le double de la valeur en utilisant `sprintf`.

Voici un exemple de l'exécution de ces programmes. Il utilise le fichier `/tmp/integer-file`.

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
valeur : 42
% cat /tmp/integer-file
84
```

Remarquez que le texte `42` a été écrit dans le fichier sur le disque sans jamais appeler `write` et à été lu par la suite sans appeler `read`. Notez que ces programmes de démonstration écrivent et lisent l'entier sous forme de chaîne (en utilisant `sprintf` et `sscanf`) dans un but d'exemple uniquement – il n'y a aucune raison pour que le contenu d'un fichier mis en correspondance avec

la mémoire soit au format texte. Vous pouvez lire et écrire de façon binaire dans un fichier mis en correspondance avec la mémoire.

5.3.3 Accès partagé à un fichier

Des processus distincts peuvent communiquer en utilisant des régions de la mémoire mises en correspondance avec le même fichier. Passez le drapeau `MAP_SHARED` afin que toute écriture dans une telle région soit immédiatement répercutée sur le disque et visible par les autres processus. Si vous n'indiquez pas ce drapeau, Linux pourrait placer les données écrites dans un tampon avant de les transférer dans le fichier.

Une alternative est de forcer Linux à intégrer les changements effectués en mémoire dans le fichier en appelant `msync`. Ses deux premiers paramètres définissent une région de la mémoire mise en correspondance avec un fichier, comme pour `munmap`. Le troisième paramètre peut prendre les valeurs suivantes :

- `MS_ASYNC` – La mise à jour est planifiée mais pas nécessairement exécutée avant la fin de la fonction.
- `MS_SYNC` – La mise à jour est immédiate ; l'appel à `msync` est bloquant jusqu'à ce qu'elle soit terminée. `MS_SYNC` et `MS_ASYNC` ne peuvent être utilisés simultanément.
- `MS_INVALIDATE` – Toutes les autres mises en correspondance avec le fichier sont invalidées afin de prendre en compte les modifications.

Par exemple, pour purger un fichier partagé mis en correspondance à l'adresse `mem_addr` et d'une longueur de `mem_length` octets, effectuez cet appel :

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

Comme pour les segments de mémoire partagée, les utilisateurs de régions de mémoire mises en correspondance avec un fichier doivent établir et suivre un protocole afin d'éviter les conditions de concurrence critique. Par exemple, un sémaphore peut être utilisé pour éviter que plus d'un processus n'accède à la région de la mémoire en même temps. Vous pouvez également utiliser `fcntl` pour placer un verrou en lecture ou en écriture sur le fichier, comme le décrit la Section 8.3, « *fcntl*: Verrous et opérations sur les fichiers », du Chapitre 8.

5.3.4 Mises en correspondance privées

Passer `MAP_PRIVATE` à `mmap` crée une région en mode copie à l'écriture. Toute écriture dans la région n'est répercutée que dans la mémoire du processus ; les autres processus qui utilisent une mise en correspondance sur le même fichier ne voient pas les modifications. Au lieu d'écrire directement sur une page partagée par tous les processus, le processus écrit sur une copie privée de la page. Toutes les lectures et écriture ultérieures du processus utilisent cette page.

5.3.5 Autres utilisations de *mmap*

L'appel `mmap` peut être utilisé dans d'autres buts que la communication interprocessus. Une utilisation courante est de le substituer à `read` et `write`. Par exemple, plutôt que de charger explicitement le contenu d'un fichier en mémoire, un programme peut mettre le fichier en correspondance avec la mémoire et l'analyser *via* des lectures en mémoire. Pour certains programmes,

cette façon de faire est plus pratique et peut également être plus rapide que des opérations d'entrées/sorties explicites sur le fichier.

Une technique puissante utilisée par certains programmes consiste à fabriquer des structures de données (des instances struct ordinaires, par exemple) dans un fichier mis en correspondance avec la mémoire. Lors d'une invocation ultérieure, le programme remet le fichier en correspondance avec la mémoire et les structures de données retrouvent leur état précédent. Notez cependant que les pointeurs de ces structures de données seront invalides à moins qu'ils ne pointent vers des adresses au sein de la même région de mémoire et que le fichier soit bien mis en correspondance à la même adresse mémoire qu'initialement.

Une autre technique utile est de mettre le fichier spécial `/dev/zero` en correspondance avec la mémoire. Ce fichier, décrit dans la Section 6.5.2, « */dev/zero* », du Chapitre 6, « Périphériques », se comporte comme s'il était un fichier de taille infinie rempli d'octets à zéro. Un programme ayant besoin d'une source d'octets à zéro peut appeler `mmap` pour le fichier `/dev/zero`. Les écritures sur `/dev/zero` sont ignorées, donc la mémoire mise en correspondance peut être utilisée pour n'importe quelle opération. Les distributeurs de mémoire personnalisés utilisent souvent `/dev/zero` pour obtenir des portions de mémoire préinitialisées.

5.4 Tubes

Un *tube* est un dispositif de communication qui permet une communication à sens unique. Les données écrites sur l'« extrémité d'écriture » du tube sont lues depuis l'« extrémité de lecture ». Les tubes sont des dispositifs séquentiels ; les données sont toujours lues dans l'ordre où elles ont été écrites. Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus père et fils.

Dans un shell, le symbole `|` crée un tube. Par exemple, cette commande provoque la création par le shell de deux processus fils, l'un pour `ls` et l'autre pour `less` :

```
% ls | less
```

Le shell crée également un tube connectant la sortie standard du processus `ls` avec l'entrée standard de `less`. Les noms des fichiers listés par `ls` sont envoyés à `less` dans le même ordre que s'ils étaient envoyés directement au terminal.

La capacité d'un tube est limitée. Si le processus écrivain écrit plus vite que la vitesse à laquelle le processus lecteur consomme les données, et si le tube ne peut pas contenir de données supplémentaires, le processus écrivain est bloqué jusqu'à ce qu'il y ait à nouveau de la place dans le tube. Si le lecteur essaie de lire mais qu'il n'y a plus de données disponibles, il est bloqué jusqu'à ce que ce ne soit plus le cas. Ainsi, le tube synchronise automatiquement les deux processus.

5.4.1 Créer des tubes

Pour créer un tube, appelez la fonction `pipe`. Passez lui un tableau de deux entiers. L'appel à `pipe` stocke le descripteur de fichier en lecture à l'indice zéro et le descripteur de fichier en écriture à l'indice un. Par exemple, examinons ce code :

```
int pipe_fds[2];
int read_fd;
```

```
int write_fd;

pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Les données écrites *via* le descripteur `write_fd` peuvent être relues *via* `read_fd`.

5.4.2 Communication entre processus père et fils

Un appel à `pipe` crée des descripteurs de fichiers qui ne sont valide qu'au sein du processus appelant et de ses fils. Les descripteurs de fichiers d'un processus ne peuvent être transmis à des processus qui ne lui sont pas liés ; cependant, lorsqu'un processus appelle `fork`, les descripteurs de fichiers sont copiés dans le nouveau processus. Ainsi, les tubes ne peuvent connecter que des processus liés.

Dans le programme du Listing 5.7, un `fork` crée un nouveau processus fils. Le fils hérite des descripteurs de fichiers du tube. Le père écrit une chaîne dans le tube et le fils la lit. Le programme exemple convertit ces descripteurs de fichiers en flux `FILE*` en utilisant `fdopen`. Comme nous utilisons des flux plutôt que des descripteurs de fichiers, nous pouvons utiliser des fonctions d'entrées/sorties de la bibliothèque standard du C de plus haut niveau, comme `printf` et `fgets`.

Listing 5.7 – (*pipe.c*) – **Utiliser un Tube pour Communiquer avec un Processus Fils**

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  /* Écrit COUNT fois  MESSAGE vers STREAM, avec une pause d'une seconde
6     entre chaque. */
7
8  void writer (const char* message, int count, FILE* stream)
9  {
10   for (; count > 0; --count) {
11     /* Écrit le message vers le flux et purge immédiatement. */
12     fprintf (stream, "%s\n", message);
13     fflush (stream);
14     /* S'arrête un instant. */
15     sleep (1);
16   }
17 }
18
19 /* Lit des chaînes aléatoires depuis le flux
20    aussi longtemps que possible.  */
21
22 void reader (FILE* stream)
23 {
24   char buffer[1024];
25   /* Lit jusqu'à ce que l'on atteigne la fin du flux. fgets lit jusqu'à
26      ce qu'une nouvelle ligne ou une fin de fichier survienne. */
27   while (!feof (stream)
28          && !ferror (stream)
29          && fgets (buffer, sizeof (buffer), stream) != NULL)
30     fputs (buffer, stdout);
31 }
32
33 int main ()
34 {
```

```
35    int fds[2];
36    pid_t pid;
37
38    /* Crée un tube. Les descripteurs de fichiers pour les deux bouts du tube
39       sont placés dans fds. */
40    pipe (fds);
41    /* Crée un processus fils. */
42    pid = fork ();
43    if (pid == (pid_t) 0) {
44      FILE* stream;
45      /* Nous sommes dans le processus fils. On ferme notre copie de
46         l'extrémité en écriture du descripteur de fichiers. */
47      close (fds[1]);
48      /* Convertit le descripteur de fichier de lecture en objet FILE
49         et lit à partir de celui-ci. */
50      stream = fdopen (fds[0], "r");
51      reader (stream);
52      close (fds[0]);
53    }
54    else {
55      /* Nous sommes dans le processus parent. */
56      FILE* stream;
57      /* Ferme notre copie de l'extrémité en lecture
58         du descripteur de fichier. */
59      close (fds[0]);
60      /* Convertit le descripteur de fichier d'écriture en objet FILE
61         et y écrit des données. */
62      stream = fdopen (fds[1], "w");
63      writer ("Coucou.", 5, stream);
64      close (fds[1]);
65    }
66
67    return 0;
68  }
```

Au début de la fonction `main`, `fds` est déclaré comme étant un tableau de deux entiers. L'appel à `pipe` crée un tube et place les descripteurs en lecture et en écriture dans ce tableau. Le programme crée alors un processus fils. Après avoir fermé l'extrémité en lecture du tube, le processus père commence à écrire des chaînes dans le tube. Après avoir fermé l'extrémité en écriture du tube, le processus fils lit les chaînes depuis le tube.

Notez qu'après l'écriture dans la fonction `writer`, le père purge le tube en appelant `fflush`. Dans le cas contraire, les données pourraient ne pas être envoyées dans le tube immédiatement.

Lorsque vous invoquez la commande `ls | less`, deux divisions de processus ont lieu : une pour le processus fils `ls` et une pour le processus fils `less`. Ces deux processus héritent des descripteurs de fichier du tube afin qu'il puissent communiquer en l'utilisant. Pour faire communiquer des processus sans lien, utilisez plutôt des FIFO, comme le décrit la Section 5.4.5, « FIFO ».

5.4.3 Rediriger les flux d'entrée, de sortie et d'erreur standards

Souvent, vous aurez besoin de créer un processus fils et de définir l'extrémité d'un tube comme son entrée ou sa sortie standard. Grâce à la fonction `dup2`, vous pouvez substituer un descripteur de fichier à un autre. Par exemple, pour rediriger l'entrée standard vers un descripteur de fichier `fd`, utilisez ce code :

```
dup2 (fd, STDIN_FILENO);
```

La constante symbolique `STDIN_FILENO` représente le descripteur de fichier de l'entrée standard, qui a la valeur `0`. L'appel ferme l'entrée standard puis la rouvre comme une copie de `fd` de façon à ce que les deux puissent être utilisés indifféremment. Les descripteurs de fichiers substitués partagent la même position dans le fichier et le même ensemble d'indicateurs de statut de fichier. Ainsi, les caractères lus à partir de `fd` ne sont pas relus à partir de l'entrée standard.

Le programme du Listing 5.8 utilise `dup2` pour envoyer des donnée d'un tube vers la commande `sort`[2]. Une fois le tube créé, le programme se divise. Le processus parent envoie quelques chaînes vers le tube. Le processus fils connecte le descripteur de fichier en lecture du tube à son entrée standard en utilisant `dup2`. Il exécute ensuite le programme `sort`.

Listing 5.8 – (*dup2.c*) – **Rediriger la Sortie d'un Tube avec *dup2***

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

int main ()
{
  int fds[2];
  pid_t pid;

  /* Crée un tube. Les descripteurs de fichiers des deux extrémités
     du tube sont placées dans fds. */
  pipe (fds);
  /* Crée un processus fils. */
  pid = fork ();
  if (pid == (pid_t) 0) {
    /* Nous sommes dans le processus fils. On ferme notre copie du
       descripteur de fichier en écriture. */
    close (fds[1]);
    /* Connexion de l'extrémité en lecture à l'entrée standard. */
    dup2 (fds[0], STDIN_FILENO);
    /* Remplace le processus fils par le programme "sort". */
    execlp ("sort", "sort", 0);
  }
  else {
    /* Processus père. */
    FILE* stream;
    /* Ferme notre copie de l'extrémité en lecture du descripteur. */
    close (fds[0]);
    /* Convertit le descripteur de fichier en écriture en objet FILE, et
       y écrit. */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "C'est un test.\n");
    fprintf (stream, "Coucou.\n");
    fprintf (stream, "Mon chien a des puces.\n");
    fprintf (stream, "Ce programme est excellent.\n");
    fprintf (stream, "Un poisson, deux poissons.\n");
    fflush (stream);
    close (fds[1]);
    /* Attend la fin du processus fils. */
    waitpid (pid, NULL, 0);
```

[2] `sort` lit des lignes de texte depuis l'entrée standard, les trie par ordre alphabétique et les affiche sur la sortie standard.

```
42    }
43
44    return 0;
45  }
```

5.4.4 *popen* et *pclose*

Une utilisation courante des tubes est d'envoyer ou de recevoir des données depuis un programme en cours d'exécution dans un sous-processus. Les fonctions `popen` et `pclose` facilitent cette pratique en éliminant le besoin d'appeler `pipe`, `fork`, `dup2`, `exec` et `fdopen`.

Comparez le Listing 5.9, qui utilise `popen` et `pclose`, à l'exemple précédent (Listing 5.8).

Listing 5.9 – (*popen.c*) – **Exemple d'Utilisation de popen**

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main ()
5  {
6    FILE* stream = popen ("sort", "w");
7    fprintf (stream, "C'est un test.\n");
8    fprintf (stream, "Coucou.\n");
9    fprintf (stream, "Mon chien a des puces.\n");
10   fprintf (stream, "Ce programme est excellent.\n");
11   fprintf (stream, "Un poisson, deux poissons.\n");
12   return pclose (stream);
13 }
```

L'appel à `popen` crée un processus fils exécutant la commande `sort`, ce qui remplace les appels à `pipe`, `fork`, `dup2` et `execlp`. Le second argument, `"w"`, indique que ce processus désire écrire au processus fils. La valeur de retour de popen est l'extrémité d'un tube ; l'autre extrémité est connectée à l'entrée standard du processus fils. Une fois que le processus père a terminé d'écrire, `pclose` ferme le flux du processus fils, attend la fin du processus et renvoie son code de retour.

Le premier argument de `popen` est exécuté comme s'il s'agissait d'une commande shell, dans un processus exécutant `/bin/sh`. Le shell recherche les programmes à exécuter en utilisant la variable d'environnement `PATH` de la façon habituelle. Si le deuxième argument est `"r"`, la fonction renvoie le flux de sortie standard du processus fils afin que le père puisse lire la sortie. Si le second argument est `"w"`, la fonction renvoie le flux d'entrée standard du processus fils afin que le père puisse envoyer des données. Si une erreur survient, `popen` renvoie un pointeur nul.

Appelez `pclose` pour fermer un flux renvoyer par `popen`. Après avoir fermé le flux indiqué, `pclose` attend la fin du processus fils.

5.4.5 FIFO

Une file *premier entré, premier sorti* (first-in, first-out, FIFO) est un tube qui dispose d'un nom dans le système de fichiers. Tout processus peut ouvrir ou fermer la FIFO ; les processus raccordés aux extrémités du tube n'ont pas à avoir de lien de parenté. Les FIFO sont également appelés canaux nommés.

Vous pouvez créer une FIFO via la commande `mkfifo`. Indiquez l'emplacement où elle doit être créée sur la ligne de commande. Par exemple, créez une FIFO dans `/tmp/fifo` en invoquant ces commandes :

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
prw-rw-rw-    1 samuel users 0 Jan 16 14:04 /tmp/fifo
```

Le premier caractère affiché par `ls` est `p` ce qui indique que le fichier est en fait une FIFO (canal nommé, *named pipe*). Dans une fenêtre, lisez des données depuis la FIFO en invoquant cette commande :

```
% cat < /tmp/fifo
```

Dans une deuxième fenêtre, écrivez dans la FIFO en invoquant cela :

```
% cat > /tmp/fifo
```

Puis, saisissez du texte. À chaque fois que vous appuyez sur Entrée, la ligne de texte est envoyé dans la FIFO et apparaît dans la première fenêtre. Fermez la FIFO en appuyant sur `Ctrl+D` dans la seconde fenêtre. Supprimez la FIFO avec cette commande :

```
% rm /tmp/fifo
```

Créer une FIFO

Pour créer une FIFO par programmation, utilisez la fonction `mkfifo`. Le premier argument est l'emplacement où créer la FIFO ; le second paramètre spécifie les permissions du propriétaire du tube, de son groupe et des autres utilisateurs, comme le décrit le Chapitre 10, Chapitre 10, Section 10.3, « Permissions du système de fichiers ». Comme un tube doit avoir un lecteur et un écrivain, les permissions doivent comprendre des autorisations en lecture et en écriture. Si le tube ne peut pas être créé (par exemple, si un fichier possédant le même nom existe déjà), `mkfifo` renvoie `-1`. Incluez `<sys/types.h>` et `<sys/stat.h>` si vous appelez `mkfifo`.

Accéder à une FIFO

L'accès à une FIFO se fait de la même façon que pour un fichier ordinaire. Pour communiquer *via* une FIFO, un programme doit l'ouvrir en écriture. Il est possible d'utiliser des fonction d'E/S de bas niveau (`open`, `write`, `read`, `close`, *etc.* listées dans l'Annexe B, « E/S de bas niveau ») ou des fonctions d'E/S de la bibliothèque C (`fopen`, `fprintf`, `fscanf`, `fclose`, *etc.*).

Par exemple, pour écrire un tampon de données dans une FIFO en utilisant des routines de bas niveau, vous pourriez procéder comme suit :

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

Pour lire une chaîne depuis la FIFO en utilisant les fonctions d'E/S de la bibliothèque C, vous pourriez utiliser ce code :

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

Une FIFO peut avoir plusieurs lecteurs ou plusieurs écrivains. Les octets de chaque écrivain sont écrits de façon atomique pour une taille inférieure à `PIPE_BUF` (4Ko sous Linux). Les paquets d'écrivains en accès simultanés peuvent être entrelacés. Les mêmes règles s'appliquent à des lectures concurrentes.

Différences avec les canaux nommés Windows

Les tubes des systèmes d'exploitation Win32 sont très similaires aux tubes Linux (consultez la documentation de la bibliothèque Win32 pour plus de détails techniques). Les principales différences concernent les canaux nommés, qui, sous Win32, fonctionnent plus comme des sockets. Les canaux nommés Win32 peuvent connecter des processus d'ordinateurs distincts connectés *via* un réseau. Sous Linux, ce sont les sockets qui sont utilisés pour ce faire. De plus, Win32 permet de multiples connexions de lecteur à écrivain sur un même canal nommé sans que les données ne soient entrelacées et les tubes peuvent être utilisés pour une communication à double sens[3].

5.5 Sockets

Un *socket* est un dispositif de communication bidirectionnel pouvant être utilisé pour communiquer avec un autre processus sur la même machine ou avec un processus s'exécutant sur d'autres machines. Les sockets sont la seule forme de communication interprocessus dont nous traiterons dans ce chapitre qui permet la communication entre processus de différentes machines. Les programmes Internet comme Telnet, rlogin, FTP, talk et le World Wide Web utilisent des sockets.

Par exemple, vous pouvez obtenir une page depuis un serveur Web en utilisant le programme Telnet car tous deux utilisent des sockets pour la communication *via* le réseau[4]. Pour ouvrir une connexion vers un serveur Web dont l'adresse est `www.codesourcery.com`, utilisez la commande `telnet www.codesourcery.com 80`. La constante magique `80` demande une connexion au serveur Web de `www.codesourcery.com` plutôt qu'à un autre processus. Essayez d'entrer `GET /` une fois la connexion établie. Cela envoie un message au serveur Web à travers le socket, celui-ci répond en envoyant la source HTML de la page d'accueil puis en fermant la connexion – par exemple :

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is "^]".
GET /
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

[3]Notez que seul Windows NT permet la création de canaux nommés ; les programmes pour Windows 9x ne peuvent créer que des connexions client.

[4]Habituellement, vous utilisez `telnet` pour vous connecter à un serveur Telnet pour une identification à distance. Mais vous pouvez également utiliser `telnet` pour vous connecter à un autre type de serveur et lui envoyer des commandes directement.

5.5.1 Concepts relatifs aux sockets

Lorsque vous créez un socket, vous devez indiquer trois paramètres : le style de communication, l'espace de nommage et le protocole.

Un style de communication contrôle la façon dont le socket traite les données transmises et définit le nombre d'interlocuteurs. Lorsque des données sont envoyées *via* le socket, elles sont découpées en morceaux appelés *paquets*. Le style de communication détermine comment sont gérés ces paquets et comment ils sont envoyés de l'émetteur vers le destinataire.

- Le style *connexion* garantit la remise de tous les paquets dans leur ordre d'émission. Si des paquets sont perdus ou mélangés à cause de problèmes dans le réseau, le destinataire demande automatiquement leur retransmission à l'émetteur.
  Un socket de type connexion ressemble à un appel téléphonique : les adresses de l'émetteur et du destinataire sont fixées au début de la communication lorsque la connexion est établie.
- Le style *datagramme* ne garantit pas la remise ou l'ordre d'arrivée des paquets. Des paquets peuvent être perdus ou mélangés à cause de problèmes dans le réseau. Chaque paquet doit être associé à sa destination et il n'y a aucune garantie quant à sa remise. Le système ne garantit que le « meilleur effort » (best effort), des paquets peuvent donc être perdus ou être remis dans un ordre différent de leur émission.
  Un socket de style datagramme se comporte plus comme une lettre postale. L'émetteur spécifie l'adresse du récepteur pour chaque message.

L'espace de nommage d'un socket spécifie comment les *adresses de socket* sont écrites. Une adresse de socket identifie l'extrémité d'une connexion par socket. Par exemple, les adresses de socket dans « l'espace de nommage local » sont des noms de fichiers ordinaires. Dans « l'espace de nommage Internet », une adresse de socket est composée de l'adresse Internet (également appelée *adresse IP*) d'un hôte connecté au réseau et d'un numéro de port. Le numéro de port permet de faire la distinction entre plusieurs sockets sur le même hôte.

Un protocole spécifie comment les données sont transmises. Parmi ces protocoles, on peut citer TCP/IP ; les deux protocoles principaux utilisés pour Internet, le protocole réseau AppleTalk ; et le protocole de communication locale d'UNIX. Toutes les combinaisons de styles, d'espace de nommage et de protocoles ne sont pas supportées.

5.5.2 Appels système

Les sockets sont plus flexibles que les techniques de communication citées précédemment. Voici les appels systèmes utiles lors de l'utilisation de sockets :

- `socket` – Crée un socket.
- `close` – Détruit un socket.
- `connect` – Crée une connexion entre deux sockets.
- `bind` – Associe un socket serveur à une adresse.
- `listen` – Configure un socket afin qu'il accepte les connexions.
- `accept` – Accepte une connexion et crée un nouveau socket pour celle-ci.

Les sockets sont représentés par des descripteurs de fichiers.

Créer et détruire des sockets

Les fonctions `socket` et `close` créent et détruisent des sockets, respectivement. Lorsque vous créez un socket, spécifiez ses trois propriétés : espace de nommage, style de communication et protocole. Pour le paramètre d'espace de nommage, utilisez les constantes commençant par `PF_` (abréviation de « protocol family », famille de protocoles). Par exemple, `PF_LOCAL` ou `PF_UNIX` spécifient l'espace de nommage local et `PF_INET` correspond à l'espace de nommage Internet. Pour le paramètre du style de communication, utilisez les constantes commençant par `SOCK_`. `SOCK_STREAM` demande un socket de style connexion, `SOCK_DGRAM` un socket de style datagramme.

Le troisième paramètre, le protocole, spécifie le mécanisme de bas niveau utilisé pour transmettre et recevoir des données. Chaque protocole est valide pour une combinaison d'espace de nommage et de type de communication particulière. Comme il y a souvent un protocole adapté pour chaque paire, indiquer `0` sélectionne généralement le bon protocole. Si l'appel à `socket` se déroule correctement, il renvoie un descripteur de fichier pour le socket. Vous pouvez lire ou écrire sur un socket en utilisant `read`, `write`, *etc.* comme avec les autres descripteurs de fichiers. Lorsque vous en avez fini avec le socket, appelez `close` pour le supprimer.

Appeler *connect*

Pour créer une connexion entre deux sockets, le client appelle `connect`, en lui passant l'adresse d'un socket serveur auquel se connecter. Un *client* est un processus initiant une connexion et un *serveur* est un processus en attente de connexions. Le client appelle connect pour initier une connexion depuis un socket local, dont le descripteur est passé en premier argument, vers le socket serveur spécifié par le deuxième argument. Le troisième argument est la longueur, en octets, de la structure d'adresse pointée par le second argument. Les formats d'adresse de sockets diffèrent selon l'espace de nommage du socket.

Envoyer des informations

Toutes les techniques valides pour écrire dans un descripteur de fichier le sont également pour écrire dans un socket. Reportez-vous à l'Annexe B pour une présentation des fonctions d'E/S de bas niveau Linux et de certains des problèmes ayant trait à leur utilisation. La fonction `send`, qui est spécifique aux descripteur de fichiers socket, est une alternative à `write` avec quelques choix supplémentaires ; reportez-vous à sa page de manuel pour plus d'informations.

5.5.3 Serveurs

Le cycle de vie d'un serveur consiste à créer un socket de type connexion, le lier à une adresse, appeler `listen` pour permettre au socket d'accepter des connexions, appeler `accept` régulièrement pour accepter les connexions entrantes, puis fermer le socket. Les données ne sont pas lues et écrites directement via le socket serveur ; au lieu de cela, chaque fois qu'un programme accepte une nouvelle connexion, Linux crée un socket séparé utilisé pour le transfert de données via cette connexion. Dans cette section, nous introduirons `bind`, `listen` et `accept`.

Une adresse doit être liée au socket serveur en utilisant `bind` afin que les clients puissent le trouver. Son premier argument est le descripteur de fichier du socket. Le second argument est

un pointeur vers une structure d'adresse de socket ; son format dépend de la famille d'adresse du socket. Le troisième argument est la longueur de la structure d'adresse en octets. Une fois qu'une adresse est liée à un socket de type connexion, il doit invoquer `listen` pour indiquer qu'il agit en tant que serveur. Son premier argument est le descripteur de fichier du socket. Le second spécifie combien de connexions peuvent être mise en file d'attente. Si la file est pleine, les connexions supplémentaires seront refusées. Cela ne limite pas le nombre total de connexions qu'un serveur peut gérer ; cela limite simplement le nombre de clients tentant de se connecter qui n'ont pas encore été acceptés.

Un serveur accepte une demande de connexion d'un client en invoquant `accept`. Son premier argument est le descripteur de fichier du socket. Le second pointe vers une structure d'adresse de socket, qui sera renseignée avec l'adresse de socket du client. Le troisième argument est la longueur, en octets, de la structure d'adresse de socket. Le serveur peut utiliser l'adresse du client pour déterminer s'il désire réellement communiquer avec le client. L'appel à `accept` crée un nouveau socket pour communiquer avec le client et renvoie le descripteur de fichier correspondant. Le socket serveur original continue à accepter de nouvelles connexions de clients. Pour lire des données depuis un socket sans le supprimer de la file d'attente, utilisez `recv`. Cette fonction prend les mêmes arguments que `read` ainsi qu'un argument `FLAGS` supplémentaire. Passer la valeur `MSG_PEEK` permet de lire les données sans les supprimer de la file d'attente.

5.5.4 Sockets locaux

Les sockets mettant en relation des processus situés sur le même ordinateur peuvent utiliser l'espace de nommage local représenté par les constantes `PF_LOCAL` et `PF_UNIX`. Ils sont appelés sockets locaux ou sockets de domaine UNIX. Leur adresse de socket, un nom de fichier, n'est utilisée que lors de la création de connexions.

Le nom du socket est spécifié dans une `struct sockaddr_un`. Vous devez positionner le champ `sun_family` à `AF_LOCAL`, qui représente un espace de nommage local. Le champ `sun_path` spécifie le nom de fichier à utiliser et peut faire au plus 108 octets de long. La longueur réelle de la `struct sockaddr_un` doit être calculée en utilisant la macro `SUN_LEN`. Tout nom de fichier peut être utilisé, mais le processus doit avoir des autorisations d'écriture sur le répertoire, qui permettent l'ajout de fichiers. Pour se connecter à un socket, un processus doit avoir des droits en lecture sur le fichier. Même si différents ordinateurs peuvent partager le même système de fichier, seuls des processus s'exécutant sur le même ordinateur peuvent communiquer via les sockets de l'espace de nommage local.

Le seul protocole permis pour l'espace de nommage local est `0`.

Comme il est stocké dans un système de fichiers, un socket local est affiché comme un fichier. Par exemple, remarquez le `s` du début :

```
% ls -l /tmp/socket
srwxrwx--x    1 user group 0 Nov 13 19:18 /tmp/socket
```

Appelez `unlink` pour supprimer un socket local lorsque vous ne l'utilisez plus.

5.5.5 Un exemple utilisant les sockets locaux

Nous allons illustrer l'utilisation des sockets au moyen de deux programmes. Le programme serveur, Listing 5.10, crée un socket dans l'espace de nommage local et attend des connexions. Lorsqu'il reçoit une connexion, il lit des messages au format texte depuis celle-ci et les affiche jusqu'à ce qu'elle soit fermée. Si l'un de ces messages est « `quit` », le programme serveur supprime le socket et se termine. Le programme `socket-server` prend en argument de ligne de commande le chemin vers le socket.

Listing 5.10 – (*socket-server.c*) – **Serveur Utilisant un Socket Local**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Lit du texte depuis le socket et l'affiche. Continue jusqu'à ce que
   le socket soit fermé. Renvoie une valeur différente de zéro si le client
   a envoyé un message "quit", zéro sinon. */

int server (int client_socket)
{
  while (1) {
    int length;
    char* text;

    /* Commence par lire la longueur du message texte depuis le socket.
       Si read renvoie zéro, le client a fermé la connexion. */
    if (read (client_socket, &length, sizeof (length)) == 0)
      return 0;
    /* Alloue un tampon pour contenir le texte. */
    text = (char*) malloc (length);

    /* Lit le texte et l'affiche. */
    read (client_socket, text, length);
    printf ("%s\n", text);
    /* Si le client a envoyé le message "quit", c'est fini.  */
    if (!strcmp (text, "quit")) {
      /* Libère le tampon. */
      free (text);
      return 1;
    }
    /* Libère le tampon. */
    free (text);
  }
}

int main (int argc, char* const argv[])
{
const char* const socket_name = argv[1];
int socket_fd;
struct sockaddr_un name;
int client_sent_quit_message;

/* Crée le socket. */
socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
/* Indique qu'il s'agit d'un serveur. */
name.sun_family = AF_LOCAL;
```

```
50  strcpy (name.sun_path, socket_name);
51  bind (socket_fd, (struct sockaddr *) &name, SUN_LEN (&name));
52  /* Se met en attente de connexions. */
53  listen (socket_fd, 5);
54
55  /* Accepte les connexions de façon répétée, lance un server() pour traiter
56     chaque client. Continue jusqu'à ce qu'un client
57      envoie un message "quit". */
58    do {
59      struct sockaddr_un client_name;
60      socklen_t client_name_len;
61      int client_socket_fd;
62
63      /* Accepte une connexion. */
64      client_socket_fd =
65        accept (socket_fd, (struct sockaddr *) &client_name, &client_name_len);
66      /* Traite la connexion. */
67      client_sent_quit_message = server (client_socket_fd);
68      /* Ferme notre extrémité. */
69      close (client_socket_fd);
70    }
71    while (!client_sent_quit_message);
72
73    /* Supprime le fichier socket. */
74    close (socket_fd);
75    unlink (socket_name);
76
77    return 0;
78  }
```

Le programme client, Listing 5.11, se connecte à un socket local et envoie un message. Le chemin vers le socket et le message sont indiqués sur la ligne de commande.

Listing 5.11 – (*socket-client.c*) – **Client Utilisant un Socket Local**

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <sys/socket.h>
4   #include <sys/un.h>
5   #include <unistd.h>
6
7   /* Écrit TEXT vers le socket indiqué par le descripteur SOCKET_FD. */
8
9   void write_text (int socket_fd, const char* text)
10  {
11    /* Écrit le nombre d'octets de la chaîne, y compris l'octet
12       nul de fin. */
13    int length = strlen (text) + 1;
14    write (socket_fd, &length, sizeof (length));
15    /* Écrit la chaîne. */
16    write (socket_fd, text, length);
17  }
18
19  int main (int argc, char* const argv[])
20  {
21    const char* const socket_name = argv[1];
22    const char* const message = argv[2];
23    int socket_fd;
24    struct sockaddr_un name;
25
26    /* Crée le socket. */
```

```
27    socket_fd = socket (PF_LOCAL, SOCK_STREAM, 0);
28    /* Stocke le nom du serveur dans l'adresse du socket. */
29    name.sun_family = AF_LOCAL;
30    strcpy (name.sun_path, socket_name);
31    /* Se connecte au socket. */
32    connect (socket_fd, (struct sockaddr*) &name, SUN_LEN (&name));
33    /* écrit le texte de la ligne de commande vers le socket. */
34    write_text (socket_fd, message);
35    close (socket_fd);
36    return 0;
37  }
```

Avant que le client n'envoie le message, il envoie sa longueur contenue dans la variable entière `length`. De même, le serveur lit la longueur du texte en lisant une variable entière depuis le socket. Cela permet au serveur d'allouer un tampon de taille adéquate pour contenir le message avant de le lire.

Pour tester cet exemple, démarrez le programme serveur dans une fenêtre. Indiquez le chemin vers le socket – par exemple, `/tmp/socket`.

```
% ./socket-server /tmp/socket
```

Dans une autre fenêtre, lancez plusieurs fois le client en spécifiant le même socket ainsi que des messages à envoyer au client :

```
% ./socket-client /tmp/socket "Coucou."
% ./socket-client /tmp/socket "Ceci est un test."
```

Le programme serveur reçoit et affiche les messages. Pour fermer le serveur, envoyez le message « `quit` » depuis un client :

```
% ./socket-client /tmp/socket "quit"
```

Le programme serveur se termine alors.

5.5.6 Sockets internet

Les sockets de domaine UNIX ne peuvent être utilisés que pour communiquer entre deux processus s'exécutant sur le même ordinateur. Les *sockets Internet*, par contre, peuvent être utilisés pour connecter des processus s'exécutant sur des machines distinctes connectées par un réseau.

Les sockets connectant des processus *via* Internet utilisent l'espace de nommage Internet représenté par `PF_INET`. Le protocole le plus courant est TCP/IP. L'*Internet Protocol* (IP), un protocole de bas niveau transporte des paquets sur Internet, en les fragmentant et les réassemblant si nécessaire. Il ne garantit que le « meilleur effort » de remise, des paquets peuvent donc disparaître ou être mélangés durant le transport. Tous les ordinateurs participants sont définis en utilisant une adresse IP unique. Le *Transmission Control Protocol* (TCP), qui s'appuie sur IP, fournit un transport fiable orienté connexion. Il permet d'établir des connexions semblables aux connexions téléphoniques entre des ordinateurs et assure que les données sont remises de façon fiable et dans l'ordre.

Noms DNS

Comme il est plus facile de se souvenir de noms que de numéros, le *Domain Name Service* (DNS, Service de Nom de Domaine) associe un nom comme `www.codesourcery.com` à l'adresse IP d'un ordinateur. Le DNS est implémenté par une hiérarchie mondiale de serveurs de noms, mais vous n'avez pas besoin de comprendre les protocoles employés par le DNS pour utiliser des noms d'hôtes Internet dans vos programmes.

Les adresses de sockets Internet comprennent deux parties : un numéro de machine et un numéro de port. Ces informations sont stockées dans une variable de type `struct sockaddr_in`. Positionnez le champ `sin_family` à `AF_INET` pour indiquer qu'il s'agit d'une adresse de l'espace de nommage Internet. Le champ `sin_addr` stocke l'adresse Internet de la machine cible sous forme d'une adresse IP entière sur 32 bits. Un *numéro de port* permet de faire la distinction entre plusieurs sockets d'une machine donnée. Comme des machines distinctes peuvent stocker les octets de valeurs multioctets dans des ordres différents, utilisez `htons` pour représenter le numéro de port dans l'*ordre des octets défini par le réseau*. Consultez la page de manuel de `ip` pour plus d'informations.

Pour convertir des noms d'hôtes, des adresses en notation pointée (comme 10.0.0.1) ou des noms DNS (comme `www.codesourcery.com`) adresse IP sur 32 bits, vous pouvez utiliser `gethostbyname`. Cette fonction renvoie un pointeur vers une structure `struct hostent` ; le champ `h_addr` contient l'IP de l'hôte. Reportez-vous au programme exemple du Listing 5.12.

Le Listing 5.12 illustre l'utilisation de sockets de domaine Internet. Le programme rapatrie la page d'accueil du serveur Web dont le nom est passé sur la ligne de commande.

Listing 5.12 – (*socket-inet.c*) – **Lecture à partir d'un Serveur WWW**

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Affiche le contenu de la page d'accueil du serveur correspondant
   au socket. */

void get_home_page (int socket_fd)
{
  char buffer[10000];
  ssize_t number_characters_read;
  /* Envoie la commande HTTP GET pour la page d'accueil. */
  sprintf (buffer, "GET /\n");
  write (socket_fd, buffer, strlen (buffer));

  /* Lit à partir du socket. L'appel à read peut ne pas renvoyer toutes
  les données en une seule fois, on continue de lire jusqu'à ce qu'il
  n'y ait plus rien. */
  while (1) {
    number_characters_read = read (socket_fd, buffer, 10000);
    if (number_characters_read == 0)
```

```
26          return;
27        /* Ecrit les données vers la sortie standard. */
28        fwrite (buffer, sizeof (char), number_characters_read, stdout);
29      }
30    }
31
32    int main (int argc, char* const argv[])
33    {
34      int socket_fd;
35      struct sockaddr_in name;
36      struct hostent* hostinfo;
37
38      /* Crée le socket. */
39      socket_fd = socket (PF_INET, SOCK_STREAM, 0);
40      /* Place le nom du serveur dans l'adresse du socket. */
41      name.sin_family = AF_INET;
42      /* Convertit la chaîne en adresse IP sur 32 bits. */
43      hostinfo = gethostbyname (argv[1]);
44      if (hostinfo == NULL)
45        return 1;
46      else
47        name.sin_addr = *((struct in_addr *) hostinfo->h_addr);
48      /* Les serveurs Web utilisent le port 80. */
49      name.sin_port = htons (80);
50
51      /* Se connecte au serveur Web. */
52      if (connect (socket_fd, (struct sockaddr*) &name,
53                                      sizeof (struct sockaddr_in)) == -1) {
54        perror ("connect");
55        return 1;
56      }
57      /* Récupère la page d'accueil du serveur. */
58      get_home_page (socket_fd);
59      return 0;
60    }
```

Ce programme lit le nom du serveur Web à partir de la ligne de commande (pas l'URL – c'est-à-dire sans le « `http ://` »). Il appelle `gethostbyname` pour traduire le nom d'hôte en adresse IP numérique puis connecte un socket de type connexion (TCP) au port 80 de cet hôte. Les serveurs Web parlent l'*Hypertext Transport Protocol* (HTTP), donc le programme émet la commande HTTP `GET` et le serveur répond en envoyant le texte correspondant à la page d'accueil.

Par exemple, pour rapatrier la page d'accueil du site Web `www.codesourcery.com`, invoquez la commande suivante :

```
% ./socket-inet www.codesourcery.com
<html>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
...
```

5.5.7 Couples de sockets

Comme nous l'avons vu précédemment, la fonction `pipe` crée deux descripteurs de fichiers chacune étant l'extrémité d'un tube. Les tubes sont limités car les descripteurs de fichiers doivent être utilisés par des processus liés et car la communication est unidirectionnelle. La fonction `socketpair` crée deux descripteurs de fichiers pour deux sockets connectés sur le même ordinateur.

Numéros de Ports Standards

Par convention, les serveurs Web attendent des connexions sur le port 80. La plupart des services réseau Internet sont associés à un numéro de port standard. Par exemple, les serveurs Web sécurisés utilisant SSL attendent les connexions sur le port 443 et les serveurs de mail (qui parlent SMTP) utilisent le port 25.
Sur les systèmes GNU/Linux, les associations entre les noms des services, les protocoles et les numéros de port standards sont listés dans le fichier `/etc/services`. La première colonne est le nom du protocole ou du service. La seconde colonne indique le numéro de port et le type de connexion : `tcp` pour le mode orienté connexion et `udp` pour le mode datagramme.
Si vous implémentez des services réseau personnalisés utilisant des sockets Internet, utilisez des numéros de ports supérieurs à 1024.

Ces descripteurs de fichiers permettent une communication à double sens entre des processus liés. Ses trois premiers paramètres sont les mêmes que pour l'appel `socket` : ils indiquent le domaine, le type de connexion et le protocole. Le dernier paramètre est un tableau de deux entiers, qui sera renseigné avec les descripteurs de fichiers des deux sockets, comme pour `pipe`. Lorsque vous appelez `socketpair`, vous devez utiliser `PF_LOCAL` comme domaine.

Deuxième partie

Maîtriser Linux

Table des Matières

Chapitre 6

Périphériques

Linux, comme la plupart des systèmes d'exploitation, interagit avec les périphériques matériels *via* des composants logiciels modulaires appelés *pilotes de périphériques*. Un pilote masque les particularités des protocoles de communication utilisés par un dispositif matériel au système d'exploitation et lui permet d'interagir avec le périphérique par le biais d'une interface standardisée.

Sous Linux, les pilotes de périphériques font partie du noyau et peuvent être intégrés de façon statique à celui-ci ou chargés à la demande sous forme de modules. Les pilotes de périphériques s'exécutent comme s'ils faisaient partie du noyau et ne sont pas accessibles directement aux processus utilisateur. Cependant, Linux propose un mécanisme à ces processus pour communiquer avec un pilote – et par là même avec le dispositif matériel – *via* des objets semblables aux fichiers. Ces objets apparaissent dans le système de fichiers et des applications peuvent les ouvrir, les lire et y écrire pratiquement comme s'il s'agissait de fichiers normaux. Vos programmes peuvent donc communiquer avec des dispositifs matériels via des objets semblables aux fichiers soit en utilisant les opérations d'E/S de bas niveau de Linux (consultez l'Annexe B, « E/S de bas niveau »), soit les opérations de la bibliothèque d'E/S standard du C.

Linux fournit également plusieurs objets semblables à des fichiers qui communiquent directement avec le noyau plutôt qu'avec des pilotes de périphériques. Ils ne sont pas liés à des dispositifs matériels ; au lieu de cela, ils fournissent différents types de comportements spécialisés qui peuvent être utiles aux applications et aux programmes systèmes.

Soyez Prudent Lorsque Vous Accédez aux Périphériques !

Les techniques présentées dans ce chapitre fournissent un accès direct aux pilotes de périphériques s'exécutant au sein du noyau Linux, et à travers eux aux dispositifs matériels connectés au système. Utilisez ces techniques avec prudence car une mauvaise manipulation peut altérer ou endommager le système GNU/Linux.
Lisez notamment le cadre « Danger des Périphériques Blocs ».

6.1 Types de périphériques

Les fichiers de périphériques ne sont pas des fichiers ordinaires – ils ne représentent pas des zones de données au sein d'un système de fichiers sur disque. Au lieu de cela, les données lues ou écrites sur un fichier de périphérique sont transmises au pilote de périphérique correspondant et, par son intermédiaire, au matériel sous-jacent. Les fichiers de périphériques se divisent en deux types :

- Un *périphérique caractère* représente un dispositif matériel qui lit ou écrit en série un flux d'octets. Les ports série et parallèle, les lecteurs de cassettes, les terminaux et les cartes son sont des exemples de périphériques caractères.
- Un *périphérique bloc* représente un dispositif matériel qui lit ou écrit des données sous forme de blocs de taille fixe. Contrairement aux périphériques caractère, un périphérique bloc fournit un accès direct aux données stockées sur le périphérique. Un lecteur de disque est un exemple de périphérique bloc.

Les programmes traditionnels n'utiliseront jamais de périphériques blocs. Bien qu'un lecteur de disque soit représenté comme un périphérique matériel, le contenu de chaque partition contient habituellement un système de fichiers monté sur l'arborescence racine de GNU/Linux. Seul le code du noyau qui implémente le système de fichiers a besoin d'accéder au périphérique bloc directement ; les programmes d'application accèdent au contenu du disque *via* des fichiers et des répertoires normaux.

Néanmoins, les applications utilisent quelquefois les périphériques caractère. Nous traiterons de plusieurs d'entre eux dans les sections suivantes.

Dangers des Périphériques Bloc

Les périphériques bloc offrent un accès direct aux données du lecteur de disque. Bien que la plupart des systèmes GNU/Linux soient configurés pour interdire aux processus non root d'accéder directement à ces périphériques, un processus root peut causer des dommages sévères en changeant le contenu du disque. En écrivant sur un périphérique bloc correspondant à un disque, un programme peut modifier ou détruire les informations de contrôle du système de fichier et même la table des partitions d'un disque et son secteur de démarrage, rendant ainsi le lecteur, ou même tout le système, inutilisable. Accédez toujours à ces périphériques avec la plus grande prudence.

6.2 Numéros de périphérique

Linux identifie les périphériques au moyen de deux nombres : le *numéro de périphérique majeur* et le *numéro de périphérique mineur*. Le numéro de périphérique majeur indique à quel pilote correspond le périphérique. Les correspondances entre les numéros de périphérique majeurs et les pilotes sont fixes et définies dans les sources du noyau Linux. Notez qu'un même numéro de périphérique majeur peut correspondre à deux pilotes différents, l'un étant un périphérique

caractère et l'autre un périphérique bloc. Les numéros de périphérique mineurs permettent de distinguer plusieurs périphériques ou composants contrôlés par le même pilote.

Par exemple, le périphérique de numéro majeur 3 correspond au contrôleur IDE primaire du système. Un contrôleur IDE peut être connecté à deux périphériques (lecteur de disque, cassette ou CD-ROM) ; le périphérique « maître » a le numéro mineur 0 et le périphérique « esclave » a le numéro mineur 64. les partitions du périphérique maître (s'il supporte les partitions) ont les numéros 1, 2, 3, etc. Les partitions du périphérique esclave sont représentées par les numéros de périphérique mineurs 65, 66, 67, *etc.*

Les numéros de périphérique majeurs sont répertoriés dans la documentation des sources du noyau Linux. Sur beaucoup de distributions GNU/Linux, ils sont décrits dans le fichier `/usr/src/linux/Documentation/devices.txt`. Le fichier spécial `/proc/devices` dresse la liste des numéros de périphérique majeurs correspondant aux pilotes de périphériques actuellement chargés dans le noyau (consultez le Chapitre 7, « Le système de fichiers */proc* » pour plus d'informations sur les entrées du système de fichiers `/proc`).

6.3 Fichiers de périphériques

Un fichier de périphérique ressemble beaucoup à un fichier classique. Vous pouvez le déplacer en utilisant la commande `mv` et le supprimer avec `rm`. Si vous essayez de copier un périphérique en utilisant `cp`, par contre, vous lirez des octets à partir de celui-ci (s'il le supporte) et les écrirez vers un fichier de destination. Si vous essayez d'écraser un fichier de périphérique, vous écrirez des octets vers le périphérique concerné.

Vous pouvez créer un fichier de périphérique en utilisant la commande `mknod` (saisissez `man 1 mknod` pour obtenir la page de manuel) ou l'appel système `mknod` (`man 2 mknod` pour la page de manuel). Créer un fichier de périphérique n'implique pas automatiquement que le pilote ou le dispositif matériel soit présent ou disponible ; le fichier de périphérique est en quelque sort un portail pour communiquer avec le pilote, s'il est présent. Seul les processus superutilisateur peuvent créer des périphériques bloc et caractère *via* la commande ou l'appel système `mknod`.

Pour créer un périphérique en utilisant la commande `mknod`, spécifiez le chemin du fichier le représentant comme premier argument de la ligne de commande. Pour le second argument, passez `b` pour un périphérique bloc ou `c` pour un périphérique caractère. Fournissez les numéros de périphérique majeur et mineur en troisième et quatrième argument, respectivement. Par exemple, la commande suivante crée un fichier de périphérique caractère appelé `lp0` dans le répertoire courant. Ce périphérique a le numéro de périphérique majeur 6 et le numéro mineur 0. Ces nombres correspondent au premier port parallèle sur le système Linux.

```
% mknod ./lp0 c 6 0
```

Souvenez-vous que seuls les processus du superutilisateur peuvent créer des périphériques bloc ou caractère, vous devez donc être connecté en tant que root pour invoquer cette commande avec succès.

La commande `ls` affiche les fichiers de périphérique d'une façon particulière. Si vous l'appelez avec les options `-l` ou `-o`, le premier caractère de chaque ligne indique le type du fichier. Rappelons que `-` (un tiret) indique un fichier classique, alors que `d` indique un répertoire.

De même, `b` désigne un périphérique bloc et `c` un périphérique caractère. Pour ces deux derniers, `ls` affiche les numéros de périphérique majeur et mineur là où se trouve habituellement la taille pour les fichiers ordinaires. Par exemple, nous pouvons afficher le périphérique caractère que nous venons juste de créer :

```
% ls -l lp0
crw-r-----  1 root root 6, 0 Mar 7 17:03 lp0
```

Dans un programme, vous pouvez déterminer si un fichier est un périphérique bloc ou caractère et donc obtenir ses numéros de périphérique via `stat`. Consultez la Section B.2, « *stat* », Annexe B, pour plus d'informations.

Pour supprimer le fichier, utilisez `rm`. Cela ne supprime pas le périphérique ou son pilote ; mais simplement le fichier de périphérique du système de fichiers.

```
% rm ./lp0
```

6.3.1 Le répertoire */dev*

Par convention, un système GNU/Linux inclut un répertoire `/dev` contenant tous les fichiers de périphériques caractère ou bloc des périphériques détectés. Les entrées de `/dev` ont des noms standardisés correspondants au numéros de périphérique majeur et mineur.

Par exemple, le périphérique maître connecté au contrôleur IDE primaire, qui dispose des numéros de périphérique majeur et mineur 3 et 0, a le nom standard `/dev/hda`. Si ce périphérique gère les partitions, la première, qui dispose du numéro de périphérique mineur 1, a le nom standard `/dev/hda1`. Vous pouvez le vérifier sur votre propre système :

```
% ls -l /dev/hda /dev/hda1
brw-rw----    1 root     disk 3, 0 May 5 1998 /dev/hda
brw-rw----    1 root     disk 3, 1 May 5 1998 /dev/hda1
```

De même, `/dev` contient une entrée pour le périphérique caractère qu'est le port parallèle que nous avons utilisé précédemment :

```
% ls -l /dev/lp0
crw-rw----    1 root daemon 6, 0 May 5 1998 /dev/lp0
```

Dans la plupart des cas, vous ne devriez pas utiliser `mknod` pour créer vos propres fichiers de périphérique. Utilisez plutôt les entrées de `/dev`. Les programmes ne disposant pas des privilèges superutilisateur n'ont pas d'autre choix que de les utiliser puisqu'ils ne peuvent pas créer leur propres entrées. Typiquement, seuls les administrateurs système et les développeurs utilisant des périphériques spécifiques ont besoin de créer leurs propres fichiers de périphérique. La plupart des distributions GNU/Linux proposent des utilitaires d'aide à la création de fichiers de périphérique standards avec les noms corrects.

6.3.2 Accéder à des périphériques en ouvrant des fichiers

Comment utiliser ces périphériques ? Dans le cas de périphériques caractère, cela peut être relativement simple : ouvrez le périphérique comme s'il s'agissait d'un fichier classique et lisez ou écrivez-y. Vous pouvez même utiliser des commandes conçues pour les fichiers traditionnels,

comme `cat` ou la syntaxe de redirection de votre shell pour envoyer ou lire des données à partir du périphérique.

Par exemple, si vous disposez d'une imprimante connectée sur le premier port parallèle de votre ordinateur, vous pouvez imprimer des fichiers en les envoyant directement sur `/dev/lp0`[1]. Pour imprimer le contenu de `document.txt`, invoquez la commande suivante :

```
% cat document.txt > /dev/lp0
```

Vous devez disposer des permissions en écriture sur le fichier de périphérique pour que la commande n'échoue pas ; sur beaucoup de systèmes GNU/Linux, les permissions sont définies de telle façon que seul root et le démon d'impression système (`lpd`) puissent écrire dans ce fichier. De plus, ce qui sort de votre imprimante dépend de la façon dont elle interprète les données que vous lui envoyez. Certaines imprimantes imprimeront les fichiers texte plats que vous leur enverrez[2], d'autres non. Les imprimantes PostScript interpréteront et imprimeront les fichiers PostScript que vous leur envoyez.

Dans un programme, envoyer des données à un périphérique est aussi simple. Par exemple, cet extrait de code utilise des fonctions d'E/S standard de bas niveau pour envoyer le contenu d'un tampon vers `/dev/lp0`.

