I

Programmation UNIX Avancée avec Linux

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.

> **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 !

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).

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[4]. 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.

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

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 :

```
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[5] 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.

---

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

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[6]. 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`[7]. Ce programme calcule l'inverse d'un entier.

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

```
#include <stdio.h>
#include <stdlib.h>
#include "reciprocal.hpp"

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

  i = atoi (argv[1]);
  printf ("L'inverse de %d est %g\n", i, reciprocal (i));
  return 0;
}
```

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

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
  // i doit être différent de zéro
  assert (i != 0);
  return 1.0/i;
}
```

6 Pour plus d'informations sur GCC, visitez http://gcc.gnu.org/.

7 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`.

Il y a également un fichier d'en-tête appelé `reciprocal.hpp` (voir Listing 1.3).

Listing 1.3 (***reciprocal.hpp***) **Fichier d'en-tête – *reciprocal.hpp***

```
#ifdef __cpluplus
extern "C" {
#endif

extern double reciprocal (int i);

#ifdef __cplusplus
}
#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'en-tête. Par défaut, GCC cherche dans le répertoire courant et dans les répertoires où les en-têtes des bibliothèques standards sont installés. Si vous avez besoin d'inclure des fichiers d'en-tête 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, « Débogage avec 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'en-tête, 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'en-tête.

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 par 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[8]. 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.

8 ... à moins que votre programme ne fonctionne toujours du premier coup.

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 caetera*.

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[9]. 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 suivant 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, donc 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.

9 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'en-tête 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'En-tête

Vous pouvez en apprendre beaucoup sur les fonctions système disponibles et comment les utiliser en observant les fichiers d'en-tête. 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'en-tête 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'en-tête placés dans les répertoires `/usr/include/bits`, `/usr/include/asm` et `/usr/include/linux`. Par exemple, les valeurs numériques des signaux (décrits dans la Section 3.3, « Signaux » du Chapitre 3, « Processus ») sont définies dans `/usr/include/bits/signum.h`. Ces fichiers d'en-tête constituent une bonne lecture pour les esprits curieux. Ne les incluez pas directement dans vos programmes, cependant ; utilisez toujours les fichiers d'en-tête 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.