```
int fd = open ("/dev/lp0", O_WRONLY);
write (fd, buffer, buffer_length);
close (fd);
```

6.4 Périphériques matériels

Quelques périphériques bloc standards sont listés dans le Tableau 6.1. Les numéros mineurs des périphériques similaires suivent un motif classique (par exemple, la seconde partition du premier périphérique SCSI est `/dev/sda2`). Il est parfois utile de savoir à quel périphérique correspondent les noms de périphériques lorsque l'on observe les systèmes de fichiers montés dans `/proc/mounts` (consultez la Section 7.5, « Lecteurs et systèmes de fichiers », du Chapitre 7, pour en savoir plus).

Le Tableau 6.2 liste quelques périphériques caractère courants.

Vous pouvez accéder à certains composants matériels *via* plus d'un périphérique caractère ; souvent, des périphériques caractère distincts ont une sémantique différente. Par exemple, lorsque vous utilisez le lecteur de cassettes IDE `/dev/ht0`, Linux rembobine automatiquement la cassette lorsque vous fermez le descripteur de fichier. Vous pouvez utiliser `/dev/nht0` pour accéder au même lecteur de cassettes, la seule différence est que Linux ne rembobine pas la cassette lors de la fermeture. Vous pourriez rencontrer des programmes utilisant `/dev/cua0` et des dispositifs similaires ; il s'agit d'anciennes interfaces vers les ports série comme `/dev/ttyS0`.

De temps à autre, vous pourriez avoir besoin d'écrire des données directement sur des périphériques caractère – par exemple :

[1] Les utilisateurs de Windows reconnaîtront là un périphérique similaire au fichier magique `LPT1` de Windows.

[2] Votre imprimante peut nécessiter l'ajout de retours chariot, code ASCII 13, à la fin de chaque ligne et l'ajout d'un caractère de saut de page, code ASCII 12, à la fin de chaque page.

TAB. 6.1 – Listing Partiel des Périphériques Bloc Courants

Périphérique	**Nom**	**N° Majeur**	**N° Mineur**
Premier lecteur de disquettes	`/dev/fd0`	2	0
Second lecteur de disquette	`/dev/fd1`	2	1
Contrôleur IDE primaire, maîtres	`/dev/hda`	3	0
Contrôleur IDE primaire, maître, première partition	`/dev/hda1`	3	1
Contrôleur IDE primaire, esclave	`/dev/hdb`	3	64
Contrôleur IDE primaire, esclave, première partition	`/dev/hdb1`	3	65
Premier lecteur SCSI	`/dev/sda`	8	0
Premier lecteur SCSI, première partition	`/dev/sda1`	8	1
Second disque SCSI	`/dev/sdb`	8	16
Second disque SCSI, première partition	`/dev/sdb1`	8	17
Premier lecteur CD-ROM SCSI	`/dev/scd0`	11	0
Second lecteur CD-ROM SCSI	`/dev/scd1`	11	1

TAB. 6.2 – Listing Partiel des Périphériques Caractère Courants

Périphérique	**Nom**	**N° Majeur**	**N° Mineur**
Port parallèle 0	`/dev/fd0` ou `/dev/par0`	6	0
Port parallèle 1	`/dev/lp1` ou `/dev/par1`	6	1
Premier port série	`/dev/ttyS0`	4	64
Second port série	`/dev/ttyS1`	4	65
Lecteur de cassettes IDE	`/dev/ht0`	37	0
Premier lecteur de cassettes SCSI	`/dev/st0`	9	0
Second lecteur de cassettes SCSI	`/dev/st1`	9	1
Console système	`/dev/console`	5	1
Premier terminal virtuel	`/dev/tty1`	4	1
Second terminal virtuel	`/dev/tty2`	4	2
Terminal du processus courant	`/dev/tty`	5	0
Carte son	`/dev/audio`	14	4

- Un programme terminal peut accéder à modem directement par le biais d'un port série. Les données écrites ou lues à partir de ces périphériques sont transmises par le modem à un ordinateur distant.
- Un programme de sauvegarde sur cassette peut écrire directement des données sur le périphérique de lecture de cassettes. Ce programme peut implémenter son propre format de compression et de vérification d'erreur.
- Un programme peut écrire directement sur le premier terminal virtuel[3] en écrivant sur `/dev/tty1`.
- Les fenêtres de terminal s'exécutant sous un environnement graphique ou les sessions distantes ne sont pas associées à des terminaux virtuels mais à des pseudos-terminaux. Consultez la Section 6.6, « PTY », pour plus d'informations.
- Parfois, un programme peut avoir besoin d'accéder au terminal auquel il est associé.
  Par exemple, votre application peut avoir besoin de demander un mot de passe à l'utilisateur. Pour des raisons de sécurité, vous ne voulez pas tenir compte des redirections d'entrée et de sortie standards et toujours lire le mot de passe à partir du terminal, peut importe la façon dont l'utilisateur appelle votre programme. Une façon de le faire est d'ouvrir `/dev/tty`, qui correspond toujours au terminal associé au processus effectuant l'ouverture. Écrivez l'invite de mot de passe sur ce périphérique et lisez le mot de passe. En ignorant l'entrée et la sortie standards, vous évitez que l'utilisateur n'alimente votre programme avec un mot de passe stocké dans un fichier avec une syntaxe comme celle-ci :

  ```
  % programme_sur < mon-motdepasse.txt
  ```

  Si vous avez besoin d'un mécanisme d'authentification dans votre programme, vous devriez vous tourner vers le dispositif PAM de GNU/Linux. Consultez la Section 10.5, « Authentifier les utilisateurs », du Chapitre 10, « Sécurité », pour plus d'informations.
- Un programme peut diffuser des sons via la carte son du système en envoyant des données audio vers `/dev/audio`. Notez que les données audio doivent être au format Sun (fichiers portant habituellement l'extension `.au`).
  Par exemple, beaucoup de distributions GNU/Linux fournissent le fichier son classique `/usr/share/sndconfig/sample.au`. Si votre système dispose de ce fichier, essayez de le jouer grâce à la commande suivante :

  ```
  % cat /usr/share/sndconfig/sample.au > /dev/audio
  ```

  Si vous devez utiliser des sons dans votre programme, cependant, vous devriez utiliser l'un des multiples bibliothèques et services de gestion de sons disponibles pour GNU/Linux. L'environnement de bureau Gnome utilise l'Enlightenment Sound Daemon (EsounD), disponible sur `http://www.tux.org/~ricdude/EsounD.html`. KDE utilise aRts, disponible sur `http://space.twc.de/~stefan/kde/arts-mcop-doc/`. Si vous utilisez l'un de ces systèmes de son au lieu d'écrire directement sur /dev/audio, votre programme pourra être utilisé plus facilement avec d'autres programmes utilisant la carte son de l'ordinateur.

[3] Sur la plupart des systèmes GNU/Linux, vous pouvez basculer vers le premier terminal en appuyant sur `Ctrl+Alt+F1`. Utilisez `Ctrl+Alt+F2` pour le second terminal virtuel, *etc.*

6.5 Périphériques spéciaux

Linux fournit également divers périphériques caractère ne correspondant à aucun périphérique matériel. Ces fichiers ont tous le numéro de périphérique majeur 1, qui est associé à la mémoire du noyau Linux et non à un pilote de périphérique.

6.5.1 */dev/null*

Le fichier `/dev/null`, le *périphérique nul*, est très pratique. Il a deux utilisations ; vous êtes probablement familiers avec la première :

- Linux ignore toute donnée écrite vers `/dev/null`. Une astuce souvent utilisée est de spécifier `/dev/null` en tant que fichier de sortie lorsque l'on ne veut pas de sortie. Par exemple, pour lancer une commande et ignorer son affichage standard (sans l'afficher ni l'envoyer vers un fichier), redirigez la sortie standard vers `/dev/null` :

  ```
  % commande_bavarde > /dev/null
  ```

- Lire depuis `/dev/null` renvoie toujours une fin de fichier. Par exemple, si vous ouvrez un descripteur de fichier correspondant à `/dev/null` en utilisant `open` puis essayez d'appeler read sur ce descripteur, aucun octet ne sera lu et `read` renverra 0. Si vous copiez `/dev/null` vers un autre fichier, la destination sera un fichier de taille nulle :

  ```
  % cp /dev/null fichier_vide
  % ls -l fichier_vide
  -rw-rw----    1 samuel   samuel 0 Mar 8 00:27 fichier_vide
  ```

6.5.2 */dev/zero*

Le fichier de périphérique `/dev/zero` se comporte comme s'il contenait une infinité d'octets à 0. Quelle que soit la quantité de données que vous essayez de lire à partir de `/dev/zero`, Linux générera suffisamment d'octets nuls.

Pour illustrer cela, exécutons le programme de capture en hexadécimal présenté dans le Listing B.4, Section B.1.4, « Lecture de données », de l'Annexe B. Ce programme affiche le contenu d'un fichier au format hexadécimal.

```
% ./hexdump /dev/zero
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

Appuyez sur `Ctrl+C` une fois convaincu qu'il continuera indéfiniment.

Mettre `/dev/zero` en correspondance avec la mémoire est une technique d'allocation avancée. Reportez-vous à la Section 5.3.5, « Autres utilisations de *mmap* », du Chapitre 5, « Communication interprocessus » pour plus d'informations, et consultez l'encadré « Obtenir de la Mémoire Alignée sur des Pages », de la Section 8.9, « *mprotect*: définir des permissions mémoire », du Chapitre 8, « Appels système Linux », pour un exemple.

6.5.3 */dev/full*

Le fichier `/dev/full` se comporte comme s'il se trouvait sur un système de fichiers ne comportant plus d'espace libre. Une écriture vers `/dev/full` échoue et positionne `errno` à `ENOSPC`, qui indique que le lecteur de destination est plein.

Par exemple, vous pouvez tenter d'écrire sur `/dev/full` en utilisant la commande `cp` :

```
% cp /etc/fstab /dev/full
cp: écriture de '/dev/full': Aucun espace disponible sur le périphérique
```

Le fichier `/dev/full` est essentiellement utile pour tester la façon dont se comporte votre programme s'il tombe à court d'espace disque lors de l'écriture d'un fichier.

6.5.4 Dispositifs de génération de nombres aléatoires

Les périphériques spéciaux `/dev/random` et `/dev/urandom` donnent accès au dispositif de génération de nombres aléatoires intégré au noyau Linux.

La plupart des fonctions logicielles chargées de générer des nombres aléatoires, comme la fonction `rand` de la bibliothèque standard du C, génèrent en fait des nombres *pseudo-aléatoires*. Bien que ces nombres aient certaines propriétés des nombres aléatoires, ils sont reproductibles : si vous relancez une série avec la même valeur d'initialisation, vous obtiendrez la même séquence de nombres pseudo-aléatoires à chaque fois. Cet inconvénient est inévitable car les ordinateurs sont intrinsèquement déterministes et prévisibles. Pour certaines applications, cependant, ce comportement n'est pas souhaitable ; par exemple, il peut être possible de casser un algorithme de chiffrement si l'on connaît la séquence de nombres aléatoires qu'il utilise.

L'obtention de nombres aléatoires plus stricts au sein de programmes informatiques nécessite une source externe d'événements aléatoires. Le noyau Linux utilise une source d'événements aléatoires particulièrement bonne : *vous* ! En mesurant les écarts temporels entre vos actions, comme l'appui sur les touches et les mouvements de la souris, Linux est capable de générer un flux de nombres aléatoires de grande qualité impossible à prédire. Vous pouvez accéder à ce flux en lisant les fichiers `/dev/random` et `/dev/urandom`. Les données que vous obtenez sont issues d'un flux d'octets généré aléatoirement.

La différence entre les deux périphériques n'est visible que lorsque Linux a épuisé son stock de nombres aléatoires. Si vous essayez de lire un nombre important d'octets à partir de `/dev/random` mais ne générez aucune action (vous n'utilisez pas le clavier, ne bougez pas la souris ni n'effectuez aucune autre action de ce type), Linux bloque l'opération de lecture. La génération ne reprendra que lorsque vous effectuerez des actions.

Par exemple, essayez d'afficher le contenu de `/dev/random` en utilisant la commande `od`[4]. Chaque ligne affiche 16 nombres aléatoires.

```
% od -t x1 /dev/random
0000000 2c 9c 7a db 2e 79 3d 65 36 c2 e3 1b 52 75 1e 1a
0000020 d3 6d 1e a7 91 05 2d 4d c3 a6 de 54 29 f4 46 04
0000040 b3 b0 8d 94 21 57 f3 90 61 dd 26 ac 94 c3 b9 3a
0000060 05 a3 02 cb 22 0a bc c9 45 dd a6 59 40 22 53 d4
```

[4]Nous utilisons `od` plutôt que le programme `hexdump` du Listing B.4, même s'il font plus ou moins la même chose, car `hexdump` se termine lorsqu'il n'y a plus de données à lire, alors que `od` attend des données supplémentaires. L'option `-t x1` indique à `od` d'afficher le contenu du fichier en hexadécimal.

Le nombre de lignes affichées varie – il peut y en avoir très peu – mais la sortie se mettra en pause dès que Linux épuisera son stock de nombres aléatoires. Essayez maintenant de déplacer votre souris ou de saisir quelque chose au clavier et vérifiez que de nouveaux nombres aléatoires apparaissent. Pour en obtenir encore plus, vous pouvez laisser votre chat marcher sur le clavier.

Une lecture à partir de `/dev/urandom`, en revanche, ne bloque jamais. Si Linux tombe à court de nombre aléatoires, il utilise un algorithme de chiffrement pour générer des octets pseudo-aléatoires à partir de la dernière séquence d'octets aléatoires. Bien que ces octets soient suffisamment aléatoires pour la plupart des utilisations, ils ne satisfont pas autant de tests que ceux obtenus à partir de `/dev/random`.

Par exemple, si vous invoquez la commande suivante, les octets aléatoires défileront en continu, jusqu'à ce que vous tuiez le programme avec `Ctrl+C` :

```
% od -t x1 /dev/urandom
0000000 62 71 d6 3e af dd de 62 c0 42 78 bd 29 9c 69 49
0000020 26 3b 95 bc b9 6c 15 16 38 fd 7e 34 f0 ba ce c3
0000040 95 31 e5 2c 8d 8a dd f4 c4 3b 9b 44 2f 20 d1 54
...
```

Utiliser des nombres aléatoires provenant de `/dev/random` dans un programme est une chose assez facile. Le Listing 6.1 présente une fonction qui génère un nombre aléatoire en utilisant les octets lus à partir de `/dev/random`. Souvenez-vous que la lecture est bloquée jusqu'à ce qu'il y ait suffisamment d'événements aléatoires pour la satisfaire ; vous pouvez utiliser `/dev/urandom` à la place si vous accordez plus de priorité à la rapidité d'exécution et que vous pouvez vous contenter de nombres aléatoires d'une qualité moindre.

Listing 6.1 – (*random_number.c*) – **Fonction Générant un Nombre Aléatoire à partir de */dev/random***

```
#include <assert.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

/* Renvoie un entier aléatoire entre MIN et MAX inclus. La source utilisée
   est /dev/random. */
int random_number (int min, int max)
{
  /* Stocke un descripteur de fichier pointant vers /dev/random dans une
     variable static. De cette façon, nous n'avons pas besoin d'ouvrir le
     fichier à chaque fois que la fonction est appelée. */
  static int dev_random_fd = -1;
  char* next_random_byte;
  int bytes_to_read;
  unsigned random_value;
  /* S'assure que MAX est plus grand que MIN.  */
  assert (max > min);
  /* S'il s'agit du premier appel de la fonction, ouvre un descripteur
     de fichier pointant vers /dev/random. */
  if (dev_random_fd == -1) {
    dev_random_fd = open ("/dev/random", O_RDONLY);
    assert (dev_random_fd != -1);
  }
  /* Lit suffisamment d'octets aléatoires pour remplir un entier. */
  next_random_byte = (char*) &random_value;
```

```
28    bytes_to_read = sizeof (random_value);
29    /* Boucle jusqu'à ce que l'on ait assez d'octets. Comme /dev/random
30       est généré à partir d'actions de l'utilisateur, la lecture peut bloquer
31       et ne renvoyer qu'un seul octet à la fois. */
32    do {
33      int bytes_read;
34      bytes_read = read (dev_random_fd, next_random_byte, bytes_to_read);
35      bytes_to_read -= bytes_read;
36      next_random_byte += bytes_read;
37    } while (bytes_to_read > 0);
38    /* Calcule un nombre aléatoire dans l'intervalle demandé.  */
39    return min + (random_value % (max - min + 1));
40  }
```

6.5.5 Périphériques loopback

Un *périphérique loopback* vous permet de simuler un périphérique bloc en utilisant un fichier disque ordinaire. Imaginez un lecteur de disque pour lequel les données sont écrites et lues à partir d'un fichier appelé `image-disque` plutôt que depuis les pistes et secteurs d'un disque physique réel ou d'une des partition d'un disque (bien sûr, le fichier `image-disque` doit se situer sur un disque physique, qui doit être plus grand que le disque simulé). Un périphérique loopback vous permet d'utiliser un fichier de cette façon.

Les périphériques loopback s'appellent `/dev/loop0`, `/dev/loop1`, *etc.* Chacun peut être utilisé pour simuler un périphérique bloc distinct. Notez que seul le superutilisateur peut paramétrer un périphérique loopback.

Un tel périphérique peut être utilisé de la même façon que n'importe quel autre périphérique bloc. En particulier, vous pouvez placer un système de fichiers sur le périphérique puis monter ce système de fichiers comme s'il résidait sur un disque ou une partition classique. Un tel système de fichiers, qui réside entièrement au sein d'un fichier sur disque ordinaire, est appelé *système de fichiers virtuel*.

Pour construire un système de fichiers et le monter à partir d'un périphérique loopback, suivez ces étapes :

1. Créez un fichier vide qui contiendra le système de fichiers virtuel. La taille du fichier sera la taille du périphérique loopback une fois monté.

   Une façon pratique de construire un fichier d'une taille prédéterminée est d'utiliser la commande `dd`. Elle copie des blocs (de 512 octets chacun, par défaut) d'un fichier vers un autre. Le fichier `/dev/zero` convient parfaitement pour être utilisé comme source d'octets nuls.

   Pour construire un fichier de 10Mo appelé `image-disque`, utilisez la commande suivante :

   ```
   % dd if=/dev/zero of=/tmp/image-disque count=20480
   20480+0 enregistrements lus.
   20480+0 enregistrements écrits.
   % ls -l /tmp/image-disque
   -rw-rw----    1 root    root 10485760 Mar 8 01:56 /tmp/image-disque
   ```

2. Le fichier que vous venez de créer est rempli avec des octets à 0. Avant de le monter, vous devez y placer un système de fichiers. Cette opération initialise diverses structures de contrôle nécessaires à l'organisation et au stockage de fichiers et crée le répertoire racine.

Vous pouvez placer n'importe quel type de système de fichiers sur votre image disque. Pour créer un système de fichiers ext3 (le type le plus courant pour les disques Linux), utilisez la commande `mke2fs`. Comme elle est habituellement exécutée sur des périphériques bloc, et non pas des fichiers ordinaires, elle demande une confirmation :

```
/sbin/mke2fs -q -j /tmp/image-disque
/tmp/image-disque n'est pas un périphérique spécial à bloc.
Procéder malgré tout? (y pour oui, n pour non) y
```

L'option `-q` supprime les informations récapitulatives sur le système de fichiers nouvellement créé. Supprimez-la si vous êtes curieux.

Désormais `image-disque` contient un nouveau système de fichiers comme s'il s'agissait d'un disque de 10 Mo tout neuf.

3. Montez le système de fichiers en utilisant un périphérique loopback. Pour cela, utilisez la commande `mount` en spécifiant le fichier de l'image disque comme périphérique à monter. Passez également l'option de montage `loop=périphérique-loopback`, en utilisant l'option `-o` pour indiquer à `mount` quel périphérique loopback utiliser.

   Par exemple, pour monter notre système de fichiers `image-disque`, utilisez ces commandes. Souvenez-vous, seul le superutilisateur peut utiliser un périphérique loopback. La première commande crée un répertoire, `/tmp/virtual-fs`, que nous allons utiliser comme point de montage pour le système de fichiers virtuel.

   ```
   % mkdir /tmp/virtual-fs
   % mount -o loop=/dev/loop0 /tmp/image-disque /tmp/virtual-fs
   ```

   Désormais, l'image disque est montée comme s'il s'agissait d'un disque de 10Mo ordinaire.

   ```
   % df -h /tmp/virtual-fs
   Sys. De fich.         Tail. Occ. Disp. %Occ. Monté sur
   /tmp/image-disque       9.7M   13k  9.2M    0% /tmp/virtual-fs
   ```

   Vous pouvez l'utiliser comme n'importe quel autre disque :

   ```
   % cd /tmp/virtual-fs
   % echo "Coucou !" > test.txt
   % ls -l
   total 13
   drwxr-xr-x     2 root     root    12288 Mar 8 02:00 lost+found
   -rw-rw----     1 root    root        14 Mar 8 02:12 test.txt
   % cat test.txt
   Coucou !
   ```

   Notez que `lost+found` est un répertoire automatiquement créé par `mke2fs`[5].

   Lorsque vous en avez fini, démontez le système de fichiers virtuel.

   ```
   % cd /tmp
   % umount /tmp/virtual-fs
   ```

   Vous pouvez supprimer `image-disque` si vous le désirez ou vous pouvez le monter plus tard pour accéder aux fichiers du système de fichiers virtuel. Vous pouvez également le copier sur un autre ordinateur où vous pourrez le monter – le système de fichiers que vous avez créé sera entièrement intact.

[5] Si le système de fichiers subit des dommage et que des données sont récupérées sans être associées à un fichier, elles sont placées dans `lost+found`.

Au lieu de créer un système de fichiers à partir de rien, vous pouvez en copier un à partir d'un périphérique existant. Par exemple, vous pouvez créer l'image du contenu d'un CD-ROM simplement en le copiant à partir d'un lecteur de CD-ROM.

Si vous disposez d'un lecteur de CD-ROM IDE, utilisez le nom de périphérique correspondant, par exemple `/dev/hda`, décrit précédemment. Si vous disposez d'un lecteur CD-ROM SCSI, le nom de périphérique sera du type `/dev/scd0`. Le lien symbolique `/dev/cdrom` peut également exister sur votre système, il pointe alors vers le périphérique approprié. Consultez le fichier `/etc/fstab` pour déterminer quel périphérique correspond au lecteur de CD-ROM de votre ordinateur.

Copiez simplement le périphérique vers un fichier. Le résultat sera une image disque complète du système de fichiers du CD-ROM situé dans le lecteur – par exemple :

```
% cp /dev/cdrom /tmp/cdrom-image
```

Cette opération peut prendre plusieurs minutes selon le CD-ROM que vous copiez et la vitesse de votre lecteur. Le fichier image résultant sera relativement gros – il fera la même taille que le contenu du CD-ROM.

Vous pouvez maintenant monter cette image sans disposer du disque original. Par exemple, pour le monter sur `/mnt/cdrom`, utilisez cette commande :

```
% mount -o loop=/dev/loop0 /tmp/cdrom-image /mnt/cdrom
```

Comme l'image est située sur le disque dur, les temps d'accès seront bien inférieurs à ceux du disque CD-ROM original. Notez que la plupart des CD-ROM utilisent le système de fichiers ISO-9660.

6.6 PTY

Si vous exécutez la commande `mount` sans arguments de ligne de commande, ce qui liste les systèmes de fichiers montés sur votre système, vous remarquerez une ligne ressemblant à cela :

```
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
```

Elle indique qu'un système de fichiers d'un type particulier, `devpts`, est monté sur `/dev/pts`. Ce système de fichiers, qui n'est pas associé avec un périphérique matériel, est un système de fichiers « magique » créé par le noyau Linux. Il est similaire au système de fichiers `/proc` ; consultez le Chapitre 7 pour plus d'informations sur son fonctionnement.

Comme le répertoire `/dev`, `/dev/pts` contient des entrées correspondant à des périphériques. Mais contrairement à `/dev`, qui est un répertoire classique, `/dev/pts` est un répertoire spécial créé dynamiquement par le noyau Linux. Le contenu du répertoire varie avec le temps et reflète l'état du système.

Les fichiers de `/dev/pts` correspondent à des pseudo-terminaux (ou pseudo-TTY, ou PTY). Linux crée un PTY pour chaque nouvelle fenêtre de terminal que vous ouvrez et place l'entrée correspondante dans `/dev/pts`. Le périphérique PTY se comporte comme un terminal classique – il accepte des entrées depuis le clavier et affiche les sorties du programme lui correspondant. Les PTY sont numérotés et leur numéro correspond au nom du fichier correspondant dans `/dev/pts`.

Vous pouvez afficher le terminal associé à un processus grâce à la commande `ps`. Indiquez `tty` comme l'un des champ de format personnalisé avec l'option `-o`. Pour afficher l'identifiant de processus, le TTY et la ligne de commande de chaque processus partageant le même terminal, invoquez `ps -o pid,tty,cmd`.

6.6.1 Exemple d'utilisation des PTY

Par exemple, vous pouvez déterminer le PTY associé à une fenêtre de terminal donnée en invoquant cette commande au sein de la fenêtre :

```
% ps -o pid,tty,cmd
  PID TT       CMD
28832 pts/4    bash
29287 pts/4    ps -o pid,tty,cmd
```

La fenêtre où est lancée la commande s'exécute au sein du PTY 4.

Le PTY a un fichier correspondant dans /dev/pts :

```
% ls -l /dev/pts/4
crw--w----    1 samuel tty 136, 4 Mar 8 02:56 /dev/pts/4
```

Notez qu'il s'agit d'un périphérique caractère et son propriétaire est celui du processus pour lequel il a été créé.

Vous pouvez lire ou écrire sur un périphérique PTY. Si vous lisez à partir de celui-ci, vous intercepterez les saisies clavier destinées au programme s'exécutant au sein du PTY. Si vous essayez d'y écrire, les données apparaîtront dans la fenêtre correspondante.

Ouvrez un nouveau terminal et déterminez son PTY en invoquant `ps -o pid,tty,cmd`. Depuis une autre fenêtre, écrivez du texte sur ce périphérique. Par exemple, si le numéro de PTY du nouveau terminal est 7, invoquez cette commande depuis une autre fenêtre :

```
% echo "Hello, other window!" > /dev/pts/7
```

La sortie apparaît dans la fenêtre de terminal. Si vous la fermez, l'entrée numéro 7 de `/dev/pts` disparaît.

Si vous invoquez `ps` pour déterminer le TTY depuis un terminal virtuel en mode texte (appuyez sur `Ctrl+Alt+F1` pour basculer vers le premier terminal virtuel, par exemple), vous remarquerez qu'il s'exécute au sein d'un périphérique de terminal ordinaire et non pas un PTY :

```
% ps -o pid,tty,cmd
  PID TT       CMD
29325 tty1     -bash
29353 tty1     ps -o pid,tty,cmd
```

6.7 *ioctl*

L'appel système `ioctl` est une interface destinée au contrôle de dispositifs matériels. Le premier argument de `ioctl` est un descripteur de fichier qui doit pointer sur le périphérique que vous voulez contrôler. Le second argument est un code de requête indiquant l'opération que vous souhaitez effectuer. Différents codes de requêtes sont disponibles pour chacun des périphériques. Selon le code, il peut y avoir des arguments supplémentaires servant à passer des données à `ioctl`.

La plupart des codes de requête disponibles pour les différents périphériques sont listés sur la page de manuel de `ioctl_list`. L'utilisation de `ioctl` nécessite généralement une connaissance approfondie du pilote de périphérique du matériel que vous souhaitez contrôler. Il s'agit d'un sujet qui dépasse le cadre de ce livre. Cependant, nous présentons un exemple vous donnant un aperçu de la façon dont `ioctl` est utilisé.

Listing 6.2 – (*cdrom-eject.c*) – **Éjecte un CD-ROM**

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
  /* Ouvre un descripteur de fichier vers le périphérique passé sur la ligne de
     commande. */
  int fd = open (argv[1], O_RDONLY);
  /* Éjecte le CD-ROM. */
  ioctl (fd, CDROMEJECT);

  /* Ferme le descripteur. */
  close (fd);
  return 0;
}
```

Le Listing 6.2 est un court programme qui éjecte le disque présent dans un lecteur de CD-ROM (si ce dernier le supporte). Il prend un argument en ligne de commande, le périphérique correspondant au lecteur de CD-ROM. Il ouvre un descripteur de fichier pointant vers le périphérique et invoque `ioctl` avec le code de requête `CDROMEJECT`. Cette requête, définie dans l'entête `<linux/cdrom.h>`, indique au périphérique d'éjecter le disque.

Par exemple, si votre système dispose d'un lecteur de CD-ROM connecté en tant que périphérique maître sur le second contrôleur IDE, le périphérique correspondant est `/dev/hdc`. Pour éjecter le disque du lecteur, invoquez cette commande :

```
% ./cdrom-eject /dev/hdc
```

Chapitre 7

Le système de fichiers */proc*

Essayez d'invoquer la commande `mount` sans argument – elle liste les systèmes de fichiers actuellement montés sur votre système GNU/Linux. Vous apercevrez une ligne de ce type :

```
proc on /proc type proc (rw)
```

Il s'agit du système de fichiers spécial `/proc`. Notez que le premier champ, `proc`, indique qu'il n'est associé à aucun périphérique matériel, comme un lecteur de disque. Au lieu de cela, `/proc` est une fenêtre sur le noyau Linux en cours d'exécution. Les fichiers de `/proc` ne correspondent pas à des fichiers réels sur un disque physique. Il s'agit plutôt d'objets magiques qui se comportent comme des fichiers mais donnent accès à des paramètres, des structures de données et des statistiques du noyau. Le « contenu » de ces fichiers n'est pas composé de blocs de données, comme celui des fichiers ordinaires. Au lieu de cela, il est généré à la volée par le noyau Linux lorsque vous lisez le fichier. Vous pouvez également changer la configuration du noyau en cours d'exécution en écrivant dans certains fichiers du système de fichiers `/proc`.

Étudions un exemple :

```
% ls -l /proc/version
-r--r--r--    1 root  root 0 Jan 17 18:09 /proc/version
```

Notez que la taille du fichier est zéro ; comme le contenu du fichier est généré par le noyau, le concept de taille de fichier n'a pas de sens. De plus, si vous essayez cette commande, vous remarquerez que la date de modification est la date courante.

Qu'y a-t-il dans ce fichier ? Le contenu de `/proc/version` est une chaîne décrivant le numéro de version du noyau Linux. Il correspond aux informations qui seraient renvoyées par l'appel système `uname`, décrit dans le Chapitre 8, « Appels système Linux », Section 8.15, « *uname* », ainsi que des informations supplémentaires comme la version du compilateur utilisé pour construire le noyau. Vous pouvez lire `/proc/version` comme n'importe quel autre fichier. Par exemple, au moyen de `cat` :

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1
```

Les différentes entrées du système de fichiers `/proc` sont décrites dans la page de manuel de `/proc` (Section 5). Pour la consulter, utilisez la commande suivante :

```
% man 5 proc
```

Dans ce chapitre, nous décrirons certaines fonctionnalités du système de fichiers `/proc` qui sont les plus susceptibles de servir à des programmeurs et nous donnerons des exemples d'utilisation. Quelques unes de ces fonctionnalités sont également utiles pour le débogage.

Si vous êtes intéressé par le fonctionnement exact de `/proc`, consultez le code source du noyau Linux situé dans `/usr/src/linux/fs/proc`.

7.1 Obtenir des informations à partir de */proc*

La plupart des fichiers de `/proc` donnent des informations formatées pour pouvoir être lues par des humains, cependant leur présentation reste suffisamment simple pour qu'elles puissent être analysées automatiquement. Par exemple, `/proc/cpuinfo` contient des informations sur le processeur (ou les processeurs dans le cas d'une machine multiprocesseur). Son contenu est un tableau de valeurs, une par ligne, avec une description de la valeur et deux points avant chacune d'entre elles.

Par exemple, son contenu peut ressembler à cela :

```
% cat /proc/cpuinfo
processor       :0
vendor_id       : GenuineIntel
cpu family      :6
model           :5
model name      : Pentium II (Deschutes)
stepping        :2
cpu MHz         : 400.913520
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
sep_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     :2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 mmx fxsr
bogomips        : 399.77
```

Nous décrirons à quoi correspondent certains de ces champs dans la Section 7.3.1, « Informations sur le processeur ».

Une méthode simple pour extraire une valeur de cette liste est de lire le fichier dans un tampon et de l'analyser en mémoire au moyen de `sscanf`. Le Listing 7.1 montre une façon de faire. Le programme déclare une fonction, `get_cpu_clock_speed`, qui charge `/proc/cpuinfo` en mémoire et en extrait la vitesse d'horloge du premier processeur.

Listing 7.1 – (***clock-speed.c***) – **Exemple d'utilisation de */proc/cpuinfo***

```c
#include <stdio.h>
#include <string.h>

/* Renvoie la vitesse d'horloge du processeur en MHZ, d'après /proc/cpuinfo.
   Sur un système multiprocesseur, renvoie la vitesse du premier.
   Renvoie zéro en cas d'erreur. */

float get_cpu_clock_speed ()
{
  FILE* fp;
  char buffer[1024];
  size_t bytes_read;
  char* match;
  float clock_speed;

  /* Charge le contenu de /proc/cpuinfo dans le tampon. */
  fp = fopen ("/proc/cpuinfo", "r");
  bytes_read = fread (buffer, 1, sizeof (buffer), fp);
  fclose (fp);
  /* Traite le cas où la lecture échoue ou le buffer est trop petit. */
  if (bytes_read == 0 || bytes_read == sizeof (buffer))
    return 0;
  /* Place un caractère nul à la fin de la chaîne. */
  buffer[bytes_read] = "\0";
  /* Recherche la ligne commençant par "cpu MHz". */
  match = strstr (buffer, "cpu MHz");
  if (match == NULL)
    return 0;
  /* Analyse la ligne pour extraire la vitesse d'horloge. */
  sscanf (match, "cpu MHz : %f", &clock_speed);
  return clock_speed;
}

int main ()
{
  printf ("CPU clock speed: %4.0f MHz\n", get_cpu_clock_speed ());
  return 0;
}
```

Soyez conscient du fait que les noms, la signification et les formats de sortie des entrées du système de fichiers `/proc` peuvent changer au fil des révisions du noyau Linux. Si vous les utilisez au sein d'un programme, assurez-vous que le comportement du programme reste cohérent si une entrée est absente ou formatée d'une façon inconnue.

7.2 Répertoires de processus

Le système de fichiers `/proc` contient un répertoire par processus s'exécutant sur le système. Le nom de chacun de ces répertoires est l'identifiant du processus auquel il correspond[1]. Ces répertoires apparaissent et disparaissent dynamiquement lors du démarrage et de l'arrêt d'un processus. Chaque dossier contient plusieurs entrées donnant accès aux informations sur le processus en cours d'exécution. C'est de ces répertoires de processus que le système de fichiers `/proc` tire son nom.

[1] Sur certains systèmes UNIX, les identifiants de processus sont alignés au moyen de zéros. Ce n'est pas le cas sous GNU/Linux.

Chaque répertoire de processus contient ces fichiers :

- `cmdline` contient la liste d'argument du processus. L'entrée `cmdline` est décrite dans la Section 7.2.2, « Liste d'arguments d'un processus ».
- `cwd` est un lien symbolique pointant vers le répertoire de travail courant du processus (qui peut être défini par exemple, via un appel à `chdir`).
- `environ` contient l'environnement du processus. Le fichier environ est décrit dans la Section 7.2.3, « Environnement de processus ».
- `exe` est un lien symbolique pointant vers l'image binaire exécutée par le processus. L'entrée `exe` est décrite dans la Section 7.2.5, « Descripteurs de fichiers d'un processus ».
- `maps` contient des informations sur les fichiers mis en correspondance avec l'espace mémoire du processus. Consultez le Chapitre 5, Section 5.3, « Mémoire mappée », pour plus de détails sur le fonctionnement des fichiers mis en correspondance avec la mémoire. Pour chaque fichier mis en correspondance, `maps` affiche l'intervalle d'adresses de l'espace mémoire du processus avec lequel le fichier est mis en correspondance, les permissions applicables à ces adresses, le nom du fichier ainsi que d'autres informations.
  Le tableau maps affiche pour chaque processus le fichier binaire en cours d'exécution, les bibliothèques partagées actuellement chargées et les autres fichiers que le processus a mis en correspondance avec sa mémoire.
- `root` est un lien symbolique vers le répertoire racine du processus. Généralement, il s'agit d'un lien vers `/`, le répertoire racine du système. Le répertoire racine d'un processus peut être changé par le biais de l'appel `chroot` ou de la commande `chroot`[2].
- `stat` contient des statistiques et des informations sur le statut du processus. Ce sont les mêmes données que celles présentées dans le fichier `status`, mais au format numérique brut, sur une seule ligne. Ce format est difficile à lire mais plus adapté au traitement automatique par des programmes.
  Si vous voulez utiliser le fichier `stat` dans vos programmes, consultez la page de manuel de `proc` qui décrit son contenu en invoquant `man 5 proc`.
- `statm` contient des informations sur la mémoire utilisée par le processus. Le fichier `statm` est décrit dans la Section 7.2.6, « Statistiques mémoire de processus ».
- `status` contient des informations statistiques et statut sur le processus formatée de façon à être lisibles par un humain. La Section 7.2.7, « Statistiques sur les processus » présente une description du fichier `status`.
- Le fichier `cpu` n'apparaît que sur les noyaux Linux SMP. Il contient un récapitulatif de la consommation en temps (utilisateur et système) du processus, par processeur.

Notez que pour des raisons de sécurité, les permissions de certains fichiers sont définies de façon à ce que seul l'utilisateur propriétaire du processus (ou le superutilisateur) puisse y accéder.

7.2.1 */proc/self*

Une entrée particulière dans le système de fichiers `/proc` facilite son utilisation par un programme pour obtenir des informations sur le processus au sein duquel il s'exécute. L'entrée

[2] L'appel et la commande `chroot` sortent du cadre de ce livre. Consultez la page de manuel de `chroot` dans la section 1 pour des informations sur la commande (`man 1 chroot`) ou la page de manuel de chroot dans la section 2 (`man 2 chroot`) pour plus d'informations sur l'appel.

`/proc/self` est un lien symbolique vers le répertoire de `/proc` correspondant au processus courant. La destination du lien `/proc/self` dépend du processus qui l'utilise : chaque processus voit son propre répertoire comme destination du lien.

Par exemple, le programme du Listing 7.2 utilise la destination du lien `/proc/self` pour déterminer son identifiant de processus (nous ne faisons cela que dans un but d'illustration ; l'appel de la fonction `getpid`, décrite dans le Chapitre 3, « Processus », Section 3.1.1, « Identifiants de processus » constitue une façon beaucoup plus simple d'arriver au même résultat). Ce programme utilise l'appel système `readlink`, présenté dans la Section 8.11, « *readlink*: lecture de liens symboliques », pour extraire la cible du lien.

Listing 7.2 – (***get-pid.c***) – **Récupérer son PID à partir de** ***/proc/self***

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

/* Renvoie l'identifiant de processus de l'appelant, déterminé à partir du
   lien symbolique /proc/self. */

pid_t get_pid_from_proc_self ()
{
  char target[32];
  int pid;
  /* Lit la cible du lien symbolique. */
  readlink ("/proc/self", target, sizeof (target));
  /* La cible est un répertoire portant de nom du PID.  */
  sscanf (target, "%d", &pid);
  return (pid_t) pid;
}
int main ()
{
  printf ("/proc/self renvoie l'identifiant %d\n",
          (int) get_pid_from_proc_self ());
  printf ("getpid() renvoie l'identifiant %d\n", (int) getpid ());
  return 0;
}
```

7.2.2 Liste d'arguments d'un processus

Le fichier `cmdline` contient la liste d'arguments du processus (consultez le Chapitre 2, « Écrire des logiciels GNU/Linux de qualité », Section 2.1.1, « La liste d'arguments »). Les arguments sont présentés sous forme d'une seule chaîne de caractères, les arguments étant séparés par des `NUL`. La plupart des fonctions de traitement de chaînes de caractères s'attendent à ce la chaîne en elle-même soit terminée par un `NUL` et ne géreront pas les `NUL` contenus dans la chaîne correctement, vous aurez donc à traiter le contenu de cette chaîne d'une façon spéciale.

Dans la Section 2.1.1, nous avons présenté un programme, Listing 2.1, qui affichait sa liste d'arguments. En utilisant l'entrée `cmdline` du système de fichiers `/proc`, nous pouvons créer un programme qui affiche la liste d'arguments d'un autre processus. Le Listing 7.3 est un programme de ce type ; il affiche la liste d'arguments du processus dont l'identifiant est passé en paramètre. Comme il peut y avoir plusieurs `NUL` dans le contenu de cmdline et non un seul en fin de chaîne,

NUL et NULL

NUL est le caractère dont la valeur décimale est 0. Il est différent de NULL, qui est un pointeur avec une valeur de 0. En C, une chaîne de caractères est habituellement terminée par un caractère NUL. Par exemple, la chaîne de caractères "Coucou␣!" occupe 9 octets car il y a un NUL implicite après le point d'exclamation indiquant la fin de la chaîne.
NULL, par contre, est une valeur de pointeur dont vous pouvez être sûr qu'elle ne correspondra jamais à une adresse mémoire réelle dans votre programme.
En C et en C++, NUL est représenté par la constante de caractère '0', ou (char) 0. La définition de NULL diffère selon le système d'exploitation ; sous Linux, NULL est défini comme ((void*)0) en C et tout simplement 0 en C++.

nous ne pouvons pas déterminer la taille de la chaîne en utilisant `strlen` (qui se contente de compter le nombre de caractères jusqu'à ce qu'elle rencontre un NUL). Au lieu de cela, nous déterminons la longueur de `cmdline` grâce à `read`, qui renvoie le nombre d'octets lus.

Listing 7.3 – (*print-arg-list.c*) – **Affiche la Liste d'Arguments d'un Processus**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7
8  /* Affiche la liste d'aguments, un par ligne, du processus dont l'identifiant
9     est passé en paramètre. */
10
11 void print_process_arg_list (pid_t pid)
12 {
13   int fd;
14   char filename[24];
15   char arg_list[1024];
16   size_t length;
17   char* next_arg;
18
19   /* Génère le nom du fichier cmdline pour le processus. */
20   snprintf (filename, sizeof (filename), "/proc/%d/cmdline", (int) pid);
21   /* Lit le contenu du fichier. */
22   fd = open (filename, O_RDONLY);
23   length = read (fd, arg_list, sizeof (arg_list));
24   close (fd);
25   /* read n'ajoute pas de NULL à la fin du tempon, nous le faisons ici. */
26   arg_list[length] = '\0';
27
28   /* Boucle sur les arguments. Ceux-ci sont séparés par des NUL. */
29   next_arg = arg_list;
30   while (next_arg < arg_list + length) {
31     /* Affiche l'argument. Chaque argument est terminé par NUL, nous le
32        traitons donc comme une chaîne classique. */
33     printf ("%s\n", next_arg);
34     /* Avance à l'argument suivant. Puisque chaque argument est terminé par
35        NUL, strlen renvoie la longueur de l'argument,
36        pas celle de la liste. */
37     next_arg += strlen (next_arg) + 1;
```

```
38    }
39  }
40
41  int main (int argc, char* argv[])
42  {
43    pid_t pid = (pid_t) atoi (argv[1]);
44    print_process_arg_list (pid);
45    return 0;
46  }
```

Supposons que le processus 372 soit le démon de journalisation système, `syslogd`.

```
% ps 372
   PID TTY      STAT   TIME COMMAND
   372 ?        S      0:00 syslogd -m 0

% ./print-arg-list 372
syslogd
-m
0
```

Dans ce cas, `syslogd` a été invoqué avec les arguments `-m 0`.

7.2.3 Environnement de processus

Le fichier `environ` contient l'environnement du processus (consultez la Section 2.1.6, « L'environnement »). Comme pour `cmdline`, les différentes variables d'environnement sont séparées par des `NUL`. Le format de chaque élément est le même que celui utilisé dans la variable `environ`, à savoir `VARIABLE=valeur`.

Le Listing 7.4 présente une généralisation du programme du Listing 2.3 de la Section 2.1.6. Cette version prend un identifiant du processus sur la ligne de commande et affiche son environnement en le lisant à partir de `/proc`.

Listing 7.4 – (*print-environment.c*) – **Affiche l'Environnement d'un Processus**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7
8  /* Affiche l'environnement du processus dont l'identifiant est passé en
9     paramètre, une variable par ligne. */
10
11 void print_process_environment (pid_t pid)
12 {
13   int fd;
14   char filename[24];
15   char environment[8192];
16   size_t length;
17   char* next_var;
18
19   /* Génère le nom du fichier environ pour le processus. */
20   snprintf (filename, sizeof (filename), "/proc/%d/environ", (int) pid);
21   /* Lit le contenu du fichier. */
22   fd = open (filename, O_RDONLY);
23   length = read (fd, environment, sizeof (environment));
```

```
24    close (fd);
25       /* read ne place pas de caractère NUL à la fin du tampon. */
26    environment[length] = '\0';
27
28    /* Boucle sur les variables. Elles sont séparées par des NUL. */
29    next_var = environment;
30    while (next_var < environment + length) {
31      /* Affiche la variable. Elle est terminée par un NUL, on la traite
32         donc comme une chaîne ordinaire. */
33      printf ("%s\n", next_var);
34      /* Passe à la variable suivante. Puisque chaque variable est terminée
35         par un NUL, strlen calcule bien la taille de la prochaine variable,
36         et non pas de toute la liste. */
37      next_var += strlen (next_var) + 1;
38    }
39  }
40  int main (int argc, char* argv[])
41  {
42    pid_t pid = (pid_t) atoi (argv[1]);
43    print_process_environment (pid);
44    return 0;
45  }
```

7.2.4 Exécutable de processus

L'entrée `exe` pointe vers le fichier binaire exécuté par le processus. Dans la Section 2.1.1, nous avons expliqué que le nom de ce fichier est habituellement passé comme premier élément de la liste d'argument. Notez cependant qu'il s'agit d'une convention ; un programme peut être invoqué avec n'importe quelle liste d'arguments. Utiliser l'entrée `exe` du système de fichiers `/proc` est une façon plus fiable de déterminer le fichier binaire en cours d'exécution.

De même, il est possible d'extraire l'emplacement absolu de l'exécutable, à partir du système de fichiers `/proc`. Pour beaucoup de programmes, les fichiers auxiliaires se trouvent dans des répertoires relatifs à l'exécutable, il est donc nécessaire de déterminer où se trouve réellement le fichier binaire. La fonction `get_executable_path` du Listing 7.5 détermine le chemin du fichier binaire s'exécutant au sein du processus appelant en examinant le lien symbolique `/proc/self/exe`.

Listing 7.5 – (*get-exe-path.c*) – **Obtient l'Emplacement du Fichier Binaire en Cours d'Exécution**

```
1   #include <limits.h>
2   #include <stdio.h>
3   #include <string.h>
4   #include <unistd.h>
5
6   /* Recherche l'emplacement du fichier binaire en cours d'exécution.
7      Ce chemin est placé dans BUFFER, de taille LEN. Renvoie le nombre de
8      caractères dans le chemin ou -1 en cas d'erreur. */
9
10  size_t get_executable_path (char* buffer, size_t len)
11  {
12    char* path_end;
13    /* Lit la cible de /proc/self/exe. */
14    if (readlink ("/proc/self/exe", buffer, len) <= 0)
15      return -1;
16    /* Recherche la dernière occurrence du caractère slash. */
17    path_end = strrchr (buffer, "/");
```

```
18    if (path_end == NULL)
19      return -1;
20    /* Se place sur le caractère suivant le dernier slash. */
21    ++path_end;
22    /* Récupère le répertoire contenant le programme en tronquant le chemin
23       après le dernier slash. */
24    *path_end = "\0";
25
26
27
28    /* La longueur du chemin est le nombre de caractères
29       jusqu'au dernier slash. */
30    return (size_t) (path_end - buffer);
31  }
32
33  int main ()
34  {
35    char path[PATH_MAX];
36    get_executable_path (path, sizeof (path));
37    printf ("ce programme se trouve dans le répertoire %s\n", path);
38    return 0;
39  }
```

7.2.5 Descripteurs de fichiers d'un processus

Le répertoire `fd` contient des sous-répertoires correspondant aux descripteurs de fichiers ouverts par un processus. Chaque entrée est un lien symbolique vers le fichier ou le périphérique désigné par le descripteur de fichier. Vous pouvez lire ou écrire sur ces liens symboliques ; cela revient à écrire ou à lire depuis le fichier ou le périphérique ouvert dans le processus cible. Les noms des entrées du sous-répertoire `fd` correspondent aux numéros des descripteurs de fichiers.

Voici une astuce amusante que vous pouvez essayer avec les entrées `fd` de `/proc`. Ouvrez une nouvelle fenêtre de terminal et récupérez l'identifiant de processus du processus shell en lançant `ps`.

```
% ps
  PID TTY      TIME CMD
 1261 pts/4 00:00:00 bash
 2455 pts/4 00:00:00 ps
```

Dans ce cas, le shell (`bash`) s'exécute au sein du processus 1261. Ouvrez maintenant une seconde fenêtre et jetez un oeil au contenu du sous-répertoire fd pour ce processus.

```
% ls -l /proc/1261/fd
total 0
lrwx------    1 samuel samuel 64 Jan 30 01:02 0 -> /dev/pts/4
lrwx------    1 samuel samuel 64 Jan 30 01:02 1 -> /dev/pts/4
lrwx------    1 samuel samuel 64 Jan 30 01:02 2 -> /dev/pts/4
```

(Il peut y avoir d'autres lignes si d'autres fichiers sont ouverts). Souvenez-vous de ce que nous avons dit dans la Section 2.1.4, « E/S standards » : les descripteurs de fichiers 0, 1 et 2 sont initialisés pour pointer vers l'entrée, la sortie et la sortie d'erreurs standards, respectivement. Donc, en écrivant dans le fichier `/proc/1261/fd/1` vous pouvez écrire sur le périphérique attaché à `stdout` pour le processus shell – dans ce cas, un pseudo TTY correspondant à la première fenêtre. Dans la seconde fenêtre, essayez d'écrire un message dans ce fichier :

```
% echo "Coucou." >> /proc/1261/fd/1
```

Le texte apparaît dans la première fenêtre.

Les descripteurs de fichiers autres que l'entrée, la sortie et la sortie d'erreurs standards apparaissent dans le sous-répertoire `fd`. Le Listing 7.6 présente un programme qui se contente d'ouvrir un descripteur pointant vers un fichier passé sur la ligne de commande et de boucler indéfiniment.

Listing 7.6 – (*open-and-spin.c*) – **Ouvre un Fichier en Lecture**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <sys/stat.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6
7  int main (int argc, char* argv[])
8  {
9    const char* const filename = argv[1];
10   int fd = open (filename, O_RDONLY);
11   printf ("dans le processus %d, le descripteur %d pointe vers %s\n",
12           (int) getpid (), (int) fd, filename);
13   while (1);
14   return 0;
15 }
```

Essayez de le lancer dans une fenêtre :

```
% ./open-and-spin /etc/fstab
dans le processus 2570, le descripteur 3 pointe vers /etc/fstab
```

Dans une autre fenêtre, observez le contenu du sous-répertoire `fd` correspondant à ce processus.

```
% ls -l /proc/2570/fd
total 0
lrwx------    1 samuel samuel 64 Jan 30 01:30 0 -> /dev/pts/2
lrwx------ 1 samuel samuel 64 Jan 30 01:30 1 -> /dev/pts/2
lrwx------ 1 samuel samuel 64 Jan 30 01:30 2 -> /dev/pts/2
lr-x------ 1 samuel samuel 64 Jan 30 01:30 3 -> /etc/fstab
```

Notez que l'entrée du descripteur de fichier 3 est liée au fichier `/etc/fstab` vers lequel pointe le descripteur.

Les descripteurs de fichiers peuvent pointer vers des sockets ou des tubes (consultez le Chapitre 5 pour plus d'informations). Dans un tel cas, la cible du lien symbolique correspondant au descripteur de fichier indiquera « socket » ou « pipe » au lieu de pointer vers un fichier ou un périphérique classique.

7.2.6 Statistiques mémoire de processus

L'entrée `statm` contient une liste de sept nombres, séparés par des espaces. Chaque valeur correspond au nombre de pages mémoire utilisées par le processus dans une catégorie donnée. Les voici , dans l'ordre :

- Taille totale du processus.
- Taille du processus résident en mémoire.

- Mémoire partagée avec d'autres processus – c'est-à-dire les pages mises en correspondance avec l'espace mémoire du processus et d'au moins un autre (comme les bibliothèques partagées ou les pages en copie à l'écriture non modifiées).
- Taille du texte du processus – c'est-à-dire la taille du code chargé.
- Taille des bibliothèques partagées chargées pour le processus.
- Mémoire utilisée pour la pile du processus.
- Le nombre de pages modifiées par le programme.

7.2.7 Statistiques sur les processus

L'entrée `status` contient un certain nombre d'informations à propos du processus, formatées de façon à être compréhensibles par un humain. Parmi ces informations on trouve l'identifiant du processus et de son père, les identifiants d'utilisateur et de groupe réel et effectif, l'utilisation mémoire et des masques binaires indiquant quels signaux sont interceptés, ignorés et bloqués.

7.3 Informations sur le matériel

Plusieurs autres fichiers du système de fichiers `/proc` donnent accès à des informations sur le matériel. Bien qu'elles n'intéressent généralement que les administrateurs système, ces informations peuvent parfois être utiles aux programmeurs. Nous présenterons ici les plus utiles.

7.3.1 Informations sur le processeur

Comme nous l'avons dit précédemment, `/proc/cpuinfo` contient des informations sur le ou les processeur(s) du système. Le champ Processor indique le numéro du processeur ; il est à 0 sur un système monoprocesseur. Les champs Vendor, CPU Family, Model et Stepping vous permettent de déterminer le modèle et la révision exacts du processeur. Plus utile, le champ Flags indique quels indicateurs sont positionnés, ils indiquent les fonctionnalités disponibles pour processeur. Par exemple, `mmx`, indique que les instructions MMX[3] sont disponibles.

La plupart des informations renvoyées par `/proc/cpuinfo` sont obtenues à partir de l'instruction assembleur x86 `cpuid`. Il s'agit d'un mécanisme de bas niveau permettant d'obtenir des informations sur le processeur. Pour mieux comprendre l'affichage de `/proc/cpuinfo`, consultez la documentation de l'instruction `cpuid` dans le *IA-32 Intel Architecture Software Developer's Manual, Volume 2 : Instruction Set Reference d'Intel* (en anglais). Ce manuel est disponible sur `http://developer.intel.com/design`.

Le dernier champ, bogomips, est une valeur propre à Linux. Il s'agit d'un indicateur de la vitesse du processeur mesurée au moyen d'une boucle et qui est donc relativement mauvais.

[3]Reportez-vous au *IA-32 Intel Architecture Software Developer's Manuel* pour une documentation complète sur les instructions MMX, et consultez le Chapitre 9, « Code assembleur en ligne », dans ce livre pour plus d'informations sur leur utilisation au sein de programmes GNU/Linux.

7.3.2 Informations sur les périphériques

Le fichier `/proc/devices` dresse la liste des numéros de périphérique majeurs pour les périphériques caractères et bloc disponibles pour le système. Consultez le Chapitre 6, « Périphériques », pour plus d'informations sur les types de périphériques et leurs numéros.

7.3.3 Informations sur le bus PCI

Le fichier `/proc/pci` donne un aperçu des périphériques attachés au(x) bus PCI. Il s'agit de cartes d'extension PCI mais cela peut aussi inclure les périphériques intégrés à la carte mère ainsi que les cartes graphiques AGP. Le listing inclut le type de périphérique ; son identifiant et son constructeur ; son nom s'il est disponible ; des informations sur ses capacités et sur les ressources PCI qu'il utilise.

7.3.4 nformations sur le port série

Le fichier `/proc/tty/driver/serial` contient des informations de configuration et des statistiques sur les ports série. Ceux-ci sont numérotés à partir de 0[4]. Les informations de configuration sur les ports série peuvent également être obtenues, ainsi que modifiées, via la commande `setserial`. Cependant, `/proc/tty/driver/serial` contient des statistiques supplémentaires sur le nombre d'interruptions reçues par chaque port série.

Par exemple, cette ligne de `/proc/tty/driver/serial` pourrait décrire le port série 1 (qui serait `COM2` sous Windows) :

```
1: uart:16550A port:2F8 irq:3 baud:9600 tx:11 rx:0
```

Elle indique que le port série est géré par un USART (Universal Synchronous Asynchronous Receiver Transmitter) de type 16550, utilise le port 0x2f8 et l'IRQ 3 pour la communication et tourne à 9 600 bauds. Le port a reçu 11 interruptions d'émission et 0 interruption de réception.

Consultez la Section 6.4, « Périphériques matériels », pour des informations sur les périphériques série.

7.4 Informations sur le noyau

La plupart des entrées de `/proc` donnent accès à des informations sur la configuration et l'état du noyau en cours d'exécution. Certaines de ces entrées sont à la racine de `/proc` ; d'autres se trouvent sous `/proc/sys/kernel`.

7.4.1 Informations de version

Le fichier `/proc/version` contient une chaîne décrivant la version du noyau et donnant des informations sur sa compilation. Elle comprend également des informations sur la façon dont il a été compilé : par qui, sur quelle machine, quand et avec quel compilateur – par exemple :

[4] Notez que sous DOS et Windows, les port série sont numérotés à partir de 1, donc `COM1` correspond au port série 0 sous Linux.

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com) (gcc version
egcs-2.91.66 19990314/Linux (egcs-1.1.2 release)) #1 Tue Mar 7
21:07:39 EST 2000
```

Cette commande indique que le système s'exécute sur une version 2.2.14 du noyau Linux, compilé avec la version 1.1.2 de EGCS (EGC, l'Experimental GNU Compiler System, était un précurseur du projet GCC actuel).

Les informations les plus intéressantes de cet affichage, le nom de l'OS et la version et la révision du noyau, sont également disponibles dans des fichiers individuels de `/proc`. Il s'agit de `/proc/sys/kernel/ostype`, `/proc/sys/kernel/osrelease` et `/proc/sys/kernel/version`, respectivement.

```
% cat /proc/sys/kernel/ostype
Linux
% cat /proc/sys/kernel/osrelease
2.2.14-5.0
% cat /proc/sys/kernel/version
#1 Tue Mar 7 21:07:39 EST 2000
```

7.4.2 Noms d'hôte et de domaine

Les fichiers `/proc/sys/kernel/hostname` et `/proc/sys/kernel/domainname` contiennent les noms d'hôte et de domaine de l'ordinateur, respectivement. Ces informations sont les mêmes que celles renvoyées par l'appel système `uname`, décrit dans la Section 8.15.

7.4.3 Utilisation mémoire

Le fichier `/proc/meminfo` contient des informations sur l'utilisation mémoire du système. Les informations sont présentées à la fois pour la mémoire physique et pour l'espace d'échange (swap). Les trois premières lignes présentent les totaux, en octets ; les lignes suivantes reprennent ces informations en kilooctets – par exemple :

```
% cat /proc/meminfo
        total:    used:    free: shared: buffers: cached:
Mem: 529694720 519610368 10084352 82612224 10977280 82108416
Swap: 271392768 44003328 227389440
MemTotal:    517280 kB
MemFree:       9848 kB
MemShared:    80676 kB
Buffers:      10720 kB
Cached:       80184 kB
BigTotal:         0 kB
BigFree:          0 kB
SwapTotal:   265032 kB
SwapFree:    222060 kB
```

Cet affichage montre un total de 512 Mo de mémoire physique, dont environ 9 Mo sont libres et 258 Mo d'espace d'échange (swap), dont 216 Mo sont libres. Dans la ligne correspondant à la mémoire physique, trois autres valeurs sont présentées :

– La colonne Shared affiche la quantité de mémoire partagée actuellement allouée sur le système (consultez la Section 5.1, « Mémoire partagée »).

- La colonne Buffers donne la mémoire allouée par Linux pour les tampons des périphériques bloc. Ces tampons sont utilisés par les pilotes de périphériques pour conserver les blocs de données lus ou écrits sur le disque.
- La colonne Cached indique la mémoire allouée par Linux pour le cache de page. Cette mémoire est utilisée pour mettre en cache les accès à des fichiers mis en correspondance avec la mémoire.

Vous pouvez utiliser la commande `free` pour afficher les mêmes informations sur la mémoire.

7.5 Lecteurs et systèmes de fichiers

Le système de fichiers `/proc` contient également des informations sur les lecteurs de disques présents sur le système et les systèmes de fichiers montés qui y correspondent.

7.5.1 Systèmes de fichiers

L'entrée `/proc/filesystems` dresse la liste de tous les types de systèmes de fichiers connus par le noyau. Notez qu'elle n'est pas très utile car elle n'est pas complète : des systèmes de fichiers peuvent être chargés et déchargés dynamiquement sous forme de modules noyau. Le contenu de `/proc/filesystems` ne reflète que les types de systèmes de fichiers liés statiquement au noyau ou actuellement chargés. D'autres types de systèmes de fichiers peuvent être disponibles sous forme de modules mais ne pas être chargés.

7.5.2 Lecteurs et partitions

Le système de fichiers `/proc` donne également des informations sur les périphériques connectés aux contrôleurs IDE ou SCSI (si le système en dispose).

Sur un système classique, le répertoire `/proc/ide` peut contenir un ou deux sous-répertoires, `ide0` et `ide1`, relatifs aux contrôleurs primaire et secondaire du système[5]. Ils contiennent d'autres sous-répertoires correspondants aux périphériques physiques connectés à ces contrôleurs. Les répertoires de chaque contrôleur ou périphérique peuvent être absents si Linux n'a détecté aucun périphérique connecté. Les chemins absolus des quatre périphériques IDE possibles sont présentés dans le Tableau 7.1.

TAB. 7.1 – Chemins Absolus des Quatre Périphériques IDE Possibles

Contrôleur	**Périphérique**	**Répertoire**
Primaire	Maître	`/proc/ide/ide0/hda/`
Primaire	Esclave	`/proc/ide/ide0/hdb/`
Secondaire	Maître	`/proc/ide/ide1/hdc/`
Secondaire	Esclave	`/proc/ide/ide1/hdd/`

[5]S'il est correctement configuré, le noyau Linux peut prendre en charge des contrôleurs IDE supplémentaires. Ils sont numérotés de façon séquentielle à partir de `ide2`.

Consultez la Section 6.4, « Périphériques Matériels », pour plus d'informations sur les noms des périphériques IDE.

Chaque répertoire de périphérique IDE contient plusieurs entrées donnant accès à des informations d'identification et de configuration sur le périphérique. Voici les plus utiles :

- `model` contient la chaîne d'identification du modèle de périphérique.
- `media` donne le type de média lu par le périphérique. Les valeurs possibles sont `disk`, `cdrom`, `tape` (cassette), `floppy` (disquette) et `UNKNOWN` (inconnu).
- `capacity` indique la capacité du périphérique en blocs de 512 octets. Notez que pour les lecteurs de CD-ROM, cette valeur sera $2^{31} - 1$, et non pas la capacité du disque présent dans le lecteur. La valeur de `capacity` représente la capacité total du disque physique ; la capacité des systèmes de fichiers contenus dans les partitions du disque sera plus petite.

Par exemple, ces commandes vous montrent comment déterminer le type de média et le périphérique primaire connecté au contrôleur IDE secondaire. Dans ce cas, il s'agit d'un lecteur CD-ROM Toshiba.

```
% cat /proc/ide/ide1/hdc/media
cdrom
% cat /proc/ide/ide1/hdc/model
TOSHIBA CD-ROM XM-6702B
```

Si des périphériques SCSI sont présents, `/proc/scsi/scsi` contient la liste de leurs identifiants. Par exemple, son contenu peut ressembler à cela :

```
% cat /proc/scsi/scsi
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: QUANTUM Model: ATLAS_V__9_WLS Rev: 0230
  Type:   Direct-Access                 ANSI SCSI revision: 03
Host: scsi0 Channel: 00 Id: 04 Lun: 00
  Vendor: QUANTUM Model: QM39100TD-SW   Rev: N491
  Type:   Direct-Access                 ANSI SCSI revision: 02
```

Cet ordinateur contient un contrôleur SCSI simple canal (appelé `scsi0`), auquel sont connectés deux disques Quantum, avec les identifiants de périphérique SCSI 0 et 4.

Le fichier `/proc/partitions` contient des informations sur les partitions des lecteurs de disque reconnus. Pour chaque partition, il décrit les numéros de périphérique majeur et mineur, le nombre de blocs de 1024 octets qu'elle contient et le nom du périphérique correspondant à cette partition.

L'entrée `/proc/sys/dev/cdrom/info` donne diverses informations sur les fonctionnalités des lecteurs CD-ROM. Les informations n'ont pas besoin d'explication.

```
% cat /proc/sys/dev/cdrom/info
CD-ROM information, Id: cdrom.c 2.56 1999/09/09

drive name: hdc
drive speed: 48
drive # of slots: 0
Can close tray: 1
Can open tray: 1
Can lock tray: 1
Can change speed: 1
Can select disk: 0
```

```
Can read multisession:  1
Can read MCN: 1
Reports media changed:  1
Can play audio: 1
```

7.5.3 Points de montage

Le fichier `/proc/mounts` donne un aperçu des systèmes de fichiers montés. Chaque ligne correspond à un *descripteur de montage* et donne le périphérique monté, le point de montage et d'autres informations. Notez que `/proc/mounts` contient les mêmes informations que le fichier traditionnel `/etc/mtab`, qui est automatiquement mis à jour par la commande mount.

Voici les différents éléments d'un descripteur de point de montage :

- Le premier élément de la ligne est le périphérique monté (consultez le autorefchap :peripheriques).
- Le second élément est le point de montage, l'emplacement dans le système de fichiers racine où le contenu du système de fichiers apparaît. Pour le système de fichiers racine lui-même, le point de montage est `/`. Pour les espaces d'échange, le point de montage est noté `swap`.
- Le troisième élément est le type de système de fichiers. Actuellement, la plupart des systèmes GNU/Linux utilisent le système de fichiers `ext2` (ou `ext3`) pour les lecteurs de disques, mais des disques DOS ou Windows peuvent être montés et seront alors de type `fat` ou `vfat`. La plupart des CD-ROM sont au format `iso9660`. Consultez la page de manuel de la commande `mount` pour une liste des types de systèmes de fichiers.

Les deux derniers éléments des lignes de `/proc/mounts` sont toujours à 0 et n'ont pas de signification.

Consultez la page de manuel de `fstab` pour plus de détail sur le format des descripteurs de point de montage[6]. GNU/Linux dispose de fonction destinées à vous aider à analyser ces descripteurs ; consultez la page de manuel de la fonction getmntent pour plus d'informations sur leur utilisation.

7.5.4 Verrous

La Section 8.3, « *fcntl*: Verrous et opérations sur les fichiers », décrit l'utilisation de l'appel système `fcntl` pour manipuler des verrous en lecture ou en écriture sur les fichiers. Le fichier `/proc/locks` décrit tous les verrous de fichiers actuellement en place sur le système. Chaque ligne correspond à un verrou.

Pour les verrous créés avec `fcntl`, la ligne commence par `POSIX ADVISORY`. Le troisième champ vaut `WRITE` ou `READ`, selon le type de verrou. Le nombre qui suit correspond à l'identifiant du processus possédant le verrou. Les trois chiffres suivants, séparés par deux-points, sont les numéros majeur et mineur du périphérique sur lequel se trouve le fichier et le numéro d'inode, qui situe le fichier dans le système de fichiers. Le reste de la ligne est constitué de valeurs utilisées en interne par le noyau et qui ne sont généralement d'aucune utilité.

[6] Le fichier `/etc/fstab` décrit la configuration des points de montage statiques d'un système GNU/Linux.

Traduire le contenu de `/proc/locks` en informations utiles demande un petit travail d'investigation. Vous pouvez observer l'évolution de `/proc/locks` en exécutant le programme du Listing 8.2 qui crée un verrou en lecture sur le fichier `/tmp/test-file`.

```
% touch /tmp/test-file
% ./lock-file /tmp/test-file
fichier /tmp/test-file
ouverture de /tmp/test-file
verrouillage
verrouillé ; appuyez sur entrée pour déverrouiller...
```

Dans une autre fenêtre, observez le contenu de /proc/locks.

```
% cat /proc/locks
1: POSIX ADVISORY WRITE 5467 08:05:181288 0 2147483647 d1b5f740 00000000
dfea7d40 00000000 00000000
```

Il peut y avoir des lignes supplémentaires si d'autres programmes ont posé des verrous. Dans notre cas, 5467 est l'identifiant de processus du programme `lock-file`. Utilisez `ps` pour observer ce qu'est en train d'exécuter ce processus.

```
% ps 5467
  PID TTY    STAT TIME COMMAND
 5467 pts/28 S    0:00 ./lock-file /tmp/test-file
```

Le fichier verrouillé, `/tmp/test-file`, se trouve sur le périphérique avec les numéros majeur et mineur 8 et 5, respectivement. Ces nombres correspondent à `/dev/sda5`.

```
% df /tmp
Filesystem           1k-blocks    Used Available Use% Mounted on
/dev/sda5              8459764 5094292   2935736 63% /
% ls -l /dev/sda5
brw-rw----    1 root     disk     8,   5 May  5  1998 /dev/sda5
```

Le fichier `/tmp/test-file` se trouve au niveau de l'inode 181 288 sur ce périphérique.

```
% ls --inode /tmp/test-file
 181288 /tmp/test-file
```

Consultez la Section 6.2, « Numéros de périphérique » pour plus d'informations sur ceux-ci.

7.6 Statistiques système

Deux entrées de `/proc` contiennent des statistiques utiles sur le système. `/proc/loadavg` donne des informations sur sa charge. Les trois premiers nombres représentent le nombre de tâches actives sur le système – processus en cours d'exécution – avec une moyenne sur les 1, 5 et 15 dernières minutes. Le chiffre suivant indique le nombre courant de tâches exécutables – processus programmés pour être exécutés, à l'opposé de ceux bloqués dans un appel système – et le nombre total de processus sur le système. Le dernier champ correspond à l'identifiant du processus ayant eu la main le plus récemment.

Le fichier `/proc/uptime` contient le temps écoulé depuis le démarrage du système, ainsi que la durée pendant laquelle le système a été inactif. Ces deux valeurs sont données sous forme décimale, en secondes.

```
% cat /proc/uptime
3248936.18 3072330.49
```

Le programme du Listing 7.7 extrait le temps depuis lequel le système est démarré ainsi que sa durée d'inactivité et les affiche de façon lisible.

Listing 7.7 – (*print-uptime.c*) – **Affiche des Informations sur les Temps Système**

```
#include <stdio.h>

/* Affiche une durée sur la sortie standard de façon lisible. TIME est la
   durée, en secondes, et LABEL est une légende courte. */

void print_time (char* label, long time)
{
  /* Constantes de conversion. */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  /* Affiche la durée. */
  printf ("%s: %ld jours, %ld:%02ld:%02ld\n", label, time / day,
          (time % day) / hour, (time % hour) / minute, time % minute);
}

int main ()
{
  FILE* fp;
  double uptime, idle_time;
  /* Lit le temps écoulé depuis le démarrage et le temps d'inactivité à
     partir de /proc/uptime. */
  fp = fopen ("/proc/uptime", "r");
  fscanf (fp, "%lf %lf\n", &uptime, &idle_time);
  fclose (fp);
  /* L'affiche. */
  print_time ("Temps écoulé depuis le démarrage ", (long) uptime);
  print_time ("Temps d'inactivité               ", (long) idle_time);
  return 0;
}
```

La commande `uptime` et l'appel système `sysinfo` (décrit dans la Section 8.14, « *sysinfo*: récupération de statistiques système ») permettent également d'obtenir le temps écoulé depuis le démarrage. La commande `uptime` affiche également les moyennes de charge système contenues dans `/proc/loadavg`.

Chapitre 8

Appels système Linux

Jusqu'ici, nous avons présenté diverses fonctions que votre programme peut utiliser pour accomplir des actions relatives au système, comme analyser des options de ligne de commande, manipuler des processus et mapper de la mémoire. Si vous y regardez de plus près, vous remarquerez que ces fonctions se répartissent en deux catégories, selon la façon dont elles sont implantées.

- Une *fonction de bibliothèque* est une fonction ordinaire qui se trouve dans une bibliothèque externe à votre programme. La plupart des fonctions que nous avons présenté jusqu'ici se trouvent dans la bibliothèque standard du C, `libc`. Par exemple, `getopt_long` et `mkstemp` en font partie.
  Un appel à une fonction de bibliothèque est identique à l'appel de n'importe quelle autre fonction. Les arguments sont placés dans des registres du processeur ou sur la pile et l'exécution est transférée au début de la fonction qui se trouve généralement dans une bibliothèque partagée.
- Un *appel système* est implanté au sein du noyau Linux. Lorsqu'un programme effectue un appel système, les arguments sont mis en forme et transférés au noyau qui prend la main jusqu'à la fin de l'appel. Un appel système n'est pas identique à un appel de fonction classique et une procédure spécifique est nécessaire pour transférer le contrôle au noyau. Cependant, la bibliothèque C GNU (l'implémentation de la bibliothèque standard du C fournie avec les systèmes GNU/Linux) masque les appels systèmes par des fonctions classiques afin qu'ils soient plus simples à utiliser. Les fonctions d'E/S de bas niveau comme `open` ou `read` font partie des appels systèmes Linux.
  Les appels système Linux constituent l'interface de base entre les programmes et le noyau Linux. À chaque appel correspond une opération ou une fonctionnalité de base.
  Certains appels sont très puissants et influent au niveau du système. Par exemple, il est possible d'éteindre le système ou d'utiliser des ressources système tout en interdisant leur accès aux autres utilisateurs. De tels appels ne sont utilisables que par des programmes s'exécutant avec les privilèges superutilisateur (lancé par root). Ils échouent si le programme ne dispose pas de ces droits.

Notez qu'une fonction de bibliothèque peut à son tour appeler une ou plusieurs fonctions de bibliothèques ou appels système.

Linux propose près de 300 appels système. La liste des appels disponibles sur votre système se trouve dans le fichier `/usr/include/asm/unistd.h`. Certains ne sont destinés qu'à être utilisés en interne par le noyau et d'autres ne servent que pour l'implémentation de certaines bibliothèques. Dans ce chapitre, nous vous présenterons ceux qui nous semblent les plus susceptibles de vous servir.

La plupart sont déclarés dans le fichier d'en-tête `/usr/include/asm/unistd.h`.

8.1 Utilisation de *strace*

Avant de commencer à parler des appels système, il est nécessaire de présenter une commande qui peut vous en apprendre beaucoup sur les appels système. La commande `strace` trace l'exécution d'un autre programme en dressant la liste des appels système qu'il effectue et des signaux qu'il reçoit.

Pour observer l'enchaînement des appels système et des signaux d'un programme invoquez simplement `strace`, suivi du nom du programme et de ses arguments. Par exemple, pour observer les appels systèmes effectués par la commande `hostname`[1], utilisez cette commande :

```
% strace hostname
```

Elle vous affichera un certain nombre d'informations. Chaque ligne correspond à un appel système. Pour chaque appel, vous trouverez son nom suivi de ses arguments (ou de leur abréviation s'ils sont trop longs) et de sa valeur de retour. Lorsque c'est possible, `strace` utilise des noms symboliques pour les arguments et la valeur de retour plutôt que leur valeur numérique et affiche les différents champs des structures passées via un pointeur à l'appel système. Notez que strace ne montre pas les appels de fonctions classiques.

La première ligne affichée par strace hostname montre l'appel système execve qui invoque le programme `hostname`[2] :

```
execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0
```

Le premier argument est le nom du programme à exécuter ; le second sa liste d'arguments, constituée d'un seul élément ; et le troisième l'environnement du programme que `strace` n'affiche pas par souci de concision. Les 30 lignes suivantes, approximativement, font partie du mécanisme qui charge la bibliothèque standard du C à partir d'un fichier de bibliothèque partagé.

Les appels systèmes effectivement utilisés par le programme pour fonctionner se trouvent vers la fin de l'affichage. L'appel système `uname` permet d'obtenir le nom d'hôte du système à partir du noyau :

```
uname({sys="Linux", node="myhostname", ...}) = 0
```

[1] Invoquée sans aucune option, la commande `hostname` affiche simplement le nom d'hôte de la machine sur la sortie standard.

[2] Sous Linux, la famille de fonction `exec` est implémentée *via* l'appel système `execve`.

Remarquez que `strace` donne le nom des champs (`sys` et `node`) de la structure passée en argument. Cette structure est renseignée par l'appel système – Linux place le nom du système d'exploitation dans le champ `sys` et le nom d'hôte du système dans le champ `node`. L'appel système `uname` est détaillé dans la Section 8.15, « *uname* ».

Enfin, l'appel système `write` affiche les informations. Souvenez-vous que le descripteur de fichier 1 correspond à la sortie standard. Le troisième argument est le nombre de caractères à écrire et la valeur de retour est le nombre de caractères effectivement écrits.

```
write(1, "myhostname\n", 11) = 11
```

L'affichage peut paraître un peu confus lorsque vous exécutez `strace` car la sortie du programme `hostname` est mélangée avec celle de `strace`.

Si le programme que vous analysez affiche beaucoup d'informations, il est parfois plus pratique de rediriger la sortie de `strace` vers un fichier. Pour cela, utilisez l'option `-o nom_de_fichier`.

La compréhension de tout ce qu'affiche `strace` nécessite une bonne connaissance du fonctionnement du noyau Linux et de l'environnement d'exécution ce qui présente un intérêt limité pour les programmeurs d'application. Cependant, une compréhension de base est utile pour déboguer des problèmes sournois et comprendre le fonctionnement d'autres programmes.

8.2 *access* : Tester les permissions d'un fichier

L'appel système `access` détermine si le processus appelant à le droit d'accéder à un fichier. Il peut vérifier toute combinaison des permissions de lecture, écriture ou exécution ainsi que tester l'existence d'un fichier.

L'appel `access` prend deux argument. Le premier est le chemin d'accès du fichier à tester. Le second un OU binaire entre `R_OK`, `W_OK` et `X_OK`, qui correspondent aux permissions en lecture, écriture et exécution. La valeur de retour est zéro si le processus dispose de toutes les permissions passées en paramètre. Si le fichier existe mais que le processus n'a pas les droits dessus, `access` renvoie -1 et positionne `errno` à `EACCES` (ou `EROFS` si l'on a testé les droits en écriture d'un fichier situé sur un système de fichiers en lecture seule).

Si le second argument est `F_OK`, `access` vérifie simplement l'existence du fichier. Si le fichier existe, la valeur de retour est 0 ; sinon, elle vaut -1 et `errno` est positionné à `ENOENT`. `errno` peut également être positionné à `EACCES` si l'un des répertoires du chemin est inaccessible.

Le programme du Listing 8.1 utilise `access` pour vérifier l'existence d'un fichier et déterminer ses permissions en lecture et en écriture. Spécifiez le nom du fichier à vérifier sur la ligne de commande.

Listing 8.1 – (***check-access.c***) – **Vérifier les Droits d'Accès à un Fichier**

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main (int argc, char* argv[])
6  {
7    char* path = argv[1];
8    int rval;
9
```

```
10    /* Vérifie l'existence du fichier. */
11    rval = access (path, F_OK);
12    if (rval == 0)
13      printf ("%s existe\n", path);
14    else {
15      if (errno == ENOENT)
16         printf ("%s n'existe pas\n", path);
17      else if (errno == EACCES)
18         printf ("%s n'est pas accessible\n", path);
19      return 0;
20    }
21
22    /* Vérifie l'accès en lecture. */
23    rval = access (path, R_OK);
24    if (rval == 0)
25      printf ("%s est accessible en lecture\n", path);
26    else
27      printf ("%s n'est pas accessible en lecture (accès refusé)\n", path);
28
29    /* Vérifie l'accès en écriture. */
30    rval = access (path, W_OK);
31    if (rval == 0)
32      printf ("%s est accessible en écriture\n", path);
33    else if (errno == EACCES)
34      printf ("%s n'est pas accessible en écriture (accès refusé)\n", path);
35    else if (errno == EROFS)
36      printf ("%s n'est pas accessible en écriture (SF en lecture seule)\n", path);
37    return 0;
38  }
```

Par exemple, pour tester les permissions d'accès à un fichier appelé `README` situé sur un CD-ROM, invoquez le programme comme ceci :

```
% ./check-access /mnt/cdrom/README
/mnt/cdrom/README existe
/mnt/cdrom/README est accessible en lecture
/mnt/cdrom/README n'est pas accessible en écriture (SF en lecture seule)
```

8.3 *fcntl* : Verrous et opérations sur les fichiers

L'appel système `fcntl` est le point d'accès de plusieurs opérations avancées sur les descripteurs de fichiers. Le premier argument de `fcntl` est un descripteur de fichiers ouvert et le second est une valeur indiquant quelle opération doit être effectuée. Pour certaines d'entre-elles, `fcntl` prend un argument supplémentaire. Nous décrirons ici l'une des opérations les plus utiles de `fcntl` : le verrouillage de fichier. Consultez la page de manuel de `fcntl` pour plus d'informations sur les autres opérations.

L'appel système `fcntl` permet à un programme de placer un verrou en lecture ou en écriture sur un fichier, d'une façon similaire à celle utilisée pour les verrous mutex traités dans le Chapitre 4, « Threads ». Un verrou en lecture se place sur un descripteur de fichier accessible en lecture et un verrou en écriture sur un descripteur de fichier accessible en écriture. Plusieurs processus peuvent détenir un verrou en lecture sur le même fichier au même moment, mais un seul peut détenir un verrou en écriture et le même fichier ne peut pas être verrouillé à la fois en lecture et en écriture. Notez que le fait de placer un verrou n'empêche pas réellement les autres

processus d'ouvrir le fichier, d'y lire des données ou d'y écrire, à moins qu'il ne demandent eux aussi un verrou avec `fcntl`.

Pour placer un verrou sur un fichier, il faut tout d'abord créer une variable `struct flock` et la remplir de zéros. Positionnez le champ `l_type` de la structure à `F_RDLCK` pour un verrou en lecture ou `F_WRLCK` pour un verrou en écriture. Appelez ensuite `fcntl` en lui passant le descripteur du fichier à verrouiller, le code d'opération `F_SETLKW` et un pointeur vers la variable `struct flock`. Si un autre processus détient un verrou qui empêche l'acquisition du nouveau, l'appel à `fcntl` bloque jusqu'à ce que ce verrou soit relâché.

Le programme du Listing 8.2 ouvre en écriture le fichier dont le nom est passé en paramètre puis place un verrou en écriture dessus. Le programme attend ensuite que l'utilisateur appuie sur la touche Entrée puis déverrouille et ferme le fichier.

Listing 8.2 – (***lock-file.c***) – **Crée un Verrou en Écriture avec *fcntl***

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  int main (int argc, char* argv[])
7  {
8    char* file = argv[1];
9    int fd;
10   struct flock lock;
11
12   printf ("ouverture de %s\n", file);
13   /* Ouvre un descripteur de fichier. */
14   fd = open (file, O_WRONLY);
15   printf ("verrouillage\n");
16   /* Initialise la structure flock. */
17   memset (&lock, 0, sizeof(lock));
18   lock.l_type = F_WRLCK;
19   /* Place un verrou en écriture sur le fichier. */
20   fcntl (fd, F_SETLKW, &lock);
21
22   printf ("verrouillé ; appuyez sur Entrée pour déverrouiller... ");
23   /* Attend l'appui sur Entrée. */
24   getchar ();
25
26   printf ("déverrouillage\n");
27   /* Libère le verrou. */
28   lock.l_type = F_UNLCK;
29   fcntl (fd, F_SETLKW, &lock);
30
31   close (fd);
32   return 0;
33 }
```

Compilez et lancez le programme sur un fichier test – par exemple, `/tmp/fichier-test` – comme suit :

```
% cc -o lock-file lock-file.c
% touch /tmp/fichier-test
% ./lock-file /tmp/fichier-test
ouverture de /tmp/fichier-test
verrouillage
verrouillé ; appuyez sur Entrée pour déverouiller...
```

Maintenant, dans une autre fenêtre, essayez de le lancer sur le même fichier :

```
% ./lock-file /tmp/fichier-test
ouverture de /tmp/fichier-test
verrouillage
```

Notez que la seconde instance est bloqué lors de la tentative de verrouillage du fichier. Revenez dans la première fenêtre et appuyez sur Entrée :

```
Déverrouillage
```

Le programme s'exécutant dans la seconde fenêtre acquiert immédiatement le verrou.

Si vous préférez que `fcntl` ne soit pas bloquant si le verrou ne peut pas être obtenu, utilisez `F_SETLK` au lieu de `F_SETLKW`. Si le verrou ne peut pas être acquis, `fcntl` renvoie `-1` immédiatement.

Linux propose une autre implémentation du verrouillage de fichiers avec l'appel `flock`. La fonction `fcntl` dispose d'un avantage majeur : elle fonctionne sur les fichiers se trouvant sur un système de fichiers NFS[3] (si tant est que le serveur NFS soit relativement récent et correctement configuré). Ainsi, si vous disposez de deux machines qui ont toutes deux le même système de fichiers monté via NFS, vous pouvez reproduire l'exemple ci-dessus en utilisant deux machines différentes. Lancez `lock-file` sur une machine en lui passant un fichier situé sur le système de fichiers NFS puis lancez le sur la seconde en lui passant le même fichier. NFS relance le second programme lorsque le verrou est relâché par le premier.

8.4 *fsync* et *fdatasync* : purge des tampons disque

Sur la plupart des système d'exploitation, lorsque vous écrivez dans un fichier, les données ne sont pas immédiatement écrites sur le disque. Au lieu de cela, le système d'exploitation met en cache les données écrites dans un tampon en mémoire, pour réduire le nombre d'écritures disque requises et améliorer la réactivité du programme. Lorsque le tampon est plein ou qu'un événement particulier survient (par exemple, au bout d'un temps donné), le système écrit les données sur le disque.

Linux fournit un système de mise en cache de ce type. Normalement, il s'agit d'une bonne chose en termes de performances. Cependant, ce comportement peut rendre instables les programmes qui dépendent de l'intégrité de données stockées sur le disque. Si le système s'arrête soudainement – par exemple, en raison d'un crash du noyau ou d'une coupure de courant – toute donnée écrite par le programme qui réside dans le cache en mémoire sans avoir été écrite sur le disque est perdue.

Par exemple, supposons que vous écriviez un programme de gestion de transactions qui tient un fichier journal. Ce dernier contient les enregistrements concernant toutes les transactions qui ont été traitées afin que si une panne système survient, le statut des données impliquées par les transactions puisse être restauré. Il est bien sûr important de préserver l'intégrité du fichier journal – à chaque fois qu'une transaction a lieu, son entrée dans le journal doit être envoyée immédiatement sur le disque dur.

[3] *Network File System* (NFS) est une technologie de partage de fichiers courante, comparable aux partages et aux lecteurs réseau Windows.

Pour faciliter l'implémentation de tels mécanismes, Linux propose l'appel système `fsync`. Celui-ci prend un seul argument, un descripteur de fichier ouvert en écriture, et envoie sur le disque toutes les données écrites dans le fichier. L'appel `fsync` ne se termine pas tant que les données n'ont pas été physiquement écrites.

La fonction du Listing 8.3 illustre l'utilisation de `fsync`. Il écrit un enregistrement d'une ligne dans un fichier journal.

Listing 8.3 – (***write_journal_entry.c***) – **Écrit et Synchronise un enregistrement**

```
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

const char* journal_filename = "journal.log";

void write_journal_entry (char* entry)
{
  int fd = open (journal_filename, O_WRONLY | O_CREAT | O_APPEND, 0660);
  write (fd, entry, strlen (entry));
  write (fd, "\n", 1);
  fsync (fd);
  close (fd);
}
```

Un autre appel système, `fdatasync`, a la même fonction. Cependant, alors que `fsync` garantit que la date de modification du fichier sera mise à jour, ce n'est pas le cas de `fdatasync` ; ce dernier ne garantit que le fait que les données seront écrites. Cela signifie qu'en général, `fdatasync` peut s'exécuter plus vite que `fsync` car il n'a qu'une seule écriture à effectuer au lieu de deux.

Cependant, sur les version actuelles de Linux, ces deux appels système font en fait la même chose, mettant tous deux à jour la date de modification du fichier.

L'appel système `fsync` vous permet de forcer explicitement l'écriture d'un tampon. Vous pouvez également ouvrir un fichier en mode entrées/sorties synchrones, ce qui signifie que toutes les écritures sont envoyées sur le disque immédiatement. Pour cela, passez l'option `O_SYNC` lors de l'ouverture du fichier avec `open`.

8.5 *getrlimit* et *setrlimit* : limites de ressources

Les appels système `getrlimit` et `setrlimit` permettent à un processus de connaître et de définir des limites sur les ressources système qu'il peut consommer. Vous connaissez peut être la commande shell `ulimit`, qui vous permet de restreindre la consommation de ressources des programmes que vous exécutez[4] ; ces appels système permettent à un programme de faire la même chose par programmation.

Pour chaque ressource il existe deux limites, la *limite stricte* et la *limite souple*. La limite souple ne doit jamais dépasser la limite dure. Typiquement, une application réduira la limite souple pour éviter une montée en puissance de sa consommation de ressources.

[4]Consultez la page de manuel de votre shell pour plus d'informations sur `ulimit`.

`getrlimit` et `setrlimit` prennent tous deux en argument un code spécifiant le type de limite de ressource et un pointeur vers une variable `struct rlimit`. L'appel `getrlimit` renseigne les champs de cette structure, tandis que `setrlimit` modifie la limite selon son contenu. La structure `rlimit` a deux champs : `rlim_cur` qui est la limite souple et `rlim_max` qui est la limite stricte.

Voici une liste des limites de ressources pouvant être modifiées les plus utiles, avec le code correspondant :

- `RLIMIT_CPU` – Temps processeur maximum, en secondes, utilisé par un programme. Il s'agit du temps pendant lequel le programme utilise effectivement le processeur, qui n'est pas forcément le temps d'exécution du programme. Si le programme dépasse cette limite, il est interrompu par un signal `SIGXCPU`.
- `RLIMIT_DATA` – Quantité maximale de mémoire qu'un programme peut allouer pour ses données. Toute allocation au-delà de cette limite échouera.
- `RLIMIT_NPROC` – Nombre maximum de processus fils pouvant être exécutés par l'utilisateur. Si le processus appel `fork` et que trop de processus appartenant à l'utilisateur sont en cours d'exécution, `fork` échouera.
- `RLIMIT_NOFILE` – Nombre maximum de descripteurs de fichiers que le processus peut ouvrir en même temps.

Consultez la page de manuel de `setrlimit` pour une liste complète des ressources système.

Le programme du Listing 8.4 illustre l'utilisation de la limite de temps processeur consommé par un programme. Il définit un temps processeur de une seconde puis entre dans une boucle infinie. Linux tue le processus peu après, lorsqu'il dépasse la seconde de temps processeur.

Listing 8.4 – (***limit-cpu.c***) – **Démonstration de la Limite de Temps Processeur**

```
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

int main ()
{
  struct rlimit rl;

  /* Récupère la limite courante. */
  getrlimit (RLIMIT_CPU, &rl);
  /* Définit une limite de temps processeur d'une seconde. */
  rl.rlim_cur = 1;
  setrlimit (RLIMIT_CPU, &rl);
  /* Occupe le programme. */
  while (1);

  return 0;
}
```

Lorsque le programme est terminé par `SIGXCPU`, le shell affiche un message interprétant le signal :

```
% ./limit_cpu
Temps UCT limite expiré
```

8.6 *getrusage* : statistiques sur les processus

L'appel système `getrusage` obtient des statistiques sur un processus à partir du noyau. Il peut être utilisé pour obtenir des statistiques pour le processus courant en passant `RUSAGE_SELF` comme premier argument ou pour les processus fils terminés qui ont été créés par ce processus et ses fils en passant `RUSAGE_CHILDREN`. Le second argument de `getrusage` est un pointeur vers une variable de type `struct rusage`, qui est renseignée avec les statistiques.

Voici quelques-uns des champs les plus intéressant d'une struct `rusage` :

- `ru_utime` – Champ de type `struct timeval` contenant la quantité de temps utilisateur, en secondes, que le processus a utilisé. Le temps utilisateur est le temps processeur passé à exécuté le programme par opposition à celui passé dans le noyau pour des appels système.
- `ru_stime` – Champ de type `struct timeval` contenant la quantité de temps système, en seconde, que le processus a utilisé. Le temps système est le temps processeur passé à exécuté des appels système à la demande du processus. `ru_maxrss` – Quantité maximale de mémoire physique occupée par le processus au cours son exécution.

La page de manuel de `getrusage` liste tous les champs disponibles. Consultez la Section 8.7, « *gettimeofday*: heure système » pour plus d'informations sur le type `struct timeval`.

La fonction du Listing 8.5 affiche les temps système et utilisateur du processus en cours.

Listing 8.5 – (***print-cpu-times.c***) – **Affiche les Temps Utilisateur et Processeur**

```
1  #include <stdio.h>
2  #include <sys/resource.h>
3  #include <sys/time.h>
4  #include <unistd.h>
5
6  void print_cpu_time()
7  {
8    struct rusage usage;
9    getrusage (RUSAGE_SELF, &usage);
10   printf ("Temps processeur : %ld.%06lds utilisateur, %ld.%06lds système\n",
11           usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
12           usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
13 }
```

8.7 *gettimeofday* : heure système

L'appel système `gettimeofday` renvoie l'heure système. Il prend un pointeur vers une variable de type `struct timeval`. Cette structure représente un temps, en secondes, séparé en deux champs. Le champ `tv_sec` contient la partie entière du nombre de secondes et le champ `tv_usec` la fraction de microsecondes. La valeur de la `struct timeval` représente le nombre de secondes écoulé depuis le début de l'epoch UNIX, c'est-à-dire le premier janvier 1970 à minuit UTC. L'appel `gettimeofday` prend également un second argument qui doit être `NULL`. Incluez `<sys/time.h>` si vous utilisez cet appel système.

Le nombre de secondes depuis l'epoch UNIX n'est généralement pas une façon très pratique de représenter les dates. Les fonctions de la bibliothèque standard `localtime` et `strftime` aident à manipuler les valeurs renvoyées par `gettimeofday`. La fonction `localtime` prend un pointeur vers un nombre de secondes (le champ `tv_sec` de `struct timeval`) et renvoie un pointeur vers

un objet `struct tm`. Cette structure contient des champs plus utiles renseignés selon le fuseau horaire courant :

- `tm_hour`, `tm_min`, `tm_sec` – Heure du jour en heures, minutes et secondes.
- `tm_year`, `tm_mon`, `tm_day` – Année, mois, jour.
- `tm_wday` – Jour de la semaine. Zéro représente le Dimanche.
- `tm_yday` – Jour de l'année.
- `tm_isdst` – Drapeau indiquant si l'heure d'été est en vigueur ou non.

La fonction `strftime` permet de produire à partir d'un pointeur vers une `struc tm` une chaîne personnalisée et formatée représentant la date et l'heure. Le format est spécifié d'une façon similaire à `printf`, sous forme d'une chaîne contenant des codes qui indiquent les champs à inclure. Par exemple, la chaîne de format

```
"%Y-%m-%d %H:%M:%S"
```

Correspond à une date de la forme :

```
2006-07-15 21:00:42
```

Passez à `strftime` un tampon pour recevoir la chaîne, la longueur du tampon, la chaîne de format et un pointeur vers une variable `struct tm`. Consultez la page de manuel de `strftime` pour une liste complète des codes qui peuvent être utilisés dans la chaîne de format. Notez que ni `localtime` ni `strftime` ne prennent en compte la partie fractionnaire de l'heure courante avec une précision supérieure à la seconde. Si vous voulez exploiter le champ `tv_usec` de la `struct timeval`, vous devrez le faire manuellement.

Incluez `<time.h>` si vous appelez `localtime` ou `strftime`.

La fonction du Listing 8.6 affiche la date et l'heure courante, avec une précision de l'ordre de la milliseconde.

Listing 8.6 – (***print-time.c***) – **Affiche la Date et l'Heure**

```
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <time.h>
4  #include <unistd.h>
5
6  void print_time ()
7  {
8    struct timeval tv;
9    struct tm* ptm;
10   char time_string[40];
11   long milliseconds;
12
13   /* Récupère l'heure courante et la convertit en struct tm. */
14   gettimeofday (&tv, NULL);
15   ptm = localtime (&tv.tv_sec);
16   /* Formate la date et l'heure à la seconde près. */
17   strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
18   /* Calcule les millisecondes à partir des microsecondes. */
19   milliseconds = tv.tv_usec / 1000;
20   /* Affiche l'heure de façon formatée, en secondes, suivie d'un point
21      décimal et des millisecondes. */
22   printf ("%s.%03ld\n", time_string, milliseconds);
23 }
```

8.8 La famille *mlock* : verrouillage de la mémoire physique

La famille d'appels système `mlock` permet à un programme de verrouiller tout ou partie de son espace d'adressage en mémoire physique. Cela évite que Linux ne l'envoie vers l'espace d'échange, même si le programme n'y accède pas pendant quelques temps.

Un programme pour lequel le temps est une ressource critique peut verrouiller la mémoire physique car le temps nécessaire au processus d'échange peut être trop long ou trop imprévisible. Un programme sensible au niveau de la sécurité pourrait vouloir empêcher l'envoi de données critiques vers un espace d'échange à partir duquel elles pourraient être récupérées après la fin du programme.

Le verrouillage d'une région de la mémoire consiste simplement à appeler `mlock` en lui passant un pointer vers le début de la région ainsi que la longueur de la région. Linux divise la mémoire en pages et ne peut verrouiller que des pages dans leur intégralité ; chaque page qui contient une partie de la région de la mémoire passée à `mlock` est verrouillée. La fonction `getpagesize` renvoie la taille de page du système qui est de 4Ko sous Linux x86.

Par exemple pour allouer 32Mo d'espace d'adressage et le verrouiller en RAM, vous utiliserez ce code :

```
const int alloc_size = 32 * 1024 * 1024;
char* memory = malloc (alloc_size);
mlock (memory, alloc_size);
```

Notez que le simple fait d'allouer une page mémoire et de la verrouiller avec `mlock` ne réserve pas de mémoire physique pour le processus appelant car les pages peuvent être en copie à l'écriture[5]. Vous devriez donc écrire une valeur quelconque sur chaque page :

```
size_t i;
size_t page_size = getpagesize ();
for (i = 0; i < alloc_size; i += page_size)
  memory[i] = 0;
```

Le fait d'écrire sur toutes les pages force Linux à allouer une page mémoire unique, non partagée, au processus pour chacune.

Pour déverrouiller une région, appelez `munlock`, qui prend les mêmes arguments que `mlock`.

Si vous voulez que l'intégralité de l'espace d'adressage de votre programme soit verrouillé en mémoire physique, appelez `mlockall`. Cet appel système ne prend qu'un seul argument : `MCL_CURRENT` verrouille toute la mémoire allouée mais les allocations suivantes ne sont pas verrouillées ; `MCL_FUTURE` verrouille toutes les pages qui sont allouées après l'appel. Utilisez `MCL_CURRENT|MCL_FUTURE` pour verrouiller en mémoire à la fois les allocation en cours et les futures allocations.

Le verrouillage de grandes quantités de mémoire, en particulier en via `mlockall`, peut être dangereux pour tout le système Linux. Le verrouillage de mémoire sans discernement est un bon moyen de ralentir considérablement le système au point de le faire s'arrêter car les autres processus en cours d'exécution doivent s'arranger avec des ressources en mémoire physique moindres et avec le fait d'être envoyé et repris depuis l'espace d'échange. Si vous verrouillez

[5]La *copie à l'écriture* (copy on write) signifie que Linux ne fait une copie privée de la page que lorsque le processus écrit une valeur à l'intérieur.

trop de mémoire, le système sera totalement à court de mémoire et Linux commencera à tuer des processus.

Pour cette raison, seuls les processus avec les privilèges de superutilisateur peuvent verrouiller de la mémoire avec `mlock` ou `mlockall`. Si un processus ne disposant pas de ces privilèges appel l'une de ces fonctions, elle échouera en renvoyant -1 et en positionnant `errno` à `EPERM`.

L'appel `munlockall` déverrouille toute la mémoire verrouillée par le processus courant, qu'elle ait été verrouillée par `mlock` ou `mlockall`.

Un moyen pratique de surveiller l'utilisation mémoire de votre programme est d'utiliser la commande `top`. La colonne `VIRT` indique la taille de l'espace d'adressage virtuel de chaque programme (cette taille inclut le code, les données et la pile dont certains peuvent être envoyés vers l'espace d'échange). La colonne `RES` (pour resident size) indique la quantité de mémoire physique effectivement occupée par le programme. La somme de toutes les valeurs présentes dans la colonne `RES` pour tous les programmes en cours d'exécution ne peut pas excéder la taille de la mémoire physique de votre ordinateur et la somme de toutes les tailles d'espace d'adressage est limitée à 2Go[6] (pour les versions 32 bits de Linux).

Incluez `<sys/mman.h>` si vous utilisez l'un des appels système `mlock`.

8.9 *mprotect* : définir des permissions mémoire

Dans la Section 5.3, « Mémoire mappée », nous avons montré comment utiliser l'appel système `mmap` pour mettre en correspondance un fichier avec la mémoire. Souvenez vous que le troisième argument de `mmap` est un ou binaire entre les indicateurs de protection mémoire `PROT_READ`, `PROT_WRITE` et `PROT_EXEC` pour des permissions en lecture, écriture ou exécution, respectivement, ou `PROT_NONE` pour empêcher l'accès à la mémoire. Si un programme tente d'effectuer une opération sur un emplacement mémoire sur lequel il n'a pas les bonnes permissions, il se termine sur la réception d'un signal `SIGSEGV` (erreur de segmentation).

Une fois que la mémoire a été mappée, ces permissions peuvent être modifiée par l'appel système `mprotect`. Les arguments de `mprotect` sont l'adresse d'une région mémoire, la taille de cette région et un jeu d'indicateurs de protection. La région mémoire consiste en un ensemble de pages complètes : l'adresse de la région doit être alignée sur la taille de page système et la longueur de la région doit être un multiple de la taille de page. Les indicateurs de protection de ces pages sont remplacés par la valeur passée en paramètre.

Par exemple, supposons que votre programme mappe `/dev/zero` pour allouer une page mémoire, comme décrit dans la Section 5.3.5, « Autres utilisations de *mmap* ». La mémoire est initialement en lecture/écriture.

```
int fd = open ("/dev/zero", O_RDONLY);
char* memory = mmap (NULL, page_size, PROT_READ | PROT_WRITE,
                     MAP_PRIVATE, fd, 0);
close (fd);
```

Plus loin, votre programme peut protéger la mémoire en écriture en appelant `mprotect` :

```
mprotect (memory, page_size, PROT_READ);
```

[6] NdT Cette limite n'est plus d'actualité avec les séries 2.4 et 2.6.

Obtenir de la Mémoire Alignée sur une Page

Notez que les régions mémoire renvoyées par `malloc` ne sont généralement pas alignées sur des pages, même si la taille de la mémoire est un multiple de la taille de page. Si vous voulez protéger de la mémoire obtenue via `malloc`, vous devez allouer une région plus importante que celle désirée et trouver une sous-région qui soit alignée sur une page.
Vous pouvez également utiliser l'appel système `mmap` pour court-circuiter `malloc` et allouer de la mémoire alignée sur des pages directement à partir du noyau Linux. Consultez la Section 5.3, « Mémoire mappée », pour plus de détails.

une technique avancée pour surveiller les accès mémoire est de protéger une région mémoire avec `mmap` ou `mprotect` puis de gérer le signal `SIGSEGV` que Linux envoie au programme lorsqu'il tente d'accéder à cette mémoire. L'exemple du Listing 8.7 illustre cette technique.

Listing 8.7 – (*mprotect.c*) – **Détecter un Accès Mémoire en Utilisant *mprotect***

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

static int alloc_size;
static char* memory;

void segv_handler (int signal_number)
{
  printf ("accès mémoire !\n");
  mprotect (memory, alloc_size, PROT_READ | PROT_WRITE);
}

int main ()
{
  int fd;
  struct sigaction sa;

  /* Installe segv_handler comme gestionnaire de signal SIGSEGV.  */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &segv_handler;
  sigaction (SIGSEGV, &sa, NULL);

  /* Alloue une page de mémoire en mappant /dev/zero. Mappe la mémoire
     en écriture seule initialement. */
  alloc_size = getpagesize ();
  fd = open ("/dev/zero", O_RDONLY);
  memory = mmap (NULL, alloc_size, PROT_WRITE, MAP_PRIVATE, fd, 0);
  close (fd);
  /* Écrit sur la page pour en obtenir une copie privée. */
  memory[0] = 0;
  /* Protège la mémoire en écriture. */
  mprotect (memory, alloc_size, PROT_NONE);

```

```
40    /* Écrit dans la région qui vient d'être allouée. */
41    memory[0] = 1;
42
43    /* Terminé ; libère la mémoire. */
44    printf ("Fini\n");
45    munmap (memory, alloc_size);
46    return 0;
47  }
```

Le programme effectue les opérations suivantes :

1. Il déclare un gestionnaire de signal pour `SIGSEGV` ;
2. Il alloue une page mémoire en mappant `/dev/zero` et en écrivant une valeur dans la page obtenue pour en obtenir une copie privée.
3. Il protège la mémoire en appelant mprotect avec l'option `PROT_NONE` ;
4. Lorsque le programme tente d'écrire en mémoire, Linux envoie un `SIGSEGV` qui est pris en charge par `segv_handler`. Le gestionnaire de signal supprime la protection de la mémoire ce qui autorise l'accès ;
5. Lorsque le gestionnaire de signal se termine, le contrôle repasse à `main`, où le programme libère la mémoire via `munmap`.

8.10 *nanosleep* : pause en haute précision

L'appel système `nanosleep` est une version en haute précision de l'appel UNIX `sleep`. Au lieu de suspendre l'exécution pendant un nombre entier de secondes, `nanosleep` prend comme argument un pointeur vers un objet de type `struct timespec`, qui peut indiquer un temps à la nanoseconde près. Cependant, en raison de détails d'implémentation du noyau Linux, la précision fournie par nanosleep n'est que de 10 millisecondes – c'est toujours mieux que celle offerte par `sleep`. Cette précision supplémentaire peut être utile, par exemple, pour ordonnancer des opérations fréquentes avec de faibles intervalles de temps entre elles.

La structure `struct timespec` a deux champs : `tv_sec`, le nombre entier de secondes et `tv_nsec` , un nombre supplémentaire de nanosecondes. La valeur de `tv_nsec` doit être inférieure à 10^9.

L'appel `nanosleep` offre un autre avantage par rapport à `sleep`. Comme pour `sleep`, l'arrivée d'un signal interrompt l'exécution de `nanosleep`, qui positionne alors `errno` à `EINTR` et renvoie -1. Cependant, `nanosleep` prend un second argument, un autre pointeur vers un objet `struct timespec`, qui, s'il n'est pas `NULL`, est renseigné avec le temps de pause qu'il restait à faire (c'est-à-dire la différence entre le temps de suspension demandé le temps de suspension effectif). Cela facilite la reprise de l'opération de suspension.

La fonction du Listing 8.8 fournit une implémentation alternative de `sleep`. Contrairement à l'appel système classique, cette fonction prend en paramètre une valeur en virgule flottante correspondant à la durée en secondes pour laquelle il faut suspendre l'exécution et reprend l'opération de suspension si elle est interrompue.

Listing 8.8 – (***better-sleep.c***) – **Fonction de Suspension Haute précision**

```
1  #include <errno.h>
2  #include <time.h>
```

```
3
4  int better_sleep (double sleep_time)
5  {
6    struct timespec tv;
7    /* Construit l'objet timespec à partir du nombre entier de seconde...  */
8    tv.tv_sec = (time_t) sleep_time;
9    /* ... et le reste en nanosecondes. */
10   tv.tv_nsec = (long) ((sleep_time - tv.tv_sec) * 1e+9);
11
12   while (1)
13   {
14     /* Suspend l'exécution pour un temps spécifié par tv. Si l'on est
15        interrompu par un signal, le temps restant est replacé dans tv. */
16     int rval = nanosleep (&tv, &tv);
17     if (rval == 0)
18       /* On a suspendu l'exécution pour le temps demandé ; terminé. */
19       return 0;
20     else if (errno == EINTR)
21       /* Interrompu par un signal. Réessaie. */
22       continue;
23     else
24       /* Autre erreur, abandon. */
25       return rval;
26   }
27   return 0;
28 }
```

8.11 *readlink* : lecture de liens symboliques

L'appel système `readlink` récupère la cible d'un lien symbolique. Il prend trois arguments : le chemin vers le lien symbolique, un tampon pour recevoir la cible du lien et sa longueur. Une particularité de `readlink` est qu'il n'insère pas de caractère `NUL` à la fin de la chaîne qu'il place dans le tampon. Cependant, il renvoie le nombre de caractères composant le chemin cible, placer soi-même le caractère `NUL` est donc simple.

Si le premier argument de `readlink` pointe vers un fichier qui n'est pas un lien symbolique, `readlink` positionne `errno` à `EINVAL` et renvoie -1.

Le petit programme du Listing 8.9 affiche la cible du lien symbolique passé sur la ligne de commande.

Listing 8.9 – (*print-symlink.c*) – **Affiche la Cible d'un Lien Symbolique**

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main (int argc, char* argv[])
6  {
7    char target_path[256];
8    char* link_path = argv[1];
9
10   /* Tente de lire la cible du lien symbolique. */
11   int len = readlink (link_path, target_path, sizeof (target_path));
12   if (len == -1) {
13     /* L'appel a échoué. */
14     if (errno == EINVAL)
```

```
15        /* Il ne s'agit pas d'un lien symbolique ; en informe l'utlisateur. */
16        fprintf (stderr, "%s n'est pas un lien symbolique\n", link_path);
17      else
18        /* Autre problème ; affiche un message générique. */
19        perror ("readlink");
20      return 1;
21    }
22    else {
23      /* Place un caractère NUL à la fin de la cible. */
24      target_path[len] = '\0';
25      /* L'affiche. */
26      printf ("%s\n", target_path);
27      return 0;
28    }
29  }
```

Par exemple, voici comment créer un lien symbolique et utiliser `print-symlink` pour le lire :

```
% ln -s /usr/bin/wc my_link
% ./print-symlink my_link
/usr/bin/wc
```

8.12 *sendfile* : transferts de données rapides

L'appel système `sendfile` constitue un mécanisme efficace pour copier des données entre deux descripteurs de fichier. Les descripteurs de fichiers peuvent pointer vers un fichier sur le disque, un socket ou tout autre dispositif.

Typiquement, pour copier des données d'un descripteur de fichier vers un autre, un programme alloue un tampon de taille fixe, y copie des données provenant d'un des descripteurs, l'écrit sur l'autre et recommence jusqu'à ce que toutes les données aient été écrites. Ce procédé n'est pas efficace que ce soit en termes de temps ou d'espace car il nécessite l'utilisation de mémoire supplémentaire pour le tampon et ajoute une copie intermédiaire pour le remplir.

En utilisant `sendfile`, le tampon intermédiaire peut être supprimé. Appelez `sendfile` en lui passant le descripteur de fichier de destination, le descripteur source, un pointeur vers une variable de déplacement et le nombre d'octets à transférer. La variable de déplacement contient le déplacement à partir duquel lire le fichier source (0 correspond au début du fichier) et est mis à jour avec la position au sein du fichier à l'issue du transfert. La valeur de retour contient le nombre d'octets transférés. Incluez `<sys/sendfile.h>` dans votre programme s'il utilise `sendfile`.

Le programme du Listing 8.10 est une implémentation simple mais extrêmement efficace de copie de fichier. Lorsqu'il est invoqué avec deux noms de fichiers sur la ligne de commande, il copie le contenu du premier dans le second. Il utilise `fstat` pour déterminer la taille, en octets, du fichier source.

Listing 8.10 – (*copy.c*) – **Copie de Fichier avec sendfile**

```
1  #include <fcntl.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <sys/sendfile.h>
5  #include <sys/stat.h>
6  #include <sys/types.h>
7  #include <unistd.h>
```

```
 8
 9  int main (int argc, char* argv[])
10  {
11    int read_fd;
12    int write_fd;
13    struct stat stat_buf;
14    off_t offset = 0;
15
16    /* Ouvre le fichier source. */
17    read_fd = open (argv[1], O_RDONLY);
18    /* Evalue le fichier afin d'obtenir sa taille. */
19    fstat (read_fd, &stat_buf);
20    /* Ouvre le fichier de destination en écriture avec les mêmes permissions
21       que le fichier source. */
22    write_fd = open (argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);
23    /* Transfère les octets d'un fichier à l'autre. */
24    sendfile (write_fd, read_fd, &offset, stat_buf.st_size);
25    /* Ferme tout. */
26    close (read_fd);
27    close (write_fd);
28
29    return 0;
30  }
```

L'appel `sendfile` peut être utilisé dans de multiples occasions pour améliorer l'efficacité des copies. Un bon exemple est un serveur Web ou tout autre service réseau, qui envoie le contenu d'un fichier à un client via le réseau. Typiquement, ce genre de programme reçoit une requête depuis un socket connecté à la machine cliente. Le programme serveur ouvre le fichier local à transférer et écrit son contenu sur un socket réseau. Utiliser `sendfile` peut accélérer cette opération de façon significative. D'autres facteurs ont une influence sur l'efficacité du transfert, comme le paramétrage correct du socket. Cependant, ils sortent du cadre de ce livre.

8.13 *setitimer* : créer des temporisateurs

L'appel système `setitimer` est une généralisation de la fonction `alarm`. Il programme l'envoi d'un signal au programme après écoulement une période de temps donnée.

Un programme peut créer trois types différents de temporisateurs :

- Si le code temporisateur est `ITIMER_REAL`, le processus reçoit un signal `SIGALRM` après que le temps spécifié s'est écoulé ;
- Si le code temporisateur est `ITIMER_VIRTUAL`, le processus reçoit un signal `SIGVTALRM` après s'être exécuté pendant un temps donné. Le temps pendant lequel le programme ne s'exécute pas (c'est-à-dire lorsque le noyau ou un autre processus est en cours d'exécution) n'est pas pris en compte ;
- Si le code temporisateur est `ITIMER_PROF`, le processus reçoit un signal `SIGPROF` lorsque le temps d'exécution du processus ou des appels système qu'il a invoqué atteint le temps spécifié.

Le premier argument de `setitimer` est un code temporisateur, indiquant le type de temporisateur à mettre en place. Le second argument est un pointeur vers un objet `struct itimerval` spécifiant les nouveaux paramètres du temporisateur. Le troisième argument, s'il n'est pas `NULL` est un pointeur vers un autre objet `struct itimerval` qui reçoit l'ancien paramétrage du temporisateur.

Une variable `struct itimerval` est constituée de deux champs :

- `it_value` est un champ de type `struct timeval` qui contient le temps avant l'expiration suivante du temporisateur et l'envoi du signal. S'il vaut 0, le temporisateur est désactivé ;
- `it_interval` est un autre champ de type `struct timeval` contenant la valeur avec laquelle sera réinitialisé après son expiration. S'il vaut 0, le temporisateur sera désactivé après expiration. S'il est différent de zéro, le temporisateur expirera de façon répétitive à chaque écoulement de l'intervalle.

Le type `struct timeval` est décrit dans la Section 8.7, « *gettimeofday*: heure système ». Le programme du Listing 8.11 illustre l'utilisation de `setitimer` pour suivre le temps d'exécution d'un programme. Un temporisateur est configuré pour expirer toutes les 250 millisecondes et envoyer un signal `SIGVTALRM`.

Listing 8.11 – (*timer.c*) – **Exemple d'Utilisation d'un Temporisateur**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void timer_handler (int signum)
{
  static int count = 0;
  printf ("le temporisateur a expiré %d fois.\n", ++count);
}

int main ()
{
  struct sigaction sa;
  struct itimerval timer;

  /* Installe timer_handler en tant que gestionnaire pour SIGVTALRM. */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &timer_handler;
  sigaction (SIGVTALRM, &sa, NULL);

  /* Configure le temporisateur pour expirer après 250ms... */
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 250000;
  /* ... puis régulièrement toutes les 250ms. */
  timer.it_interval.tv_sec = 0;
  timer.it_interval.tv_usec = 250000;
  /* Démarre un temporisateur virtuel. Il décompte dès que le processus
     s'exécute. */
  setitimer (ITIMER_VIRTUAL, &timer, NULL);

  /* Perd du temps.  */
  while (1);
}
```

8.14 *sysinfo* : récupération de statistiques système

L'appel système `sysinfo` renseigne une structure avec des statistiques sur le système. Son seul argument est un pointeur vers une variable `struct sysinfo`. Voici quelques uns des champs les plus intéressants de cette structure :

– `uptime` – Temps écoulé depuis le démarrage du système, en secondes ;
– `totalram` – Mémoire physique disponible au total ;
– `freeram` – Mémoire physique libre ;
– `procs` – Nombre de processus s'exécutant sur le système.

Consultez la page de manuel de `sysinfo` pour une description complète du type `struct sysinfo`. Incluez `<linux/kernel.h>`, `<linux/sys.h>` et `<sys/sysinfo.h>` si vous utilisez `sysinfo`.

Le programme du Listing 8.12 affiche des statistiques sur le système courant.

Listing 8.12 – (*sysinfo.c*) – **Affiche des Statistiques Système**

```
#include <linux/kernel.h>
#include <linux/sys.h>
#include <stdio.h>
#include <sys/sysinfo.h>

int main ()
{
  /* Facteurs de conversion. */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  const double megabyte = 1024 * 1024;
  /* Récupère les statistiques système. */
  struct sysinfo si;
  sysinfo (&si);
  /* Affiche un résumé des informations intéressantes. */
  printf ("uptime système : %ld jours, %ld:%02ld:%02ld\n",
          si.uptime / day, (si.uptime % day) / hour,
          (si.uptime % hour) / minute, si.uptime % minute);
  printf ("RAM totale    : %5.1f Mo\n", si.totalram / megabyte);
  printf ("RAM libre     : %5.1f Mo\n", si.freeram / megabyte);
  printf ("nb processus  : %d\n", si.procs);
  return 0;
}
```

8.15 *uname*

L'appel système `uname` renseigne une structure avec diverses informations système, comme le nom de l'ordinateur, le nom de domaine et la version du système d'exploitation en cours d'exécution. Ne passez qu'un seul argument à uname : un pointeur vers un objet `struct utsname` . Incluez `<sys/utsname.h>` si vous utilisez `uname`.

L'appel à `uname` renseigne les champs suivants :
– `sysname` – Nom du système d'exploitation (par exemple, `Linux`) ;
– `release`, `version` – Numéros de version majeure et mineure du noyau ;
– `machine` – Informations sur la plateforme système. Pour un Linux x86, ce sera i386 ou i686 selon le processeur ;
– `node` – Nom non qualifié de l'ordinateur ;
– `__domainname` – Nom de domaine de l'ordinateur.

Chacun de ces champs est une chaîne de caractères.

Le petit programme du Listing 8.13 affiche les numéros de version du noyau et des informations sur le matériel.

Listing 8.13 – (*print-uname.c*) – **Affiche le Numéro de Version et des Infos Matérielles**

```
1  #include <stdio.h>
2  #include <sys/utsname.h>
3
4  int main ()
5  {
6    struct utsname u;
7    uname (&u);
8    printf ("%s version %s.%s sur système %s\n", u.sysname, u.release,
9            u.version, u.machine);
10   return 0;
11 }
```

Chapitre 9

Code assembleur en ligne

Aujourd'hui, peu de programmeurs utilisent le langage assembleur. Des langages de plus haut niveau comme le C ou le C++ s'exécutent sur quasiment toutes les architecture et permettent une productivité plus importante lors de l'écriture et de la maintenance du code. Parfois, les programmeurs ont besoin d'utiliser des instructions assembleur dans leurs programmes, la GNU Compiler Collection permet aux programmeurs d'ajouter des instructions en langage assembleur dépendantes de l'architecture à leurs programmes.

Les instructions assembleur GCC en ligne ne doivent pas être utilisées de façon excessive. Les instructions en langage assembleur dépendent de l'architecture, aussi, des programmes utilisant des instructions x86 ne peuvent pas être compilés sur des PowerPC. Pour les utiliser, vous aurez besoin d'un dispositif permettant de les traduire dans le jeu d'instruction de votre architecture. Cependant, les instructions assembleur vous permettent d'accéder au matériel directement et peuvent permettre de produire du code plus performant.

L'instruction `asm` vous permet d'insérer des instructions assembleur dans des programmes C ou C++. Par exemple, l'instruction

```
asm ("fsin" : "=t" (answer) : "0" (angle));
```

est une façon spécifique à l'architecture x86 de coder cette instruction C[1] :

```
answer = sin (angle);
```

Notez que contrairement aux instructions assembleur classiques, les constructions `asm` vous permettent de spécifier des opérandes en utilisant la syntaxe du C.

Pour en savoir plus sur le jeu d'instructions x86, que nous utiliserons dans ce chapitre, consultez `http://developer.intel.com/design/pentiumii/manuals/` et `http://www.x86-64.org/documentation`.

[1] L'expression `sin (angle)` est généralement implémentée par un appel à la bibliothèque `math`, mais si vous demandez une option `-O1` ou plus, GCC remplacera cet appel par une unique instruction assembleur `fsin`.

9.1 Quand utiliser du code assembleur ?

Bien qu'il ne faille pas abuser des constructions `asm`, elles permettent à vos programmes d'accéder au matériel directement et peuvent produire des programmes qui s'exécutent rapidement. Vous pouvez les utiliser lors de l'écriture du code faisant partie du système d'exploitation qui a besoin d'interagir avec le matériel. Par exemple, `/usr/include/asm/io.h` contient des instructions assembleur pour accéder aux ports d'entrée/sortie directement. Le fichier source du noyau situé dans `/usr/src/linux/arch/i386/kernel/process.s` offre un autre exemple en utilisant `hlt` dans une boucle d'inactivité. Consultez d'autres fichiers source du noyau Linux situés dans `/usr/src/linux/arch/` et `/usr/src/linux/drivers/`.

Les instructions assembleur peuvent également accélérer la boucle de traitement interne de certains programmes. Par exemple, si la majorité du temps d'exécution d'un programme est consacré au calcul du sinus et du cosinus du même angle, il est possible d'utiliser l'instruction x86 `fsincos`[2]. Consultez par exemple `/usr/include/bits/mathinline.h` qui encapsule des séquences assembleur au sein de macros afin d'accélérer le calcul de certaines fonctions transversales.

Vous ne devriez utiliser des instructions assembleur en ligne pour accélérer l'exécution qu'en dernier ressort. Les compilateurs actuels sont sophistiqués et connaissent les détails des processeurs pour lesquels ils génèrent du code. Ainsi, ils peuvent souvent sélectionner des séquences de code qui paraissent contre-intuitives mais qui s'exécutent en fait plus vite que d'autres. À moins que vous ne compreniez le jeu d'instructions et les propriétés d'ordonnancement du processeur cible dans les détails, vous feriez sans doute mieux de laisser les optimiseurs du compilateur générer du code assembleur à votre place pour la plupart des opérations.

De temps à autre, une ou deux instructions assembleur peuvent remplacer plusieurs lignes de code d'un langage de plus haut niveau. Par exemple, déterminer la position du bit non nul le plus significatif d'un entier en utilisant le C nécessite une boucle ou des calculs en virgule flottante. Beaucoup d'architectures, y compris le x86 disposent d'une instruction assembleur (`bsr`) qui calcule cette position. Nous illustrerons son utilisation dans la Section 9.3.5, « Exemple ».

9.2 Assembleur en ligne simple

Nous allons maintenant présenter la syntaxe des instructions assembleur `asm` avec un exemple décalant une valeur de 8 bits vers la droite sur architecture x86 :

```
asm ("shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc");
```

Le mot-clef `asm` est suivi par une expression entre parenthèses constituée de sections séparées par deux-points. La première section contient une instruction assembleur et ses opérandes, dans cet exemple, `shrl` décale à droite les bits du premier opérande. Celui-ci est représenté par `%0`, le second est la constante immédiate `$8`.

La deuxième section indique les sorties. Ici, la première sortie de l'instruction sera placée dans la variable C `answer`, qui doit être une lvalue. La chaîne `"=r"` contient un signe égal indiquant un opérande de sortie et un `r` indiquant que `answer` est stocké dans un registre.

[2]Les améliorations au niveau des algorithmes ou des structures de données sont souvent plus efficaces dans la réduction du temps d'exécution d'un programme que l'utilisation d'instructions assembleur.

La troisième section spécifie les entrées. La variable C `operand` donne la valeur à décaler. La chaîne `"r"` indique qu'elle est stockée dans un registre mais ne contient pas le signe égal car il s'agit d'un opérande d'entrée et non pas de sortie.

La quatrième et dernière section indique que l'instruction modifie la valeur du registre de code condition `cc`.

9.2.1 Convertir un *asm* en instructions assembleur

Le traitement des constructions `asm` par GCC est très simple. Il produit des instructions assembleur pour traiter les opérandes de l'`asm` et replace la construction `asm` par l'instruction que vous spécifiez. Il n'analyse pas du tout l'instruction.

Par exemple, GCC convertit cet extrait de code :

```
double foo, bar;
asm ("mycool_asm %1, %0" : "=r" (bar) : "r" (foo));
```

en la séquence d'instructions x86 suivante :

```
        movl  -8(%ebp),%edx
        movl  -4(%ebp),%ecx
#APP
        mycool_asm %edx, %edx
#NO_APP
        movl %edx,-16(%ebp)
        movl %ecx,-12(%ebp)
```

Souvenez-vous que `foo` et `bar` requièrent chacun deux mots d'espace de stockage dans la pile sur une architecture x86 32 bits. Le registre `ebp` pointe vers les données sur la pile.

Les deux premières instructions copient `foo` dans les registres `EDX` et `ECX` qu'utilise `mycool_asm`. Le compilateur a décidé d'utiliser les mêmes registres pour stocker la réponse, qui est copiée dans `bar` par les deux dernières instructions. Il choisit les registres adéquats, peut même les réutiliser, et copie les opérandes à partir et vers les emplacements appropriés automatiquement.

9.3 Syntaxe assembleur avancée

Dans les sous-sections qui suivent, nous décrivons les règles de syntaxe pour les constructions `asm`. Leurs sections sont séparées par deux-points.

Nous utiliserons l'instruction `asm` suivante qui calcule l'expression booléenne $x > y$:

```
asm ("fucomip %%st(1), %%st; seta %%al" :
     "=a" (result) : "u" (y), "t" (x) : "cc", "st");
```

Tout d'abord, `fucomip` compare ses deux opérandes `x` et `y` et stocke les valeurs indiquant le résultat dans le registre de code condition. Puis, `seta` convertit ces valeurs en 0 ou 1.

9.3.1 Instructions assembleur

La première section contient les instructions assembleur, entre guillemets. La construction `asm` exemple contient deux instructions assembleur, `fucomip` et `seta`, séparées par un point-virgule. Si l'assembleur n'autorise pas les points-virgule, utilisez des caractères de nouvelle ligne (`\n`)

pour séparer les instructions. Le compilateur ignore le contenu de cette première section, excepté qu'il supprime un niveau de signes pourcent, %% devient donc %. La signification de %%st(1) et d'autres termes similaires dépend de l'architecture.

GCC se plaindra si vous spécifiez l'option `-traditional` ou `-ansi` lors de la compilation d'un programme contenant des constructions `asm`. Pour éviter de telles erreurs, utilisez le mot clé alternatif `__asm__` comme dans les fichiers d'en-tête cités précédemment.

9.3.2 Sorties

La seconde section spécifie les opérandes de sortie des instructions en utilisant une syntaxe C. Chaque opérande est décrit par une contrainte d'opérande sous forme de chaîne suivie d'une expression C entre parenthèses. Pour les opérandes de sortie, qui doivent être des lvalues, la chaîne de contrainte doit commencer par un signe égal. Le compilateur vérifie que l'expression C pour chaque opérande de sortie est effectivement une lvalue.

Les lettres spécifiant les registres pour une architecture donnée peuvent être trouvée dans le code source de GCC, dans la macro `REG_CLASS_FROM_LETTER`. Par exemple, le fichier de configuration `gcc/config/i386/i386.h` de GCC liste les lettres de registre pour l'architecture x86[3]. Le Tableau 9.1 en fait le résumé.

TAB. 9.1 – Lettres correspondant aux Registres sur l'Architecture Intel x86

Lettre	**Registres Éventuellement Utilisés par GCC**
R	Registre général (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Registre de données général (EAX, EBX, ECX, EDX)
f	Registre virgule flottante
t	Registre en virgule flottante supérieur
u	Second registre en virgule flottante
a	Registre EAX
b	Registre EBX
c	Registre ECX
d	Registre EDX
x	Registre SSE (Streaming SIMD Extension)
y	Registres multimédias MMX
A	Valeur de 8 octets formée à partir de EAX et EDX
D	Pointeur de destination pour les opérations sur les chaînes (EDI)
S	Pointeur source pour les opérations sur les chaînes (ESI)

Lorsqu'une structure `asm` utilise plusieurs opérandes, elles doivent être décrites par une chaîne de contrainte et une expressions C et séparées par des virgules, comme l'illustre l'exemple donné précédemment. Vous pouvez spécifier jusqu'à 10 opérandes, numérotées de `%0` à `%9` dans les

[3]Vous devrez avoir une certaine familiarité avec le fonctionnement de GCC pour comprendre le contenu de ce fichier.

sections d'entrée et de sortie. S'il n'y a pas d'opérandes de sortie ou de registre affectés, laissez la section de sortie vide ou signalez la avec un commentaire du type `/* no outputs */`.

9.3.3 Entrées

La troisième section décrit les entrées des instructions assembleur. La chaîne de contrainte d'un opérande d'entrée ne doit pas contenir de signe égal, ce qui indique une lvalue. Mis à part cela, leur syntaxe est identique à celle des opérandes de sortie.

Pour indiquer qu'un registre est à la fois lu et écrit par la même construction asm, utilisez l'indice de l'opérande de sortie comme chaîne de contrainte d'entrée. Par exemple, pour indiquer qu'un registre d'entrée est le même que le premier registre de sortie, utilisez 0. Spécifier la même expression C pour des opérandes d'entrée et de sortie ne garantit pas que les deux valeurs seront placées dans le même registre.

La section d'entrée peut être omise s'il n'y a pas d'opérandes d'entrée et que le section de déclaration des modifications est vide.

9.3.4 Déclaration des modifications

Si une instruction modifie les valeur d'un ou plusieurs registres par effet de bord, spécifiez ces registres dans la quatrième section de la structure `asm`. Par exemple, l'instruction `fucomip` modifie le registre de code condition, qui est désigné par `cc`. Les registres modifiés sont décrits dans des chaînes individuelles séparées par des virgules. Si l'instruction est susceptible de modifier un emplacement mémoire arbitraire, spécifiez `memory`. En utilisant les informations sur la modification des registres, le compilateur détermine les valeurs qui doivent être restaurées après l'exécution du bloc `asm`. Si vous ne renseignez pas ces informations correctement, GCC pourrait supposer que certains registres contiennent des valeurs qui ont en fait été écrasées, ce qui pourrait affecter le fonctionnement de votre programme.

9.3.5 Exemple

L'architecture x86 dispose d'instructions qui déterminent les positions du bit non nul le plus significatif ou le moins significatif dans un mot. Le processeur peut exécuter ces instructions de façon relativement efficace. Par contre, l'implémentation de la même opération en C nécessite une boucle et un décalage binaire.

Par exemple, l'instruction assembleur `bsrl` calcule la position du bit le plus significatif de son premier opérande et place cette position (à partir de 0 qui représente le bit le moins significatif) dans le second. Pour placer la position du bit non nul le plus significatif de number dans position, nous pourrions utiliser cette structure asm :

```
asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
```

Une façon d'implémenter la même opération en C, est d'utiliser une boucle de ce genre :

```
long i;
for (i = (number >> 1), position = 0; i != 0; ++position)
  i >>= 1;
```

Pour comparer les vitesses de ces deux versions, nous allons les placer dans une boucle qui calcule les positions pour de grands nombres. Le Listing 9.1 utilise l'implémentation en C. Le programme boucle sur des entiers, en partant de 1 jusqu'à la valeur passée sur la ligne de commande. Pour chaque valeur de number, il calcule la position du bit non nul le plus significatif. Le Listing 9.2 effectue les mêmes opérations en utilisant une construction assembleur en ligne. Notez que dans les deux versions, nous plaçons la position calculée à une variable `volatile result`. Cela permet d'éviter que l'optimiseur du compilateur ne supprime le calcul ; si le résultat n'est pas utilisé ou stocké en mémoire, l'optimiseur élimine le calcul en le considérant comme du code mort.

Listing 9.1 – (***bit-pos-loop.c***) – **Recherche d'un Bit en Utilisant une Boucle**

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

int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
  long i;
  unsigned position;
  volatile unsigned result;

  /* Répète l'opération pour un nombre important de valeurs. */
  for (number = 1; number <= max; ++number) {
    /* Décale le nombre vers la droite jusqu'à ce qu'il vaille
       zéro. Mémorise le nombre de décalage qu'il a fallu faire. */
    for (i = (number >> 1), position = 0; i != 0; ++position)
      i >>= 1;
    /* La position du nombre non nul le plus significatif est le nombre
       de décalages qu'il a fallu après le premier. */
    result = position;
  }

  return 0;
}
```

Listing 9.2 – (***bit-pos-asm.c***) – **Recherche d'un Bit en Utilisant *bsrl***

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
  unsigned position;
  volatile unsigned result;
  /* Répète l'opération pour un nombre important de valeurs. */
  for (number = 1; number <= max; ++number) {
    /* Calcule la position du bit non nul le plus significatif
       en utilisant l'instruction assembleur bsrl. */
    asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
    result = position;
  }
  return 0;
}
```

Compilons les deux versions avec les optimisations actives :

```
% cc -O2 -o bit-pos-loop bit-pos-loop.c
% cc -O2 -o bit-pos-asm bit-pos-asm.c
```

À présent, lançons-les en utilisant la commande `time` pour mesurer le temps d'exécution. Il est nécessaire de passer une valeur importante comme argument afin de s'assurer que chaque version mette un minimum de temps à s'exécuter.

```
% time ./bit-pos-loop 250000000
real    0m27.042s
user    0m24.583s
sys     0m0.135s
% time ./bit-pos-asm 250000000
real    0m1.472s
user    0m1.442s
sys     0m0.013s
```

Voyez comme la version utilisant l'assembleur s'exécute beaucoup plus vite (vos propres résultats peuvent varier).

9.4 Problèmes d'optimisation

L'optimiseur de GCC tente de réordonner et de réécrire le code du programme pour minimiser le temps d'exécution même en présence d'expressions `asm`. Si l'optimiseur détermine que les valeurs de sortie ne sont pas utilisées, l'instruction sera supprimée à moins que le mot clé `volatile` ne figure entre `asm` et ses arguments (par ailleurs, GCC ne déplacera pas une structure `asm` sans opérande de sortie en dehors d'une boucle). Toute structure `asm` peut être déplacée de façon difficile à prévoir, même d'un saut à l'autre. La seule façon de garantir l'ordre d'un bloc assembleur est d'inclure toutes les instructions dans la même structure `asm`.

L'utilisation de constructions `asm` peut limiter l'efficacité de l'optimiseur car le compilateur ne connaît pas leur sémantique. GCC est forcé de faire des suppositions qui peuvent interdire certaines optimisations. *Caveat emptor* !

9.5 Problèmes de maintenance et de portabilité

Si vous décidez d'utiliser des constructions `asm` non-portables et dépendantes de l'architecture, les encapsuler dans des macros peut aider à la maintenance et améliorer la portabilité.

Placer toutes ces macros dans un fichier et les documenter facilitera le portage de l'application vers une autre architecture, quelque chose qui arrive souvent même pour les programmes « à la va vite ». De cette façon, le programmeur n'aura besoin de réécrire qu'un seul fichier pour adapter le logiciel à une autre architecture.

Par exemple, la plupart des instructions `asm` du code source Linux sont regroupées dans les fichiers d'en-tête situés sous `/usr/src/linux/include/asm/` et `/usr/src/linux/include/asm-i386/` et les fichiers source situés sous `/usr/src/linux/arch/i386/` et `/usr/src/linux/drivers/`.

Chapitre 10

Sécurité

Une grande partie de la puissance d'un système GNU/Linux vient de sa capacité à gérer plusieurs utilisateurs et de son bon support pour le réseau. Beaucoup de gens peuvent utiliser le système en même temps et se connecter au système à distance. Malheureusement, cette puissance présente des risques, en particulier pour les système connectés à Internet. Dans certaines circonstances, un « hacker » distant peut se connecter au système et lire, modifier ou supprimer des fichiers stockés sur la machine. Ou bien, un utilisateur pourrait lire, modifier ou supprimer les fichier d'un autre sans y être autorisé. Lorsque ce type d'événement se produit, la sécurité du système est dite *compromise*.

Le noyau Linux fournit divers mécanismes afin de s'assurer que de telles situations ne se présentent pas. Mais les applications classiques doivent elles aussi être attentives à éviter les failles de sécurité. Par exemple, imaginons que vous développiez un logiciel de comptabilité. Bien que vous puissiez vouloir permettre à tous les utilisateurs de remplir des notes de frais, il n'est certainement pas souhaitables qu'ils puissent tous les *valider*. De même, les utilisateurs ne doivent pouvoir consulter que leur propre bulletin de paie et les managers ne connaître que les salaires des employés de leur département.

Pour forcer ce type de contrôle, vous devez être très attentif. Il est étonnamment facile de faire une erreur permettant à des utilisateurs de faire quelque chose qu'ils ne devraient pas pouvoir faire. La meilleur approche est de demander de l'aide à des experts en sécurité. Toutefois, tout développeur d'application doit maîtriser quelques bases.

10.1 Utilisateurs et groupes

Chaque utilisateur sous Linux est identifié par un numéro unique appelé *user ID*, ou *UID*. Bien sûr, lorsque vous vous connectez, vous utilisez un nom d'utilisateur plutôt que cet UID. Le système effectue la conversion entre ce nom d'utilisateur et l'UID, et à partir de ce moment, seul l'user ID est pris en compte.

Vous pouvez en fait faire correspondre plus d'un nom d'utilisateur au même UID. En ce qui concerne le système, seuls les UID comptent. Il n'y a aucune façon de donner plus de droits à un utilisateur qu'à un autre s'ils ont tous deux le même UID.

Vous pouvez contrôler l'accès à un fichier ou à toute autre ressource en l'associant à un UID particulier. Dans ce cas, seul l'utilisateur disposant de cet UID peut accéder à la ressource. Par exemple, vous pouvez créer un fichier que vous serez le seul à pouvoir lire ou un répertoire où vous seul pourrez créer de nouveau fichier. Cela suffit dans la plupart des cas simples.

Parfois, cependant, vous avez besoin de partager une ressource entre plusieurs utilisateurs. Par exemple, si vous êtes un manager, vous pourriez vouloir créer un fichier que tous les managers pourraient lire mais pas les employés ordinaires. Linux ne vous permet pas d'associer plusieurs UID au même fichier, vous ne pouvez donc pas créer une liste de tous les gens auxquels vous souhaitez donner accès au fichier et les lier au fichier.

Par contre, vous pouvez créer un *groupe*. À chaque groupe est associé un numéro unique, appelé group ID ou GID. Chaque groupe contient un ou plusieurs user ID. Un même UID peut faire partie de plusieurs groupes mais un groupe ne peut pas contenir d'autres groupes ; ils ne peuvent contenir que des utilisateurs. Comme les utilisateurs, les groupes ont des noms. Tout comme pour les noms d'utilisateurs, le nom d'un groupe n'a pas d'importance, le système utilise toujours le GID en interne.

Pour continuer avec notre exemple, vous pourriez créer un groupe managers et y placer les UID de tous les managers. Vous pourriez alors créer un fichier qui peut être lu par n'importe qui dans ce groupe mais pas par les gens qui y sont extérieurs. En général, vous ne pouvez associer qu'un seul groupe à une ressource. Il n'y a aucun moyen de spécifier que les utilisateurs peuvent accéder à un fichier s'ils font partie du groupe 7 ou du groupe 42, par exemple.

Si vous êtes curieux de connaître votre UID et les groupes auxquels vous appartenez, vous pouvez utiliser la commande `id`. Voici un exemple de la sortie de cette commande :

```
% id
uid=501(mitchell) gid=501(mitchell) groups=501(mitchell),503(csl)
```

la première partie indique que l'UID de l'utilisateur ayant invoqué la commande est 501. La commande détermine le nom d'utilisateur correspondant et l'affiche entre parenthèses. On voit ici que l'utilisateur 501 est dans deux groupes : le groupe 501 (appelé `mitchell`) et le groupe 503 (appelé `csl`). Vous vous demandez certainement pourquoi le groupe 501 apparaît deux fois : une première fois dans le champ `gid` et une seconde dans le champ `groups`. Nous y reviendrons plus tard.

10.1.1 Le superutilisateur

Un des comptes utilisateur est très spécial[1]. Cet utilisateur a l'UID 0 est a généralement le nom d'utilisateur root. On y fait parfois référence en parlant du *compte superutilisateur*. L'utilisateur root peut littéralement tout faire : lire ou supprimer n'importe quel fichier, ajouter de nouveaux utilisateurs, désactiver l'accès réseau, *etc.* Beaucoup d'opérations spéciales ne peuvent être réalisées que par des processus s'exécutant avec les privilèges root – c'est-à-dire s'exécutant sous le compte utilisateur root.

[1]Le fait qu'il n'y ait qu'un seul utilisateur spécial est à l'origine du nom d'UNIX donné par AT&T à son système d'exploitation. Un système d'exploitation plus ancien qui disposait de plusieurs utilisateurs spéciaux a été baptisé MULTICS. GNU/Linux, bien sûr, est compatible avec UNIX.

Le problème de cette conception est qu'un nombre important de programmes doivent être exécutés par root car beaucoup de programmes ont besoin d'accéder à ces opérations spéciales. Si l'un de ses programmes ne se comporte pas correctement, les conséquences peuvent être dramatique. Il n'y a aucun moyen de contenir un programme lorsqu'il est exécuté par root ; il peut faire *n'importe quoi*. Les programmes lancés par root doivent être écrit avec beaucoup de précautions.

10.2 Identifiant de groupe et utilisateur de processus

Jusqu'à maintenant, nous n'avons parlé que des commandes exécutées par un utilisateur en particulier. Cela ne colle pas exactement à la réalité car l'ordinateur ne sait jamais réellement quel utilisateur l'exécute. Si Ève découvre le nom d'utilisateur et le mot de passe d'Alice, Ève peut se connecter en tant qu'Alice et le système laisserait faire à Ève tout ce qu'Alice peut faire. Il ne connaît que l'UID en cours et non pas l'utilisateur qui saisit les commandes. Si on ne peut pas faire confiance à Alice pour garder secret son mot de passe, alors rien de ce que vous pourrez faire en tant que programmeur d'application ne pourra empêcher Ève d'accéder aux fichiers d'Alice. La responsabilité de la sécurité du système est partagée entre le développeur, les utilisateurs du système et les administrateurs du système.

À chaque processus est associé un UID et un GID. Lorsque vous invoquez une commande, elle lance un processus dont l'UID et le GID sont les vôtres. Lorsque nous disons qu'un utilisateur effectue une opération quelconque, nous voulons dire en réalité qu'un processus avec l'UID correspondant effectue cette opération. Lorsque le processus exécute un appel système, le noyau décide s'il y est autorisé. Pour cela, il examine les permissions associées aux ressources auxquelles le processus tente d'accéder et vérifie l'UID et le GID associés au processus tentant d'exécuter l'appel.

Désormais, vous savez ce que signifie le champ `gid` de la sortie de la commande `id`. Il indique le GID du processus courant. Même si l'utilisateur 501 fait partie de plusieurs groupes, le processus courant ne peut avoir qu'un seul GID. Dans l'exemple présenté précédemment, le GID en cours est 501.

Si vous devez manipuler des UID ou des GID dans votre programme (et vous aurez à le faire si vous écrivez des programmes concernant la sécurité), vous devez utiliser les types `uid_t` et `gid_t` définis dans l'en-tête `<sys/types.h>` – même si les UID et GID ne sont en fait que des entiers, évitez de faire des suppositions sur le nombre de bits utilisé dans ces types ou d'effectuer des opérations arithmétiques en les utilisant. Traitez les comme un moyen obscur de manipuler les identifiants de groupe et d'utilisateur.

Pour récupérer l'UID et le GID du processus courant, vous pouvez utiliser les fonctions `geteuid` et `getegid` définies dans `<unistd.h>`. Ces fonctions ne prennent aucun paramètre et n'échouent jamais, il n'y a pas d'erreur à contrôler. Le Listing 10.1 présente un simple programme qui fournit un sous-ensemble des fonctionnalités de la commande `id` :

Listing 10.1 – (***simpleid.c***) – **Affiche les Identifiants d'Utilisateur et de Groupe**

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

```
4  int main()
5  {
6    uid_t uid = geteuid ();
7    gid_t gid = getegid ();
8    printf ("uid=%d gid=%d\n", (int) uid, (int) gid);
9    return 0;
10 }
```

Lorsque ce programme est exécuté (par le même utilisateur que celui qui a lancé le programme `id` dans l'exemple précédemment), la sortie est la suivante :

```
% ./simpleid
uid=501 gid=501
```

10.3 Permissions du système de fichiers

Un bon moyen de voir les utilisateurs et les groupes en actions est d'étudier les permissions du système de fichiers. En examinant la façon dont le système associe les permissions avec chaque fichier et en observant comment le noyau vérifie qui est autorisé à accéder à quels fichiers, le concept d'identifiant utilisateur et de groupe devrait devenir clair.

Chaque fichier a exactement un utilisateur propriétaire et un groupe propriétaire. Lorsque vous créez un nouveau fichier, il est détenu par l'utilisateur et le groupe du processus créateur[2].

Les opérations de base sur les fichiers, en ce qui concerne Linux, sont la lecture, l'écriture et l'exécution (Notez que la création et la suppression ne sont pas considérées comme des opérations sur le fichier, elles sont considérées comme des opérations sur le répertoire contenant le fichier. Nous en parlerons plus loin). Si vous ne pouvez pas lire un fichier, Linux ne vous laissera pas en examiner le contenu. Si vous ne pouvez pas y écrire, vous ne pouvez pas modifier son contenu. Si vous ne disposez pas des droits d'exécution sur un fichier contenant le code d'un programme, vous ne pouvez pas exécuter ce programme.

Linux vous permet d'indiquer lesquelles de ces trois actions – lire, écrire et exécuter – peuvent être réalisées par l'utilisateur propriétaire, le groupe propriétaire et toute autre personne. Par exemple, vous pouvez dire que l'utilisateur propriétaire aura tous les droits, que les membres du groupe propriétaire pourront lire et exécuter le fichier (mais pas y écrire) et que personne d'autre n'y a accès.

Vous pouvez consulter ces bits de permission de façon interactive avec la commande `ls` en utilisant les options `-l` ou `-o` et par programmation via l'appel système `stat`. Vous pouvez définir de façon interactive les bits de permission avec le programme chmod[3] et par programmation avec l'appel système du même nom. Pour examiner les permissions d'un fichier appelé `hello`, utilisez `ls -l hello`. Voici un exemple de sortie :

```
% ls -l hello
-rwxr-x---    1 samuel csl 11734 Jan 22 16:29 hello
```

[2]En fait, il existe de rares exceptions faisant intervenir les *sticky bits*, décrits plus loin dans la Section 10.3.2, « Sticky bits ».

[3]On fait parfois référence aux bits de permission en utilisant le terme *mode*. Le nom de la commande `chmod` est un diminutif pour « change mode ».

Les indications `samuel` et `csl` signifient que l'utilisateur propriétaire est samuel et que le groupe propriétaire est csl.

La chaîne de caractères a début de la ligne indique les permissions associées au fichier. Le premier tiret indique que le fichier est un fichier classique. Il serait remplacé par `d` pour un répertoire ou d'autres lettres dans le cas de fichiers spéciaux comme les périphériques (reportez-vous au Chapitre 6, « Périphériques ») ou les canaux nommés (voyez le Chapitre 5, « Communication interprocessus », Section 5.4, « Tubes »). Les trois caractères suivants représentent les permissions de l'utilisateur propriétaire : ils indiquent que samuel peut lire, écrire et exécuter le fichier. Les trois caractères d'après donnent les permissions des membres du groupe csl ; ses membres ne peuvent que lire et exécuter le fichier. Enfin, les trois derniers caractères indiquent les permissions de toute autre personne ; ici, tout utilisateur n'étant pas samuel et ne faisant pas partie du groupe csl ne peut rien faire avec le fichier `hello`.

Voyons comment cela fonctionne. Tout d'abord, essayons d'accéder au fichier en tant qu'utilisateur nobody, qui ne fait pas partie du groupe csl :

```
% id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
% cat hello
cat: hello: Permission denied
% echo hi > hello
sh: ./hello: Permission denied
% ./hello
sh: ./hello: Permission denied
```

Nous n'avons pas le droit de lire le fichier, c'est pourquoi `cat` échoue ; nous ne pouvons pas écrire dans le fichier, c'est pourquoi `echo` échoue ; et nous n'avons pas le droit d'exécuter le fichier, c'est pourquoi `./hello` échoue.

Les choses s'arrangent si nous accédons au fichier en tant que mitchell, qui n'est pas membre du groupe csl :

```
% id
uid=501(mitchell) gid=501(mitchell) groups=501(mitchell),503(csl)
% cat hello
#!/bin/bash
echo "Hello, world."
% ./hello
Hello, world.
% echo hi > hello
bash: ./hello: Permission denied
```

Nous pouvons afficher le contenu du fichier et nous pouvons l'exécuter (il s'agit d'un simple script shell) mais nous ne pouvons toujours pas y écrire.

Si nous sommes identifiés en tant que propriétaire (samuel), nous pouvons même écraser le fichier :

```
% id
uid=502(samuel) gid=502(samuel) groups=502(samuel),503(csl)
% echo hi > hello
% cat hello
hi
```

Vous pouvez modifier les permissions associées avec un fichier si vous êtes son propriétaire (ou le superutilisateur). Par exemple, si vous voulez désormais permettre à tout le monde d'exécuter le fichier, vous pouvez faire ce qui suit :

```
% chmod o+x hello
% ls -l hello
-rwxr-x--x    1 samuel csl 3 Jan 22 16:38 hello
```

Notez qu'il y a maintenant un `x` à la fin de la première chaîne de caractères. L'option `o+x` signifie que vous voulez donner la permission d'exécution à tous les autres gens (ni le propriétaire, ni les membres du groupe propriétaire). Pour révoquer les permissions en écriture du groupe, vous utiliseriez `g-w`. Consultez la page de manuel de la section 1 sur `chmod` pour plus de détails sur sa syntaxe :

```
% man 1 chmod
```

Dans un programme, vous pouvez utiliser l'appel système `stat` pour déterminer les permissions associées à un fichier. Cette fonction prend deux paramètres : le nom du fichier sur lequel vous voulez des renseignements et l'adresse d'une structure de données renseignée avec des informations sur le fichier. Consultez l'Annexe B, « E/S de bas niveau », Section B.2, « *stat* », pour une présentation des autres informations que vous pouvez obtenir via `stat`. Le Listing 10.2 montre un exemple d'utilisation de `stat` pour déterminer les permissions associées à un fichier.

Listing 10.2 – (*stat-perm.c*) – **Déterminer si le Propriétaire a les Droits d'Écriture**

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3
4  int main (int argc, char* argv[])
5  {
6    const char* const filename = argv[1];
7    struct stat buf;
8
9    /* Récupère les informations sur le fichier. */
10   stat (filename, &buf);
11   /* Affiche un message si les permissions sont définies de façons
12      à ce que le propriétaire puisse y écrire. */
13   if (buf.st_mode & S_IWUSR)
14     printf ("Le propriétaire peut écrire dans '%s'.\n", filename);
15   return 0;
16 }
```

Si vous exécutez ce programme sur notre script `hello`, il indique :

```
% ./stat-perm hello
Le propriétaire peut écrire dans 'hello'.
```

La constante `S_IWUSR` correspond à la permission en écriture pour le propriétaire. Il y a une constante pour chaque bit. Par exemple, `S_IRGRP` correspond à la permission en lecture pour le groupe propriétaire et `S_IXOTH` à la permission en exécution pour les utilisateurs qui ne sont ni propriétaires, ni membre du groupe. Si vous stockez les permissions dans une variable, utilisez le `typedef mode_t`. Comme la plupart des appels système, `stat` renverra `-1` et positionnera `errno` s'il ne peut pas obtenir les informations sur le fichier.

Vous pouvez utiliser la fonction chmod pour modifier les bits de permission d'un fichier existant. Appelez `chmod` avec le nom du fichier dont vous voulez changer les permission et les bits à activer sous forme d'un OU binaire entre les différentes constantes citées précédemment. Par exemple, l'extrait suivant rend `hello` lisible et exécutable par le propriétaire mais désactive toutes les autres permissions associées à hello :

```
chmod ("hello", S_IRUSR | S_IXUSR);
```

Le même système de bits de permission s'applique aux répertoires mais ces bits ont des significations différentes. Si un utilisateur est autorisé à lire le répertoire, alors il peut consulter la liste des fichiers présents dans ce répertoire. Avec les droits en écriture, il est possible d'ajouter ou de supprimer des fichiers. Notez qu'un utilisateur peut supprimer des fichiers d'un répertoire s'il a les droits en écriture sur celui-ci, même s'il n'a pas la permission de modifier le fichier qu'il supprime. Si un utilisateur a les droits d'exécution sur un répertoire, il a le droit d'y entrer et d'accéder aux fichiers qu'il contient. Sans droit d'exécution sur un répertoire, un utilisateur ne peut pas accéder aux fichiers qu'il contient, indépendamment des droits qu'il détient sur les fichiers eux-mêmes.

Pour conclure, observons comment le noyau décide d'autoriser ou non un processus à accéder à un fichier donné. Tout d'abord, il détermine si l'utilisateur demandant l'accès est le propriétaire du fichier, un membre du groupe propriétaire ou quelqu'un d'autre. La catégorie dans laquelle tombe l'utilisateur est utilisée pour déterminer quel ensemble de bits lecture/écriture/exécution est vérifié. Puis, le noyau contrôle l'opération effectuée par rapport aux permissions accordées à l'utilisateur[4].

Il y a une exception qu'il convient de signaler : les processus s'exécutant en tant que root (avec l'user ID 0) sont toujours autorisés à accéder à n'importe quel fichier, quelques soient les permissions qui y sont associées.

10.3.1 Faille de sécurité : les programmes non exécutables

Voici un premier exemple de situation ou la sécurité se complique. Vous pourriez penser que si vous interdisez l'exécution d'un programme, personne ne pourra le lancer. Après tout, c'est ce que sous-entend l'interdiction d'exécution. Mais un utilisateur ingénieux pourrait copier le programme, modifier les permissions pour rendre la copie exécutable et la lancer ! Si vous interdisez l'exécution de programmes sans interdire aux utilisateurs de les copier, vous créez une *faille de sécurité* – un moyen pour les utilisateurs de faire des choses que vous n'aviez pas prévu.

10.3.2 Sticky bits

En plus des permissions en lecture, écriture et exécution, il existe un bit magique appelé *sticky bit*[5]. Ce bit ne concerne que les répertoires.

Un répertoire pour lequel le sticky bit est actif ne vous autorise à détruire un fichier que si vous en êtes le propriétaire. Comme nous l'avons dit précédemment, vous pouvez normalement supprimer un fichier si vous avez un accès en écriture sur le répertoire qui le contient, même si vous n'en êtes pas le propriétaire. Lorsque le sticky bit est activé, vous devez *toujours* avoir les

[4]Le noyau peut également refuser l'accès à un fichier si un répertoire dans le chemin du fichier est inaccessible. Par exemple, si un processus n'a pas le droit d'accéder au répertoire `/tmp/private`, il ne pourra pas lire `/tmp/private/data`, même si les permissions de ce dernier sont définies de façon à l'y autoriser.

[5]Ce terme est anachronique ; il remonte à un temps où l'activation du sticky bit (bit « collant ») permettait de conserver un programme en mémoire même lorsqu'il avait terminé de s'exécuter. Les pages allouées pour le programme étaient « collées » en mémoire. Les sticky bits sont également parfois appelés « bits de rappel ».

droits en écriture sur le répertoire, mais vous devez en plus être le propriétaire du fichier que vous voulez supprimer.

Sur un système GNU/Linux seuls certains répertoires ont le sticky bit actif. Par exemple, le répertoire `/tmp`, dans lequel tout utilisateur peut placer des fichiers temporaires, en fait partie. Ce répertoire est spécifiquement conçu pour pouvoir être utilisé par tous les utilisateur, tout le monde doit donc y écrire. Mais il n'est pas souhaitable qu'un utilisateur puisse supprimer les fichiers d'un autre, le sticky bit est donc activé pour ce répertoire. De cette façon, seul le propriétaire (ou root, bien sûr) peut supprimer le fichier.

Vous pouvez voir que le sticky bit est actif grâce au `t` à la fin de la liste des permissions si vous lancez ls sur `/tmp` :

```
% ls -ld /tmp
drwxrwxrwt   12 root root 2048 Jan 24 17:51 /tmp
```

La constante correspondante à utiliser avec `stat` et `chmod` est `S_ISVTX`.

Si votre programme crée des répertoires qui se comportent comme `/tmp`, c'est-à-dire que beaucoup de personne doivent y écrire sans pouvoir supprimer les fichier des autres, vous devez activer le sticky bit sur ce répertoire. Vous pouvez le faire en utilisant la commande `chmod` de cette façon :

```
% chmod o+t répertoire
```

Pour le faire par programmation, appelez `chmod` avec le drapeau de mode `S_ISVTX`. Par exemple, pour activer le sticky bit du répertoire spécifié par `dir_path` et donner un accès complet à tous les utilisateur, effectuez l'appel suivant :

```
chmod (dir_path, S_IRWXU | S_IRWXG | S_IRWXO | S_ISVTX);
```

10.4 Identifiants réels et effectifs

Jusqu'à maintenant, nous avons parlé de l'UID et du GID associés avec un processus comme s'il n'y en avait qu'un seul de chaque. Mais, en réalité, ce n'est pas aussi simple.

Chaque processus a en réalité deux user ID : l'*user ID effectif* et l'*user ID réel* (bien sûr, il y a également un *group ID effectif* et un *group ID réel* ; presque tout ce qui est vrai pour les user ID l'est également pour les group ID). La plupart du temps, le noyau ne se préoccupe que du user ID effectif. Par exemple, si un processus tente d'ouvrir un fichier, le noyau vérifie l'user ID effectif pour décider s'il laisse le processus accéder au fichier.

Les fonctions `geteuid` et `getegid` décrites précédemment renvoient l'user ID et le group ID effectifs. Les fonctions analogues `getuid` et `getgid` renvoient l'user ID et le group ID réels.

Si le noyau ne s'occupe que de l'user ID effectif, il ne semble pas très utile de faire la distinction entre user ID réel et effectif. Cependant, il y a un cas très important dans lequel le user ID réel est pris en compte. Si vous voulez changer l'user ID effectif d'un processus en cours d'exécution, le noyau vérifie l'user ID réel et l'user ID effectif.

Avant d'observer *comment* vous pouvez changer l'user ID effectif d'un processus, examinons *pourquoi* vous pourriez vouloir le faire en reprenant l'exemple de notre application de comptabilité. Supposons qu'il y ait un processus serveur qui ait besoin de consulter tout fichier présent

sur le système, peu importe qui l'ait créé. Un tel processus doit s'exécuter en tant que root car lui seul est sûr de pouvoir accéder à n'importe que fichier. Mais supposons maintenant qu'une requête arrive de la part d'un utilisateur particulier (disons mitchell) pour accéder à des fichiers quelconques. Le processus serveur pourrait examiner avec attention les permissions associées avec les fichiers concernés et essayer de déterminer si mitchell devrait être autorisé à accéder à ces fichiers. Mais cela signifierait dupliquer tous les traitements que le noyau ferait de toutes façons. Réimplémenter cette logique serait complexe, sujet à des erreurs et pénible.

Une meilleure approche est simplement de modifier temporairement l'user ID effectif du processus pour ne plus qu'il soit celui de root mais celui de mitchell puis d'essayer d'effectuer les opérations demandées. Si mitchell n'a pas le droit d'accéder aux données, le noyau interdira l'accès au processus et renverra des informations adéquates sur l'erreur. Une fois que les opérations demandées par mitchell sont terminées, le processus peut récupérer son user ID effectif original qui est root.

Les programmes qui authentifient les utilisateurs lorsqu'ils essaient de se connecter tirent eux aussi avantage de cette possibilité de modifier les user ID. Ces programmes s'exécutent en tant que root – lorsque l'utilisateur saisit un login et un mot de passe, le programme de connexion vérifie le nom d'utilisateur et le mot de passe dans la base de données du système. Puis il change à la fois ses user ID réel et effectif afin de devenir cet utilisateur. Enfin, le programme de connexion appelle `exec` pour lancer le shell de l'utilisateur, ce qui permet à l'utilisateur d'avoir un environnement dans lequel les user ID réel et effectif sont les siens.

La fonction utilisée pour modifier les user ID d'une processus est `setreuid` (il y a, bien sûr, une fonction `setregid` similaire). Cette fonction prend deux arguments. Le premier argument est l'user ID réel désiré ; le second est l'user ID effectif demandé. Par exemple, voici comment vous échangeriez les user ID réel et effectif :

```
setreuid (geteuid(), getuid ());
```

Bien sûr, le noyau ne laisse pas n'importe quel processus changer son user ID. Si un processus pouvait modifier son user ID effectif à volonté, alors, toute personne pourrait facilement prendre l'identité de n'importe quel autre utilisateur simplement en changeant l'user ID effectif de l'un de ses processus. Le système laissera un processus disposant d'un user ID effectif de 0 modifier son user ID à sa guise (encore une fois, remarquez la puissance dont dispose un processus s'exécutant en tant que root ! Un processus dont l'user ID effectif est 0 peut faire tout ce qu'il lui plaît). Tout autre processus, par contre, ne peut faire que l'une des actions suivantes :

- définir son user ID effectif pour qu'il soit le même que son user ID réel ;
- définir son user ID réel pour qu'il soit le même que son user ID effectif ;
- échanger les deux identifiants.

La première possibilité serait utilisée par notre système de comptabilité lorsqu'il a terminé d'accéder au fichiers en tant que mitchell et veut redevenir root. La seconde par un programme de connexion une fois qu'il a défini l'user ID effectif pour qu'il soit celui de l'utilisateur qui s'est connecté. Définir l'user ID réel permet de s'assurer que l'utilisateur ne pourra jamais redevenir root. L'échange des deux identifiants est avant tout une fonctionnalité historique, les programmes modernes l'utilisent rarement.

Vous pouvez passer -1 à la place de l'un des deux arguments de `setreuid` si vous ne voulez pas modifier l'user ID correspondant. Il existe également une fonction raccourci appelée `seteuid`.

Cette fonction définit l'user ID effectif mais ne modifie pas l'user ID réel. Les deux instructions suivantes font toutes deux la même chose :

```
seteuid (id);
setreuid (-1, id);
```

10.4.1 Programmes setuid

En utilisant les techniques présentées ci-dessus, vous êtes en mesure de créer des processus root qui s'exécutent sous une autre identité de façon temporaire puis redeviennent root. Vous pouvez également faire abandonner tous ses privilèges à un processus root en redéfinissant ses user ID réel et effectif.

Voici une énigme : un processus non root peut il devenir root ? Cela semble impossible, en utilisant les techniques précédentes, mais voici une preuve que ça l'est :

```
% whoami
mitchell
% su
Password: ...
% whoami
root
```

La commande `whoami` est similaire à `id`, excepté qu'elle n'affiche que l'user ID effectif, pas les autres informations. La commande `su` vous permet de devenir le superutilisateur si vous connaissez le mot de passe root.

Comment fonctionne `su` ? Comme nous savons que le shell initial s'exécute avec des user ID réel et effectif qui sont ceux de mitchell, `setreuid` ne nous permettra pas de les changer.

L'astuce est que le programme `su` est un programme *setuid*. Cela signifie que lorsqu'il s'exécute, son user ID effectif sera celui du propriétaire du fichier et non l'user ID du processus qui effectue l'appel `exec` (l'user ID réel est cependant toujours déterminé par ce dernier). Pour créer un programme `setuid`, utilisez la commande `chmod +s` ou l'option `S_ISUID` si vous appelez `chmod` par programmation[6].

Étudions le programme du Listing 10.3.

Listing 10.3 – (*setuid-test.c*) – **Programme de Démonstration de setuid**

```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
  printf ("uid=%d euid=%d\n", (int) getuid (), (int) geteuid ());
  return 0;
}
```

Supposons maintenant que ce programme est en mode setuid et que root en est le propriétaire. Dans ce cas, la sortie de `ls -l` ressemblerait à cela :

```
-rwsrws--x 1 root root 11931 Jan 24 18:25 setuid-test
```

[6]Bien sûr, il existe la notion parallèle de programme *setgid*. Lorsqu'un tel programme s'exécute, son group ID effectif est celui du groupe propriétaire du fichier. La plupart des programmes setuid sont également des programmes setgid.

Le bit `s` indique que le fichier n'est pas seulement exécutable (comme l'indiquerait un `x`) mais qu'il est également setuid et setgid. Lorsque nous utilisons ce programme, il affiche quelque chose de ce genre :

```
% whoami
mitchell
% ./setuid-test
uid=501 euid=0
```

Notez que l'user ID effectif est à 0 lorsque le programme s'exécute.

Vous pouvez utiliser la commande `chmod` avec les arguments `u+s` ou `g+s` pour activer les bits setuid et setgid sur un fichier exécutable, respectivement – par exemple :

```
% ls -l program
-rwxr-xr-x    1 samuel csl 0 Jan 30 23:38 program
% chmod g+s program
% ls -l program
-rwxr-sr-x    1 samuel csl 0 Jan 30 23:38 program
% chmod u+s program
% ls -l program
-rwsr-sr-x    1 samuel csl 0 Jan 30 23:38 program
```

Vous pouvez également utiliser l'appel `chmod` avec les indicateurs de mode `S_ISUID` et `S_ISGID`.

`su` est capable de modifier l'user ID effectif par le biais de ce mécanisme. Il s'exécute initialement avec un user ID effectif à 0. Puis il vous demande un mot de passe. Si le mot de passe concorde avec celui de root, il positionne son user ID réel de sorte qu'il soit celui de root puis lance un nouveau shell. Dans le cas contraire, il se termine, vous renvoyant à votre shell d'utilisateur non privilégié.

Observons les permissions du programme su :

```
% ls -l /bin/su
-rwsr-xr-x    1 root root 14188 Mar 7 2000 /bin/su
```

Notez que root est le propriétaire et que le bit setuid est actif.

Remarquez que `su` ne change pas réellement l'user ID du shell à partir duquel il a été lancé. Au lieu de cela, il lance un nouveau shell avec le nouvel user ID. Le shell original est bloqué jusqu'à ce que le nouveau se termine.

10.5 Authentifier les utilisateurs

Souvent, lorsque vous créez un programme setuid, vous souhaitez n'en autoriser l'accès qu'à certains utilisateurs. Par exemple, le programme `su` ne vous laisse devenir root que si vous disposez du mot de passe correspondant. Le programme vous oblige à prouver que vous pouvez devenir root avant de vous permettre de faire quoi que ce soit. Ce mécanisme est appelé authentification – le programme su vérifie que vous êtes celui que vous prétendez être.

Si vous administrez un système très sécurisé, l'authentification des utilisateurs par le biais d'un simple mot de passe ne vous satisfera probablement pas. Les utilisateurs ont tendance à noter leur mot de passe et les personnes mal intentionnées à les découvrir. Les utilisateurs

choisissent généralement des mots de passe comme leur date de naissance, le nom de leur animal de compagnie, *etc*[7]. Les mot de passe ne sont finalement pas une bonne garantie de sécurité.

Par exemple, beaucoup d'organisations utilisent désormais un système de mots de passe à « usage unique » générés par des cartes d'identité électronique que les utilisateurs gardent sur eux. Le même mot de passe ne peut être utilisé deux fois et vous ne pouvez obtenir de mot de passe valide à partir de la carte qu'en entrant un code d'identification. L'attaquant doit donc obtenir la carte et le code pour accéder au système. Dans des complexes extrêmement sécurisés, les scan rétiniens et d'autres types de systèmes d'identification biométriques sont utilisés.

Si vous écrivez un programme qui doit authentifier ses utilisateurs, vous devriez permettre à l'administrateur système de sélectionner le moyen d'authentification qu'il souhaite utiliser. GNU/Linux propose une bibliothèque très utile pour vous faciliter la tâche. Ce mécanisme, appelé *Pluggable Authentication Modules* (Modules d'Authentification Enfichable), ou PAM, facilite l'écriture d'application qui authentifient leurs utilisateurs comme le désire l'administrateur système.

Il est plus simple de comprendre le fonctionnement de PAM en étudiant une application PAM simple. Le Listing 10.4 illustre l'utilisation de PAM.

Listing 10.4 – (*pam.c*) – **Exemple d'Utilisation de PAM**

```
1  #include <security/pam_appl.h>
2  #include <security/pam_misc.h>
3  #include <stdio.h>
4
5  int main ()
6  {
7    pam_handle_t* pamh;
8    struct pam_conv pamc;
9
10   /* Initialise la conversation PAM. */
11   pamc.conv = &misc_conv;
12   pamc.appdata_ptr = NULL;
13   /* Démarre une nouvelle session d'authentification. */
14   pam_start ("su", getenv ("USER"), &pamc, &pamh);
15   /* Authentifie l'utilisateur. */
16   if (pam_authenticate (pamh, 0) != PAM_SUCCESS)
17     fprintf (stderr, "Échec de l'authentification !\n");
18   else
19     fprintf (stderr, "Authentification OK.\n");
20   /* Fini. */
21   pam_end (pamh, 0);
22   return 0;
23 }
```

Pour compiler ce programme, vous devez le lier à deux bibliothèques : `libpam` et une bibliothèque utilitaire appelée `libpam_misc` :

```
% gcc -o pam pam.c -lpam -lpam_misc
```

Ce programme commence par construire un objet de conversation PAM. Cet objet est utilisé par la bibliothèque PAM lorsqu'elle a besoin d'obtenir des informations de la part des utilisateurs.

[7] On a découvert que les administrateurs système avaient tendance à choisir le mot *dieu* pour mot de passe plutôt que n'importe quel autre (pensez-en ce que vous voulez). Aussi, si vous avez besoin d'un accès root sur une machine et que l'administrateur n'est pas là, une inspiration divine pourra peut être vous aider.

La fonction `misc_conv` utilisée dans cet exemple est une fonction standard utilisant le terminal pour les entrées sorties. Vous pouvez écrire votre propre fonction qui affiche une boîte de dialogue, utilise la parole pour les entrées/sorties ou propose même des méthodes d'interactions plus exotiques.

Le programme appelle ensuite `pam_start`. Cette fonction initialise la bibliothèque PAM. Le premier argument est un nom de service. Vous devez utiliser un nom qui identifie de façon unique votre application. Par exemple, si votre application s'appelle `whizbang`, vous devriez utiliser ce nom pour le service. Cependant, le programme ne fonctionnera probablement pas à moins qu'un administrateur système ne configure explicitement le système pour fonctionner avec votre service. Revenons à notre exemple, nous utilisons le service `su`, qui indique que notre programme authentifie les utilisateurs de la même façon que la commande `su`. Vous ne devriez pas utiliser cette technique dans un programme réel. Sélectionnez un nom de service qui vous est propre et concevez vos scripts d'installation pour aider l'administrateur à configurer PAM correctement pour votre application.

Le second argument est le nom de l'utilisateur que vous voulez authentifier. Dans cet exemple, nous utilisons la valeur de la variable d'environnement `USER` (en théorie, il s'agit du nom d'utilisateur correspondant à l'user ID effectif du processus courant, mais ce n'est pas toujours le cas). Dans la plupart des programmes réels, vous afficheriez une invite pour saisir le nom d'utilisateur. Le troisième argument est la conversation PAM, que nous avons présentée précédemment. L'appel à `pam_start` renseigne le handle passé en quatrième argument. Passez ce handle aux appels ultérieurs aux fonctions de la bibliothèque PAM.

Ensuite, le programme appelle `pam_authenticate`. Le second argument vous permet de spécifier diverses options ; la valeur 0 demande l'utilisation des valeurs par défaut. La valeur de retour de cette fonction indique le résultat de l'authentification.

Finalement, le programme appelle `pam_end` pour libérer toutes les structures de données allouées.

Supposons que le mot de passe pour l'utilisateur courant soit « `password` » (un mot de passe extrêmement faible). Alors, l'exécution de ce programme avec le mot de passe correct produit le comportement attendu :

```
% ./pam
Password: password
Authentification OK.
```

Si vous exécutez ce programme dans un terminal, le mot de passe n'apparaîtra probablement pas lorsque vous le saisirez : il est masqué pour éviter qu'une autre personne ne puisse l'apercevoir alors que vous l'entrez.

Par contre, si un hacker tente d'utiliser un mauvais mot de passe, la bibliothèque PAM signalera l'échec correctement :

```
% ./pam
Password: raté
Échec de l'authentification !
```

Les bases que nous avons présenté sont suffisantes pour la plupart des programmes simples. Une documentation complète sur le fonctionnement de PAM est disponible sous `/usr/doc/pam` sur la plupart des systèmes GNU/Linux.

10.6 Autres failles de sécurité

Bien que ce chapitre présente quelques failles de sécurité répandues, vous ne devez en aucun cas compter sur ce livre pour couvrir toutes les failles possibles. Beaucoup on déjà été trouvés et beaucoup plus attendent de l'être. Si vous essayez d'écrire du code sécurisé, il n'y a réellement pas d'autre solution que de faire appel à un expert pour un audit de code.

10.6.1 Dépassement de tampon

Pratiquement toutes les applications Internet majeures, y compris `sendmail`, `finger`, `tal` et d'autres, ont a un moment donné été victimes de failles dites de *dépassement de tampon*.

Si vous écrivez du code destiné à être exécuté en tant que root, vous devez absolument être familier avec ce type de failles de sécurité. Cela s'applique également si vous écrivez un programme qui utilise les mécanismes de communication interprocessus. Si vous écrivez un programme qui lit des fichiers (ou pourrait lire des fichiers) vous devez là aussi connaître les concepts de cette faille. Ce dernier critère s'applique à presque tous les programmes. Fondamentalement, si vous avez l'intention d'écrire des applications GNU/Linux, vous devez connaître ce type de failles.

L'idée sous-jacente d'une attaque par dépassement de tampon est de faire exécuter à un programme du code qu'il n'était pas censé exécuter. Le mode opératoire habituel est d'écraser une partie de la pile du processus. La pile du programme contient, entre autres, l'adresse mémoire à laquelle le programme doit transférer le contrôle à la fin de la fonction en cours. Ainsi, si vous placez le code que vous voulez exécuter quelque part en mémoire et que vous modifiez l'adresse de retour pour pointer à cet emplacement, vous pouvez faire exécuter n'importe quoi au programme. Lorsque le programme terminera la fonction en cours d'exécution, il sautera vers le nouveau code et exécutera ce qu'il contient, avec les privilèges du processus en cours. Il est clair que si le programme s'exécute en tant que root, ce serait un désastre. Si le processus s'exécute avec les privilèges d'un autre utilisateur, ce n'est un désastre « que » pour cet utilisateur – et par conséquent pour les utilisateurs dépendants de ses fichiers.

Si le programme s'exécute en tant que démon en attente de connexions réseau, la situation est encore pire. Un démon s'exécute habituellement en tant que root. S'il contient des bugs de type dépassement de tampon, n'importe quelle personne pouvant se connecter à l'ordinateur exécutant le démon peut en prendre le contrôle en envoyant une séquence de données au démon *via* le réseau. Un programme qui n'utilise pas les communications réseau est plus sûr car seuls les utilisateurs disposant d'un compte sur la machine peuvent l'attaquer.

Les versions de `finger`, `talk` et `sendmail` concernées par ce type de bug partageaient toutes la même faille. Toutes utilisaient un tampon de taille fixe pour lire une chaîne, ce qui impliquait une limite supérieure constante pour la taille de la chaîne, mais permettaient quand même aux clients d'envoyer des chaînes plus grandes que le tampon. Voici le genre de code qu'elles pouvaient contenir :

```
#include <stdio.h>

int main ()
{
  /* Personne de censé n'aurait plus de 32 caractères dans son nom
     d'utilisateur. De plus, il me semble qu'UNIX ne permet que des
```

```
    nom de 8 caractères. Il y a donc suffisamment de place. */
  char username[32];
  /* Demande le nom de l'utilisateur. */
  printf ("Saisisez votre identifiant de connexion : ");
  /* Lit une ligne saisie par l'utilisateur. */
  gets (username);
  /* Traitements divers... */

  return 0;
}
```

L'utilisation conjointe d'un tampon de 32 caractères et de la fonction `gets` ouvre la porte à un dépassement de tampon. La fonction `gets` lit la saisie de l'utilisateur jusqu'à ce qu'un caractère de nouvelle ligne apparaîsse et stocke le résultat dans le tampon `username`. Les commentaires dans le code sont corrects en ce sens que les utilisateurs ont généralement des identifiants courts, aucun utilisateur bien intentionné ne saisirait plus de 32 caractères. Mais vous écrivez un logiciel sécurisé, vous devez adopter le point de vue d'un attaquant. Dans ce cas, l'attaquant pourrait délibérément saisir un nom d'utilisateur très long. Les variables locales comme `username` sont stockées dans la pile, aussi, en dépassant les limites du tableau, il est possible de placer des octets arbitraires sur la pile au delà de la zone réservée à la variable username. Le nom d'utilisateur dépasse alors le tampon et écrase une partie de la pile, permettant une attaque telle que celle décrite précédemment.

Heureusement, il est facile d'éviter les dépassements de tampon. Lorsque vous lisez des chaînes, vous devriez toujours utiliser soit une fonction, comme `getline`, qui alloue dynamiquement suffisamment d'espace soit une fonction qui interrompt la lecture lorsque le tampon est plein. Voici un exemple d'utilisation de getline :

```
char* username = getline (NULL, 0, stdin);
```

Cet appel utilise automatiquement `malloc` pour allouer un tampon suffisamment grand pour contenir la ligne et vous le renvoie. Vous ne devez pas oublier d'appeler free pour libérer le tampon afin d'éviter les fuites mémoire.

Vous vous faciliterez la vie si vous utiliser le C++ ou un autre langage proposant des primitives simples pour lire les saisies utilisateur. En C++, par exemple, vous pouvre utiliser cette simple instruction :

```
string username;
getline (cin, username);
```

La chaîne username sera également désallouée automatiquement, vous n'avez pas besoin d'appeler `free`[8].

Bien sûr, les dépassements de tampon peuvent survenir avec n'importe quel tableau dimensionné de façon statique, pas seulement avec les chaînes de caractères. Si vous voulez produire un code sécurisé, vous ne devriez jamais écrire dans une structure de données, sur la pile ou ailleurs, sans vérifier que vous n'allez pas dépasser ses limites.

[8]Certains programmeurs pensent que le C++ est un langage horrible et compliqué. Leurs arguments sur l'héritage multiple et d'autres complication ont un certain mérite, mais il est plus simple d'écrire du code évitant les dépassements de tampon et autres problèmes similaires en C++ qu'en C.

10.6.2 Conditions de concurrence critique dans */tmp*

Une autre problème très répandu converne la création de fichiers avec des noms prédictibles, typiquement dans le répertoire `/tmp`. Supposons que votre programme `prog`, qui s'exécute avec les droits root, crée toujours un fichier temporaire appelé `/tmp/prog` et y écrive des informations vitales. Un utilisateur mal intentionné pourrait créer un lien symbolique sous `/tmp/prog` vers n'importe quel fichier du système. Lorsque votre programme tente de créer le fichier, l'appel système `open` n'échouera pas. Cependant, les données que vous écrirez n'iront pas vers `/tmp/prog` mais seront écrites dans le fichier choisi par l'attaquant.

On dit de ce genre d'attaque qu'elle exploite un condition de concurrence critique. Il y a une concurrence implicite entre vous et l'attaquant. Celui qui arrive à créer le fichier en premier gagne.

Cette attaque est généralement utilisée pour détruire des éléments importants du système de fichiers. En créant les liens appropriés, l'attaquant peut utiliser un programme s'exécutant en tant que root croyant écrire dans un fichier temporaire pour écraser un fichier système important. Par exemple, en créant un lien symbolique vers `/etc/passwd`, l'attaquant peut effacer la base de données des mots de passe du système. Il existe également des moyens pour l'attaquant d'obtenir un accès root en utilisant cette technique.

Une piste pour éviter ce genre d'attaque serait d'utiliser un nom aléatoire pour le fichier. Par exemple, vous pourriez utiliser `/dev/random` pour injecter une partie aléatoire dans le nom du fichier. Cela complique bien sûr la tâche de l'attaquant pour deviner le nom du fichier, mais cela ne l'en empêche pas. Il pourrait créer un nombre conséquent de liens symboliques en utilisant beaucoup de nom potentiels. Même s'il doit essayer 10 000 fois avant d'obtenir des conditions de concurrence critique, cette seule fois peut être désastreuse.

Une autre approche est d'utiliser l'option `O_EXCL` lors de l'appel à `open`. Cette option provoque l'échec de l'ouverture si le fichier existe déjà. Malheureusement, si vous utilisez le Network File System (NFS), ou si un utilisateur de votre programme est susceptible d'utiliser NFS, cette approche n'est pas assez robuste car `O_EXCL` n'est pas fiable sur un système de fichier NFS. Vous ne pouvez pas savoir avec certitude si votre code sera utilisé sur un système disposant de NFS, aussi, si vous êtes paranoïaque, ne vous reposez pas sur `O_EXCL`.

Dans le Chapitre 2, « Écrire des logiciels GNU/Linux de qualité », Section 2.1.7, « Utilisation de fichiers temporaires », nous avons présenté `mkstemp`. Malheureusement, sous Linux, `mkstemp` ouvre le fichier avec l'option `O_EXCL` après avoir déterminé un nom suffisamment dur à deviner. En d'autres termes, l'utilisation de `mkstemp` n'est pas sûre si `/tmp` est monté via NFS[9]. L'utilisation de `mkstemp` est donc mieux que rien mais n'est pas totalement sûre.

Une approche qui fonctionne est d'utiliser `lstat` sur le nouveau fichier (`lstat` est présenté dans la Section B.2, « *stat* »). La fonction `lstat` est similaire à `stat`, excepté que si le fichier est lien symbolique, `lstat` vous donne des informations sur ce lien et non sur le fichier vers lequel il pointe. Si `lstat` vous indique que votre nouveau fichier est un fichier ordinaire, pas un lien symbolique, et que vous en êtes le propriétaire, alors tout devrait bien se passer.

Le Listing 10.5 présente une fonction tentant d'ouvrir un fichier dans `/tmp` de façon sécurisée. Les auteurs de ce livre ne l'ont pas fait audité de façon professionnelle et ne sont pas non plus

[9]Bien sûr, si vous êtes administrateur système, vous ne devriez pas monter un système NFS sur `/tmp`.

des experts en sécurité, il y a donc de grandes chances qu'elle ait une faiblesse. Nous ne vous recommandons pas son utilisation sans l'avoir faite auditer, mais elle devrait vous convaincre que l'écriture de code sécurisé est complexe. Pour vous dissuader encore plus, nous avons délibérément défini l'interface de façon à ce qu'elle soit complexe à utiliser dans un programme réel. La vérification d'erreurs tient une place importante dans l'écriture de logiciels sécurisés, nous avons donc inclus la logique de contrôle d'erreurs dans cet exemple.

Listing 10.5 – (*temp-file.c*) – **Créer un Fichier Temporaire**

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

/* Renvoie le descripteur de fichier d'un nouveau fichier temporaire.
   Le fichier pourra être lu ou écrit par l'user ID effectif du processus
   courant et par personne d'autre.

   Renvoie -1 si le fichier temporaire ne peut pas être créé.  */

int secure_temp_file ()
{
  /* Ce descripteur de fichier pointe vers /dev/random et nous permet
     de disposer d'une bonne source de nombres aléatoires. */
  static int random_fd = -1;
  /* Entier aléatoire. */
  unsigned int random;
  /* Tampon utilisé pour convertir random en chaîne de caractères.
     Le tampon à une taille fixe, ce qui signifie que nous sommes
     potentiellement vulnérables à un bug de dépassement de tampon si
     les entiers de la machine d'exécution tiennent sur un nombre
     *conséquent* de bits. */
  char filename[128];
  /* Descripteur de fichier du nouveau fichier temporaire. */
  int fd;
  /* Informations sur le nouveau fichier. */
  struct stat stat_buf;

  /* Si nous n'avons pas encore ouvert /dev/random nous le faisons
     maintenant. Cette façon de faire n'est pas threadsafe. */
  if (random_fd == -1) {
  /* Ouvre /dev/random. Notez que nous supposons ici que /dev/random est
     effectivement une source de bits aléatoire et non pas un fichier
     rempli de zéros placé ici par l'attaquant. */
  random_fd = open ("/dev/random", O_RDONLY);
  /* Abandonne si l'on ne peut pas ouvrir /dev/random. */
  if (random_fd == -1)
    return -1;
  }

  /* Lit un entier à partir de /dev/random. */
  if (read (random_fd, &random, sizeof (random)) !=
      sizeof (random))
    return -1;
  /* Crée un fichier à partir du nombre aléatoire. */
  sprintf (filename, "/tmp/%u", random);

  /* Tente d'ouvrir le fichier. */
  fd = open (filename,
             /* Nous utilisons O_EXECL,
```

```
52                  même si cela ne fonctionne pas avec NFS. */
53               O_RDWR | O_CREAT | O_EXCL,
54               /* Personne ne doit pouvoir lire ou écrire dans le fichier. */
55               S_IRUSR | S_IWUSR);
56    if (fd == -1)
57      return -1;
58    /* Appelle lstat sur le fichier afin de s'assurer qu'il
59       ne s'agit pas d'un lien symbolique. */
60    if (lstat (filename, &stat_buf) == -1)
61      return -1;
62    /* Si le fichier n'est pas un fichier traditionnel, quelqu'un
63       a tenté de nous piéger. */
64    if (!S_ISREG (stat_buf.st_mode))
65      return -1;
66    /* Si le fichier ne nous appartient pas, quelqu'un d'autre pourrait
67       le supprimer, le lire ou le modifier alors que nous nous en
68       servons. */
69    if (stat_buf.st_uid != geteuid () || stat_buf.st_gid != getegid ())
70      return -1;
71    /* Si il y a d'autres bits de permissions actifs,
72       quelque chose cloche. */
73    if ((stat_buf.st_mode & ~(S_IRUSR | S_IWUSR)) != 0)
74      return -1;
75
76    return fd;
77  }
```

Cette fonction appelle `open` pour créer le fichier puis appelle `lstat` quelques lignes plus loin pour s'assurer que le fichier n'est pas un lien symbolique. Si vous réfléchissez attentivement, vous réaliserez qu'il semble y avoir une condition de concurrence critique dans ce cas. En effet, un attaquant pourrait supprimer le fichier et le remplacer par un lien symbolique entre le moment où nous appelons `open` et celui où nous appelons `lstat`. Cela n'aurait pas d'impact direct sur cette fonction car nous avons déjà un descripteur de fichier ouvert pointant sur le nouveau fichier, mais nous indiquerions une erreur à l'appelant. Cette attaque ne causerait pas de dommages directs mais rendrait impossible le fonctionnement de l'appelant. Une telle attaque est dite déni de service (DoS, Denial of Service).

Heureusement, le sticky bit vient à notre aide. Comme le sticky bit est actif sur `/tmp`, personne d'autre que nous ne peut supprimer les fichiers de ce répertoire. Bien sûr, root peut toujours supprimer des fichiers, mais si l'attaquant dispose déjà des privilèges root, il n'y a rien qui puisse protéger votre programme.

Si vous choisissez de supposer que l'administrateur système est compétent, alors `/tmp` ne sera pas monté *via* NFS. Et si l'administrateur système est suffisamment stupide pour monter `/tmp` en NFS, il y a de bonnes chances que le sticky bit ne soit pas actif non plus. Aussi, pour la plupart des utilisations, nous pensons qu'il est sûr d'utiliser `mkstemp`. Mais vous devez être conscient de ces problèmes et ne devez pas vous reposer sur le fonctionne de `O_EXCL` pour un fonctionnement correct si le répertoire utilisé n'est pas `/tmp` – pas plus que vous ne devez supposer que le sticky bit est actif ailleurs.

10.6.3 Utilisation de *system* ou *popen*

La troisième faille de sécurité que tout programme devrait avoir en tête est l'utilisation du shell pour exécuter d'autres programme. Prenons l'exemple fictif d'un serveur dictionnaire. Ce

programme est conçu pour accepter les connexions venant d'Internet. Chaque client envoie un mot et le serveur indique s'il s'agit d'un mot anglais valide. Comme tout système GNU/Linux dispose d'une liste d'environ 45000 mots anglais dans `/usr/share/dict/word`, une façon simple de créer ce serveur est d'invoquer le programme `grep`, comme ceci :

```
% grep -x word /usr/dict/words
```

Ici, `word` est le mot que souhaite valider l'utilisateur. Le code de sortie de `grep` nous indiquera si le mot figure dans `/usr/share/dict/words`[10].

Le Listing 10.6 vous montre comment vous pourriez coder la partie du serveur invoquant `grep` :

Listing 10.6 – (*grep-dictionary.c*) – **Cherche un Mot dans le Dictionnaire**

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

/* Renvoie une valeur différente de 0 si et seulement si WORD
   figure dans /usr/dict/words. */

int grep_for_word (const char* word)
{
  size_t length;
  char* buffer;
  int exit_code;

/* Construit la chaîne "grep -x WORD /usr/dict/words". Alloue la chaîne
   dynamiquement pour éviter les dépassements de tampon. */
  length =
    strlen ("grep -x ") + strlen (word) + strlen (" /usr/dict/words") + 1;
  buffer = (char*) malloc (length);
  sprintf (buffer, "grep -x %s /usr/dict/words", word);

  /* Exécute la commande. */
  exit_code = system (buffer);
  /* Libère le tampon. */
  free (buffer);
  /* Si grep a renvoyé 0, le mot était présent dans le dictionnaire. */
  return exit_code == 0;
}
```

Remarquez qu'en calculant le nombre de caractères dont nous avons besoin et en allouant le tampon dynamiquement, nous sommes sûrs d'éviter les dépassements de tampon.

Malheureusement, l'utilisation de la fonction `system` (décrite dans le Chapitre 3, « Processus », Section 3.2.1, « Utiliser *system* ») n'est pas sûr. Cette fonction invoque le shell système standard pour lancer la commande puis renvoyer la valeur de sortie. Mais que se passe-t-il si un attaquant envoie un « mot » qui est fait la ligne suivante ou quelque chose du même type ?

```
foo /dev/null; rm -rf /
```

Dans ce cas, le serveur exécutera cette commande :

```
grep -x foo /dev/null; rm -rf / /usr/dict/words
```

[10] Si vous ne connaissez pas `grep`, vous devriez consulter les pages de manuel. C'est un programme incoyablement utile.

Le problème est maintenant évident. L'utilisateur a transformé une commande, l'invocation de `grep`, en deux commandes car le shell traite le point virgule comme un séparateur de commandes. La première commande est toujours l'invocation inoffensive de `grep`, mais la seconde supprime tous les fichiers du système ! Même si le serveur ne s'exécute pas en tant que root, tous les fichiers qui peuvent être supprimés par l'utilisateur sous lequel s'exécute le serveur seront supprimés. Le même problème peut survenir avec `popen` (décrit dans la Section 5.4.4, « *popen* et *pclose* »), qui crée un pipe entre le processus père et le fils mais utilise quand même le shell pour lancer la commande.

Il y a deux façons d'éviter ces problèmes. La première est d'utiliser les fonctions de la famille `exec` au lieu de `system` et `popen`. Cette solution contourne le problème car les caractères considérés comme spéciaux par le shell (comme le point-virgule dans la commande précédente) ne sont pas traités lorsqu'ils apparaissent dans les arguments d'un appel à exec. Bien sûr, vous abandonnez le côté pratique de `system` et `popen`.

L'autre alternative est de valider la chaîne pour s'assurer qu'elle n'est pas dangereuse. Dans l'exemple du serveur dictionnaire, vous devriez vous assurer que le mot ne contient que des caractères alphabétiques, en utilisant la fonction `isalpha`. Si elle ne contient pas d'autres caractères, il n'y a aucun moyen de piéger le shell en lui faisant exécuter une autre commande. N'implémentez pas la vérification en recherchant des caractères dangereux ou inattendus ; il est toujours plus sûr de rechercher explicitement les caractères dont vous savez qu'ils sont sûrs que d'essayer d'anticiper sur tous les caractères pouvant être problématiques.

Chapitre 11

Application GNU/Linux d'Illustration

Ce chapitre est celui ou tout s'imbrique. Nous allons spécifier et implanter un programme GNU/Linux complet qui fait appel à la plupart des techniques décrites dans ce livre. Ce programme fournit des informations sur le système sur lequel il s'exécute par le biais d'une interface Web.

L'application est une application pratique de certaines des méthodes que nous avons décrites dans le cadre de la programmation GNU/Linux et illustrées dans des programmes plus courts. Celui-ci se rapproche plus du "monde réel", contrairement à la plupart des listings des chapitres précédents. Il peut vous servir de tremplin pour écrire vos propres logiciels GNU/Linux.

11.1 Présentation

Ce programme fait partie d'un système de surveillance pour un système GNU/Linux en cours d'éxécution. Voici ses fonctionalités :

- Il incorpore un serveur Web minimal. Les clients locaux ou distants accèdent aux informations en demandant des pages au server *via* le protocole HTTP ;
- Il n'utilise pas de pages HTML statique mais les génère à la volée par le biais de modules, chacun d'eux produisant une page informant sur un aspect de l'état du système ;
- Les modules ne sont pas liés statiquement avec l'exécutable serveur. Ils sont chargés dynamiquement à partir de bibliothèques partagées. Des modules peuvent être ajoutés, supprimés ou remplacés alors que le serveur est en cours d'exécution ;
- Chaque connexion est traitée dans un processus fils. Cela permet au serveur de rester réactif même lorsque certaines requêtes nécessitent un temps de traitement important et cela protège le serveur contre d'éventuels problèmes dans les modules ;
- Le serveur ne nécessite pas les privilèges superutilisateur pour s'exécuter (tant qu'il n'utilise pas un port privilégié). Cependant, cela limite les informations pouvant être collectées.

Nous présentons quatre modules illustrant la façon dont ils peuvent être écrits. Ils exploitent plus en profondeur certaines techniques de collecte d'informations présentées précédemment dans ce livre. Le module `time` utilise l'appel système `gettimeofday`, le module `issue` exploite les E/S de bas niveau et l'appel système `sendfile`, `diskfree` illustre l'utilisation de `fork`, `exec` et `dup2` en

lançant une commande dans un processus fils, enfin, le module `processes` utilise le système de fichiers `/proc` et divers appels système.

11.1.1 Limitations

Ce programme dispose d'un nombre important de fonctionalité que vous pouvez trouver dans une application réelle, comme l'analyse de la ligne de commande et la vérification d'erreurs. Mais, dans le même temps, nous avons simplifier les choses afin d'améliorer la lisibilité et nous concentrer sur les sujets spécifiques à GNU/Linux traités tout au long de ce livre. Gardez en tête les limitations suivantes lors de votre lecture du code :

- Nous n'essayons pas d'implanter le protocole HTTP dans son intégralité. Au contraire, nous en implémentons juste assez pour que le serveur puisse interagir avec des clients Web. Un programme réel fournirait une implantation plus aboutie ou s'interfacerait avec l'un des nombreux serveurs Web[1] disponibles ;
- De même, nous ne visons pas une compatibilité complète avec les spécifications HTML (consultez `http://www.w3.org/MarkUppourplusd'informations`. Nous générons un HTML simple pris en charge par les navigateurs les plus répandus ;
- Le serveur n'est pas conçu pour maximiser les performances ou minimiser l'utilisation des ressources. En particulier, nous avons intentionnellement omis une partie du code de configuration du réseau auquel vous pourriez vous attendre dans un serveur Web. Ce sujet sort du cadre de ce livre. Consultez une des nombreuses excellentes références sur le développement d'application réseau, comme *Unix Network Programming, Volume 1 : Networking APIx – Sockets and XTI* de W. Richard Stevens (Prentice Hall, 1997) pour plus d'informations.
- Nous ne tentons pas de réguler l'utilisation des ressources (nombre de processus, consommation mémoire, *etc.*) par le serveur ou ses modules. Beaucoup d'implantations de serveurs Web répondent aux connexions en utilisant un nombre fixe de processus plutôt que d'en créer un nouveau à chaque connexion ;
- Le serveur charge la bibliothèque partagée correspondant à un module à chaque fois qu'une requête y faisant appel est reçue puis le décharge immédiatement une fois qu'elle a été traitée. Une implantation plus efficace utiliserait certainement un cache pour les modules chargés.

11.2 Implantation

Excepté lorsqu'ils sont très concis, les programmes écrits en C demande une organisation soigneuse afin d'assurer la modularité et la maintenabilité du code. Ce programme est divisé en quatre fichiers source principaux.

Chaque fichier exporte des fonctions ou des variables qui peuvent accéder depuis d'autres endroits du programme. Pour des raisons de simplicité, toutes les fonctions et variables exportées

[1] Le serveur Web open source le plus populaire sous GNU/Linux est le serveur Apache, disponible sur `http://www.apache.org`

HTTP

L'*Hypertext Transport Protocol* (HTTP) est utilisé pour la communication entre clients et serveurs Web. Le client se connecte au serveur *via* un port connu (généralement le port 80, mais il peut s'agir de n'importe quel port). Les requêtes et les en-têtes HTTP sont au format texte.

Une fois connecté, le client envoie une requête au serveur. Une requête classique est `GET /page HTTP/1.0`. La méthode `GET` indique que le client demande au serveur l'envoi d'une page. Le second élément est le chemin de la page sur le serveur et le troisième est constitué du protocole et de sa version. Les lignes suivantes sont constituées d'en-têtes, dont le format est similaire à celui des en-têtes mail, qui contiennent des informations supplémentaires sur le client. L'en-tête se termine par une ligne vide.

Le serveur renvoie une réponse indiquant le résultat du traitement de la requête. Une réponse classique est `HTTP/1.0 200 OK`. Le premier élément est la version du protocol et les deux suivants indiquant le résultat ; dans ce cas, 200 signifie que la requête a été correctement traitée. Les lignes suivantes contiennent des en-têtes, là encore similaires aux en-têtes mail. L'en-tête se termine par une ligne blanche. Le serveur peut envoyer des données quelconques pour satisfaire la requête.

Typiquement, le serveur répond à la demande d'une page en envoyant sa source HTML. Dans ce cas, les en-tête de réponse incluront `Content-type: text/html` indiquant que le résultat est au format HTML. La source HTML du document suit immédiatement l'en-tête.

Consultez la spécification HTTP disponible sur `http://www.w3.org/Protocols/` pour plus d'informations.

sont définies dans un unique fichier d'en-tête, `server.h` (*cf.* Listing 11.1), qui est inclus par les autres fichiers. Les fonctions destinées à n'être utilisées que dans une seule unité de compilation sont déclarées comme `static` et ne sont donc pas mentionnées dans `server.h`.

Listing 11.1 – (*server.h*) – **Déclarations des Fonctions et Variables**

```
1  #ifndef SERVER_H
2  #define SERVER_H
3
4  #include <netinet/in.h>
5  #include <sys/types.h>
6
7  /*** Symboles definis dans common.c.   ***********************************/
8
9  /* Nom du programme. */
10 extern const char* program_name;
11
12 /* Mode verbeux si différent de zéro.   */
13 extern int verbose;
14
15 /* Identique à malloc, sauf que le programme se termine en cas d'échec.   */
16 extern void* xmalloc (size_t size);
17
18 /* Identique à realloc, sauf que le programme se termine en cas d'échec.   */
19 extern void* xrealloc (void* ptr, size_t size);
20
21 /* Identique à strdup, sauf que le programme se termine en cas d'échec.   */
22 extern char* xstrdup (const char* s);
23
24 /* Affiche un message d'erreur suite à l'appel de OPERATION, utilise
25     la valeur de errno et termine le programme. */
26 extern void system_error (const char* operation);
27
28 /* Affiche un message d'erreur lors d'un échec dont la cause est CAUSE,
29     le MESSAGE descriptif est affiché et le programme terminé. */
30 extern void error (const char* cause, const char* message);
31
32 /* Renvoie le répertoire contenant l'exécutable du programme.
33     La valeur de retour est un tampon mémoir que l'appelant doit
34     libérer. Cette fonction appelle abort en cas d'échec. */
35 extern char* get_self_executable_directory ();
36
37
38 /*** Symboles définis dans module.c   ************************************/
39
40 /* Instance de module chargé. */
41 struct server_module {
42    /* Référence sur la bibliothèque partagée correspondant au module.  */
43    void* handle;
44    /* Nom du module. */
45    const char* name;
46    /* Fonction générant les résultats au format HTML pour le module.  */
47 void (* generate_function) (int);
48 };
49
50 /* Répêrtoire à partir duquel sont chargés les modules.  */
51 extern char* module_dir;
52
53 /* Tente de charger un module dont le nom est MODULE_PATH. Si un module
54    existe à cet emplacement, charge le module et renvoie une structure
55    server_module le représentant. Sinon, renvoie NULL. */
```

```
56  extern struct server_module* module_open (const char* module_path);
57
58  /* Décharge un module et libère l'objet MODULE.  */
59  extern void module_close (struct server_module* module);
60
61
62  /*** Symboles definis dans server.c.  ***********************************/
63
64  /* Lance le serveur sur l'adresse LOCAL_ADDRESS et le port PORT. */
65  extern void server_run (struct in_addr local_address, uint16_t port);
66
67  #endif  /* SERVER_H */
```

11.2.1 Fonctions génériques

`common.c` (*cf.* Listing 11.2 contient des fonctions génériques utilisées dans tous le programme.

Listing 11.2 – (***common.c***) – **Fonctions Génériques**

```
1   #include <errno.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5   #include <unistd.h>
6
7   #include "server.h"
8
9   const char* program_name;
10
11  int verbose;
12
13  void* xmalloc (size_t size)
14  {
15    void* ptr = malloc (size);
16    /* Termine le programme si l'allocation échoue. */
17    if (ptr == NULL)
18      abort ();
19    else
20      return ptr;
21  }
22
23  void* xrealloc (void* ptr, size_t size)
24  {
25    ptr = realloc (ptr, size);
26    /* Termine le programme si l'allocation échoue. */
27    if (ptr == NULL)
28      abort ();
29    else
30      return ptr;
31  }
32
33  char* xstrdup (const char* s)
34  {
35    char* copy = strdup (s);
36    /* Termine le programme si l'allocation échoue. */
37    if (copy == NULL)
38      abort ();
39    else
40      return copy;
41  }
```

```
42
43 void system_error (const char* operation)
44 {
45   /* Génère le message d'erreur correspondant à errno.  */
46   error (operation, strerror (errno));
47 }
48
49 void error (const char* cause, const char* message)
50 {
51   /* Affiche un message d'erreur sur stderr. */
52   fprintf (stderr, "%s: error: (%s) %s\n", program_name, cause, message);
53   /* Termine le programme. */
54   exit (1);
55 }
56
57 char* get_self_executable_directory ()
58 {
59   int rval;
60   char link_target[1024];
61   char* last_slash;
62   size_t result_length;
63   char* result;
64
65   /* Lit la cible du lien symbolique /proc/self/exe. */
66   rval = readlink ("/proc/self/exe", link_target, sizeof (link_target));
67   if (rval == -1)
68     /* L'appel à readlink a échoué, terminé. */
69     abort ();
70   else
71     /* Ajoute un octet nul à la fin de la chaîne. */
72     link_target[rval] = '\0';
73   /* Nous voulons supprimer le nom du fichier exécutable afin d'obtenir
74      le nom du répertoire qui le contient. On recherche le slash le plus à droite.
          */
75   last_slash = strrchr (link_target, '/');
76   if (last_slash == NULL || last_slash == link_target)
77     /* Quelque chose d'inattendu s'est produit. */
78     abort ();
79   /* Alloue un tampon pour accueillir le chemin obtenu. */
80   result_length = last_slash - link_target;
81   result = (char*) xmalloc (result_length + 1);
82   /* Copie le résultat. */
83   strncpy (result, link_target, result_length);
84   result[result_length] = '\0';
85   return result;
86 }
```

Vous pouvez utiliser ces fonctions dans d'autres programmes ; le contenu de ce fichier peut être inclus dans une bibliothèque constituant une base de code pour un certain nombre de projets :

– `xmalloc`, `xrealloc` et `xstrdup` sont des versions avec vérification d'erreur des fonctions `malloc`, `realloc` et `strdup`, respectivement. Contrairement aux fonctions standard qui renvoient un pointeur `NULL` lorsque l'allocation échoue, ces fonctions terminent immédiatement le programme lorsqu'il n'y a pas assez de mémoire disponible.
  La détection précoce de l'échec des allocations mémoire est une bonne idée. Si ce n'est pas fait, les allocations ayant échoué introduisent des pointeurs `NULL` à des endroits inattendus. Comme les échecs d'allocation ne sont pas simples à reproduire, il peut être difficile de réparer de tels problèmes. Les échecs d'allocaltion sont généralement catastrophiques, aussi,

l'arrêt du programme est souvent une réaction appropriée ;
- La fonction `error` sert à signaler une erreur fatale. Elle affiche un message sur `stderr` et termine le programme. Pour les erreurs causées par des appels systèmes ou des fonctions de bibliothèques, `system_error` génère une partie du message d'erreur à partir de la valeur de `errno` (reportez-vous à la Section 2.2.3, « Codes d'erreur des appels système », du Chapitre 2, « Écrire des logiciels GNU/Linux de qualité ») ;
- `get_self_executable_directory` détermine le répertoire contenant le fichier exécutable du processus courant. Ce chemin peut être utilisé pour localiser d'autres composants du programmes installés au même endroit. Cette fonction utilise le lien symbolique `/proc/self/exe` du système de fichiers `/proc` (*cf.* Section 7.2.1, namerefsec :procself du Chapitre 7, « Le système de fichiers */proc* »).

`common.c` définit également deux variables globales très utiles :
- La valeur de `program_name` est le nom du programme en cours d'exécution, déterminé à partir de la liste d'arguments (*cf.* Section 2.1.1, « La liste d'arguments », dans le Chapitre 2). Lorsque le programme est invoqué à partir d'un shell, il s'agit du chemin et du nom de programme saisis par l'utilisateur ;
- La variable `verbose` est différente de zéro si le programme s'exécute en mode verbeux. Dans ce cas, diverses portions du programme affichent des messages sur `stdout`.

11.2.2 Chargement de modules serveur

`module.c` (*cf.* Listing 11.3) fournit l'implémentation des modules serveurs pouvant être chargés dynamiquement. Un module serveur chargé est représenté par une instance de `struct server_module` dont la définition se trouve dans `server.h`.

Listing 11.3 – (*module.c*) – Chargement et Déchargement de Module Serveur

```
1  #include <dlfcn.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  #include "server.h"
7
8  char* module_dir;
9
10 struct server_module* module_open (const char* module_name)
11 {
12   char* module_path;
13   void* handle;
14   void (* module_generate) (int);
15   struct server_module* module;
16
17   /* Détermine le chemin absolu de la bibliothèque partagée correspondant
18      au module que nous essayons de charger. */
19   module_path =
20     (char*) xmalloc (strlen (module_dir) + strlen (module_name) + 2);
21   sprintf (module_path, "%s/%s", module_dir, module_name);
22
23   /* Tente d'ouvrir la bibliothèque partagée MODULE_PATH. */
24   handle = dlopen (module_path, RTLD_NOW);
25   free (module_path);
26   if (handle == NULL) {
```

```
27      /* Échec ; le chemin n'existe pas ou il ne s'agit pas
28         d'une bibliothèque partagée. */
29      return NULL;
30    }
31
32    /* Charge le symbole module_generate depuis la bibliothèque partagée. */
33    module_generate = (void (*) (int)) dlsym (handle, "module_generate");
34    /* S'assure que le symbole existe. */
35    if (module_generate == NULL) {
36    /* Le symbole n'a pas été trouvé. Bien que l'on soit en présence d'une
37       bibliothèque partagée, il ne s'agit probablement pas d'un module
38       serveur. Ferme et indique l'échec de l'opération. */
39      dlclose (handle);
40      return NULL;
41    }
42    /* Alloue et initialise un objet server_module. */
43    module = (struct server_module*) xmalloc (sizeof (struct server_module));
44    module->handle = handle;
45    module->name = xstrdup (module_name);
46    module->generate_function = module_generate;
47    /* Le renvoie, indiquant que tout s'est déroulé correctement. */
48    return module;
49  }
50
51  void module_close (struct server_module* module)
52  {
53    /* Ferme la bibliothèque partagée. */
54    dlclose (module->handle);
55    /* Libère la mémoire allouée pour le nom du module. */
56    free ((char*) module->name);
57    /* Libère la mémoire allouée pour le module. */
58    free (module);
59  }
```

Chaque module est un fichier de bibliothèque partagée (reportez-vous à la Section 2.3.2, « Bibliothèques partagées » du Chapitre 2) et doit définir et exporter une fonction appelée `module_generate`. Cette fonction génère une page HTML et l'écrit vers un descripteur de fichiers correspondant à un socket client.

`module.c` contient deux fonctions :

- `module_open` tente de charger un module serveur dont le nom est connu. Le nom se termine normalement par l'extension `.so` car les modules sont implantés sous forme de bibliothèques partagées. La fonction ouvre la bibliothèque partagée avec `dlopen` et recherche un symbole nommé `module_generate` à partir de cette bibliothèque en utilisant `dlsym` (consultez la Section 2.3.6, « Chargement et déchargement dynamiques », du Chapitre 2). Si la bibliothèque ne peut pas être ouverte ou si `module_generate` n'est pas exporté par celle-ci, l'appel échoue et `module_open` renvoie un pointeur `NULL`. Si tout ce passe bien, elle alloue et renvoie un objet module ;
- `module_close` ferme la bibliothèque partagée correspondant au module serveur et libère l'objet `struct server_module`.

`module.c` définit également une variable globale `module_dir`. Il s'agit du chemin du répertoire dans lequel `module_open` recherche les bibliothèques partagées correspondant aux modules serveur.

11.2.3 Le serveur

`server.c` (*cf.* Listing 11.4) est l'implantation du serveur HTTP minimal.

Listing 11.4 – (*server.c*) – **Implantation du Serveur**

```
1  #include <arpa/inet.h>
2  #include <assert.h>
3  #include <errno.h>
4  #include <netinet/in.h>
5  #include <signal.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <sys/wait.h>
11 #include <unistd.h>
12
13 #include "server.h"
14
15 /* En-tête et réponse HTTP dans le cas d'une requête traitée avec succès. */
16
17 static char* ok_response =
18   "HTTP/1.0 200 OK\n"
19   "Content-type: text/html\n"
20   "\n";
21
22 /* Réponse, en-tête et corps de la réponse HTTP indiquant que nous n'avons
23    pas compris la requête. */
24
25 static char* bad_request_response =
26   "HTTP/1.0 400 Bad Request\n"
27   "Content-type: text/html\n"
28   "\n"
29   "<html>\n"
30   " <body>\n"
31   " <h1>Bad Request</h1>\n"
32   " <p>This server did not understand your request.</p>\n"
33   " </body>\n"
34   "</html>\n";
35
36 /* Réponse, en-tête et modèle de corps indiquant que le document
37    demandé n'a pas été trouvé. */
38
39 static char* not_found_response_template =
40   "HTTP/1.0 404 Not Found\n"
41   "Content-type: text/html\n"
42   "\n"
43   "<html>\n"
44   " <body>\n"
45   " <h1>Not Found</h1>\n"
46   " <p>The requested URL %s was not found on this server.</p>\n"
47   " </body>\n"
48   "</html>\n";
49
50 /* Réponse, en-tête et modèle de corps indiquant que la méthode
51    n'a pas été comprise. */
52
53 static char* bad_method_response_template =
54   "HTTP/1.0 501 Method Not Implemented\n"
55   "Content-type: text/html\n"
56   "\n"
```

```
57    "<html>\n"
58    " <body>\n"
59    " <h1>Method Not Implemented</h1>\n"
60    " <p>The method %s is not implemented by this server.</p>\n"
61    " </body>\n"
62    "</html>\n";
63
64  /* Gestionnaire de SIGCHLD, libère les ressources occupées par les
65     processus fils qui se sont terminés. */
66
67  static void clean_up_child_process (int signal_number)
68  {
69    int status;
70    wait (&status);
71  }
72
73  /* Traite une requête HTTP "GET" pour PAGE et envoie les résultats
74     vers le descripteur de fichier CONNECTION_FD. */
75  static void handle_get (int connection_fd, const char* page)
76  {
77    struct server_module* module = NULL;
78
79    /* S'assure que la page demandée débute avec un slash et n'en
80       contient pas d'autre - nous ne prenons pas en charge les
81       sous-répertoires. */
82    if (*page == '/' && strchr (page + 1, '/') == NULL) {
83      char module_file_name[64];
84
85      /* Le nom de la page a l'air correct. Nous construisons le nom du
86         module en ajoutant ".so" au nom de la page. */
87      snprintf (module_file_name, sizeof (module_file_name),
88                "%s.so", page + 1);
89      /* Tente d'ouvrir le module. */
90      module = module_open (module_file_name);
91    }
92
93    if (module == NULL) {
94      /* Soit la page demandée était malformée, soit nous n'avons pas pu
95         ouvrir le module demandé. Quoi qu'il en soit, nous renvoyons
96         une réponse HTTP 404, Introuvable. */
97      char response[1024];
98
99      /* Génère le message de réponse. */
100     snprintf (response, sizeof (response), not_found_response_template, page);
101     /* L'envoie au client. */
102     write (connection_fd, response, strlen (response));
103   }
104   else {
105     /* Le module demandé a été correctement chargé.  */
106
107     /* Envoie la réponse HTTP indiquant que tout s'est bien passé
108        ainsi que l'en-tête HTTP pour une page HTML. */
109     write (connection_fd, ok_response, strlen (ok_response));
110     /* Invoque le module, il générera la sortie HTML et l'enverra
111        au descripteur de fichier client. */
112     (*module->generate_function) (connection_fd);
113     /* Nous en avons terminé avec le module. */
114     module_close (module);
115   }
116 }
117
118 /* Prend en charge une connexion client sur le descripteur de fichier
```

```
      CONNECTION_FD. */
119
120 static void handle_connection (int connection_fd)
121 {
122   char buffer[256];
123   ssize_t bytes_read;
124
125   /* Lis les données envoyées par le client. */
126   bytes_read = read (connection_fd, buffer, sizeof (buffer) - 1);
127   if (bytes_read > 0) {
128     char method[sizeof (buffer)];
129     char url[sizeof (buffer)];
130     char protocol[sizeof (buffer)];
131
132     /* Les données ont été lues correctement. Ajoute un octet nul à la
133        fin du tampon pour pouvoir l'utiliser avec les fonctions de manipulation
134        de chaîne classiques. */
135     buffer[bytes_read] = '\0';
136     /* La première ligne qu'envoie le client correspond à la requête HTTP composée
137        d'une méthode, de la page demandée et de la version du protocole. */
138     sscanf (buffer, "%s %s %s", method, url, protocol);
139     /* Le client peut envoyer divers en-têtes d'information à la suite de la
         requête.
140        Pour notre implantation du protocole HTTP, nous n'en tenons pas compte.
141        Cependant, nous devons lire toutes les données envoyées par le client.
142        Nous continuons donc à lire les données jusqu'à la fin de l'en-tête
143        qui se termine par une ligne blanche. Le protocole HTTP définit
144        la combinaison CR/LF comme délimiteur de ligne. */
145     while (strstr (buffer, "\r\n\r\n") == NULL)
146        bytes_read = read (connection_fd, buffer, sizeof (buffer));
147     /* S'assure que la dernière lecture n'a pas échoué. Si c'est le cas,
148        il y a un problème avec la connexion, abandonne. */
149     if (bytes_read == -1) {
150        close (connection_fd);
151        return;
152     }
153     /* Vérifie le champ protocole. Nous ne comprenons que les versions 1.0
154        et 1.1. */
155     if (strcmp (protocol, "HTTP/1.0") && strcmp (protocol, "HTTP/1.1")) {
156        /* Nous ne comprenons pas ce protocole. Renvoie une indication
157           de requête incorrecte. */
158        write (connection_fd, bad_request_response,
159               sizeof (bad_request_response));
160     }
161     else if (strcmp (method, "GET")) {
162        /* Ce serveur n'implante que la méthode GET. Le client en a spécifié une
             autre.
163           Indique un échec. */
164        char response[1024];
165
166        snprintf (response, sizeof (response),
167                  bad_method_response_template, method);
168        write (connection_fd, response, strlen (response));
169     }
170     else
171        /* La requête est valide, nous la traitons.  */
172        handle_get (connection_fd, url);
173   }
174   else if (bytes_read == 0)
175     /* Le client a fermé la connexion avant d'envoyer des données.
176        Rien à faire. */
177     ;
```

```
178     else
179       /* L'appel à read a échoué. */
180       system_error ("read");
181   }
182 
183   void server_run (struct in_addr local_address, uint16_t port)
184   {
185     struct sockaddr_in socket_address;
186     int rval;
187     struct sigaction sigchld_action;
188     int server_socket;
189     /* Définit un gestionnaire pour SIGCHLD qui libère les ressources des
190        processus fils terminés. */
191     memset (&sigchld_action, 0, sizeof (sigchld_action));
192     sigchld_action.sa_handler = &clean_up_child_process;
193     sigaction (SIGCHLD, &sigchld_action, NULL);
194 
195     /* Crée un socket TCP. */
196     server_socket = socket (PF_INET, SOCK_STREAM, 0);
197     if (server_socket == -1)
198       system_error ("socket");
199     /* Construit une structure socket address pour recevoir l'adresse
200        locale sur laquelle nous voulons attendre les connexions. */
201     memset (&socket_address, 0, sizeof (socket_address));
202     socket_address.sin_family = AF_INET;
203     socket_address.sin_port = port;
204     socket_address.sin_addr = local_address;
205     /* Lie le socket à cette adresse. */
206     rval = bind (server_socket, &socket_address, sizeof (socket_address));
207     if (rval != 0)
208       system_error ("bind");
209     /* Demande au socket d'accepter les connexions. */
210     rval = listen (server_socket, 10);
211     if (rval != 0)
212       system_error ("listen");
213 
214     if (verbose) {
215       /* En mode verbeux, affiche l'adresse locale et le numéro du port
216          sur lesquels nous écoutons. */
217       socklen_t address_length;
218 
219       /* Détermine l'adresse locale du socket. */
220       address_length = sizeof (socket_address);
221       rval = getsockname (server_socket, &socket_address, &address_length);
222       assert (rval == 0);
223       /* Affiche un message. Le numéro du port doit être converti à partir de
224          l'ordre des octets réseau (big endian) vers l'ordre des octets de l'hôte.
               */
225       printf ("server listening on %s:%d\n",
226               inet_ntoa (socket_address.sin_addr),
227               (int) ntohs (socket_address.sin_port));
228     }
229 
230     /* Boucle indéfiniment, en attente de connexions.  */
231     while (1) {
232       struct sockaddr_in remote_address;
233       socklen_t address_length;
234       int connection;
235       pid_t child_pid;
236 
237       /* Accepte une connexion. Cet appel est bloquant jusqu'à ce qu'une
238          connexion arrive. */
```

```
239       address_length = sizeof (remote_address);
240       connection = accept (server_socket, &remote_address, &address_length);
241       if (connection == -1) {
242         /* L'appel à accept a échoué. */
243         if (errno == EINTR)
244            /* L'appel a été interrompu par un signal. Recommence.  */
245            continue;
246         else
247            /* Quelque chose est arrivé. */
248            system_error ("accept");
249       }
250 
251       /* Nous avons reçu une connexion. Affiche un message si nous sommes
252          en mode verbeux. */
253       if (verbose) {
254         socklen_t address_length;
255 
256         /* Récupère l'addresse distante de la connexion. */
257         address_length = sizeof (socket_address);
258         rval = getpeername (connection, &socket_address, &address_length);
259         assert (rval == 0);
260         /* Affiche un message. */
261         printf ("connection accepted from %s\n",
262                  inet_ntoa (socket_address.sin_addr));
263       }
264 
265       /* Crée un processus fils pour prendre en charge la connexion. */
266       child_pid = fork ();
267       if (child_pid == 0) {
268         /* Nous sommes dans le processus fils. Il n'a pas besoin de stdin ou stdout,
269            nous les fermons donc. */
270         close (STDIN_FILENO);
271         close (STDOUT_FILENO);
272         /* De même, le socket d'écoute est inutile pour le processus fils. */
273         close (server_socket);
274         /* Traite une requête pour la connexion. Nous avons notre propre copie
275            du descripteur de socket. */
276         handle_connection (connection);
277         /* Terminé, nous fermons le socket de connexion et terminons le
278            processus fils. */
279         close (connection);
280         exit (0);
281       }
282       else if (child_pid > 0) {
283         /* Nous sommes dans le processus parent. Le processus fils gère la connexion,
284            nous n'avons donc pas besoin de notre copie du descripteur de socket. Nous la
285            fermons puis nous reprenons la boucle pour accepter une autre connexion. */
286         close (connection);
287       }
288       else
289         /* L'appel à fork a échoué. */
290         system_error ("fork");
291     }
292   }
```

Voici les fonctions définies dans `server.c` :

– `server_run` est le point d'entrée du serveur. Cette fonction démarre le serveur, accepte des connexions et ne se termine pas tant qu'une erreur sérieuse ne survient pas. Le serveur

utilise un socket serveur TCP (consultez la Section 5.5.3, « Serveurs », du Chapitre 5, « Communication interprocessus ») ;
Le premier argument de `server_run` correspond à l'adresse locale sur laquelle les connexions sont acceptées. Une machine sous GNU/Linux peut avoir plusieurs adresses réseau, chacune pouvant être liée à une interface réseau différente[2]. Pour obliger le serveur à n'accepter les connexions qu'à partir d'une certaine interface, spécifiez son adresse réseau. Passez l'adresse locale `INADDR_ANY` pour accepter les connexions depuis n'importe quelle adresse locale.
Le second argument de `server_run` est le numéro de port sur lequel accepter les connexions. Si le port est déjà en cours d'utilisation ou s'il s'agit d'un port privilégié et que le serveur ne s'exécute pas avec les privilèges superutilisateur, le démarrage échoue. La valeur spéciale 0 demande à Linux de sélectionner automatiquement un port inutilisé. Consultez la page de manuel `inet` pour plus d'informations sur les adresses Internet et les numéros de ports.
Le serveur traite chaque connexion dans un processus fils créé par le biais de `fork` (*cf.* Section 3.2.2, « Utiliser *fork* et *exec* » dans le Chapitre 3, « Processus »). Le processus principal (parent) continue à accepter des connexions tandis que celles déjà acceptées sont traitées. Le processus fils appelle `handle_connection` puis ferme le socket et se termine.

- `handle_connection` traite une connexion client en utilisant le descripteur de socket qui lui est passé en argument. Cette fonction lit les données à partir du socket et tente de les interpréter en tant que demande de page HTTP.
  Le serveur ne prend en charge que les versions 1.0 et 1.1 du protocole HTTP. Lorsque le protocole ou la version est incorrect, il répond en envoyant le code HTTP 400 et le message `bad_request_response`. Le serveur ne comprend que la méthode HTTP GET. Si le client utilise une autre méthode, le code réponse HTTP 500 lui est renvoyé ainsi que le message `bad_method_response_template` ;
- Si le client envoie une requête GET valide, `handle_connection` appelle `handle_get` pour la traiter. Cette fonction tente de charger un module serveur dont le nom est dérivé de la page demandée. Par exemple, si le client demande une page information, elle tente de charger le module `information.so`. Si le module ne peut pas être chargé, `handle_get` renvoie au client le code réponse HTTP 404 et le message `not_found_response_template`.
  Si le client demande une page à laquelle correspond un module, `handle_get` renvoie un code réponse 200, signifiant que la requête a été correctement traitée, puis invoque la fonction `module_generate` du module. Celle-ci génère le HTML de la page WEb et l'envoie au client Web.
- `server_run` installe `clean_up_child_process` en tant que gestionnaire de signal pour `SIGCHLD`. Cette fonction se contente de libérer les ressources des processus fils terminés (*cf.* Section 3.4.4, « Libérer les ressources des fils de façon asynchrone » du Chapitre 3).

11.2.4 Le programme principal

`main.c` (*cf.* Listing 11.5) définit la fonction `main` du programme serveur. Il lui incombe d'analyser les options de la ligne de commande, de détecter et signaler les erreurs et de configurer

[2]Votre ordinateur peut êter configuré pour disposer d'interfaces telles que `eth0`, une carte Ethernet, `lo`, le réseau de bouclage ou `ppp0`, une connexion réseau à distance.

et lancer le serveur.

Listing 11.5 – (*main.c*) – **Programme principal du serveur et analyse de la ligne de commande**

```
#include <assert.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#include "server.h"

/* Description des options longues pour getopt_long. */

static const struct option long_options[] = {
   { "address",          1, NULL, 'a' },
   { "help",             0, NULL, 'h' },
   { "module-dir",       1, NULL, 'm' },
   { "port",             1, NULL, 'p' },
   { "verbose",          0, NULL, 'v' },
};

/* Description des options courtes pour getopt_long. */

static const char* const short_options = "a:hm:p:v";

/* Usage summary text. */

static const char* const usage_template =
  "Usage: %s [ options ]\n"
  " -a, --address ADDR          Bind to local address (by default, bind\n"
  "                               to all local addresses).\n"
  " -h, --help            Print this information.\n"
  " -m, --module-dir DIR Load modules from specified directory\n"
  "                         (by default, use executable directory).\n"
  " -p, --port PORT       Bind to specified port.\n"
  " -v, --verbose         Print verbose messages.\n";

/* Affiche des informations sur l'utilisation. Si IS_ERROR est différent de zéro,
   écrit sur stderr et utilise un code de sortie d'erreur, sinon utilise un code
   de sortie normale. Ne se termine jamais. */

static void print_usage (int is_error)
{
  fprintf (is_error ? stderr : stdout, usage_template, program_name);
  exit (is_error ? 1 : 0);
}

int main (int argc, char* const argv[])
{
  struct in_addr local_address;
  uint16_t port;
  int next_option;

  /* Stocke le nom du programme qui sera utilisé dans les messages d'erreur. */
  program_name = argv[0];

  /* Définit les valeurs par défaut des options. Le serveur écoute sur toutes les
```

```
58        adresse et récupère automatiquement un numéro de port inutilisé. */
59      local_address.s_addr = INADDR_ANY;
60      port = 0;
61      /* N'affiche pas les messages verbeux. */
62      verbose = 0;
63      /* Charge les modules depuis le répertoire contenant l'exécutable. */
64      module_dir = get_self_executable_directory ();
65      assert (module_dir != NULL);
66
67      /* Analyse les options. */
68      do {
69        next_option =
70           getopt_long (argc, argv, short_options, long_options, NULL);
71        switch (next_option) {
72        case 'a':
73           /* L'utilisateur a spécifié -a ou --address. */
74           {
75             struct hostent* local_host_name;
76
77             /* Recherche le nom d'hôte demandé par l'utilisateur. */
78             local_host_name = gethostbyname (optarg);
79             if (local_host_name == NULL || local_host_name->h_length == 0)
80               /* Impossible de le résoudre. */
81               error (optarg, "invalid host name");
82             else
83               /* Nom d'hôte OK, nous l'utilisons. */
84               local_address.s_addr =
85                  *((int*) (local_host_name->h_addr_list[0]));
86           }
87           break;
88
89         case 'h':
90           /* L'utilisateur a passé -h ou --help.    */
91           print_usage (0);
92
93         case 'm':
94           /* L'utilisater a passé -m ou --module-dir.    */
95           {
96             struct stat dir_info;
97
98             /* Vérifie que le répertoire existe... */
99             if (access (optarg, F_OK) != 0)
100              error (optarg, "module directory does not exist");
101            /* ... qu'il est accessible... */
102            if (access (optarg, R_OK | X_OK) != 0)
103              error (optarg, "module directory is not accessible");
104            /* ... et qu'il s'agit d'un répertoire. */
105            if (stat (optarg, &dir_info) != 0 || !S_ISDIR (dir_info.st_mode))
106              error (optarg, "not a directory");
107            /* Tout semble correct, nous l'utilisons. */
108            module_dir = strdup (optarg);
109          }
110          break;
111
112        case 'p':
113          /* L'utilisateur a passé -p ou --port.    */
114          {
115            long value;
116            char* end;
117
118            value = strtol (optarg, &end, 10);
119            if (*end != '\0')
```

```
120             /* Présence de caractères non numériques dans le numéro de port. */
121             print_usage (1);
122           /* Le numéro de port doit être converti à l'ordre des octets du réseau
123              (big endian). */
124           port = (uint16_t) htons (value);
125         }
126         break;
127
128       case 'v':
129         /* L'utilisateur a spécifié -v ou --verbose. */
130         verbose = 1;
131         break;
132
133       case '?':
134         /* L'utilisateur a passé une option inconnue.  */
135         print_usage (1);
136
137       case -1:
138         /* Nous en avons terminé avec les options.  */
139         break;
140       default:
141         abort ();
142       }
143     } while (next_option != -1);
144
145     /* Ce programme ne prend aucun argument supplémentaire. Affiche une erreur
146        si l'utilisateur en a passé. */
147     if (optind != argc)
148       print_usage (1);
149
150     /* Affiche le répertoire des modules, si nous sommes en mode verbeux. */
151     if (verbose)
152       printf ("modules will be loaded from %s\n", module_dir);
153     /* Lance le serveur. */
154     server_run (local_address, port);
155     return 0;
156   }
```

`main.c` définit les fonctions suivantes :

– `main` invoque `getopt_long` (*cf.* la Section 2.1.3, « Utiliser *getopt_long* » du Chapitre 2) afin d'analyser les options de ligne de commande. Ces options sont disponibles au format long ou court, comme défini par le tableau `long_options` pour les premières et la chaîne `short_options` pour les secondes.
La valeur par défaut pour le port d'écoute est 0 et l'adresse locale est `INADDR_ANY`. Ces valeurs peuvent être redéfinies au moyen des options `-port` (`-p`) et `-address` (`-a`), respectivement. Si l'utilisateur spécifie une adresse, `main` appelle la fonction `gethostbyname` pour la convertir en adresse Internet numérique[3].
La valeur par défaut pour le répertoire à partir duquel charger les modules serveur est le répertoire qui contient l'exécutable du serveur, déterminé par `get_self_executable_directory` . L'utilisateur peut modifier cette valeur grâce à l'option `-module-dir` (`-m`) ; `main` s'assure que le répertoire indiqué est bien accessible.
Par défaut, les messages verbeux ne sont pas affichés. L'utilisateur peut les activer au moyen de l'option `-verbose` (`-v`) ;

[3] `gethostbyname` effectue une résolution DNS si nécessaire.

– Si l'utilisateur spécifie l'option `-help` (`-h`) ou passe des options invalides, `main` invoque `print_usage` qui affiche des explications sur l'utilisation du programme et le termine.

11.3 Modules

Nous vous proposons quatre modules pour présenter le type de fonctionnalités que vous pourriez implanter en utilisant ce serveur. La création de votre propre module serveur consiste simplement à définir une fonction `module_generate` pour renvoyer le texte adéquat au format HTML.

11.3.1 Afficher l'heure

Le module `time.so` (*cf.* Listing 11.6 génère une page affichant simplement l'heure locale du serveur. La fonction `module_generate` de ce module appelle `gettimeofday` pour obtenir l'heure courante (reportez-vous à la Section 8.7, « *gettimeofday*: heure système » du Chapitre 8, « Appels système Linux ») et utilise `localtime` et `strftime` pour générer une représentation textuelle de celle-ci. Cette représentation est ensuite injectée dans le modèle HTML `page_template`.

Listing 11.6 – (*time.c*) – **Module serveur affichant l'heure courante**

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <sys/time.h>
4  #include <time.h>
5
6  #include "server.h"
7
8  /* Modèle de la page générée par le module. */
9
10 static char* page_template =
11   "<html>\n"
12   " <head>\n"
13   " <meta http-equiv=\"refresh\" content=\"5\">\n"
14   " </head>\n"
15   " <body>\n"
16   " <p>The current time is %s.</p>\n"
17   " </body>\n"
18   "</html>\n";
19
20 void module_generate (int fd)
21 {
22   struct timeval tv;
23   struct tm* ptm;
24   char time_string[40];
25   FILE* fp;
26
27   /* Récupère l'heure courante et la convertit en struct tm. */
28   gettimeofday (&tv, NULL);
29   ptm = localtime (&tv.tv_sec);
30
31   /* Formate l'heure avec une précision à la seconde. */
32   strftime (time_string, sizeof (time_string), "%H:%M:%S", ptm);
33
34   /* Crée un flux correspondant au descripteur de fichier du
35      socket client. */
```

```
36    fp = fdopen (fd, "w");
37    assert (fp != NULL);
38    /* Génère la sortie HTML. */
39    fprintf (fp, page_template, time_string);
40    /* Terminé ; purge le flux. */
41    fflush (fp);
42  }
```

Ce module utilise les fonctions d'E/S de la bibliothèque C standard pour des raisons pratiques. L'appel `fdopen` génère un pointeur sur un flux (`FILE*`) correspondant au socket client (*cf.* Annexe B, « E/S de bas niveau » ; Section B.4, « Lien avec les fonctions d'E/S standards du C »). Le module envoie des données au descripteur en utilisant `fprintf` et le purge au moyen de `fflush` pour éviter une perte des données mises dans le tampon lorsque le socket est germé.

La page HTML renvoyée par le module `time.so` inclue élement `<meta>` qui demande au client de recharger la page toutes les 5 secondes. De cette façon, le client reste à l'heure.

11.3.2 Afficher la distribution utilisée

Le module `issue.so` (Listing 11.7 affiche des informations sur la distribution sur laquelle tourne le serveur. Ces informations sont habituellement stockées dans le fichier `/etc/issue`. Le module envoie le contenu de ce fichier encapsulé dans un élément `<pre>`.

Listing 11.7 – (*issue.c*) – **Module permettant d'afficher des informations sur la distribution GNU/Linux**

```
1  #include <fcntl.h>
2  #include <string.h>
3  #include <sys/sendfile.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <unistd.h>
7
8  #include "server.h"
9
10 /* Source HTML du début de la page que nous générons. */
11
12 static char* page_start =
13   "<html>\n"
14   " <body>\n"
15   "  <pre>\n";
16
17 /* Source HTML de la fin de la page que nous générons. */
18
19 static char* page_end =
20   " </pre>\n"
21   " </body>\n"
22   "</html>\n";
23
24 /* Source HTML de la page indiquant qu'il y a eu un problème à l'ouverture de
25    /etc/issue. */
26
27 static char* error_page =
28   "<html>\n"
29   " <body>\n"
30   " <p>Error: Could not open /etc/issue.</p>\n"
31   " </body>\n"
32   "</html>\n";
```

```
33
34 /* Source HTML indiquant une erreur.  */
35
36 static char* error_message = "Error reading /etc/issue.";
37
38 void module_generate (int fd)
39 {
40   int input_fd;
41   struct stat file_info;
42   int rval;
43
44   /* Ouvre /etc/issue. */
45   input_fd = open ("/etc/issue", O_RDONLY);
46   if (input_fd == -1)
47     system_error ("open");
48   /* Récupère des informations sur le fichier. */
49   rval = fstat (input_fd, &file_info);
50
51   if (rval == -1)
52     /* Soit nous n'avons pas pu ouvrir le fichier, soit ne n'avons pas pu y lire.
          */
53     write (fd, error_page, strlen (error_page));
54   else {
55     int rval;
56     off_t offset = 0;
57
58     /* Écrit le début de la page. */
59     write (fd, page_start, strlen (page_start));
60     /* Copie le contenu de /proc/issue vers le socket client. */
61     rval = sendfile (fd, input_fd, &offset, file_info.st_size);
62     if (rval == -1)
63       /* Quelque chose s'est mal passé lors de l'envoi du contenu de /etc/issue.
64          Envoie un message d'erreur. */
65       write (fd, error_message, strlen (error_message));
66     /* Fin de la page. */
67     write (fd, page_end, strlen (page_end));
68   }
69
70   close (input_fd);
71 }
```

Le module commence par essayer d'ouvrir `/etc/issue`. Si ce fichier ne peut pas être ouvert, le module envoie une page d'erreur au client. Sinon, il envoie le début de la page HTML contenu dans `page_start`. Puis le contenu de `/etc/issue` est transmis au moyen de `sendfile` (*cf.* Chapitre 8, Section 8.12, « *sendfile*: transferts de données rapides »). Enfin, il envoie la fin de la page HTML contenue dans `page_end`.

Vous pouvez facilement adapter ce module pour envoyer le contenu d'un autre fichier. Si celui-ci contient une page HTML complète, supprimez la partie qui envoie `page_start` et `page_end`. Vous pourriez également adapter le serveur principal pour qu'il puisse servir des fichier statiques, comme les serveurs Web traditionnels. L'utilisation de `sendfile` fournit un degré d'efficacité supplémentaire.

11.3.3 Afficher l'espace libre

Le modules `diskfree.so` (Listing 11.8) génère une page affichant des informations sur l'espace disque libre sur les systèmes de fichiers montées sur le serveur. Ces informations sont simplement

le résultat de l'invocaton de la commande `df -h`. Tout comme `issue.so`, le module encapsule les informations dans une balise `<pre>`.

Listing 11.8 – (*diskfree.c*) – **Module affichant des informations sur l'espace disque libre**

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6
7  #include "server.h"
8
9  /* Source HTML du début de la page générée. */
10
11 static char* page_start =
12   "<html>\n"
13   " <body>\n"
14   " <pre>\n";
15
16 /* Source HTML de la fin de la page générée. */
17
18 static char* page_end =
19   " </pre>\n"
20   " </body>\n"
21   "</html>\n";
22
23 void module_generate (int fd)
24 {
25   pid_t child_pid;
26   int rval;
27
28   /* Écrit le début de la page. */
29   write (fd, page_start, strlen (page_start));
30   /* Crée un processus fils. */
31   child_pid = fork ();
32   if (child_pid == 0) {
33     /* Nous sommes le processus fils. */
34     /* Crée la liste d'arguments pour l'invocation de df.  */
35     char* argv[] = { "/bin/df", "-h", NULL };
36
37     /* Duplique stdout et stderr afin d'envoyer les données au socket client. */
38     rval = dup2 (fd, STDOUT_FILENO);
39     if (rval == -1)
40       system_error ("dup2");
41     rval = dup2 (fd, STDERR_FILENO);
42     if (rval == -1)
43       system_error ("dup2");
44     /* Lance df pour afficher l'espace libre sur les systèmes de fichiers montés.
         */
45     execv (argv[0], argv);
46     /* Un appel à execv ne se termine jamais à moins d'une erreur. */
47     system_error ("execv");
48   }
49   else if (child_pid > 0) {
50     /* Nous sommes dans le processus parent, nous
51        attendons la fin du processus fils. */
52     rval = waitpid (child_pid, NULL, 0);
53     if (rval == -1)
54       system_error ("waitpid");
55   }
56   else
```

```
57      /* L'appel à fork a échoué. */
58      system_error ("fork");
59    /* Écrit la fin de la page. */
60    write (fd, page_end, strlen (page_end));
61  }
```

Alors que `issue.so` envoie le contenu d'un fichier en utilisant `sendfile`, ce module doit invoquer une commande et rediriger sa sortie vers le client. Pour cela, il suit les étapes suivantes :

1. Tout d'abord, le module crée un processus fils au moyen de `fork` (*cf.* Chapitre 3, Section 3.2.2, « Utiliser *fork* et *exec* ») ;
2. Le processus fils copie le descripteur de fichier du socket vers les descripteurs `STDOUT_FILENO` et `STDERR_FILENO` qui correspondernt à la sortie standard et à la sortie des erreurs standard (*cf.* Chapitre 2, Section 2.1.4, « E/S standards »). Les descripteurs sont copiés en utilisant l'appel `dup2` (*cf.* Chapitre 5, Section 5.4.3, « Rediriger les flux d'entrée, de sortie et d'erreur standards »). Tout affichage ultérieur depuis le processus vers un de ces flux est envoyé au socket client ;
3. Le processus fils invoque la commande `df` avec l'option `-h` en apelant `execv` (*cf.* Chapitre 3, Section 3.2.2, « Utiliser *fork* et *exec* ») ;
4. Le processus parent attend la fin du processus fils en appelant `waitpid` (*cf.* Chapitre 3, Section 3.4.2, « Les appels système *wait* »).

Vous pouvez facilement adapter ce module pour qu'il invoque une commande différente et redirige sa sortie vers le client.

11.3.4 Afficher la liste des processus en cours d'exécution

Le module `processes.so` (Listing 11.9) est un module plus riche. Il génère une page contenant un tableau présentant les processus en cours d'exécution sur le serveur. Chaque processus apparaît dans une ligne du tableau affichant son PID, le nom de l'exécutable, l'utilisateur et le groupe propriétaires ainsi que la taille de l'empreinte mémoire résidente.

Listing 11.9 – (*processes.c*) – **Module listant les processus**

```
1  #include <assert.h>
2  #include <dirent.h>
3  #include <fcntl.h>
4  #include <grp.h>
5  #include <pwd.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <sys/stat.h>
10 #include <sys/types.h>
11 #include <sys/uio.h>
12 #include <unistd.h>
13
14 #include "server.h"
15
16 /* Place l'identifiant du groupe et de l'utilisateur propriétaire du processus PID
17    dans *UID et *GID, respectivement. Renvoie 0 en cas de succès, une valeur
       différente
18    de 0 sinon. */
```

```
19
20 static int get_uid_gid (pid_t pid, uid_t* uid, gid_t* gid)
21 {
22   char dir_name[64];
23   struct stat dir_info;
24   int rval;
25
26   /* Génère le nom du répertoire du processus dans /proc. */
27   snprintf (dir_name, sizeof (dir_name), "/proc/%d", (int) pid);
28   /* Récupère des informations sur le répertoire. */
29   rval = stat (dir_name, &dir_info);
30   if (rval != 0)
31     /* Répertoire introuvable ; ce processus n'existe peut être plus. */
32     return 1;
33   /* S'assure qu'il s'agit d'un répertoire ; si ce n'est pas le cas
34      il s'agit d'une erreur. */
35   assert (S_ISDIR (dir_info.st_mode));
36
37   /* Récupère les identifiants dont nous avons besoin.  */
38   *uid = dir_info.st_uid;
39   *gid = dir_info.st_gid;
40   return 0;
41 }
42
43 /* Renvoie le nom de l'utilisateur  UID. La valeur de retour est un tampon
44    que l'appelant doit libérer avec free. UID doit être un identifiant utilisateur
45    valide. */
46
47 static char* get_user_name (uid_t uid)
48 {
49   struct passwd* entry;
50
51   entry = getpwuid (uid);
52   if (entry == NULL)
53     system_error ("getpwuid");
54   return xstrdup (entry->pw_name);
55 }
56
57 /* Renvoie le nom du groupe GID. La valeur de retour est un tampon que
58    l'appelant doit libérer avec free. GID doit être un identifiant de groupe
59    valide. */
60
61 static char* get_group_name (gid_t gid)
62 {
63   struct group* entry;
64
65   entry = getgrgid (gid);
66   if (entry == NULL)
67     system_error ("getgrgid");
68   return xstrdup (entry->gr_name);
69 }
70
71 /* Renvoie le nom du programme s'exécutant dans le processus PID ou NULL en cas
72    d'erreur. La valeur de retour est un tampon que l'appelant doit libérer avec
73    free. */
74
75 static char* get_program_name (pid_t pid)
76 {
77   char file_name[64];
78   char status_info[256];
79   int fd;
80   int rval;
```

```
81    char* open_paren;
82    char* close_paren;
83    char* result;
84
85    /* Génère le nom du fichier "stat" dans le répertoire du processus sous /proc
86       et l'ouvre. */
87    snprintf (file_name, sizeof (file_name), "/proc/%d/stat", (int) pid);
88    fd = open (file_name, O_RDONLY);
89    if (fd == -1)
90      /* Impossible d'ouvrir le fichier stat pour ce processus. Peut être qu'il
91         n'existe plus. */
92      return NULL;
93    /* Lit le contenu. */
94    rval = read (fd, status_info, sizeof (status_info) - 1);
95    close (fd);
96    if (rval <= 0)
97      /* Impossible à lire pour une raison quelconque. Échec. */
98      return NULL;
99    /* Ajout un caractère nul à la fin du contenu du fichier. */
100   status_info[rval] = '\0';
101
102   /* Le nom du programme est le second élément dans le fichier et est
103      encadré par des parenthèses. Recherche les positions des parenthèses
104      dans le fichier. */
105   open_paren = strchr (status_info, '(');
106   close_paren = strchr (status_info, ')');
107   if (open_paren == NULL
108       || close_paren == NULL
109       || close_paren < open_paren)
110     /* Impossible de les trouver ; Échec. */
111     return NULL;
112   /* Alloué de la mémoire pour le résultat. */
113   result = (char*) xmalloc (close_paren - open_paren);
114   /* Copie le nom du programme dans le résultat. */
115   strncpy (result, open_paren + 1, close_paren - open_paren - 1);
116   /* strncpy n'ajoute pas d'octet nul à la fin du résultat, nous le faisons. */
117   result[close_paren - open_paren - 1] = '\0';
118   /* Terminé. */
119   return result;
120 }
121
122 /* Renvoie la taille de l'empreinte mémoire résidente (RSS) en kilooctets, du
123    processus PID. Renvoie -1 en cas d'échec. */
124
125 static int get_rss (pid_t pid)
126 {
127   char file_name[64];
128   int fd;
129   char mem_info[128];
130   int rval;
131   int rss;
132
133   /* Génère le nom de l'entrée "statm" du processus dans le répertoire /proc. */
134   snprintf (file_name, sizeof (file_name), "/proc/%d/statm", (int) pid);
135   /* L'ouvre. */
136   fd = open (file_name, O_RDONLY);
137   if (fd == -1)
138     /* Impossible d'ouvrir le fichier "statm" pour ce processus. Peut être qu'il
139        n'existe plus. */
140     return -1;
141   /* Lit le contenu du fichier. */
142   rval = read (fd, mem_info, sizeof (mem_info) - 1);
```

```
143     close (fd);
144     if (rval <= 0)
145       /* Impossible de lire le fichier ; échec. */
146       return -1;
147     /* Ajoute un octet nul. */
148     mem_info[rval] = '\0';
149     /* Extrait la RSS. Il s'agit du second élément. */
150     rval = sscanf (mem_info, "%*d %d", &rss);
151     if (rval != 1)
152       /* Le contenu de statm est formaté d'une façon inconnue. */
153       return -1;
154 
155     /* Les valeurs de statm sont en nombre de pages système.
156        Convertit le RSS en kilooctets. */
157     return rss * getpagesize () / 1024;
158   }
159 
160   /* Génère une ligne de tableau HTML pour le processus PID. Renvoie un
161      pointeur vers un tampon que l'appelant doit libérer avec free ou
162      NULL si une erreur survient. */
163 
164   static char* format_process_info (pid_t pid)
165   {
166     int rval;
167     uid_t uid;
168     gid_t gid;
169     char* user_name;
170     char* group_name;
171     int rss;
172     char* program_name;
173     size_t result_length;
174     char* result;
175 
176     /* Récupère les UID et GID du processus. */
177     rval = get_uid_gid (pid, &uid, &gid);
178     if (rval != 0)
179       return NULL;
180     /* Obtient la RSS du processus. */
181     rss = get_rss (pid);
182     if (rss == -1)
183       return NULL;
184     /* Récupère le nom du programme du processus. */
185     program_name = get_program_name (pid);
186     if (program_name == NULL)
187       return NULL;
188     /* Convertit les UID et GID en noms symboliques.  */
189     user_name = get_user_name (uid);
190     group_name = get_group_name (gid);
191 
192     /* Calcule la taille de la chaîne nécessaire pour stocker le résultat
193        et alloue la mémoire correspondante. */
194     result_length = strlen (program_name)
195       + strlen (user_name) + strlen (group_name) + 128;
196     result = (char*) xmalloc (result_length);
197     /* Formate le résultat. */
198     snprintf (result, result_length,
199               "<tr><td align=\"right\">%d</td><td><tt>%s</tt></td><td>%s</td>"
200               "<td>%s</td><td align=\"right\">%d</td></tr>\n",
201               (int) pid, program_name, user_name, group_name, rss);
202     /* Libération des ressources. */
203     free (program_name);
204     free (user_name);
```

```
205   free (group_name);
206   /* Terminé. */
207   return result;
208 }
209 
210 /* Source HTML du début de la page de listing. */
211 
212 static char* page_start =
213   "<html>\n"
214   " <body>\n"
215   " <table cellpadding=\"4\" cellspacing=\"0\" border=\"1\">\n"
216   "   <thead>\n"
217   "    <tr>\n"
218   "     <th>PID</th>\n"
219   "     <th>Program</th>\n"
220   "     <th>User</th>\n"
221   "     <th>Group</th>\n"
222   "     <th>RSS (KB)</th>\n"
223   "    </tr>\n"
224   "   </thead>\n"
225   "   <tbody>\n";
226 
227 /* Source HTML de la fin de la page. */
228 
229 static char* page_end =
230   "   </tbody>\n"
231   " </table>\n"
232   " </body>\n"
233   "</html>\n";
234 
235 void module_generate (int fd)
236 {
237   size_t i;
238   DIR* proc_listing;
239 
240   /* Crée un tableau d'iovec. Nous le renseignerons avec les
241      tampons qui seront assemblés pour générer la sortie, en adaptant
242      sa taille de façon dynamique. */
243 
244   /* Nombre d'éléments utilisés dans le tableau. */
245   size_t vec_length = 0;
246   /* Taille du tableau. */
247   size_t vec_size = 16;
248   /* Tableau d'iovec. */
249   struct iovec* vec =
250     (struct iovec*) xmalloc (vec_size * sizeof (struct iovec));
251 
252   /* Le premier buffer est la source HTML du début de la page. */
253   vec[vec_length].iov_base = page_start;
254   vec[vec_length].iov_len = strlen (page_start);
255   ++vec_length;
256 
257   /* Commence le listing du répertoire /proc.  */
258   proc_listing = opendir ("/proc");
259   if (proc_listing == NULL)
260     system_error ("opendir");
261 
262   /* Parcourt les entrées de /proc.  */
263   while (1) {
264     struct dirent* proc_entry;
265     const char* name;
266     pid_t pid;
```

```
267       char* process_info;
268
269       /* Récupère l'entrée suivante. */
270       proc_entry = readdir (proc_listing);
271       if (proc_entry == NULL)
272         /* Nous sommes à la fin du répertoire.  */
273         break;
274       /* Si cette entrée n'est pas composée uniquement de chiffre, il ne
275          s'agit pas d'un répertoire de processus. Nous le laissons de côté. */
276       name = proc_entry->d_name;
277       if (strspn (name, "0123456789") != strlen (name))
278         continue;
279       /* Le nom de l'entrée est l'identifiant du processus. */
280       pid = (pid_t) atoi (name);
281       /* Génère une ligne de tableau HTML décrivant ce processus. */
282       process_info = format_process_info (pid);
283       if (process_info == NULL)
284         /* Quelque chose s'est mal passé. Le processus peut s'être terminé
285            pendant que nous l'analysions. Affiche une ligne d'erreur. */
286         process_info = "<tr><td colspan=\"5\">ERROR</td></tr>";
287
288       /* S'assure que le tableau iovec est suffisamment long pour accueillir
289          le tampon (plus un car nous aurons besoin d'un élément supplémentaire
290          à la fin du listing). Si ce n'est pas le cas, double sa taille. */
291       if (vec_length == vec_size - 1) {
292         vec_size *= 2;
293         vec = xrealloc (vec, vec_size * sizeof (struct iovec));
294       }
295       /* Stocke le tampon en tant qu'élément suivant dans le tableau. */
296       vec[vec_length].iov_base = process_info;
297       vec[vec_length].iov_len = strlen (process_info);
298       ++vec_length;
299     }
300
301     /* Ferme le répertoire à la fin du listing.  */
302     closedir (proc_listing);
303
304     /* Ajout un dernier tampon avec le HTML de fin de page.  */
305     vec[vec_length].iov_base = page_end;
306     vec[vec_length].iov_len = strlen (page_end);
307     ++vec_length;
308
309     /* Envoie toute la page au descripteur client en une seule fois.  */
310     writev (fd, vec, vec_length);
311
312     /* Libère les différents tampons. Le premier et le dernier contiennent des
           données
313        statique et ne doivent pas être libérés. */
314     for (i = 1; i < vec_length - 1; ++i)
315       free (vec[i].iov_base);
316     /* Libère la mémoire occupée par le tableau d'iovec. */
317     free (vec);
318   }
```

La récolte d'informations sur un processus et leur formatage en HTML sont divisés en plusieurs opérations plus simples :

– `get_uid_gid` extrait les identifiants de l'utilisateur et du groupe propriétaires d'un processus. Pour cela, la fonction utilise `stat` (*cf.* Annexe B, Section B.2, « *stat* ») sur le répertoire de `/proc` correspondant au processus (*cf.* Chapitre 7, Section 7.2, « Répertoires

de processus »). L'utilisateur et le groupe propriétaires de ce répertoires sont identiques aux propriétaires du processus ;

- `get_user_name` renvoie le nom d'utilisateur correspondant à un UID. Cette fonction se contente d'appeler la fonction de la bibliothèque C `getpwuid`, qui consulte le fichier `/etc/passwd` du système et renvoie une copie du résultat. `get_group_name` renvoie le nom du groupe correspondant à unGID. Il utilise l'appel `getrgid`.
- `get_program_name` renvopie le nom du programme s'exécutant dans un processus donné. Ces informations sont obtenues à partir de l'entrée `stat` dans le répertoire du processus sous `/proc` (*cf.* Chapitre 7, Section 7.2, « Répertoires de processus »). Nous utilisons cette entrée plutôt que le lien symbolique `exe` (*cf.* Chapitre 7, Section 7.2.4, « Exécutable de processus ») ou l'entrée `cmdline` (*cf.* Chapitre 7, Section 7.2.2, « Liste d'arguments d'un processus ») car ils sont inaccessibles si le processus serveur n'appartient pas au même utilisateur que le celui qui est examiné. De plus, la lecture de `stat` ne force pas Linux à remonter le processus examiné en mémoire s'il avait été déchargé de la mémoire ;
- `get_rss` renvoie la taille de l'empreinte mémoire résidente d'un processus. Cette information est donnée par le second élément du contenu de l'entrée `statm` dans le répertoire `/proc` pour le processus (*cf.* Chapitre 7, Section 7.2.6, « Statistiques mémoire de processus ») ;
- `format_process_info` génère une chaîne contenant les éléments HTML à placer dans une ligne du tableau représentant un processus. Après l'appel aux fonctions présentées ci-dessus pour obtenir les informations nécessaires, elle alloue un tampon et génère le HTML en utilisant `snprintf` ;
- `module_generate` génère la page HTML proprement dite, y compris le tableau. La sortie est composée d'une chaîne contenant le début de la page et du tableau (`page_start`), d'une chaîne pour chaque ligne du tableau (générée par `format_process_info`) et une chaîne contenant la fin du tableau et de la page (`page_end`).
  `module_generate` détermine les PID des processus en cours d'exécution sur le système en lisant le contenu de `/proc`. Elle utilise `opendir` et `readdir` (*cf.* Annexe B, Section B.6, « Lire le contenu d'un répertoire ») pour parcourir le répertoire. Elle analyse son contenu à la recherche d'entrées composées uniquement de chiffres ; qui sont supposées être des entrées correspondant à des processus.
  Un nombre potentiellement important de chaînes doivent être envoyées vers le socket client – une pour le début et une pour la fin de la page, ainsi qu'une par processus. Si chaque chaîne devait être écrite avec un appel distinct à `write` cela générerait une surcharge réseau inutile car chaque chaîne devrait être envoyée dans un paquet distinct.
  Pour optimiser l'encapsulation des données dans des paquets, nous utilisons un unique appel à `writev` (*cf.* Annexe B, Section B.3, « Écriture et lecture vectorielles »). Pour cela, nous devons construire un tableau `vec` d'objets `struct iovec`. Cependant, comme nous ne connaissons pas à l'avance le nombre de processus, nous devons commencer avec un tableau très petit que nous étendons au fur et à mesure de l'ajout de processus. La variable `vec_length` contient le nombre d'éléments utilisés dans `vec` tandis que `vec_size` contient la taille réelle de `vec`. Lorsque `vec_length` est en passe de devenir plus grand que `vec_size`, nous doublons la taille de `vec` par le viais de `xrealloc`. Lorsque nous avons terminé l'écriture des données, nous devons libérer toutes les chaînes allouées dynamiquement sur lesquelles

pointent les différentes entrées de `vec` puis libérer `vec` proprement dit.

11.4 Utilisation du serveur

Si nous avions l'intention de distribuer les sources du programme, de les maintenir et de les porter sur d'autres plateformes, nous utiliserions probablement GNU Automake et GNU Autoconf ou un système semblable d'automatisation de la configuration. De tels outils dépassent du cadre de ce livre ; pour plus d'informations, consultez *GNU Autoconf, Automake, and Libtool*[4] (de Vaughan, Elliston, Tromey et Taylor, chez New Riders, 2000).

11.4.1 Le fichier *Makefile*

Au lieu d'utiliser Autoconf ou un outil voisin, nous fournissons un fichier `Makefile` compatible avec GNU Make[5] afin qu'il soit simple de compiler et lier le serveur et ses modules. Le `Makefile` est présenté dans le Listing 11.10. Consultez la page info de GNU Make pour plus de détails sur la syntaxe de ce type de fichiers.

Listing 11.10 – (*Makefile*) – **Fichier de confiuguration GNU Make pour l'exemple de serveur**

```
1  ### Configuration.  ######################################################
2
3  # Options par défaut du compilateur C.
4  CFLAGS                  = -Wall -g
5  # Fichiers sources C pour le serveur.
6  SOURCES                 = server.c module.c common.c main.c
7  # Fichiers objets correspondants.
8  OBJECTS                 = $(SOURCES:.c=.o)
9  # Fichiers de bibliothèques partagées des modules.
10 MODULES                 = diskfree.so issue.so processes.so time.so
11
12 ### Règles.  ############################################################
13
14 # Les cibles phony ne correspondent pas à des fichiers à construire; il
15 # s'agit de noms pour des cibles conceptuelles.
16 .PHONY:          all clean
17
18 # Cible par défaut: construit tout.
19 all:             server $(MODULES)
20
21 # Nettoie les objets construits.
22 clean:
23         rm -f $(OBJECTS) $(MODULES) server
24
25 # Programme serveur principal. Fait l'édition de liens avec les options
26 # -Wl,-export-dynamic afin que les modules puissent utiliser des symboles
27 # définis par le programme. Utilise libdl qui contient les appels nécessaires
28 # au chargement dynamique.
29 server:          $(OBJECTS)
30         $(CC) $(CFLAGS) -Wl,-export-dynamic -o $@ $^ -ldl
31
32 # Tous les fichiers objets du serveur dépendent de server.h. Les fichiers
```

[4] NdT. en anglais

[5] GNU Make est généralement installé sur les systèmes GNU/Linux.

```
33  # objets sont construits à partir des fichiers sources grâce à une règle par
34  # défaut.
35  $(OBJECTS):     server.h
36
37  # Règle pour construire les bibliothèques partagées des modules à partir des
38  # fichiers sources correspondants. Compile avec -fPIC et génère un fichier
39  # objet partagé.
40  $(MODULES): \
41  %.so:           %.c server.h
42          $(CC) $(CFLAGS) -fPIC -shared -o $@ $<
```

Le `Makefile` définit les cibles suivantes :

- `all` (la cible par défaut si vous invoquez `make` sans arguments car il s'agit de la première cible dans le `Makefile`) comprend l'exécutable du serveur et tous les modules. Les modules sont définis par la variable `MODULES` ;
- `clean` supprime tout élément construit par le `Makefile` ;
- `server` crée l'exécutable du serveur. Les fichiers sources listés dans la variables `SOURCES` sont compilés et liés ;
- la dernière ligne est un modèle générique afin de compiler les fichiers objets partagés des modules du serveur à partir des fichiers sources correspondants.

Notez que les fichiers sources des modules serveur sont compilés en utilisant l'option `-fPIC` car il s'agit de bibliothèques partagées (*cf.* Chapitre 2, Section 2.3.2, « Bibliothèques partagées »).

Remarquez également que l'exécutable du serveur est lié avec les options de compilation `-Wl` et `-export-dynamic`. Grâce à cela, GCC passe l'option `-export-dynamic` à l'éditeur de liens qui crée un fichier exécutable qui exporte ses symboles comme le fait une bibliothèque partagée. Cela permet aux modules, qui sont chargés dynamiquement sous forme de bibliothèques partagées, de faire référence à des fonctions de `common.c` qui sont liées statiquement à l'exécutable du serveur.

11.4.2 Construire le serveur

La construction du programme est relativement simple. Depuis le répertoire où se situent les sources, invoquez simplement `make` :

```
% make
cc -Wall -g   -c -o server.o server.c
cc -Wall -g   -c -o module.o module.c
cc -Wall -g   -c -o common.o common.c
cc -Wall -g   -c -o main.o main.c
cc -Wall -g -Wl,-export-dynamic -o server server.o module.o common.o main.o -ldl
cc -Wall -g -fPIC -shared -o diskfree.so diskfree.c
cc -Wall -g -fPIC -shared -o issue.so issue.c
cc -Wall -g -fPIC -shared -o processes.so processes.c
cc -Wall -g -fPIC -shared -o time.so time.c
```

Cela lance la construction du serveur et des bibliothèques partagées des modules.

```
% ls -l server *.so
-rwxr-xr-x    1 samuel samuel 25769 Mar 11 01:15 diskfree.so
-rwxr-xr-x    1 samuel samuel 31184 Mar 11 01:15 issue.so
-rwxr-xr-x    1 samuel samuel 41579 Mar 11 01:15 processes.so
-rwxr-xr-x    1 samuel samuel 71758 Mar 11 01:15 server
-rwxr-xr-x    1 samuel samuel 13980 Mar 11 01:15 time.so
```

11.4.3 Lancer le serveur

Pour lancer le serveur, invoquez simplement l'exécutable `server`.

Si vous ne spécifiez pas de port au moyen de l'option `-port` (`-p`) ; Linux en choisira un pour vous ; dans ce cas, passez l'option `-verbose` (`-v`) afin que le serveur l'affiche.

Si vous ne demandez pas d'adresse particulière au moyen de `-address` (`-a`), le serveur s'exécute sur toutes vos adresses réseau. Si votre ordinateur est relié à un réseau, cela signifie que d'autres personnes pourront y accéder, si tant est qu'ils connaissent le port à utiliser et une page à demander. Pour des raisons de sécurité, il est conseillé d'utiliser l'adresse `localhost` jusqu'à ce que vous soyez sûr que le serveur fonctionne correctement et ne révèle pas d'informations confidentielles. L'utilisation de cette adresse oblige le serveur à utiliser le périphérique réseau de bouclage (appelé `lo`) – seuls les programmes s'exécutant sur le même ordinateur peuvent s'y connecter. Si vous spécifiez une adresse différente, elle doit correspondre à votre ordinateur :

```
% ./server --address localhost --port 4000
```

Le serveur est désormais opérationnel. Ouvrez un navigateur et essayer de vous y connecter en utilisant le bon numéro de port. Demandez une page correspondant à l'un des modules. Par exemple, pour invoquer le module `diskfree.so`, utilisez cette URL :

```
http://localhost:4000/diskfree
```

Au lieu de `4000`, utilisez le numéro de port que vous avez spécifiez (ou le port choisi pour vous par Linux). Appuyez sur `Ctrl+C` pour tuer le serveur lorsque vous en avez fini.

Si vous ne spécifiez pas `localhost` comme adresse pour le serveur, vous pouvez également vous y connecter à partir d'un autre ordinateur en utilisant le nom du vôtre dans l'URL – par exemple :

```
http://host.domain.com:4000/diskfree
```

Si vous passez l'option `-verbose` (`-v`), le serveur affiche des informations au démarrage et affiche l'adresse IP de chaque client se connectant. Si vous vous connecter *via* `localhost`, l'adresse sera toujours `127.0.0.1`.

Si vous tentez d'écrire vos propres modules, vous pourriez les placer dans un répertoire différent de celui contenant le serveur. Dans ce cas, indiquez ce répertoire au moyen de l'option `-module-dir` (`-m`). Le serveur recherchera les modules dans ce répertoire.

Si vous oubliez la syntaxe des options en ligne de commande, invoquez `server` avec l'option `-help` (`-h`) :

```
% ./server --help
Usage: ./server [ options ]
  -a, --address ADDR        Bind to local address (by default, bind
                              to all local addresses).
  -h, --help                Print this information.
  -m, --module-dir DIR      Load modules from specified directory
                              (by default, use executable directory).
  -p, --port PORT           Bind to specified port.
  -v, --verbose             Print verbose messages.
```

11.5 Pour finir

Si vous aviez réellement l'intention de publier ce programme, vous devriez écrire de la documentation. Beaucoup de gens ne se rendent pas compte que l'écriture d'une bonne documentation est aussi difficile et prend autant de temps – et est aussi important – que l'écriture de bons logiciels. Cependant, la documentation des logiciels pourrait faire l'objet d'un livre entier, nous vous laisserons donc avec quelques références vous permettant d'en apprendre plus sur la façon de documenter un logiciel GNU/Linux.

Vous aurez probablement besoin d'écrire une page de manuel pour le programme `server`, par exemple. C'est le premier endroit où la plupart des utilisateurs vont rechercher des informations sur un programme. Les pages de manuel sont formatées en utilisant un système classique sous UNIX, `troff`. Pour consulter la page de manuel de `troff`, qui décrit le format des fichiers troff, utilisez la commande suivante :

```
% man troff
```

Pour en savoir plus sur l'organisation des pages de manuel sous GNU/Linux, consultez la page de manuel de la commande `man` *via* :

```
% man man
```

Vous pouvez également écrire des pages info pour le serveur et les modules, en utilisant le système Info GNU. Naturellement, le système Info est documenté au format Info ; pour consulter la documentation, invoquez :

```
% info info
```

Beaucoup de programmes GNU/Linux proposent également une documentation texte ou HTML.

Bonne programmation sous GNU/Linux !

Troisième partie

Annexes

Table des Matières

Annexe A

Autres outils de développement

Le développement de programmes GNU/Linux rapides et fiables en C ou en C++ nécessite plus que la compréhension en surface du fonctionnement du système d'exploitation et des appels système. Dans cette annexe, nous présenterons des outils de développement permettant de détecter les erreurs à l'exécution comme l'utilisation incorrecte d'une zone mémoire allouée dynamiquement et de déterminer quelles parties du programme monopolisent le plus de temps d'exécution. L'analyse du code source d'un programme peut révéler certaines de ces informations ; en utilisant ces outils d'analyse dynamique et en exécutant effectivement le programme, vous pouvez en obtenir beaucoup plus.

A.1 Analyse statique de programmes

Certaines erreurs de programmation peuvent être détectées en utilisant des outils d'analyse statique qui étudient le code source du programme. Si vous invoquez GCC avec l'option `-Wall` ou `-pedantic`, le compilateur affiche des avertissement sur les structures de programmation risquées ou potentiellement fausses. En éliminant de telles constructions, vous réduirez les risques de bugs et faciliterez la compilation de vos programmes sur d'autres variantes de GNU/Linux et même sur d'autres systèmes d'exploitation.

En utilisant diverses options, vous pouvez faire en sorte que GCC émette des avertissements sur un nombre impressionnant de structures de programmation douteuses. L'option `-Wall` active la plupart des vérifications. Par exemple, le compilateur affichera un avertissement sur un commentaire commençant au sein d'un autre commentaire, sur un type de retour incorrect pour la fonction `main` ou pour une fonction non `void` qui ne dispose pas d'instruction `return`. Si vous utilisez l'option `-pedantic`, GCC émet des avertissements pour tout ce qui ne respecte pas les normes ANSI C et ISO C++. Par exemple, l'utilisation de l'extension GNU `asm` provoque l'émission d'un avertissement avec cette option. Un petit nombre d'extensions GNU, comme l'utilisation des mots clés alternatifs commençant par `__` (deux tirets bas), ne déclencheront aucun message d'avertissement. Bien que la page `info` de GCC marque l'utilisation de ces options comme dépréciée, nous vous recommandons de les utiliser pour éviter la plupart des extensions

GNU du langage car ces extensions ont tendance à changer au cours du temps et à interagir de façon néfaste sur l'optimisation du code.

Listing A.1 – (*hello.c*) – **Programme Coucou**

```
1  main()
2  {
3    printf("Coucou.\n");
4  }
```

Compilez le programme "Coucou" du Listing A.1. Bien que GCC le compile sans rien dire, le code source n'obéit pas aux règles ANSI C. Si vous activez les avertissement en compilant avec les options `-Wall -pedantic`, GCC indique trois constructions douteuses.

```
% gcc -Wall -pedantic hello.c
hello.c:2: warning: return type defaults to "int"
hello.c: In function "main":
hello.c:3: warning: implicit declaration of function "printf"
hello.c:4: warning: control reaches end of non-void function
```

Ces avertissements indiquent les problèmes suivants :

- Le type de retour de main n'est pas précisé ;
- La fonction `printf` est déclarée de façon implicite car `<stdio.h>` n'est pas inclus ;
- La fonction, implicitement déclarée comme renvoyant un `int`, ne renvoie en fait aucune valeur.

L'analyse du code source d'un programme ne permet pas de détecter toutes les erreurs de programmation et les lacunes. Dans la section suivante, nous présentons quatre outils aidant à à détecter les erreurs commises dans l'utilisation de mémoire allouée dynamiquement. Dans les sections suivantes, nous montrerons comment analyser le temps d'exécution d'un programme à l'aide du profiler `gprof`.

A.2 Détection des erreurs d'allocation dynamique

Lorsque vous écrivez un programme, vous ne savez généralement pas quelle quantité de mémoire il nécessitera pour s'exécuter. Par exemple, une ligne lue à partir d'un fichier au moment de l'exécution peut avoir n'importe quelle taille. Les programmes C et C++ utilisent `malloc`, `free` et leurs variantes pour allouer dynamiquement de la mémoire pendant l'exécution du programme. Voici quelques règles sur l'utilisation de mémoire allouée dynamiquement :

- Le nombre d'allocations (appels à `malloc`) doit correspondre exactement au nombre de libérations (appels à `free`) ;
- Les lectures et écritures doivent se faire dans l'espace alloué, pas au delà ;
- La mémoire allouée ne doit pas être utilisée avant son allocation ou après sa libération.

Comme l'allocation et la libération dynamiques ont lieu durant l'exécution, les analyses statiques ne peuvent que rarement détecter les violations de ces règles. C'est pourquoi il existe des programmes de vérification mémoire qui exécutent le programme et collectent des informations pour déterminer si une telle violation a lieu. Voici un exemple de types de violations pouvant être détectés :

- Lecture d'un emplacement mémoire avant allocation ;

- Écriture dans un emplacement mémoire avant son allocation ;
- Lecture d'une position se trouvant avant la zone allouée ;
- Écriture d'une position se trouvant avant la zone allouée ;
- Lecture d'une position se trouvant après la fin de la zone allouée ;
- Écriture d'une position se trouvant après la fin de cette zone ;
- Lecture d'un emplacement mémoire après sa libération ;
- Écriture dans un emplacement après sa libération ;
- Échec de la libération de la mémoire ;
- Double libération de mémoire ;
- Libération de mémoire non allouée.

Il est également utile d'être averti lors d'une demande d'allocation de zéro octet car il s'agit certainement d'une erreur de la part du programmeur.

Le Tableau A.1 indique les fonctionnalités des quatre outils de diagnostic. Malheureusement, il n'existe pas d'outil diagnostiquant toutes les erreurs d'utilisation de la mémoire. De plus, aucun outil ne détecte l'écriture ou la lecture d'une zone non allouée, toutefois, une telle opération déclencherait probablement une erreur de segmentation. Ces outils ne détectent que les erreurs survenant durant l'exécution du programme. Si vous exécutez le programme avec des données qui ne nécessitent pas d'allocation mémoire, les outils ne détecteront aucun problème. Pour tester un programme de façon exhaustive, vous devez l'exécuter en utilisant différentes données d'entrée pour vous assurer que tous les chemins possible au sein de votre programme sont parcourus. De plus, vous ne pouvez utiliser qu'un seul outil à la fois, vous devez donc recommencer les mêmes tests avec plusieurs outils pour vous assurer d'avoir le plus de vérifications possibles.

TAB. A.1 – Fonctionnalités de Différents Outils de Vérification Mémoire (X : prise en charge totale, O : détection occasionnelle)

Comportement erroné	**malloc**	**mtrace**	**ccmalloc**	**Electric Fence**
Lecture avant allocation				
Écriture avant allocation				
Lecture avant la zone allouée				X
Écriture avant la zone alloué	O		O	X
Lecture après la zone allouée				X
Écriture après la zone allouée			X	X
Lecture après libération				X
Écriture après libération				X
Échec de libération		X	X	
Double libération	X		X	
Libération de mémoire non allouée		X	X	
Allocation de taille nulle			X	X

Dans les sections qui suivent, nous décrirons comment utiliser la vérification fournie par `malloc` et `mtrace`, qui sont les deux plus simples, puis nous nous intéresserons à `ccmalloc` et Electric Fence.

A.2.1 Programme de test d'allocation et de libération mémoire

Nous utiliserons les programme `malloc-use` du Listing A.2 pour illustrer l'allocation, la libération et l'utilisation de la mémoire. Pour le lancer, passez-lui le nombre maximum de régions mémoire à allouer en premier argument. Par exemple, `malloc-use 12` crée un tableau `A` avec 12 pointeurs de caractères qui ne pointent sur rien. Le programme accepte cinq commandes différentes :

- Pour allouer b octets sur lesquels pointe l'entrée `A[i]`, saisissez `a i b`. L'indice i peut être n'importe quel nombre non négatif inférieur à l'argument de ligne de commande ;
- Pour libérer la mémoire se situant à l'indice i, entrez `d i` ;
- Pour lire le $p^{\text{ème}}$ caractère de la mémoire allouée à l'indice i (comme avec `A[i][p]`), saisissez `r i p`. Ici, p peut avoir n'importe quelle valeur entière ;
- Pour écrire un caractère à la $p^{\text{ème}}$ position de la mémoire allouée à l'indice i, entrez `w i p` ;
- Lorsque vous avez terminé, saisissez `q`.

Nous présenterons le code du programme plus loin dans la Section A.2.7 et illustrerons comment l'utiliser.

A.2.2 Vérification par *malloc*

Les fonctions d'allocation mémoire fournies avec la bibliothèque C GNU peut détecter l'écriture en mémoire avant une zone allouée et la double libération.

La définition de la variable d'environnement `MALLOC_CHECK_` à la valeur 2 provoque l'arrêt d'un programme lorsqu'une telle erreur est détectée (attention, la variable d'environnement se termine par un tiret bas). Il n'est pas nécessaire de recompiler le programme.

Voici un exemple de détection d'écriture à un emplacement mémoire juste avant le début d'une zone allouée :

```
% export MALLOC_CHECK_=2
% ./malloc-use 12
Please enter a command: a 0 10
Please enter a command: w 0 -1
Please enter a command: d 0
Aborted (core dumped)
```

la commande `export` active les vérifications de `malloc`. La valeur 2 provoque l'arrêt immédiat du programme lors de la détection d'une erreur.

L'utilisation de la vérification par `malloc` est avantageuse car elle ne nécessite aucune recompilation, mais l'étendue des vérifications est assez limitée. Fondamentalement, il s'agit de s'assurer que les structures de données utilisées pour l'allocation mémoire ne sont pas corrompues. C'est pourquoi il est possible de détecter une double libération du même emplacement mémoire. De plus, l'écriture juste avant le début d'une zone allouée peut être détectée facilement car le dispositif d'allocation mémoire stocke la taille de chaque zone allouée juste avant celle-ci. Aussi, une écriture juste avant la zone corrompt cette taille. Malheureusement, la vérification de cohérence ne peut avoir lieu que lorsque le programme appelle des routines d'allocation, pas lorsqu'il accède à la mémoire, aussi, un nombre important de lectures et d'écritures peuvent

avoir lieu avant qu'une erreur ne soit détectée. Dans l'exemple précédent, l'écriture illégale n'a été détectée que lorsque la mémoire a été libérée.

A.2.3 Recherche de fuites mémoire avec *mtrace*

L'outil `mtrace` aide à diagnostiquer les erreurs les plus courantes lors de l'allocation dynamique de mémoire : la non concordance entre le nombre d'allocations et de libérations. L'utilisation de `mtrace` se fait en quatre étapes, `mtrace` est fourni avec la bibliothèque C GNU :

1. Modifier le code source pour inclure `<mcheck.h>` et invoquer `mtrace()` dès le début du programme, au début de `main`. L'appel à `mtrace` active la surveillance des allocations et libérations mémoire ;
2. Indiquer le nom d'un fichier pour stocker les informations sur les allocations et libérations mémoire :

```
% export MALLOC_TRACE=memory.log
```

3. Exécuter le programme. Toutes les allocations et libérations sont stockées dans le fichier journal.
4. Utiliser la commande `mtrace` pour analyser les allocations et libérations mémoire pour s'assurer que leur nombre concorde.

```
% mtrace mon_programme $MALLOC_TRACE
```

Les messages émis par `mtrace` sont relativement simples à comprendre. Par exemple, avec notre exécution de `malloc-use`, ils seraient de ce type :

```
- 0000000000 Free 3 was never alloc'd malloc-use.c:39
Memory not freed:
-----------------
   Address     Size     Caller
0x08049d48      0xc  at malloc-use.c:30
```

Ces messages indiquent une libération de mémoire qui n'a jamais été allouée à la ligne 39 de `malloc-use.c` et une zone de mémoire allouée à la ligne 30 non libérée.

`mtrace` détecte les erreurs grace à l'analyse du fichier spécifié par la variable d'environnement `MALLOC_TRACE`, celui-ci contient la liste de toutes les allocations et libérations mémoire du programme. L'exécutable doit se terminer normalement pour que les données soient écrites. La commande `mtrace` analyse le fichier et dresse la liste des allocations et libérations qui n'ont pas de réciproque.

A.2.4 Utiliser *ccmalloc*

La bibliothèque `ccmalloc` détecte les erreurs mémoire en remplaçant les appels à `malloc` et `free` avec des instructions traçant leur utilisation. Si le programme se termine correctement, elle crée un rapport concernant les fuites mémoire et d'autres erreurs. La bibliothèque `ccmalloc` a été écrite par Armin Bierce.

Vous devrez probablement télécharger et installer la bibliothèque `ccmalloc` vous-même. Les sources sont disponibles sur `http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/`.

Une fois celles-ci décompressées, lancez `configure`. Exécutez `make` et `make install`, copiez le fichier `ccmalloc.cfg` dans le répertoire d'où vous lancerez le programme que vous voulez contrôler et renommez la copie en `.ccmalloc`. Vous êtes maintenant prêt à utiliser `ccmalloc`.

Les fichiers objets du programme doivent être liés avec la bibliothèque `ccmalloc` et la bibliothèque de liaison dynamique. Ajoutez `-lccmalloc` et `-ldl` à votre commande d'édition de liens, par exemple :

```
% gcc -g -Wall -pedantic malloc-use.o -o ccmalloc-use -lccmalloc -ldl
```

Lancez votre programme pour créer un rapport. Par exemple, l'exécution de notre programme `malloc-use` de façon à ce qu'il ne libère pas une zone mémoire produit le rapport suivant :

```
% ./ccmalloc-use 12
file-name=a.out does not contain valid symbols
trying to find executable in current directory ...
using symbols from "ccmalloc-use"
(to speed up this search specify "file ccmalloc-use"
 in the startup file ".ccmalloc")
Please enter a command: a 0 12
Please enter a command: q.
---------------.
|ccmalloc report|
=========================================================
| total # of|   allocated | deallocated |      garbage |
+-----------+-------------+-------------+--------------+
|     bytes |          60 |          48 |           12 |
+-----------+-------------+-------------+--------------+
|allocations|            2|            1|             1|
+-------------------------------------------------------+
| number of checks: 1                                   |
| number of counts: 3                                   |
| retrieving function names for addresses ... done.     |
| reading file info from gdb ... done.                  |
| sorting by number of not reclaimed bytes ... done.    |
| number of call chains: 1                              |
| number of ignored call chains: 0                      |
| number of reported call chains: 1                     |
| number of internal call chains: 1                     |
| number of library call chains: 0                      |
=========================================================
|
*100.0% = 12 Bytes of garbage allocated in 1 allocation
|       |
|       |        0x400389cb in <???>
|       |
|       |        0x08049198 in <main>
|       |                   at malloc-use.c:89
|       |
|       |        0x08048fdc in <allocate>
|       |                   at malloc-use.c:30
|       |
|       +-----> 0x08049647 in <malloc>
|                           at src/wrapper.c:284
|
+-------------------------------------------------------
```

Les dernières lignes donnent la liste des appels de fonctions qui ont alloué de la mémoire sans la libérer.

Pour utiliser `ccmalloc` afin de diagnostiquer des écriture avant le début ou après la fin d'une zone mémoire allouée, vous devrez modifier le fichier `.ccmalloc` dans le répertoire du programme. Ce fichier est lu au démarrage du programme.

A.2.5 Electric Fence

Écrit par Bruce Perens, Electric Fence stoppe l'exécution du programme à la ligne exacte de la lecture ou de l'écriture en dehors d'une zone allouée. Il s'agit du seul outil qui détecte les lectures illégales. Il est inclus dans la plupart des distributions GNU/Linux, le code source est tout de même disponible sur `http://www.perens.com/FreeSoftware/`.

Comme pour `ccmalloc`, les fichiers objets de votre programme doivent être liés à la bibliothèque Electric Fence en ajoutant `-lefence` à la commande d'édition de liens, par exemple :

```
% gcc -g -Wall -pedantic malloc-use.o -o emalloc-use -lefence
```

Electric Fence vérifie la validité des utilisations de la mémoire au cours de l'exécution du programme. Une mauvaise utilisation provoque une erreur de segmentation :

```
% ./emalloc-use 12
  Electric Fence 2.0.5 Copyright (C) 1987-1998 Bruce Perens.
Please enter a command: a 0 12
Please enter a command: r 0 12
Segmentation fault
```

En utilisant un débogueur, vous pouvez déterminer l'emplacement de l'action illégale.

Par défaut, Electric Fence ne diagnostique que les accès à des emplacements situés après la zone allouée. Pour activer la détection des accès à des emplacements situés avant la zone allouée à la place des accès à des zones situées après, utilisez cette commande :

```
% export EF_PROTECT_BELOW=1
```

Pour détecter les accès à des emplacements mémoire libérés, positionnez `EF_PROTECT_FREE` à 1. Des fonctionnalités supplémentaires sont décrites dans la page de manuel de `libefence`.

Electric Fence peut diagnostiquer des accès illégaux en mobilisant au moins deux pages mémoire pour toute allocation. Il place la zone allouée à la fin de la première page, ainsi, tout accès après la fin de la zone allouée provoque une erreur de segmentation. Si vous positionnez `EF_PROTECT_BELOW` à 1, il place la zone allouée au début de la seconde page. Comme chaque appel à `malloc` utilise deux pages mémoire, Electric Fence peut consommer une quantité importante de mémoire. N'utilisez cette bibliothèque qu'à des fins de débogage.

A.2.6 Choisir parmi les différents outils de débogage mémoire

Nous avons présenté quatre outils distincts, incompatibles destinés à diagnostiquer de mauvaises utilisations de mémoire allouée dynamiquement. Alors comment un programmeur GNU/-Linux peut-il s'assurer que la mémoire est utilisée correctement ? Aucun outil ne garantit la détection de toutes les erreurs, mais l'utilisation de n'importe lequel d'entre eux augmente la probabilité d'en découvrir. Pour faciliter la détection d'erreurs concernant la mémoire allouée dynamiquement, développez et testez le code l'utilisant séparément du reste. Cela réduit le

volume de code à analyser lors de la recherche de telles erreurs. Si vous écrivez vos programmes en C++, dédiez une classe à la gestion de la mémoire dynamique. Si vous programmez en C, minimisez le nombre de fonctions utilisant l'allocation et la libération. Lors des tests de ce code, assurez-vous qu'un seul outil s'exécute à la fois car ils sont incompatibles. Lorsque vous testez le programme, assurez-vous de varier la façon dont vous l'utilisez afin de tester toutes les portions de code.

Lequel de ces outils devriez-vous utiliser ? Comme l'absence d'équilibre entre les allocations et les libérations est l'erreur la plus courante en matière de gestion dynamique de la mémoire, utilisez `mtrace` au début du processus de développement. Ce programme est disponible pour tous les système GNU/Linux et a été éprouvé. Après vous être assuré que les nombres d'allocations et de libérations sont identiques, utilisez Electric Fence pour détecter les accès mémoire invalides. Vous éliminerez ainsi presque toutes les erreurs mémoire. Lorsque vous utilisez Electric Fence, vous devez être attentif à ne pas effectuer trop d'allocations et de libérations car chaque allocation nécessite au moins deux pages mémoire. L'utilisation de ces deux outils vous permettra de détecter la plupart des erreurs.

A.2.7 Code source du programme d'allocation dynamique de mémoire

Le Listing A.2 présente le code source d'un programme illustrant l'allocation dynamique, la libération et l'utilisation de mémoire. Consultez la Section A.2.1, « Programme de test d'allocation et de libération mémoire », pour une description de son utilisation.

Listing A.2 – (*malloc-use.c*) – **Dynamic Memory Allocation Checking Example**

```
1  /* Utilisation des fonctions C d'allocation mémoire.  */
2  /* Invoquez le programme en utilisant un argument précisant la taille du tableau.
3     Ce tableau est composé de pointeurs sur des tableaux pouvant
4     être alloués par la suite.
5     Le programme accepte les commandes suivantes :
6     o allouer de la mémoire :   a <indice> <taille>
7     o libérer de la mémoire : d <indice>
8     o lire un emplacement mémoire :  r <indice> <position-dans-la-mémoire-allouée>
9     o écrire à un emplacement :   w <indice> <position-dans-la-mémoire-allouée>
10    o quitter :               q
11    L'utilisateur a la responsabilité de respecter les règles de l'allocation
12    dynamique de la mémoire (ou non). */
13 #ifdef MTRACE
14 #include <mcheck.h>
15 #endif /* MTRACE */
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <assert.h>
19 /* Alloue de la mémoire pour la taille spécifiée, renvoie une
20    valeur différentes de zéro en cas de succès. */
21 void allocate (char** array, size_t size)
22 {
23   *array = malloc (size);
24 }
25 /* Libère la mémoire.  */
26 void deallocate (char** array)
27 {
28   free ((void*) *array);
29 }
30 /* Lit un emplacement mémoire.  */
```

```
31  void read_from_memory (char* array, int position)
32  {
33    char character = array[position];
34  }
35  /* Écrit à un emplacement mémoire.  */
36  void write_to_memory (char* array, int position)
37  {
38    array[position] = 'a';
39  }
40  int main (int argc, char* argv[])
41  {
42    char** array;
43    unsigned array_size;
44    char command[32];
45    unsigned array_index;
46    char command_letter;
47    int size_or_position;
48    int error = 0;
49  #ifdef MTRACE
50    mtrace ();
51  #endif /* MTRACE */
52    if (argc != 2) {
53      fprintf (stderr, "%s: taille-du-tableau\n", argv[0]);
54      return 1;
55    }
56    array_size = strtoul (argv[1], 0, 0);
57    array = (char **) calloc (array_size, sizeof (char *));
58    assert (array != 0);
59    /* Effectue ce que l'utilisateur demande. */
60    while (!error) {
61      printf ("Entrez une commande : ");
62      command_letter = getchar ();
63      assert (command_letter != EOF);
64      switch (command_letter) {
65      case 'a':
66        fgets (command, sizeof (command), stdin);
67        if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
68            && array_index < array_size)
69          allocate (&(array[array_index]), size_or_position);
70        else
71          error = 1;
72        break;
73      case 'd':
74        fgets (command, sizeof (command), stdin);
75        if (sscanf (command, "%u", &array_index) == 1
76            && array_index < array_size)
77          deallocate (&(array[array_index]));
78        else
79          error = 1;
80        break;
81      case 'r':
82        fgets (command, sizeof (command), stdin);
83        if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
84            && array_index < array_size)
85          read_from_memory (array[array_index], size_or_position);
86        else
87          error = 1;
88        break;
89      case 'w':
90        fgets (command, sizeof (command), stdin);
91        if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
92            && array_index < array_size)
```

```
93            write_to_memory (array[array_index], size_or_position);
94          else
95            error = 1;
96          break;
97        case 'q':
98          free ((void *) array);
99          return 0;
100       default:
101         error = 1;
102       }
103   }
104   free ((void *) array);
105   return 1;
106 }
```

A.3 Profilage

Maintenant que votre programme est correct (espérons-le), nous allons voir comment améliorer ses performances. Avec l'aide du profileur `gprof`, vous pouvez déterminer les fonctions qui monopolisent le plus de temps d'exécution. Cela peut vous aider à déterminer les portions du programme à optimiser ou à réécrire pour qu'elles s'exécutent plus rapidement. Cela peut également vous aider à trouver des erreurs. Par exemple, vous pourriez détecter qu'une fonction est appelée beaucoup plus souvent que vous ne le supposiez.

Dans cette sections, nous décrirons comment utiliser `gprof`. La réécriture de code pour accélerer son exécution nécessite de la créativité et un certain soin dans le choix des algorithmes.

L'obtention d'informations de profilage demande de suivre trois étapes :

1. Compiler et lier votre programme de façon à activer le profilage ;
2. Exécuter votre programme de façon à générer les données de profilage ;
3. Utiliser `gprof` pour analyser et afficher les données de profilage.

Avant d'illustrer ces différentes étapes, nous allons présenter une programme suffisamment important pour qu'il soit possible de le profiler.

A.3.1 Une calculatrice simplifiée

Pour illustrer le profilage, nous allons utiliser un programme faisant office de calculatrice. Pour nous assurer que son exécution prend suffisamment de temps, nous utiliserons des nombres monadiques pour les calculs, chose que nous ne ferions jamais dans un programme réel. Le code de ce programme est présenté à la fin de ce chapitre.

Un *nombre monadique* (ou unaire) est représenté par autant de symboles que la valeur qu'il représente. Par exemple, le nombre 1 est représenté par "x", 2 par "xx" et 3 par "xxx". Au lieu d'utiliser des x, notre programme représentera un nombre positif en utilisant une liste chaînée constituée d'autant d'éléments que la valeur du nombre. Le fichier `number.c` contient les fonctions pour créer le nombre 0, ajouter 1 à un chiffre, soustraire 1 d'un nombre et ajouter, soustraire et multiplier des nombres. Une autre fonction convertit une chaine représentant une nombre décimal positif en un nombre unaire et enfin, une dernière fonction permet de passer d'un nombre unaire à un `int`. L'addition est implantée en effectuant des additions successives du nombre un tandis

que la soustraction utilise des soustractions répétitives du nombre 1. La multiplication utilise une répétition d'addtions. Les prédicats `even` et `odd` renvoient le nombre unaire 1 si et seulement si leur opérande est paire ou impaire (respectivement) ; sinon, ils renvoient le nombre unaire représentant 0. Ces deux prédicats s'appellent l'un l'autre, par exemple, un nombre est pair s'il vaut zéro ou si ce nombre moins un est impair.

La calculatrice accepte des expression postfixées[1] sur une ligne et affiche la valeur de chaque expression – par exemple :

```
% ./calculator
Veuillez saisir une expression postfixée :
2 3 +
5
Veuillez saisir une expression postfixée :
2 3 + 4 -
1
```

La calculatrice, définie dans `calculator.c`, lit chaque expression et stocke les valeurs intermédiaires sur une pile de nombres unaires, définie dans `stack.c`. La pile stocke ses nombres unaires dans une liste chaînée.

A.3.2 Collecter des informations de profilage

La première étape dans le profilage d'un programme est de marquer son exécutable de façon à ce qu'il collecte des informations de profilage. Pour cela, utilisez l'option de compilation `-pg` lors de la compilation et de l'édition de liens. Par exemple :

```
% gcc -pg -c -o calculator.o calculator.c
% gcc -pg -c -o stack.o stack.c
% gcc -pg -c -o number.o number.c
% gcc -pg calculator.o stack.o number.o -o calculator
```

Cette séquence de commandes active la collecte d'informations sur les appels de fonction et l'horodatage. Pour collecter des informations d'utilisation ligne par ligne, utilisez l'option de débogage `-g`. Pour compter le nombre d'exécutions de blocs de base, comme le nombre d'itérations des boubles `do`, utilisez `-a`.

La seconde étape est l'exécution du programme. Pendant son exécution, les données sont collectées dans un fichier nommé `gmon.out`, uniquement pour les portions de code qui ont été traversées. Vous devez varier les entrées du programme ou les commandes pour exécuter les sections du code que vous souhaitez profiler. Le programme doit se terminer normalement pour que le fichier de données de profilage soient écrites correctement.

A.3.3 Affichage des données de profilage

À partir du nom d'un exécutable, `gprof` analyse le fichier `gmon.out` pour afficher des informations sur le temps passé dans chaque fonction. Par exemple, examinons les données de profilage "brutes" pour le calcul de `1787 x 13 - 1918` en utilisant notre programme, elles sont fournies par la commande `grpof ./calculator` :

[1] Dans la notation *postfixée*, un opérateur binaire est placé après ses opérandes plutôt qu'entre elles. Ainsi, pour multiplier 6 par 8, vous utiliseriez 6 8 x. Pour multiplier 6 et 8 puis ajouter 5 au résultat, vous utiliseriez 6 8 x 5 +.

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls ms/call ms/call name
 26.07      1.76     1.76 20795463    0.00     0.00 decrement_number
 24.44      3.41     1.65     1787    0.92     1.72 add
 19.85      4.75     1.34 62413059    0.00     0.00 zerop
 15.11      5.77     1.02     1792    0.57     2.05 destroy_number
 14.37      6.74     0.97 20795463    0.00     0.00 add_one
  0.15      6.75     0.01     1788    0.01     0.01 copy_number
  0.00      6.75     0.00     1792    0.00     0.00 make_zero
  0.00      6.75     0.00       11    0.00     0.00 empty_stack
```

Le parcours de la fonction `decrement_number` et des sous fonctions qu'elle appelle occupe 26,07% du temps d'exécution total du programme. Elle a été appelé 20795463 fois. Chaque exécution a nécessité 0,0 seconde – autrement dit, un temps trop faible pour être mesuré. La fonction `add` a été invoquée 1787 fois, certainement pour calculer le produit. Chaque appel a demandé 0.92 secondes. La fonction `copy_number` a été invoquée 1788 fois, tandis que son exécution n'a nécessité que 0,15% du temps total d'exécution. Parfois, les fonctions `mcount` et `profil` utilisées pour le profilage apparaissent dans les données.

En plus des *données brutes de profilage*, qui indique le temps passé dans chacune des fonctions, `gprof` produit des *graphes d'appel* indiquant le temps passé dans chaque fonction et les fonctions qu'elle appelle dans le cadre d'une chaîne d'appels de fonctions :

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    6.75                 main [1]
                0.00    6.75       2/2           apply_binary_function [2]
                0.00    0.00       1/1792        destroy_number [4]
                0.00    0.00       1/1           number_to_unsigned_int [10]
                0.00    0.00       3/3           string_to_number [12]
                0.00    0.00       3/5           push_stack [16]
                0.00    0.00       1/1           create_stack [18]
                0.00    0.00       1/11          empty_stack [14]
                0.00    0.00       1/5           pop_stack [15]
                0.00    0.00       1/1           clear_stack [17]
-----------------------------------------------
                0.00    6.75       2/2           main [1]
[2]    100.0    0.00    6.75       2         apply_binary_function [2]
                0.00    6.74       1/1           product [3]
                0.00    0.01       4/1792        destroy_number [4]
                0.00    0.00       1/1           subtract [11]
                0.00    0.00       4/11          empty_stack [14]
                0.00    0.00       4/5           pop_stack [15]
                0.00    0.00       2/5           push_stack [16]
-----------------------------------------------
                0.00    6.74       1/1           apply_binary_function [2]
[3]     99.8    0.00    6.74       1         product [3]
                1.02    2.65    1787/1792        destroy_number [4]
                1.65    1.43    1787/1787        add [5]
                0.00    0.00    1788/62413059     zerop [7]
                0.00    0.00       1/1792        make_zero [13]
```

Le premier cadre indique que l'exécution de `main` a nécessité 100% des 6,75 secondes qu'a duré l'exécution du programme. Elle a appelé `apply_binary_function` deux fois, celle-ci ayant

été appelé deux fois pendant toute la durée d'exécution du programme. L'appelant de `main` était `<spontaneous>`, ce qui signifie que le profileur n'a pas été capable de le déterminer. Le premier cadre montre également que `string_to_number` a appelé `push_stack` trois fois sur les cinq fois où celle-ci a été appelée. Le troisième cadre montre que l'exécution de `product` et des fonctions qu'il appelle a nécessité 99,8% du temps d'exécution total. Elle a été invoquée une seule fois depuis `apply_binary_function`.

Le graphe d'appel indique le temps total d'exécution d'une fonction et de ses fils. Si le graphe d'appel est un arbre, ce temps est simple à calculer, mais les fonction récursives doivent être traitées d'une manière spéciale. Par exemple, la fonction `even` appelle `odd` qui appelle `even` à son tour. Chaque cycle d'appel de ce genre bénéficie de son propre horodatage et est affiché individuellement dans le graphe d'appel. Considérons ces données venant du profilage de la séquence visant à déterminer si 1787 x 13 x 3 est pair :

```
-----------------------------------------------
                0.00    0.02       1/1           main [1]
[9]      0.1    0.00    0.02       1         apply_unary_function [9]
                0.01    0.00       1/1           even <cycle 1> [13]
                0.00    0.00       1/1806        destroy_number [5]
                0.00    0.00       1/13          empty_stack [17]
                0.00    0.00       1/6           pop_stack [18]
                0.00    0.00       1/6           push_stack [19]
-----------------------------------------------
[10]     0.1    0.01    0.00       1+69693   <cycle 1 as a whole> [10]
                0.00    0.00   34847             even <cycle 1> [13]
-----------------------------------------------
                               34847             even <cycle 1> [13]
[11]     0.1    0.01    0.00   34847         odd <cycle 1> [11]
                0.00    0.00   34847/186997954     zerop [7]
                0.00    0.00       1/1806        make_zero [16]
                               34846             even <cycle 1> [13]
```

La valeur `1+69693` dans le cadre `[10]` indique que le cycle 1 a été appelé une fois, tandis qu'au sein de ce cycle il y a eu 69693 appels de fonction. Le cycle a appelé la fonction `even`. L'entrée suivante montre que la fonction `odd` a été appelée 34847 par `even`.

Dans cette section, nous avons brièvement présenté une partie des fonctionalités de `gprof`. Les pages `info` donnent des informations sur d'autres fonctionalités utiles :

- L'option `-s` affiche la somme des résultats pour plusieurs exécutions consécutives ;
- L'option `-c` permet d'identifier les fils qui auraient pu être appelés mais ne l'ont pas été ;
- L'option `-l` pour afficher des informations de profilage ligne par ligne.
- L'option `-A` pour afficher le code source annoté avec les pourcentages de temps d'exécution.

Les pages `info` donnent également plus d'informations sur la façon d'interpréter les résultats de l'analyse.

A.3.4 Comment *gprof* collecte les données

Lors du profilage d'un exécutable, à chaque fois qu'une fonction est appelée, le compteur qui y est associé est incrémenté. De plus, `gprof` interrompt régulièrement l'exécution pour déterminer la fonction en cours. Ces exemples montrent comment le temps d'exécution est déterminé. Comme les interruptions d'horloge de Linux surviennent toutes les 0,01 secondes, l'arrêt de l'exécution

ne peut avoir lieu qu'au maximum toutes les 0,01 secondes. Ainsi, le profilage de programmes s'exécutant très rapidement ou appelant peu souvent des fonctions qui s'exécutent rapidement pourrait être imprécis. Pour éviter cela, exécutez le programme plus longtemps ou additionnez les données de profilage de plusieurs exécutions. Reportez-vous à la documentation concernant l'option `-s` dans les pages `info` de `groff` pour plus d'informations.

A.3.5 Code source de la calculatrice

Le Listing A.3 présente un programme qui calcule la valeur d'expressions postfixées.

Listing A.3 – (*calculator.c*) – **Programme Principal de la calculatrice**

```
/* Effectue des calculs en utilisant des nombres unaires.  */
/* Saisissez des expressions utilisant la notation postfixée sur une ligne,
   par exemple : 602 7 5 - 3 * +
   Les nombres positifs sont saisis en utilisant une notation décimal. Les
   opérateurs "+", "-" et "*" sont acceptés. Les opérateurs unaires
   "even" et "odd" renvoient le nombre 1 si leur opérande est pair ou impair,
   respectivement. Les différents éléments doivent être séparés par des espaces.
   Les nombres négatifs ne sont pas pris en charge. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "definitions.h"
/* Applique la fonction binaire demandée aux opérandes obtenues depuis la pile
   et place le résultat sur la pile. Renvoie une valeur différente de zéro si
   tout se passe bien. */
int apply_binary_function (number (*function) (number, number),
                           Stack* stack)
{
  number operand1, operand2;
  if (empty_stack (*stack))
    return 0;
  operand2 = pop_stack (stack);
  if (empty_stack (*stack))
    return 0;
  operand1 = pop_stack (stack);
  push_stack (stack, (*function) (operand1, operand2));
  destroy_number (operand1);
  destroy_number (operand2);
  return 1;
}
/* Applique la fonction unaire demandée aux opérandes obtenues depuis la pile
   et place le résultat sur la pile. Renvoie une valeur différente de zéro si
   tout se passe bien. */
int apply_unary_function (number (*function) (number),
                          Stack* stack)
{
  number operand;
  if (empty_stack (*stack))
    return 0;
  operand = pop_stack (stack);
  push_stack (stack, (*function) (operand));
  destroy_number (operand);
  return 1;
}
int main ()
{
```

```
48    char command_line[1000];
49    char* command_to_parse;
50    char* token;
51    Stack number_stack = create_stack ();
52    while (1) {
53      printf ("Veuillez saisir une opération postfixée :\n");
54      command_to_parse = fgets (command_line, sizeof (command_line), stdin);
55      if (command_to_parse == NULL)
56        return 0;
57      token = strtok (command_to_parse, " \t\n");
58      command_to_parse = 0;
59      while (token != 0) {
60        if (isdigit (token[0]))
61          push_stack (&number_stack, string_to_number (token));
62        else if (((strcmp (token, "+") == 0) &&
63                  !apply_binary_function (&add, &number_stack)) ||
64                 ((strcmp (token, "-") == 0) &&
65                  !apply_binary_function (&subtract, &number_stack)) ||
66                 ((strcmp (token, "*") == 0) &&
67                  !apply_binary_function (&product, &number_stack)) ||
68                 ((strcmp (token, "even") == 0) &&
69                  !apply_unary_function (&even, &number_stack)) ||
70                 ((strcmp (token, "odd") == 0) &&
71                  !apply_unary_function (&odd, &number_stack)))
72          return 1;
73        token = strtok (command_to_parse, " \t\n");
74      }
75      if (empty_stack (number_stack))
76        return 1;
77      else {
78        number answer = pop_stack (&number_stack);
79        printf ("%u\n", number_to_unsigned_int (answer));
80        destroy_number (answer);
81        clear_stack (&number_stack);
82      }
83    }
84    return 0;
85  }
```

Les fonctions du Listing A.4 implante les nombres unaires en utilisant des listes chaînées vides.

Listing A.4 – (*number.c*) – **Implantation d'un Nombre Unaire**

```
1  /* Opérations sur les nombres unaires.  */
2  #include <assert.h>
3  #include <stdlib.h>
4  #include <limits.h>
5  #include "definitions.h"
6  /* Crée un nombre représentant zéro.  */
7  number make_zero ()
8  {
9    return 0;
10 }
11 /* Renvoie une valeur différente de zéro si le nombre représente un zéro.  */
12 int zerop (number n)
13 {
14   return n == 0;
15 }
16 /* Soustrait 1 à un nombre positif.  */
17 number decrement_number (number n)
```

```
18 {
19   number answer;
20   assert (!zerop (n));
21   answer = n->one_less_;
22   free (n);
23   return answer;
24 }
25 /* Ajoute 1 à un nombre.  */
26 number add_one (number n)
27 {
28   number answer = malloc (sizeof (struct LinkedListNumber));
29   answer->one_less_ = n;
30   return answer;
31 }
32 /* Détruit un nombre.  */
33 void destroy_number (number n)
34 {
35   while (!zerop (n))
36     n = decrement_number (n);
37 }
38 /* Copie un nombre. Cette fonction n'est nécessaire qu'à
39    cause de l'allocation mémoire. */
40 number copy_number (number n)
41 {
42   number answer = make_zero ();
43   while (!zerop (n)) {
44     answer = add_one (answer);
45     n = n->one_less_;
46   }
47   return answer;
48 }
49 /* Additionne deux nombres.  */
50 number add (number n1, number n2)
51 {
52   number answer = copy_number (n2);
53   number addend = n1;
54   while (!zerop (addend)) {
55     answer = add_one (answer);
56     addend = addend->one_less_;
57   }
58   return answer;
59 }
60 /* Soustrait un nombre d'un autre.  */
61 number subtract (number n1, number n2)
62 {
63   number answer = copy_number (n1);
64   number subtrahend = n2;
65   while (!zerop (subtrahend)) {
66     assert (!zerop (answer));
67     answer = decrement_number (answer);
68     subtrahend = subtrahend->one_less_;
69   }
70   return answer;
71 }
72 /* Renvoie le produit de deux nombres.  */
73 number product (number n1, number n2)
74 {
75   number answer = make_zero ();
76   number multiplicand = n1;
77   while (!zerop (multiplicand)) {
78     number answer2 = add (answer, n2);
79     destroy_number (answer);
```

```
80        answer = answer2;
81        multiplicand = multiplicand->one_less_;
82      }
83      return answer;
84    }
85    /* Renvoie une valeur différente de zéro si un nombre est
86       pair.  */
87    number even (number n)
88    {
89      if (zerop (n))
90        return add_one (make_zero ());
91      else
92        return odd (n->one_less_);
93    }
94    /* Renvoie une valeur différente de zéro si un nombre est
95       impair.  */
96    number odd (number n)
97    {
98      if (zerop (n))
99        return make_zero ();
100     else
101       return even (n->one_less_);
102   }
103   /* Convertit une chaîne représentant un entier décimal en un "number". */
104   number string_to_number (char * char_number)
105   {
106     number answer = make_zero ();
107     int num = strtoul (char_number, (char **) 0, 0);
108     while (num != 0) {
109       answer = add_one (answer);
110       --num;
111     }
112     return answer;
113   }
114   /* Convertit un "number" en "unsigned int".  */
115   unsigned number_to_unsigned_int (number n)
116   {
117     unsigned answer = 0;
118     while (!zerop (n)) {
119       n = n->one_less_;
120       ++answer;
121     }
122     return answer;
123   }
```

La fonction du Listing A.5 implante une pile de nombre unaires en utilisant une liste chaînée.

Listing A.5 – (*stack.c*) – **Pile de Nombres Unaires**

```
1   /* Implante une pile de "number"s.  */
2   #include <assert.h>
3   #include <stdlib.h>
4   #include "definitions.h"
5   /* Crée une pile vide.  */
6   Stack create_stack ()
7   {
8     return 0;
9   }
10  /* Renvoie une valeur différente de zéro si la pile est vide.  */
11  int empty_stack (Stack stack)
12  {
13    return stack == 0;
```

```
14  }
15  /* Supprime le number situé au sommet d'une pile non vide.
16     Échoue si la pile est vide. */
17  number pop_stack (Stack* stack)
18  {
19    number answer;
20    Stack rest_of_stack;
21    assert (!empty_stack (*stack));
22    answer = (*stack)->element_;
23    rest_of_stack = (*stack)->next_;
24    free (*stack);
25    *stack = rest_of_stack;
26    return answer;
27  }
28  /* Ajoute un number au début de la pile.  */
29  void push_stack (Stack* stack, number n)
30  {
31    Stack new_stack = malloc (sizeof (struct StackElement));
32    new_stack->element_ = n;
33    new_stack->next_ = *stack;
34    *stack = new_stack;
35  }
36  /* Supprime tous les éléments de la pile.  */
37  void clear_stack (Stack* stack)
38  {
39    while (!empty_stack (*stack)) {
40      number top = pop_stack (stack);
41      destroy_number (top);
42    }
43  }
```

Le Listing A.6 contient les déclaration de la pile et des nombres.

Listing A.6 – (*definitions.h*) – **Fichier d'En-Tête pour number.c et stack.c**

```
1   #ifndef DEFINITIONS_H
2   #define DEFINITIONS_H 1
3   /* Implante un number en utilisant une liste chaînée.  */
4   struct LinkedListNumber
5   {
6      struct LinkedListNumber*
7                     one_less_;
8   };
9   typedef struct LinkedListNumber* number;
10  /* Implante une pile de numbers sous forme de liste chaînée. Utilise 0 pour
11     représenter une pile vide. */
12  struct StackElement
13  {
14     number         element_;
15     struct         StackElement* next_;
16  };
17  typedef struct StackElement* Stack;
18  /* Opérations sur les piles de numbers. */
19  Stack create_stack ();
20  int empty_stack (Stack stack);
21  number pop_stack (Stack* stack);
22  void push_stack (Stack* stack, number n);
23  void clear_stack (Stack* stack);
24  /* Operations sur les numbers. */
25  number make_zero ();
26  void destroy_number (number n);
27  number add (number n1, number n2);
```

```
28 number subtract (number n1, number n2);
29 number product (number n1, number n2);
30 number even (number n);
31 number odd (number n);
32 number string_to_number (char* char_number);
33 unsigned number_to_unsigned_int (number n);
34 #endif /* DEFINITIONS_H */
```

Annexe B

E/S de bas niveau

Les programmeurs C sous GNU/Linux ont deux jeux de fonctions d'entrées/sorties à leur disposition. La bibliothèque C standard fournit des fonctions d'E/S : `printf`, `fopen`, *etc*[1]. Le noyau Linux fournit un autre ensemble d'opérations d'E/S qui opèrent à un niveau inférieur à celui des fonctions de la bibliothèque C.

Comme ce livre est destiné à des gens connaissant déjà le langage C, nous supposerons que vous avez déjà rencontré et savez comment utiliser les fonctions d'E/S de la bibliothèque C.

Il y a souvent de bonnes raisons d'utiliser les fonctions d'E/S de bas niveau de Linux. Elles sont pour la plupart des appels systèmes au noyau[2] et fournissent un accès direct aux possibilités sous-jacentes offertes par le système aux applications. En fait, les fonctions d'E/S de la bibliothèque C standard sont implantées par dessus les appels systèmes d'E/S de bas niveau de Linux. L'utilisation de cette dernière est généralement la façon la plus efficace d'effectuer des opérations d'entrée/sortie – et est parfois également plus pratique.

Tout au long de ce livre, nous supposons que vous êtes familier avec les appels décrits dans cette annexe. Vous êtes peut être déjà familiers avec eux car ils sont très proches de ceux fournis avec d'autres système d'exploitation de type unique (ainsi qu'avec la plateforme Win32). Si vous n'êtes pas coutumier de ces appels, cependant, continuez votre lecture ; le reste du livre n'en sera que plus simple à comprendre si vous commencez par prendre connaissance de ce chapitre.

B.1 Lire et écrire des données

La première fonction d'E/S que vous avez rencontré lorsque vous avez commencé à apprendre le langage C était certainement `printf`. Elle formate une chaîne de texte puis l'affiche sur la sortie standard. La version générique, `fprintf`, peut afficher le texte sur un autre flux que la sortie standard. Un flus est représenté par un pointeur `FILE*`. Vous obtenez un tel pointeur en ouvrant un fichier avec `fopen`. Lorsque vous en avez fini, vous pouvez le fermer avec `fclose`. En plus de

[1] La bibliothèque C++ standard fournit les *flux d'E/S* (iostreams) qui proposent des fonctionnalités similaires. La bibliothèque C standard est également disponible avec le langage C++.

[2] Consultez le Chapitre 8, « Appels système Linux » pour des explications concernant la différence entre un appel système et un appel de fonction traditionnelle.

`fprintf`, vous pouvez utiliser d'autres fonctions comme `fputc`, `fputs` ou `fwrite` pour écrire des données dans un flux, ou `fscanf`, `fgetc`, `fgets` ou `fread` pour lire des données.

Avec les opérations d'E/S de bas niveau de Linux, vous utilisez *descripteur de fichier* au lieu d'un pointeur `FILE*`. Un descripteur de fichier est un entier qui fait référence à une instance donnée d'un fichier ouvert au sein d'un processus. Il peut être ouvert en lecture, en écriture ou en lecture/écriture. Un descripteur de fichier ne fait pas forcément référence à un fichier ouvert ; il peut représenter une connexion vers un composant d'un autre système qui est capable d'envoyer ou de recevoir des données. Par exemple, une connexion vers un dispositif matériel est représentée par un descripteur de fichier (voir Chapitre 6, « Périphériques »), tout comme l'est un socket ouvert (voir Chapitre 5, « Communication interprocessus », Section 5.5, « Sockets ») ou l'extrémité d'un tube (voir Section 5.4, « Tubes »).

Incluez les fichiers d'en-tête `<fcntl.h>`, `<sys/types.h>`, `<sys/stat.h>` et `<unistd.h>` si vous utilisez l'une des fonctions d'E/S de bas niveau décrites ici.

B.1.1 Ouvrir un fichier

Pour ouvrir un fichier et obtenir un descripteur de fichier pouvant y accéder, utilisez l'appel `open`. Il prend le chemin du fichier à ouvrir sous forme d'une chaîne de caractères et des indicateurs spécifiant comment il doit l'être. Vous pouvez utiliser `open` pour créer un nouveau fichier ; pour cela, passez un troisième argument décrivant les droits d'accès à appliquer au nouveau fichier.

Si le second argument est `O_RDONLY`, le fichier est ouvert en lecture seul ; un erreur sera signalée si vous essayez d'y écrire. De même, `O_WRONLY` ouvre le descripteur de fichier en écriture seule. Passer `O_RDWR` crée un descripteur de fichier pouvant être utilisé à la fois en lecture et en écriture. Notez que tous les fichiers ne peuvent pas être ouverts dans les trois modes. Par exemple, les permissions d'un fichier peuvent interdire à un processus de l'ouvrir en lecture ou en écriture ; un fichier sur un périphérique en lecture seule, comme un lecteur CD-ROM ne peut pas être ouvert en lecture.

Vous pouvez passer des options supplémentaires en utilisant un OU binaire de ces valeurs avec d'autres indicateurs. Voici les valeurs les plus courantes :

- Passez `O_TRUNC` pour tronquer le fichier ouvert, s'il existait auparavant. Les données écrites remplaceront le contenu du fichier ;
- Passez `O_APPEND` pour ajouter les données au contenu d'un fichier existant. Elles sont écrites à la fin du fichier ;
- Passez `O_CREAT` pour créer un nouveau fichier. Si le nom de fichier que vous passez à `open` correspond à un fichier inexistant, un nouveau fichier sera créé, si tant est que le répertoire le contenant existe et que le processus a les permissions nécessaires pour créer des fichiers dans ce répertoire. Si le fichier existe déjà, il est ouvert ;
- Passez `O_EXCL` et `O_CREATE` pour forcer la création d'un nouveau fichier. Si le fichier existe déjà, l'appel à `open` échouera.

Si vous appelez `open` en lui passant `O_CREATE`, fournissez un troisième argument indiquand les permissions applicable au nouveau fichier. Consultez le Chapitre 10, « Sécurité », Section 10.3, « Permissions du système de fichiers », pour une description des bits de permission et de la façon de les utiliser.

Par exemple, le programme du Listing B.1 crée un nouveau fichier avec le nom de fichier indiqué sur la ligne de commande. Il utilise l'indicateur `O_EXCL` avec `open` afin de signaler une erreur si le fichier existe déjà. Le nouveau fichier dispose des autorisations en lecture/écriture pour l'utilisateur et le groupe propriétaires et est en lecture seule pour les autres utilisateurs (si votre umask est positionné à une valeur différente de zéro, les permissions effectives pourraient être plus restrictives).

Umasks

Lorsque vous créez un nouveau fichier avec open, certains des bits de permissions peuvent être désacrivés. Cela survient lorsque votre umask est différent de zéro. L'umask d'un processus spécifie les bits de permissions qui sont masqués lors de n'importe quelle création de fichier. Les permissions effectives sont obtenues en appliquant un ET binaire entre les permissions que vous passez à open et le complément à un du umask.
Pour changer la valeur de votre umask à partir d'un shell, utilisez la commande `umask` et indiquez la valeur numérique du masque en notation octale. Pour changer le umask d'un processus en cours d'exécution, utilisez l'appel `umask` en lui passant la valeur du masque à utiliser pour les appels suivants.
Par exemple, cette ligne :
`umask (S_IRWXO | S_IWGRP);`
dans un programme ou l'invocation de cette commande : `% umask 027`
indiquent que les permissions en écriture pour le groupe et toutes les permissions pour les autres seront toujours masquée pour les créations de fichiers.

Listing B.1 – (*create-file.c*) – **Create a New File**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <sys/stat.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  int main (int argc, char* argv[])
7  {
8    /* Chemin vers le nouveau fichier. */
9    char* path = argv[1];
10   /* Permissions du nouveau fichier. */
11   mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |  S_IROTH;
12   /* Crée le fichier. */
13   int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
14   if (fd == -1) {
15     /* Une erreur est survenue, affiche un message et quitte.  */
16     perror ("open");
17     return 1;
18   }
19   return 0;
20 }
```

Voici le programme en action :

```
% ./create-file testfile
% ls -l testfile
-rw-rw-r--    1 samuel   users 0 Feb 1 22:47 testfile
% ./create-file testfile
open: Le fichier existe
```

Notez que le fichier fait zéro octet car le programme n'y a écrit aucune donnée.

B.1.2 Fermer un fichier

Lorsque vous en avez fini avec un descripteur de fichier, fermez-le avec `close`. Dans certains cas, comme dans le cas du programme du Listing B.1 il n'est pas nécessaire d'appeler `close` explicitement car Linux ferme tous les descripteurs de fichiers lorsqu'un processus se termine (c'est-à-dire à la fin du programme). Bien sûr, une fois que vous avez fermé un descripteur de fichier, vous ne pouvez plus l'utiliser.

La fermeture d'un descripteur de fichier peut déclencher des actions spécifiques de la part de Linux, selon la nature du descripteur de fichier. Par exemple, lorsque vous fermez un descripteur correspondant à un socket réseau, Linux ferme la connexion entre les deux ordinateurs communicant *via* le socket.

Linux limite le nombre de descripteurs de fichiers qu'un processus peut maintenir ouverts en même temps. Les descripteurs de fichiers ouverts utilisent des ressources noyau, il est donc conseillé de fermer les descripteurs de fichiers dès que vous avez terminer de les utiliser. La limite classique est de 1024 descripteurs par processus. Vous pouvez l'ajuster avec l'appel système `setrlimit` ; consultez la Section 8.5, « *getrlimit* et *setrlimit*: limites de ressources », pour plus d'informations.

B.1.3 Écrire des données

Écrire des données dans un fichier se fait par le biais de l'appel `write`. Passz lui un descripteur de fichier, un pointeur vers un tampon de données et le nombre d'octets à écrire. Les données écrites n'ont pas besoin d'être une chaine de caractères ; `write` copie des octets quelconques depuis le tampon vers le descripteur de fichier.

Le programme du Listing B.2 écrit l'heure courante à la fin du fichier passé sur la ligne de commande. Si le fichier n'existe pas, il est créé. Ce programme utilise également les fonctions `time`, `localtime` et `asctime` pour obtenir et formater l'heure courante ; consultez leurs pages de manuel respectives pour plus d'informations.

Listing B.2 – (*timestamp.c*) – **Ajoute l'Heure Courante à un Fichier**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6  #include <time.h>
7  #include <unistd.h>
8  /* Renvoie une chaine représentant l'heure courante. */
9  char* get_timestamp ()
10 {
11   time_t now = time (NULL);
```

```
12    return asctime (localtime (&now));
13  }
14  int main (int argc, char* argv[])
15  {
16    /* Fichier auquel ajouter l'horodatage. */
17    char* filename = argv[1];
18    /* Récupère l'heure courante. */
19    char* timestamp = get_timestamp ();
20    /* Ouvre le fichier en écriture. S'il existe, ajoute les données
21       à la fin, sinon, un nouveau fichier est créé. */
22    int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
23    /* Calcule la longueur de la chaîne d'horodatage. */
24    size_t length = strlen (timestamp);
25    /* L'écrit dans le fichier. */
26    write (fd, timestamp, length);
27    /* Fini. */
28    close (fd);
29    return 0;
30  }
```

Voici comment fonctionne ce programme :

```
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

Notez que la première fois que nous invoquons `timestamp`, il crée le fichier `tsfile`, alors que la seconde fois, les données sont ajoutées à la fin.

L'appel `write` renvoie le nombre d'octets effectivement écrits ou -1 si une erreur survient. Pour certains types de descripteurs de fichiers, le nombre d'octets écrits peut être inférieur au nombre d'octets demandés. Dans ce cas, c'est à vous d'appeler `write` encore une fois pour écrire le reste des données. La fonction du Listing B.3 montre une façon de le faire. Notez que pour certaines applications, vous pourriez avoir à effectuer des contrôles supplémentaires avant la reprise de l'écriture. Par exemple, si vous utilisez un socket réseau, il faudrait ajouter le code permettant de détecter si la connexion a été fermée pendant l'écriture et, si c'est le cas, prendre les mesures adéquates.

Listing B.3 – (*write-all.c*) – **Écrit tout le Contenu d'un Tampon**

```
1  /* Écrit l'intégralité des COUNT octets de BUFFER vers le descripteur
2     de fichier FD. Renvoie -1 en cas d'erreur ou le nombre d'octets
3     écrits. */
4  ssize_t write_all (int fd, const void* buffer, size_t count)
5  {
6    size_t left_to_write = count;
7    while (left_to_write > 0) {
8      size_t written = write (fd, buffer, count);
9      if (written == -1)
10       /* Une erreur est survenue. Terminé. */
11       return -1;
12     else
13       /* Mémorise le nombre d'octets restant à écrire. */
14       left_to_write -= written;
15   }
```

```
16    /* Nous ne devons pas avoir écrit plus de COUNT octets ! */
17    assert (left_to_write == 0);
18    /* Le nombre d'octets écrits est exactement COUNT. */
19    return count;
20  }
```

B.1.4 Lecture de données

L'appel permettant de lire des données est `read`. Tout comme `write`, il prend en arguments un descripteur de fichier, un pointeur vers un tampon et un nombre d'octets. Ce dernier indique le nombre d'octets à lire à partir du descripteur. L'appel à `read` renvoie -1 en cas d'erreur ou le nombre d'octets effectivement lus. Il peut être inférieur au nombre d'octets demandé, par exemple, s'il ne reste pas suffisamment d'octets à lire dans le fichier.

Lire des Fichiers Texte DOS/Windows

Une fois que vous aurez lu ce livre, nous sommes convaincus que vous déciderez d'écrire tous vos programmes pour GNU/Linux. Cependant, il pourrait vous arriver d'avoir à lire des fichiers texte générés par des programmes DOS ou Windows. Il est important d'anticiper une différence majeure entre les deux plateformes dans la façon de structurer les fichiers texte.

Dans les fichiers texte GNU/Linux, chaque ligne est séparée de la suivante par un caractère de nouvelle ligne. Ce dernier est représenté par la constante de caractère `'\n'` dont le code ASCII est 10. Sou Windows, par contre, les lignes sont séparées par une séquence de deux caractères : un retour chariot (le caractère `'\r'`, dont le code ASCII est 13), suivi d'un caractère de nouvelle ligne.

Certains éditeurs de texte GNU/Linux affichent un `^M` à la fin de chaque ligne lorsqu'ils affichent le contenu d'un fichier texte Windows – il s'agit du caractère de retour chariot. Emacs affiche les fichiers texte Windows correctement mais les signale en affichant `(DOS)` dans la barre de mode en bas du tampon. Certains éditeurs Windows, comme Notepad, affichent tout le texte des fichiers GNU/Linux sur une seule ligne car ils ne trouvent pas le caractère de retour chariot à la fin de chaque ligne. D'autres programmes, que ce soit sous GNU/Linux ou Windows, peuvent signaler des erreurs étranges lorsque les fichiers qui leur sont fournis en entrée ne sont pas au bon format.

Si votre programme lit des fichiers texte générés par des programmes Windows, vous voudrez probablement remplacer la séquence `'\r\n'` par un seul caractère de nouvelle ligne. De même, si votre programme écrit des fichiers texte qui doivent être lus par des programmes Windows, remplacez le caractère de nouvelle ligne par la séquence `'\r\n'`. Vous devrez le faire, que vous utilisiez les appels d'E/S de bas niveau présentés dans cette annexe ou les fonctions de la bibliothèque C standard.

Le Listing B.4 présente un programme utilisant l'appel `read`. Il affiche une image hexadécimale du contenu du fichier passé sur la ligne de commande. Chaque ligne affiche le déplacement dans le fichier et les 16 octets suivants.

Listing B.4 – (*hexdump.c*) – **Affichage de l'Image Hexadécimale d'un Fichier**

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  unsigned char buffer[16];
  size_t offset = 0;
  size_t bytes_read;
  int i;
  /* Ouvre le fichier en lecture. */
  int fd = open (argv[1], O_RDONLY);
  /* Lit le fichier morceau par morceau, jusqu'à ce que la lecture
     soit "trop courte", c'est-à-dire que l'on lise moins que ce que
     l'on a demandé, ce qui indique que la fin du fichier est atteinte. */
  do {
    /* Lis la "ligne" suivante. */
    bytes_read = read (fd, buffer, sizeof (buffer));
    /* Affiche le déplacement dans le fichier, suivi des octets correspondants. */
    printf ("0x%06x : ", offset);
    for (i = 0; i < bytes_read; ++i)
       printf ("%02x ", buffer[i]);
    printf ("\n");
    /* Conserve notre position dans le fichier. */
    offset += bytes_read;
  }
  while (bytes_read == sizeof (buffer));
  /* Terminé.   */
  close (fd);
  return 0;
}
```

Voici `hexdump` en action. Il affiche l'image de son propre exécutable :

```
% ./hexdump hexdump
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...
```

La sortie peut être différente chez vous, selon le compilateur utilisé pour construire `hexdump` est les options de compilation.

B.1.5 Se déplacer dans un fichier

Un descripteur de fichier connait sa position dans le fichier. Lorsque vous y écrivez ou que vous y lisez, sa position est modifiée selon le nombre d'octets lus ou écrits. Parfois, cependant, vous pouvez avoir besoin de vous déplacer dans un fichier sans lire ou écrire de données. Par exemple, vous pouvez vouloir écrire au milieu d'un fichier sans en modifier le début, ou vous pouvez avoir besoin de retourner au début d'un fichier et de le relire sans avoir à le réouvrir.

L'appel `lseek` vous permet de modifier votre position dans un fichier. Passez lui le descripteur deux fichier et deux autres arguments indiquand la nouvelle position.

- Si le troisième argument est `SEEK_SET`, `lseek` interprète le second argument comme une position, en octets, depusi le début du fichier ;
- Si le troisième argument est `SEEK_CUR`, `lseek` interprète le second argument comme un déplacement, positif ou négatif, depuis la position courante ;
- Si le troisième argument est `SEEK_END`, `lseek` interprète le second argument comme un déplacement à partir de la fin du fichier. Une valeur positive indique une position au-delà de la fin du fichier.

L'appel à `lseek` renvoie la nouvelle position, sous forme d'un déplacement par rapport au début du fichier.Le type de ce déplacement est `off_t`. Si une erreur survient, `lseek` renvoie -1. Vous ne pouvez pas utiliser `lseek` avec certains types de descripteurs de fichiers comme les sockets.

Si vous voulez obtenir votre position dans un fichier sans la modifier, indiquez un déplacement de 0 par rapport à votre position actuelle – par exemple :

```
off_t position = lseek (file_descriptor, 0, SEEK_CUR);
```

Linux autorise l'utilisation de `lseek` pour spécifier une position au-delà de la fin du fichier. Normalement, si un descripteur de fichier est positionné à la fin d'un fichier et que vous y écrivez, Linux augmente automatiquement la taille du fichier pour faire de la place pour les nouvelles données. Si vous indiquez une position au-delà de la fin du fichier puis que vous y écrivez, Linux commence par agrandir le fichier de la taille du "trou" que vous avez créé en appelant `lseek` puis écrit à la fin de celui-ci. Ce trou n'occupe toutefois pas de place sur le disque ; Linux note simplement sa taille. Si vous essayez de lire le fichier, il apparait comme s'il était rempli d'octets nuls.

En utilisant cette propriété de `lseek`, il est possible de créer des fichier extrèmement grands qui n'occupent pratiquement aucun espace disque. Le programme `lseek-huge` du Listing B.5 le fait. Il prend en arguments de ligne de commande un nom de fichier et sa taille, en mégaoctets. Le programme ouvre un nouveau fichier, se place après la fin de celui-ci en utilisant `lseek` puis écrit un octet à 0 avant de fermer le fichier.

Listing B.5 – (*lseek-huge.c*) – **Créer de Gros Fichiers avec *lseek***

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  int zero = 0;
  const int megabyte = 1024 * 1024;
  char* filename = argv[1];
  size_t length = (size_t) atoi (argv[2]) * megabyte;
  /* Ouvre un nouveau fichier. */
  int fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
  /* Se place un octet avant la fin désirée du fichier. */
  lseek (fd, length - 1, SEEK_SET);
  /* Écrit un octet nul. */
  write (fd, &zero, 1);
  /* Terminé. */
  close (fd);
  return 0;
}
```

Voici comment créer un fichier d'un gigaoctet (1024 Mo) en utilisant `lseek-huge`. Notez l'espace libre avant et après l'opération.

```
% df -h .
Filesystem            Tail. Occ. Disp. %Occ. Monté sur
/dev/hda5             2.9G  2.1G  655M  76% /
% ./lseek-huge grosfichier 1024
% ls -l grosfichier
-rw-r-----    1 samuel   samuel  1073741824 Feb 5 16:29 grosfichier
% df -h .
Filesystem            Tail. Occ. Disp. %Occ. Monté sur
/dev/hda5             2.9G  2.1G  655M  76% /
```

Aucun espace disque significatif n'est utilisé en dépit de la taille conséquente de `grosfichier`. Si nous ouvrons `grosfichier` et que nous y lisons des connées, il apparait comme étant rempli de 1 Go d'octets nuls. Par exemple, nous pouvons inspecter son contenu avec le programme `hexdump` du Listing B.4.

```
% ./hexdump grosfichier | head -10
0x000000 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
0x000040 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
0x000050 : 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00
...
```

Si vous exécutez cette commande vous-même, vous devrez probabelement la tuer avec Ctrl+C plutôt de la regarder affichier 2^{30} octets nuls.

Notez que ces trous magiques dans les fichiers sont une fonctionalité spéciale du système de fichier `ext2` qui est traditionnelement utilisé pour les disques GNU/Linux. Si vous utilisez `lseek-huge` pour créer un fichier sur un autre type de système de fichiers, comme `fat` ou `vfat` qui sont utilisés sur certaines partitions DOS et Windows, vous constaterez que le fichier occupe effectivement autant d'espace que sa taille.

Linux n'autorise pas le positionnement avant le début du fichier avec `lseek`.

B.2 *stat*

En utilisant `open` et `read`, vous pouvez extraire le contenu d'un fichier. Mais qu'en est-il des autres informations ? Par exemple, la commande `ls -l` affiche des informations comme la taille, la date de dernière modification, les permissions ou le propriétaire pour les fichiers du répertoire courant.

L'appel `stat` récupère ces informations pour un fichier. Appelez `stat` en lui passant le chemin du fichier sur lequel vous voulez des informations et un pointeur sur une variable de type `struct stat`. Si l'appel à `stat` se déroule correctement, il revoie 0 et renseigne les champs de la structure avec des informations sur le fichier ; sinon, il renvoie -1.

Voici les champs les plus intéressant d'une `struct stat` :

– `st_mode` contient les permissions du fichier. Ces permissions sont décrites dans la Section 10.3, « Permissions du système de fichiers » ;

– En plus des permissions clasiques, le champ `st_mode` encode le type de fichier dans les bits de poids fort. Consultez les explication ci-dessous pour savoir comment le décoder ;
– `st_uid` et `st_gid` contiennent les identifiants de l'utilisateur et du groupe auxquels le fichier appartient, respectivement. Les identifiants de groupe et d'utilisateur sont détaillés dans la Section 10.1, « Utilisateurs et groupes » ;
– `st_size` contient la taille du fichier, en octets ;
– `st_atime` contient la date de dernier accès au fichier (en lecture ou écriture) ;
– `st_mtime` contient la date de dernière modification du fichier.

Un certain nombre de macros analysent la valeur du champ `st_mode` pour déterminer le type de fichier sur lequel `stat` a été invoqué. Une macro vaut vrai si le fichier est de ce type :

`S_ISBLK (mode)`	périphérique bloc
`S_ISCHR (mode)`	périphérique caractère
`S_ISDIR (mode)`	répertoire
`S_ISFIFO (mode)`	fifo (canal nommé)
`S_ISLNK (mode)`	lien symbolique
`S_ISREG (mode)`	fichier classique
`S_ISSOCK (mode)`	socket

Le champ `st_dev` contient les numéros de périphérique majeur et mineur du dispositif matériel sur lequel le fichier se situe. Les numéros de périphériques sont traités dans le Chapitre 6. Le numéro majeur de périphérique est décalé à gauche de 8 bits, tandis que le numéro mineur occupe les 8 bits de poids faible. Le champ `st_ino` contient le *numéro d'inode* du fichier. Cela situe le fichier sur le système de fichiers.

Si vous appelez `stat` sur un lien symbolique, `stat` suit le lien et vous renvoie les informations sur le fichier sur lequel il pointe, pas sur le lien symbolique. Cela signifie que `S_ISLNK` ne sera jamais vrai pour une valeur renvoyée par `stat`. Utilisez la fonction `lstat` si vous ne voulez pas suivre les liens symboliques ; cette fonction renvoie les informations sur le lien et non pas sur le fichier cible. Si vous appelez `lstat` sur un fichier qui n'est pas un lien symbolique, elle a le même comportement que `stat`. Appeler `stat` sur un lien invalide (un lien qui pointe vers un fichier inexistant ou inaccessible) provoque une erreur, alors que l'appel de `lstat` sur le même fichier n'a aucune incidence.

Si vous disposez déjà d'un fichier ouvert en lecture/écriture, appelez `fstat` au lieu de `stat`. Il prend un descripteur de fichier en premier argument à la place du chemin vers le fichier.

Le Listing B.6 présente une fonction qui alloue un tampon suffisamment long pour recevoir le contenu du fichier et charge les données à l'intérieur. Elle utilise `fstat` pour déterminer la taille du tampon à allouer et vérifier que le fichier est bien un fichier classique.

Listing B.6 – (***read-file.c***) – **Charge un Fichier dans un Tampon**

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <sys/stat.h>
4  #include <sys/types.h>
5  #include <unistd.h>
6  /* Charge le contenu de FILENAME dans un tampon nouvellement alloué. La
7     taille du tampon est stockée dans *LENGTH. Renvoie le tampon, qui devra
8     être libéré par l'appelant. Renvoie NULL si FILENAME ne correspond pas
9     à un fichier régulier. */
10 char* read_file (const char* filename, size_t* length)
```

```
11 {
12   int fd;
13   struct stat file_info;
14   char* buffer;
15   /* Ouvre le fichier. */
16   fd = open (filename, O_RDONLY);
17   /* Récupère les informations sur le fichier. */
18   fstat (fd, &file_info);
19   *length = file_info.st_size;
20   /* S'assure que le fichier est un fichier ordinaire.  */
21   if (!S_ISREG (file_info.st_mode)) {
22     /* Ce n'en est pas un, abandonne. */
23     close (fd);
24     return NULL;
25   }
26   /* Alloue un tampon suffisamment grand pour recevoir le contenu du fichier. */
27   buffer = (char*) malloc (*length);
28   /* Charge le fichier dans le tampon. */
29   read (fd, buffer, *length);
30   /* Terminé. */
31   close (fd);
32   return buffer;
33 }
```

B.3 Écriture et lecture vectorielles

L'appel `write` prend en arguments un pointeur vers le début d'un tampon de données et la taille de ce tampon. Il écrit le contenu d'une région contigüe de mémoire vers un descripteur de fichier. Cependant, un programme a souvent besoin d'écrir plusieurs éléments de données, chacun se trouvant à un endroit différent. Pour utiliser `write`, un tel programme devrait soit copier tous les objets dans une région contigüe, ce qui gaspillerait des cycles CPU et de la mémoire, soit effectuer de multiples appels à `write`.

Pour certaines applications, appeler `write` plusieurs fois peut être inefficace ou peu souhaitable. Par exemple, lors de l'écriture vers un socket réseau, deux appels à `write` peut provoquer l'envoi de deux paquets sur le réseau, alors que les mêmes données auraient pu être envoyées en une fois si un seul appel à `write` avait été possible.

L'appel `writev` vous permet d'envoyer le contenu de plusieurs zones mémoire non-contigües vers un descripteur de fichier en une seul opération. C'est ce que l'on appelle l'*écriture vectorielle*. La contrepartie de l'utilisation de `writev` est que vous devez créer une structure de données indiquand le début et la taille de chaque région mémoire. Cette structure est un tableau d'éléments `struct iovec`. Chaque élément correspond à un emplacement mémoire à écrire ; les champs `iov_base` et `iov_len` correspondent respectivement à l'adresse du début de la région et à sa taille. Si vous connaissez à l'avance le nombre de zones mémoire à écrire, vous pouvez vous contenter de déclarer un tableau de `struct iovec` ; si le nombre de régions peut varier, vous devez allouer le tableau dynamiquement.

Appelez `writev` en lui passant le descripteur de fichier vers lequel envoyer les données, le tableau de `struct iovec` et le nombre d'éléments contenus dans ce tableau. La valeur de retour correspond au nombre d'octets écrits.

Le programme du Listing B.7 écrit ses arguments de ligne de commande dans un fichier en utilisant un unique appel à `writev`. Le premier argument est le nom du fichier dans lequel écrire les arguments qui suivent. Le programme alloue un tableau de `struct iovec` qui fait le double du nombre d'arguments à écrire – pour chaque argument, le texte de l'argument proprement dit est écrit, suivit d'un caractère de nouvelle ligne. Comme nous ne savons pas à l'avance le nombre d'arguments, le tableau est créé en utilisant `malloc`.

Listing B.7 – (*write-args.c*) – **Écrit la Liste d'Arguments dans un Fichier avec *writev***

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  int fd;
  struct iovec* vec;
  struct iovec* vec_next;
  int i;
  /* Nous aurons besoin d'un "tampon" contenant un caractère de nouvelle ligne.
     Nous utilisons une variable char normale pour cela. */
  char newline = '\n';
  /* Le premier argument est le nom du fichier de sortie. */
  char* filename = argv[1];
  /* Ignore les deux premiers éléments de la liste d'arguments. L'élément
     à l'indice 0 est le nom du programme et celui à l'indice 1 est
     le nom du fichier de sortie. */
  argc -= 2;
  argv += 2;

  /* Alloue un tableau d'éléments iovec. Nous aurons besoin de deux élements pour
     chaque argument, un pour le texte proprement dit et un pour
     la nouvelle ligne. */
  vec = (struct iovec*) malloc (2 * argc * sizeof (struct iovec));

  /* Boucle sur la liste d'arguments afin de construire les éléments iovec. */
  vec_next = vec;
  for (i = 0; i < argc; ++i) {
    /* Le premier élément est le texte de l'argument. */
    vec_next->iov_base = argv[i];
    vec_next->iov_len = strlen (argv[i]);
    ++vec_next;
    /* Le second élement est un caractère de nouvelle ligne. Il est possible
       de faire pointer plusieurs éléments du tableau de struct iovec vers
       la même région mémoire. */
    vec_next->iov_base = &newline;
    vec_next->iov_len = 1;
    ++vec_next;
  }

  /* Écrit les arguments dans le fichier. */
  fd = open (filename, O_WRONLY | O_CREAT);
  writev (fd, vec, 2 * argc);
  close (fd);

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

Voici un exemple d'exécution de `write-args`.

```
% ./write-args fichiersortie "premier arg" "deuxième arg" "troisème arg"
% cat outputfile
premier arg
deuxième arg
troisième arg
```

Linux propose une fonction équivalente pour la lecture, `readv` qui permet de charger des données dans des zones mémoire non-contigües en une seule fois. Comme pour `writev`, un tableau d'éléments `struct iovec` indique les zones mémoire dans lesquelles charger les données à partir du descripteur de fichier.

B.4 Lien avec les fonctions d'E/S standards du C

Nous avons évoqué le fait que les fonctions d'E/S standards du C sont implémentées comme une surcouche de ces fonctions d'E/S de bas niveau. Parfois, cependant, il peut être pratique d'utiliser les fonctions de la bibliothèque standard avec des descripteurs de fichiers ou les fonctions de bas niveau sur un flux `FILE*`. GNU/Linux autorise ces deux pratiques.

Si vous avez ouvert un fichier en utilisant `fopen`, vous pouvez obtenir le descripteur de fichier sous-jacent par le biais de la fonction `fileno`. Elle prend un paramètre `FILE*` et renvoie le descripteur de fichier. Par exemple, pour ouvrir un fichier avec l'appel standard `fopen` mais y écrire avec `writev`, vous pourriez utiliser une séquencence telle que :

```
FILE* flux = fopen (nomfichier, "w");
int descripteur = fileno (flux);
writev (descripteur, tableau, taille_tableau);
```

Notez que `flux` et `descripteur` correspondent tous deux au même fichier. Si vous appelez `fclose` comme ceci, vous ne pourrez plus écrire dans `descripteur` :

```
fclose (flux);
```

De même, si vous appelez `close`, vous ne pourrez plus écrire dans `flux` :

```
close (descripteur);
```

Pour effectuer l'opération inverse, obtenir un flux à partir d'un descripteur, utilisez la fonction `fdopen`. Elle produit un pointeur `FILE*` correspondant à un descripteur de fichier. La fonction `fdopen` prend en paramètres un descripteur de fichier et une chaîne correspondant au mode dans lequel le flux doit être ouvert. La syntaxe de l'argument de mode est la même que pour le second argument de `fopen` et il doit être compatbile avec le descripteur de fichier. Par exemple, passez la chaine de mode `r` pour un descripteur de fichier en lecture ou `w` pour un descripteur de fichier en écriture. Comme pour `fileno`, le flux et le descripteur de fichier font référence au même fichier, ainsi, si vous en fermez l'un des deux, vous ne pouvez plus utiliser l'autre.

B.5 Autres opérations sur les fichiers

Un petit nombre d'opérations supplémentaires sur les fichiers et répertoires peut s'avérer utile :

- `getcwd` renvoie le répertoire de travail courant. Il prend deux argument, un tampon de `char` et sa longueur. Il copie le chemin du répertoire de travail courant dans le tampon ;
- `chdir` change le répertoire de travail courant pour qu'il corresponde au chemin passé en paramètre ;
- `mkdir` crée un nouveau répertoire. Son premier argument est le chemin de celui-ci, le second les permissions à y appliquer. L'interprétation des permissions est la même que celle du troisième argument de `open`, elles sont affectées par l'umask du processus ;
- `rmdir` supprime le répertoire dont le chemin est passé en paramètre ;
- `unlink` supprime le fichier dont le chemin est passé en paramètre. Cet appel peut également être utilisé pour supprimer d'autres objets du système de fichiers, comme les canaux nommés (référez-vous à la Section 5.4.5, « FIFO ») ou les périphériques (voir le Chapitre 6) ; En fait, `unlink` ne supprime pas forcément le contenu du fichier. Comme son nom l'indique, il rompt le lien entre le fichier et le répertoire qui le contient. Le fichier n'apparait plus dans le listing du répertoire mais si un processus détient un descripteur de fichier ouvert sur ce fichier, le contenu n'est pas effacé du disque. Cela n'arrive que lorsque qu'aucun processus ne détient de descripteur de fichier ouvert. Ainsi, si un priocessus ouvre un fichier pour y écrire ou y lire et qu'un second processus supprime le fichier avec `unlink` et crée un nouveau fichier avec le même nom, le premier processus "voit" l'ancien contenu du fichier et non pas le nouveau (à moins qu'il ne ferme le fichier et le ré-ouvre) ;
- `rename` renomme ou déplace un fichier. Le premier argument correspond au chemin courant vers le fichier, le second au nouveau chemin. Si les deux chemins sont dans des répertoires différents, `rename` déplace le fichier, si tant est que le nouveau chemin est sur le même système de fichiers que l'ancien. Vous pouvez utiliser `rename` pour déplacer des répertoires ou d'autres objets du système de fichiers.

B.6 Lire le contenu d'un répertoire

GNU/Linux dispose de fonctions pour lire le contenu des répertoires. Bien qu'elles ne soient pas directement liées aux fonctions de bas niveau décrites dans cet appendice, nous les présentons car elles sont souvent utiles.

Pour lire le contenu d'un répertoire, les étapes suivantes sont nécessaires :

1. Appelez `opendir` en lui passant le chemin du répertoire que vous souhaitez explorer. `opendir` renvoie un descripteur `DIR*`, dont vous aurez besoin pour accéder au contenu du répertoire. Si une erreur survient, l'appel renvoie `NULL` ;
2. Appelez `readdir` en lui passant le descripteur `DIR*` que vous a renvoyé `opendir`. À chaque appel, `readdir` renvoie un pointeur vers une instance de `struct dirent` correspondant à l'entrée suivante dans le répertoire. Lorsque vous atteignez la fin du contenu du répertoire, `readdir` renvoie `NULL`.

   La `struct dirent` que vous optenez *via* `readdir` dispose d'un champ `d_name` qui contient le nom de l'entrée.
3. Appelez `closedir` en lui passant le descripteur `DIR*` à la fin du parcours.

Incluez `<sys/types.h>` et `<dirent.h>` si vous utilisez ces fonctions dans votre programme.

Notez que si vous avez besoin de trier les entrées dans un ordre particulier, c'est à vous de le faire.

Le programme du Listing B.8 affiche le contenu d'un répertoire. Celui-ci peut être spécifié sur la ligne de commande mais, si ce n'est pas le cas, le répertoire courant est utilisé. Pour chaque entrée, son type et son chemin est affiché. La fonction `get_file_type` utilise `lstat` pour déterminer le type d'une entrée.

Listing B.8 – (*listdir.c*) – **Affiche le Contenu d'un Répertoire**

```
#include <assert.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
/* Renvoie une chaine qui décrit le type du fichier PATH. */
const char* get_file_type (const char* path)
{
  struct stat st;
  lstat (path, &st);
  if (S_ISLNK (st.st_mode))
    return "lien symbolique";
  else if (S_ISDIR (st.st_mode))
    return "répertoire";
  else if (S_ISCHR (st.st_mode))
    return "périphérique caractère";
  else if (S_ISBLK (st.st_mode))
    return "périphérique bloc";
  else if (S_ISFIFO (st.st_mode))
    return "fifo";
  else if (S_ISSOCK (st.st_mode))
    return "socket";
  else if (S_ISREG (st.st_mode))
    return "fichier ordinaire";
  else
    /* Impossible. Toute entrée doit être de l'un des type ci-dessus. */
    assert (0);
}
int main (int argc, char* argv[])
{
  char* dir_path;
  DIR* dir;
  struct dirent* entry;
  char entry_path[PATH_MAX + 1];
  size_t path_len;
  if (argc >= 2)
    /* Utilise le répertoire spécifié sur la ligne de commande s'il y a lieu. */
    dir_path = argv[1];
  else
    /* Sinon, utilise le répertoire courant. */
    dir_path = ".";
  /* Copie le chemin du répertoire dans entry_path. */
  strncpy (entry_path, dir_path, sizeof (entry_path));
  path_len = strlen (dir_path);
  /* Si le répertoire ne se termine pas par un slash, l'ajoute. */
  if (entry_path[path_len - 1] != '/') {
    entry_path[path_len] = '/';
    entry_path[path_len + 1] = '\0';
    ++path_len;
```

```
52    }
53    /* Démarre l'affichage du contenu du répertoire. */
54    dir = opendir (dir_path);
55    /* Boucle sur les entrées du répertoire. */
56    while ((entry = readdir (dir)) != NULL) {
57      const char* type;
58      /* Construit le chemin complet en concaténant le chemin du répertoire
59         et le nom de l'entrée. */
60      strncpy (entry_path + path_len, entry->d_name,
61               sizeof (entry_path) - path_len);
62      /* Détermine le type de l'entrée. */
63      type = get_file_type (entry_path);
64      /* Affiche le type et l'emplacement de l'entrée. */
65      printf ("%-18s: %s\n", type, entry_path);
66    }
67    /* Fini. */
68    closedir (dir);
69    return 0;
70  }
```

Voici les premières lignes affichées lors de l'affichage du contenu de `/dev` (elles peuvent être différentes chez vous) :

```
% ./listdir /dev
directory        : /dev/.
directory        : /dev/..
socket           : /dev/log
character device : /dev/null
regular file     : /dev/MAKEDEV
fifo             : /dev/initctl
character device : /dev/agpgart
...
```

Pour vérifier les résultats, vous pouvez utiliser la commande `ls` sur le même répertoire. Passez l'indicateur `-U` pour demander à `ls` de ne pas trier les entrées et passez l'indicateur `-a` pour inclure le répertoire courant (`.`) et le répertoire parent (`..`).

```
% ls -lUa /dev
total 124
drwxr-xr-x    7 root root   36864 Feb 1 15:14 .
drwxr-xr-x   22 root root    4096 Oct 11 16:39 ..
srw-rw-rw-    1 root root       0 Dec 18 01:31 log
crw-rw-rw-    1 root root  1,   3 May 5 1998 null
-rwxr-xr-x    1 root root   26689 Mar 2 2000 MAKEDEV
prw-------    1 root root       0 Dec 11 18:37 initctl
crw-rw-r--    1 root root 10, 175 Feb 3 2000 agpgart
...
```

Le premier caractère de chaque ligne affichée par `ls` indique le type de l'entrée.

Annexe C

Tableau des signaux

Le Tableau C.1 présente une partie des signaux Linux que vous avez le plus de chances de rencontrer. Notez que certains signaux peuvent être interprétés de différentes façons selon l'endroit où ils surviennent.

Les noms des signaux présentés ici sont définis sous forme de macros préprocesseur. Pour les utiliser dans votre programme utilisez `<signal.h>`. Les définitions proprement dites se trouvent dans le fichier `/usr/include/sys/signum.h`, qui est lui-même inclus par `<signal.h>`.

Pour une liste complète des signaux Linux, accompagnés d'une courte description et du comportement associé par défaut à leur réception, consultez la page de manuel de `signal` de la Section 7 par le biais de la commande suivante :

```
% man 7 signal
```

Tab. C.1 – Signaux Linux

Nom	Description
SIGHUP	Linux envoie ce signal à un processus lorsqu'il est déconnecté d'un terminal. La plupart des programmes Linux utilisent SIGHUP pour tout autre chose : indiquer à un programme en cours d'exécution qu'il doit recharger ses fichiers de configuration.
SIGINT	Ce signal est envoyé lorsque l'utilisateur tente d'arrêter le programme en utilisant Ctrl+C.
SIGILL	Reçu par un processus lorsqu'il tente d'exécuter une instruction illégale, cela peut indiquer une corruption de la pile du programme.
SIGABRT	Reçu lors d'un appel à abort.
SIGFPE	Reçu lorsque le processus exécute une instruction en virgule flottante invalide. Selon la manière dont le CPU est configuré, une opération en virgule flottante peut renvoyer une valeur non-numérique spéciale comme inf (infini) ou NaN (not a number) au lieu de lever un SIGFPE
SIGKILL	Ce signal termine un processus immédiatement et ne peut pas être intercepté.
SIGUSR1	Réservé à l'usage par l'application.
SIGUSR2	Réservé à l'usage par l'application.
SIGSEGV	Le programme a effectué un accès invalide à la mémoire. Il peut s'agir d'un accès à une adresse invalide dans l'espace d'adressage du processus ou l'accès peut être interdit par les permissions appliquées à la mémoire. Libérer un "pointeur sauvage" peut provoquer un SIGSEGV.
SIGPIPE	Le programme a tenté d'accéder à un flux de données invalide, comme une connexion qui a été fermée par l'autre protagoniste.
SIGALRM	L'appel système alarm programme l'envoi de ce signal. Consultez la Section 8.13, « *setitimer*: créer des temporisateurs » dans le Chapitre 8, « Appels système Linux », pour plus d'informations sur setitimer, une version générique de alarm.
SIGTERM	Ce signal demande à un processus de se terminer. Il s'agit du signal envoyé par défaut par la commande kill.
SIGCHLD	Linux envoie ce signal à un processus lorsqu'un processus fils se termine. Consultez la Section 3.4.4, « Libérer les ressources des fils de façon asynchrone » du Chapitre 3, « Processus ».
SIGXCPU	Linux envoie ce signal à un processus lorsqu'il dépasse la limite de temps CPU qui lui a été allouée. Consultez la Section 8.5, « *getrlimit* et *setrlimit*: limites de ressources », du Chapitre 8 pour plus d'informations sur les limites de temps CPU.
SIGVTALRM	La fonction setitimer programme l'envoi de ce signal. Consultez la Section 8.13, « *setitimer*: créer des temporisateurs ».

Annexe D

Ressources en ligne

Cette annexe dresse une liste de ressources présentes sur Internet permettant d'en savoir plus sur la programmation sur le système GNU/Linux.

D.1 Informations générales

- `http://www.advancedlinuxprogramming.com` est le site de ce livre. Vous pourrez y télécharger l'intégrale de ce livre et les sources des programmes, trouver des liens vers d'autres ressources et obtenir plus d'informations sur la programmation GNU/Linux. Ces informations sont également disponibles sur `http://www.newriders.com` ;
- `http://www.tldp.org` héberge le Linux Documentation Project. Ce site regroupe une grande variété de documents, des listes de FAQ, HOWTO et autres concernant les systèmes et logiciels GNU/Linux ;
- `http://www.advancedlinuxprogramming-fr.org` est le site de la version française de ce livre, vous y trouverez un wiki reprenant son contenu amélioré par les lecteurs.

D.2 Informations sur les logiciels GNU/Linux

- `http://www.gnu.org` est le site officiel du projet GNU. À partir de ce site, vous pouvez télécharger une quantité impressionnante d'applications libres sophistiquées. On y trouve entre autre la bibliothèque C GNU, qui fait partie de tout système GNU/Linux et une part importante des fonctions présentées dans ce livre. Le site du Projet GNU fournit également des informations sur la façon de contribuer au développement du système GNU/Linux en écrivant du code ou de la documentation, en utilisant des logiciels libres et en faisant passer le message des logiciels libres ;
- `http://www.kernel.org` est le site principal de distribution du code source du noyau Linux. Pour les questions les plus tordues et les plus techniques sur le fonctionnement de Linux, il s'agit d'une mine. Consultez également le répertoire `Documentation` pour plus d'explications sur le fonctionnement interne du noyau ;

- `http://www.linuxhq.com` distribue également des sources patch et informations sur le noyau Linux ;
- `http://gcc.gnu.org` hébere le projet de la GNU Compiler Collection (GCC). GCC est le principal compilateur utilisé sur les système GNU/Linux et inclut des compilateurs C, C++, l'Objective C, Java, Chill, Fortran, *etc.* ;
- `http://www.gnome.org` et `http://www.kde.org` hébergent les deux environnements de bureau les plus populaires sous Linux, Gnome et KDE. Si vous prévoyez d'écrire une application avec une interface graphique, vous devriez commencer par vous familiariser avec l'un des deux (ou les deux).

D.3 Autres sites

- `http://developer.intel.com` fournit des informations sur l'architecture des processeurs Intel, y compris pour les x86 (IA32). Si vous développez pour Linux sur x86 et que vous utilisez des instructions assembleur en ligne, les manuels techniques qui y sont disponibles peuvent vous être très utiles ;
- `http://developer.amd.com` fournit le même genre d'information sur les processeurs AMD et leurs fonctionnalités spécifiques ;
- `http://freshmeat.net` recense des projets open source, généralement pour GNU/Linux. Ce site est l'un des meilleurs endroits pour se tenir au courant des nouvelles version de logiciels GNU/Linux, depuis le système de base à des applications plus obscures et spécialisées ;
- `http://www.linuxsecurity.com` donne des informations, des techniques et recense des liens concernant des logiciels liés à la sécurité sous GNU/Linux. Ce site peut être intéressant pour les utilisateurs, les administrateurs système et les développeurs.

Annexe E

Licence Open Publication version 1.0

Cette traduction de l'Open Publication License n'est pas officielle. Seule la version (en anglais) disponible sur `http://www.opencontent.org/openpub/` fait foi. L'auteur ne pourra être tenu responsable d'erreurs de traduction.

I. Conditions applicables aux versions modifiées ou non

Les travaux sous Open Publication License peuvent être reproduits et distribués, entièrement ou partiellement, sous quelle que forme que ce soit, physique ou électronique, à la condition que les termes de cette licence soient respectés et que cette licence ou une référence à celle-ci (mentionnant les options éxecées par les auteurs et/ou l'éditeur) soit attachée à la reproduction.

La forme de l'incorporation doit être la suivante :

> Copyright © <année> <auteur ou mandataire>. Ce document ne peut être distribué que dans le respect des termes et conditions définies par l'Open Poublication License, vX.Y ou ultérieure (la dernière version est disponible sur `http://www.opencontent.org/openpub/`).

La référence doit être immédiatement suivie de la mention des options exercées par les auteurs ou l'éditeur du document (voir Section VI., « Options possibles »).

La redistribution de documents sous Open Publication License est autorisée.

Toute publication sous une forme classique (papier) impose la citation de l'éditeur et de l'auteur original. Les noms de l'éditeur et des auteurs doivent apparaître sur toutes les couvertures du livre. Sur les couvertures, le nom de l'éditeur original devrait être aussi grand que le titre de l'ouvrage et adopter une forme possessive vis à vis du titre.

II. Copyright

Le copyright d'un travail sous Open Publication License est détenu par ses auteurs ou mandataires.

III. Portée de cette licence

Les termes de cette licence s'appeliquent à tous les travaux sous licence Open Publication sauf mention contraire explicite dans le document.

La mise en commun de travaux sous licence Open Publication ou d'un document sous licence Open Publication avec des travaux sous une autre licence sur le même support ne provoque pas l'application de la licence Open Publication aux autres travaux. Les documents résultants doivent contenir une note stipulant l'inclusion de documents sous licence Open Publication et la mention de copyright adéquate.

Divisibilité . Si une partie de cette licence s'avère inapplicable en raison des lois en vigueur, le reste de la licence reste en vigueur ;

Absence de Garantie . Les travaux sous licence Open Publication sont fournis "en l'état" sans aucune garantie d'aucune sort, explicite ou implicite, y compris, mais ne se limitant pas à, les garanties de commercialisation ou d'adaptation à un but particulier ou une garantie de non violation de copyright.

IV. Conditions applicables aux travaux modifiés

Toute version modifiée des documents couverts par cette licence, y compris les traduction, les anthologies, les compilations et les reproductions partielles doivent répondre aux conditions suivantes :

1. La version modifiée doit être indiquée comme telle ;
2. La personne effectuant les modifications doit être identifiée et les modifications datées ;
3. Des remerciements vis à vis des auteurs auteurs et de l'éditeur originaux, s'il y a lieu, doivent être maintenus en accord avec les pratiques habituelles en matière de citations académiques ;
4. L'emplacement du document original doit être clairement identifié ;
5. Les noms des auteurs originaux ne doit pas être utilisé pour cautionner le document modifié sans leur permission.

V. Bonnes pratiques

En plus des contraintes imposées par cette licence, il est demandé et fortement recommandé aux ditributeurs :

1. Si vous distribuez des travaux sous licence Open Publication sous forme papier ou numérique, prévenez les auteurs de vos intention au moins trente jours avant le gel de la maquette, afin de donner aux auteur le temps de fournir des documents à jour. Cette notification doit mentionner, s'il y a lieu, les modifications apportées au document ;
2. Toute modification substancielle (y compris les suppressions) doit être clairement signalée dans le document ou décrite dans une note attachée au document ;

3. Enfin, bien que cela ne soit pas imposé par cette licence, il est de bon ton d'offrir une copie gratuite de toute version papier ou numérique d'un travail sous licence Open Publication à ses auteurs.

VI. Options possibles

Les auteurs ou l'éditeur d'un document sous licence Open Publication peut choisir d'apporter des modifications à la licence sous forme d'options signalées à la suite de la référence ou de la copie de cette licence. Ces options font partie de la licence et doivent être incluses avec celle-ci (ou avec ses références) dans les travaux dérivés.

A. Interdire la distribution de version modifiées de façon substancielle sans l'autorisation explicite des auteurs. Le terme "modifications substancielles" est définie comme étant une modification du contenu sémantique du document et exclut les modifications ne touchant que le formatage ou les corrections typographiques.

   Pour cela, ajoutez la mention "La distribution de versions de ce document modifiées de manière substancielle est interdite sans l'autorisation explicite du détenteur du copyright" à la suite de la référence à la licence ou de sa copie ;

B. Interdire toute distribution des travaux, dérivés ou non, dans leur intégralité ou non au format papier dans un but commercial est interdit sauf autorisation préalable du détenteur du copyright.

   Pour appliquer cette option, ajoutez la mention "La distribution de ce document ou de travaux dérivés s'y rapportant sous forme papier est interdite sauf autorisation préalable du détenteur du copyright" à la suite de la référence à la licence ou de sa copie.

VII. Annexe à la licence Open Publication

(Cette section n'est pas considéré comme faisant partie de la licence).

Les sources de travaux sous licence Open Publication sont disponibles sur la page d'accueil d'Open Publication sur `http://works.opencontent.org/`.

Les auteurs voulant proposer leurs propres licences appliquées à des travaux Open Publication peuvent le faire à la condition que les termes ne soient pas plus restrictifs que ceux de la licence Open Publication.

Si vous avez des questions concernant la Licence Open Publication, veuillez contacter David Wiley ou la Liste des Auteurs Open Publication par email à l'adresse `opal@opencontent.org`.

Pour souscrire à cette liste, envoyez un email contenant le mot "subscribe" à l'adresse `opal-request@opencontent.org`.

Pour envoyer un email à la liste, envoyez un email à `opal@opencontent.org` ou répondez simplement à un email.

Pour résilier votre abonnement à cette liste, envoyez un email contenant le mot "unsubscribe" à l'adresse `opal-request@opencontent.org`.

Annexe F

Licence Publique Générale GNU

La version originale de cette licence est disponible sur `http://www.gnu.org/copyleft/gpl.html`, cette traduction est disponible sur `http://fsffrance.org/gpl/gpl-fr.fr.html`.

> Ceci est une traduction non officielle de la GNU General Public License en français. Elle n'a pas été publiée par la Free Software Foundation, et ne détermine pas les termes de distribution pour les logiciels qui utilisent la GNU GPL, seul le texte anglais original de la GNU GPL déterminent ces termes. Cependant, nous espérons que cette traduction aidera les francophones à mieux comprendre la GNU GPL.

Préambule

Les licences de la plupart des logiciels sont conçues pour vous enlever toute liberté de les partager et de les modifier. A contrario, la Licence Publique Générale est destinée à garantir votre liberté de partager et de modifier les logiciels libres, et à assurer que ces logiciels soient libres pour tous leurs utilisateurs. La présente Licence Publique Générale s'applique à la plupart des logiciels de la Free Software Foundation, ainsi qu'à tout autre programme pour lequel ses auteurs s'engagent à l'utiliser. (Certains autres logiciels de la Free Software Foundation sont couverts par la GNU Lesser General Public License à la place.) Vous pouvez aussi l'appliquer aux programmes qui sont les vôtres.

Quand nous parlons de logiciels libres, nous parlons de liberté, non de prix. Nos licences publiques générales sont conçues pour vous donner l'assurance d'être libres de distribuer des copies des logiciels libres (et de facturer ce service, si vous le souhaitez), de recevoir le code source ou de pouvoir l'obtenir si vous le souhaitez, de pouvoir modifier les logiciels ou en utiliser des éléments dans de nouveaux programmes libres et de savoir que vous pouvez le faire.

Pour protéger vos droits, il nous est nécessaire d'imposer des limitations qui interdisent à quiconque de vous refuser ces droits ou de vous demander d'y renoncer. Certaines responsabilités vous incombent en raison de ces limitations si vous distribuez des copies de ces logiciels, ou si vous les modifiez.

Par exemple, si vous distribuez des copies d'un tel programme, à titre gratuit ou contre une rémunération, vous devez accorder aux destinataires tous les droits dont vous disposez. Vous

devez vous assurer qu'eux aussi reçoivent ou puissent disposer du code source. Et vous devez leur montrer les présentes conditions afin qu'ils aient connaissance de leurs droits.

Nous protégeons vos droits en deux étapes : (1) nous sommes titulaires des droits d'auteur du logiciel, et (2) nous vous délivrons cette licence, qui vous donne l'autorisation légale de copier, distribuer et/ou modifier le logiciel.

En outre, pour la protection de chaque auteur ainsi que la nôtre, nous voulons nous assurer que chacun comprenne que ce logiciel libre ne fait l'objet d'aucune garantie. Si le logiciel est modifié par quelqu'un d'autre puis transmis à des tiers, nous voulons que les destinataires soient mis au courant que ce qu'ils ont reçu n'est pas le logiciel d'origine, de sorte que tout problème introduit par d'autres ne puisse entacher la réputation de l'auteur originel.

En définitive, un programme libre restera à la merci des brevets de logiciels. Nous souhaitons éviter le risque que les redistributeurs d'un programme libre fassent des demandes individuelles de licence de brevet, ceci ayant pour effet de rendre le programme propriétaire.

Pour éviter cela, nous établissons clairement que toute licence de brevet doit être concédée de façon à ce que l'usage en soit libre pour tous ou bien qu'aucune licence ne soit concédée.

Les termes exacts et les conditions de copie, distribution et modification sont les suivants :

Conditions de copie, distribution et modification de la Licence Publique Générale GNU.

0. La présente Licence s'applique à tout programme ou tout autre ouvrage contenant un avis, apposé par le titulaire des droits d'auteur, stipulant qu'il peut être distribué au titre des conditions de la présente Licence Publique Générale. Ci-après, le "Programme" désigne l'un quelconque de ces programmes ou ouvrages, et un "ouvrage fondé sur le Programme" désigne soit le Programme, soit un ouvrage qui en dérive au titre des lois sur le droit d'auteur : en d'autres termes, un ouvrage contenant le Programme ou une partie de ce dernier, soit à l'identique, soit avec des modifications et/ou traduit dans un autre langage. (Ci-après, le terme "modification" implique, sans s'y réduire, le terme traduction) Chaque concessionaire sera désigné par "vous".

Les activités autres que la copie, la distribution et la modification ne sont pas couvertes par la présente Licence ; elles sont hors de son champ d'application. L'opération consistant à exécuter le Programme n'est soumise à aucune limitation et les sorties du programme ne sont couvertes que si leur contenu constitue un ouvrage fondé sur le Programme (indépendamment du fait qu'il ait été réalisé par l'exécution du Programme). La validité de ce qui précède dépend de ce que fait le Programme.

1. Vous pouvez copier et distribuer des copies à l'identique du code source du Programme tel que vous l'avez reçu, sur n'importe quel support, du moment que vous apposiez sur chaque copie, de manière ad hoc et parfaitement visible, l'avis de droit d'auteur adéquat et une exonération de garantie ; que vous gardiez intacts tous les avis faisant référence à la présente Licence et à l'absence de toute garantie ; et que vous fournissiez à tout destinataire du Programme autre que vous-même un exemplaire de la présente Licence en même temps que le Programme.

Vous pouvez faire payer l'acte physique de transmission d'une copie, et vous pouvez, à votre discrétion, proposer une garantie contre rémunération.

2. Vous pouvez modifier votre copie ou des copies du Programme ou n'importe quelle partie de celui-ci, créant ainsi un ouvrage fondé sur le Programme, et copier et distribuer de telles modifications ou ouvrage selon les termes de l'Article 1 ci-dessus, à condition de vous conformer également à chacune des obligations suivantes :

- a) Vous devez munir les fichiers modifiés d'avis bien visibles stipulants que vous avez modifié ces fichiers, ainsi que la date de chaque modification ;
- b) Vous devez prendre les dispositions nécessaires pour que tout ouvrage que vous distribuez ou publiez, et qui, en totalité ou en partie, contient ou est fondé sur le Programme - ou une partie quelconque de ce dernier - soit concédé comme un tout, à titre gratuit, à n'importe quel tiers, au titre des conditions de la présente Licence.
- c) Si le programme modifié lit habituellement des instructions de façon interactive lorsqu'on l'exécute, vous devez, quand il commence son exécution pour ladite utilisation interactive de la manière la plus usuelle, faire en sorte qu'il imprime ou affiche une annonce comprenant un avis de droit d'auteur ad hoc, et un avis stipulant qu'il n'y a pas de garantie (ou bien indiquant que c'est vous qui fournissez la garantie), et que les utilisateurs peuvent redistribuer le programme en respectant les présentes obligations, et expliquant à l'utilisateur comment voir une copie de la présente Licence. (Exception : si le Programme est lui-même interactif mais n'imprime pas habituellement une telle annonce, votre ouvrage fondé sur le Programme n'est pas obligé d'imprimer une annonce).

Ces obligations s'appliquent à l'ouvrage modifié pris comme un tout. Si des éléments identifiables de cet ouvrage ne sont pas fondés sur le Programme et peuvent raisonnablement être considérés comme des ouvrages indépendants distincts en eux mêmes, alors la présente Licence et ses conditions ne s'appliquent pas à ces éléments lorsque vous les distribuez en tant qu'ouvrages distincts. Mais lorsque vous distribuez ces mêmes éléments comme partie d'un tout, lequel constitue un ouvrage fondé sur le Programme, la distribution de ce tout doit être soumise aux conditions de la présente Licence, et les autorisations qu'elle octroie aux autres concessionnaires s'étendent à l'ensemble de l'ouvrage et par conséquent à chaque et toute partie indifférement de qui l'a écrite.

Par conséquent, l'objet du présent article n'est pas de revendiquer des droits ou de contester vos droits sur un ouvrage entièrement écrit par vous ; son objet est plutôt d'exercer le droit de contrôler la distribution d'ouvrages dérivés ou d'ouvrages collectifs fondés sur le Programme.

De plus, la simple proximité du Programme avec un autre ouvrage qui n'est pas fondé sur le Programme (ou un ouvrage fondé sur le Programme) sur une partition d'un espace de stockage ou un support de distribution ne place pas cet autre ouvrage dans le champ d'application de la présente Licence.

3. Vous pouvez copier et distribuer le Programme (ou un ouvrage fondé sur lui, selon l'Article 2) sous forme de code objet ou d'exécutable, selon les termes des Articles 1 et 2 ci-dessus, à condition que vous accomplissiez l'un des points suivants :

- a) L'accompagner de l'intégralité du code source correspondant, sous une forme lisible par un ordinateur, lequel doit être distribué au titre des termes des Articles 1 et 2 ci-dessus, sur un support habituellement utilisé pour l'échange de logiciels ; ou,
- b) L'accompagner d'une proposition écrite, valable pendant au moins trois ans, de fournir à tout tiers, à un tarif qui ne soit pas supérieur à ce que vous coûte l'acte physique de

réaliser une distribution source, une copie intégrale du code source correspondant sous une forme lisible par un ordinateur, qui sera distribuée au titre des termes des Articles 1 et 2 ci-dessus, sur un support habituellement utilisé pour l'échange de logiciels ; ou,

- c) L'accompagner des informations reçues par vous concernant la proposition de distribution du code source correspondant. (Cette solution n'est autorisée que dans le cas d'une distribution non commerciale et seulement si vous avez reçu le programme sous forme de code objet ou d'exécutable accompagné d'une telle proposition - en conformité avec le sous-Article b ci-dessus.)

Le code source d'un ouvrage désigne la forme favorite pour travailler à des modifications de cet ouvrage. Pour un ouvrage exécutable, le code source intégral désigne la totalité du code source de la totalité des modules qu'il contient, ainsi que les éventuels fichiers de définition des interfaces qui y sont associés, ainsi que les scripts utilisés pour contrôler la compilation et l'installation de l'exécutable. Cependant, par exception spéciale, le code source distribué n'est pas censé inclure quoi que ce soit de normalement distribué (que ce soit sous forme source ou binaire) avec les composants principaux (compilateur, noyau, et autre) du système d'exploitation sur lequel l'exécutable tourne, à moins que ce composant lui-même n'accompagne l'exécutable.

Si distribuer un exécutable ou un code objet consiste à offrir un accès permettant leur copie depuis un endroit particulier, alors l'offre d'un accès équivalent pour copier le code source depuis le même endroit compte comme une distribution du code source - même si les tiers ne sont pas contraints de copier le source en même temps que le code objet.

4. Vous ne pouvez copier, modifier, concéder en sous-licence, ou distribuer le Programme, sauf tel qu'expressément prévu par la présente Licence. Toute tentative de copier, modifier, concéder en sous-licence, ou distribuer le Programme d'une autre manière est réputée non valable, et met immédiatement fin à vos droits au titre de la présente Licence. Toutefois, les tiers ayant reçu de vous des copies, ou des droits, au titre de la présente Licence ne verront pas leurs autorisations résiliées aussi longtemps que ledits tiers se conforment pleinement à elle.

5. Vous n'êtes pas obligé d'accepter la présente Licence étant donné que vous ne l'avez pas signée. Cependant, rien d'autre ne vous accorde l'autorisation de modifier ou distribuer le Programme ou les ouvrages fondés sur lui. Ces actions sont interdites par la loi si vous n'acceptez pas la présente Licence. En conséquence, en modifiant ou distribuant le Programme (ou un ouvrage quelconque fondé sur le Programme), vous signifiez votre acceptation de la présente Licence en le faisant, et de toutes ses conditions concernant la copie, la distribution ou la modification du Programme ou d'ouvrages fondés sur lui.

6. Chaque fois que vous redistribuez le Programme (ou n'importe quel ouvrage fondé sur le Programme), une licence est automatiquement concédée au destinataire par le concédant originel de la licence, l'autorisant à copier, distribuer ou modifier le Programme, sous réserve des présentes conditions. Vous ne pouvez imposer une quelconque limitation supplémentaire à l'exercice des droits octroyés au titre des présentes par le destinataire. Vous n'avez pas la responsabilité d'imposer le respect de la présente Licence à des tiers.

7. Si, conséquement à une décision de justice ou l'allégation d'une transgression de brevet ou pour toute autre raison (non limitée à un probleme de brevet), des obligations vous sont imposées (que ce soit par jugement, conciliation ou autre) qui contredisent les conditions de la présente Licence, elles ne vous excusent pas des conditions de la présente Licence. Si vous ne pouvez

distribuer de manière à satisfaire simultanément vos obligations au titre de la présente Licence et toute autre obligation pertinente, alors il en découle que vous ne pouvez pas du tout distribuer le Programme. Par exemple, si une licence de brevet ne permettait pas une redistribution sans redevance du Programme par tous ceux qui reçoivent une copie directement ou indirectement par votre intermédiaire, alors la seule façon pour vous de satisfaire à la fois à la licence du brevet et à la présente Licence serait de vous abstenir totalement de toute distribution du Programme.

Si une partie quelconque de cet article est tenue pour nulle ou inopposable dans une circonstance particulière quelconque, l'intention est que le reste de l'article s'applique. La totalité de la section s'appliquera dans toutes les autres circonstances.

Cet article n'a pas pour but de vous induire à transgresser un quelconque brevet ou d'autres revendications à un droit de propriété ou à contester la validité de la moindre de ces revendications ; cet article a pour seul objectif de protéger l'intégrité du système de distribution du logiciel libre, qui est mis en oeuvre par la pratique des licenses publiques. De nombreuses personnes ont fait de généreuses contributions au large spectre de logiciels distribués par ce système en se fiant à l'application cohérente de ce système ; il appartient à chaque auteur/donateur de décider si il ou elle veut distribuer du logiciel par l'intermédiaire d'un quelconque autre système et un concessionaire ne peut imposer ce choix.

Cet article a pour but de rendre totalement limpide ce que l'on pense être une conséquence du reste de la présente Licence.

8. Si la distribution et/ou l'utilisation du Programme est limitée dans certains pays que ce soit par des brevets ou par des interfaces soumises au droit d'auteur, le titulaire originel des droits d'auteur qui décide de couvrir le Programme par la présente Licence peut ajouter une limitation géographique de distribution explicite qui exclue ces pays afin que la distribution soit permise seulement dans ou entre les pays qui ne sont pas ainsi exclus. Dans ce cas, la présente Licence incorpore la limitation comme si elle était écrite dans le corps de la présente Licence.

9. La Free Software Foundation peut, de temps à autre, publier des versions révisées et/ou nouvelles de la Licence Publique Générale. De telles nouvelles versions seront similaires à la présente version dans l'esprit mais pourront différer dans le détail pour prendre en compte de nouvelles problématiques ou inquiétudes.

Chaque version possède un numéro de version la distinguant. Si le Programme précise le numéro de version de la présente Licence qui s'y applique et "une version ultérieure quelconque", vous avez le choix de suivre les conditions de la présente version ou de toute autre version ultérieure publiée par la Free Software Foundation. Si le Programme ne spécifie aucun numéro de version de la présente Licence, vous pouvez choisir une version quelconque publiée par la Free Software Foundation à quelque moment que ce soit.

10. Si vous souhaitez incorporer des parties du Programme dans d'autres programmes libres dont les conditions de distribution sont différentes, écrivez à l'auteur pour lui en demander l'autorisation. Pour les logiciels dont la Free Software Foundation est titulaire des droits d'auteur, écrivez à la Free Software Foundation ; nous faisons parfois des exceptions dans ce sens. Notre décision sera guidée par le double objectif de préserver le statut libre de tous les dérivés de nos logiciels libres et de promouvoir le partage et la réutilisation des logiciels en général.

ABSENCE DE GARANTIE

11. COMME LA LICENCE DU PROGRAMME EST CONCEDEE A TITRE GRATUIT, AUCUNE GARANTIE NE S'APPLIQUE AU PROGRAMME, DANS LES LIMITES AUTORISEES PAR LA LOI APPLICABLE. SAUF MENTION CONTRAIRE ECRITE, LES TITULAIRES DU DROIT D'AUTEUR ET/OU LES AUTRES PARTIES FOURNISSENT LE PROGRAMME "EN L'ETAT", SANS AUCUNE GARANTIE DE QUELQUE NATURE QUE CE SOIT, EXPRESSE OU IMPLICITE, Y COMPRIS, MAIS SANS Y ETRE LIMITE, LES GARANTIES IMPLICITES DE COMMERCIABILITE ET DE LA CONFORMITE A UNE UTILISATION PARTICULIERE. VOUS ASSUMEZ LA TOTALITE DES RISQUES LIES A LA QUALITE ET AUX PERFORMANCES DU PROGRAMME. SI LE PROGRAMME SE REVELAIT DEFECTUEUX, LE COUT DE L'ENTRETIEN, DES REPARATIONS OU DES CORRECTIONS NECESSAIRES VOUS INCOMBENT INTEGRALEMENT.

12. EN AUCUN CAS, SAUF LORSQUE LA LOI APPLICABLE OU UNE CONVENTION ECRITE L'EXIGE, UN TITULAIRE DE DROIT D'AUTEUR QUEL QU'IL SOIT, OU TOUTE PARTIE QUI POURRAIT MODIFIER ET/OU REDISTRIBUER LE PROGRAMME COMME PERMIS CI-DESSUS, NE POURRAIT ETRE TENU POUR RESPONSABLE A VOTRE EGARD DES DOMMAGES, INCLUANT LES DOMMAGES GENERIQUES, SPECIFIQUES, SECONDAIRES OU CONSECUTIFS, RESULTANT DE L'UTILISATION OU DE L'INCAPACITE D'UTILISER LE PROGRAMME (Y COMPRIS, MAIS SANS Y ETRE LIMITE, LA PERTE DE DONNEES, OU LE FAIT QUE DES DONNEES SOIENT RENDUES IMPRECISES, OU LES PERTES EPROUVEES PAR VOUS OU PAR DES TIERS, OU LE FAIT QUE LE PROGRAMME ECHOUE A INTEROPERER AVEC UN AUTRE PROGRAMME QUEL QU'IL SOIT) MEME SI LE DIT TITULAIRE DU DROIT D'AUTEUR OU LE PARTIE CONCERNEE A ETE AVERTI DE L'EVENTUALITE DE TELS DOMMAGES. FIN DES CONDITIONS

Comment appliquer ces conditions à vos nouveaux programmes

Si vous développez un nouveau programme, et si vous voulez qu'il soit de la plus grande utilité possible pour le public, le meilleur moyen d'y parvenir est d'en faire un logiciel libre que chacun peut redistribuer et modifier au titre des présentes conditions.

Pour ce faire, munissez le programme des avis qui suivent. Le plus sûr est de les ajouter au début de chaque fichier source pour véhiculer le plus efficacement possible l'absence de toute garantie ; chaque fichier devrait aussi contenir au moins la ligne "copyright" et une indication de l'endroit où se trouve l'avis complet.

[Une ligne donnant le nom du programme et une courte idée de ce qu'il fait.]

Copyright (C) [année] [nom de l'auteur]

Ce programme est un logiciel libre ; vous pouvez le redistribuer et/ou le modifier au titre des clauses de la Licence Publique Générale GNU, telle que publiée par la Free Software Foundation ; soit la version 2 de la Licence, ou (à votre discrétion) une version ultérieure quelconque.

Ce programme est distribué dans l'espoir qu'il sera utile, mais SANS AUCUNE GARANTIE ; sans même une garantie implicite de COMMERCIABILITE ou DE CONFORMITE A UNE UTILISATION PARTICULIERE. Voir la Licence Publique Générale GNU pour plus de détails.

Vous devriez avoir reçu un exemplaire de la Licence Publique Générale GNU avec ce programme ; si ce n'est pas le cas, écrivez à la Free Software Foundation Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Ajoutez aussi des informations sur la manière de vous contacter par courrier électronique et courrier postal.

Si le programme est interactif, faites en sorte qu'il affiche un court avis tel que celui-ci lorsqu'il démarre en mode interactif :

Gnomovision version 69, Copyright (C) année nom de l'auteur

Gnomovision n'est accompagné d'ABSOLUMENT AUCUNE GARANTIE ; pour plus de détails tapez "show w". Ceci est un logiciel libre et vous êtes invité à le redistribuer en respectant certaines obligations ; pour plus de détails tapez "show c".

Les commandes hypothétiques "show w" et "show c" sont supposées montrer les parties ad hoc de la Licence Publique Générale. Bien entendu, les instructions que vous utilisez peuvent porter d'autres noms que "show w" et "show c" ; elles peuvent même être des clics de souris ou des éléments d'un menu – ou tout ce qui convient à votre programme.

Vous devriez aussi obtenir de votre employeur (si vous travaillez en tant que développeur) ou de votre école, si c'est le cas, qu'il (ou elle) signe une "renonciation aux droits d'auteur" concernant le programme, si nécessaire. Voici un exemple (changez les noms) :

Yoyodyne, Inc., déclare par la présente renoncer à toute prétention sur les droits d'auteur du programme

"Gnomovision" (qui fait des avances aux compilateurs) écrit par James Hacker.

[signature de Ty Coon], 1er avril 1989

Ty Coon, Président du Vice

La présente Licence Publique Générale n'autorise pas l'incorporation de votre programme dans des programmes propriétaires. Si votre programme est une bibliothèque, vous pouvez considérer plus utile d'autoriser l'édition de liens d'applications propriétaires avec la bibliothèque. Si c'est ce que vous voulez faire, utilisez la GNU Lesser General Public License au lieu de la présente Licence.