Christophe Blaess

Développement système sous Linux

Ordonnancement multitâche, gestion mémoire, communications, programmation réseau

4e édition

EYROLLES

Développement système sous Linux

4e édition

Tirer le meilleur parti de l'environnement Linux

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

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

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

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

À qui s'adresse cet ouvrage ?

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

Sur le site www.editions-eyrolles.com

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

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

Sommaire

Développement système sous

Linux

Ordonnancement multitâche,
gestion mémoire, communications,
programmation réseau

Dans la collection *Les guides de formation Tsoft*

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

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

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

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

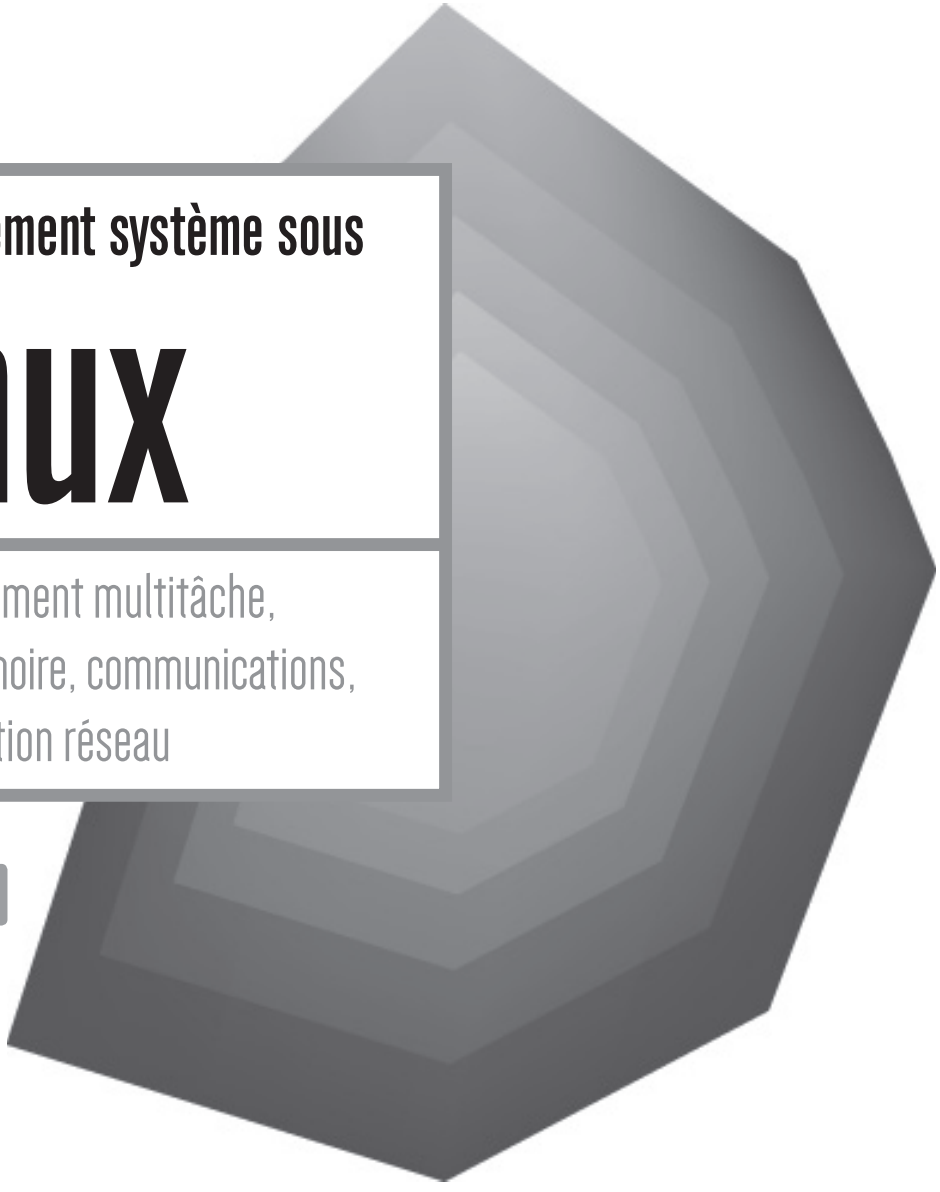
Autres ouvrages

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

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

I. Hurbain, E. Dreyfus. – **Mémento Unix/Linux**
N°13306, 2e édition, 2011, 14 pages.

Christophe Blaess



Développement système sous Linux

Ordonnancement multitâche, gestion mémoire, communications, programmation réseau

4e édition

EYROLLES

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

Avant-propos

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

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

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

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

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

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

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

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

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

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

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

Le standard qui s'impose de nos jours est une fusion de Posix et de la norme *Unix 98*, ayant évoluées ensembles pour donner naissance à SUSv4 (*Single Unix Specifications version 4*). Non seulement ce document décrit des fonctionnalités bien respectées sur les systèmes Unix en général et Linux en particulier, mais il est en outre disponible gratuitement sur Internet, sur le site de l'association Open Group (www.opengroup.org).

De nombreux lecteurs des éditions précédentes m'ont écrit pour me communiquer leurs observations et me faire part de leurs remarques, je les en remercie de tout cœur.

Ris-Orangis, avril 2016
Christophe@Blaess.fr
http://www.blaess.fr/christophe/

Table des matières

1

Concepts et outils

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

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

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

Généralités sur le développement sous Linux

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

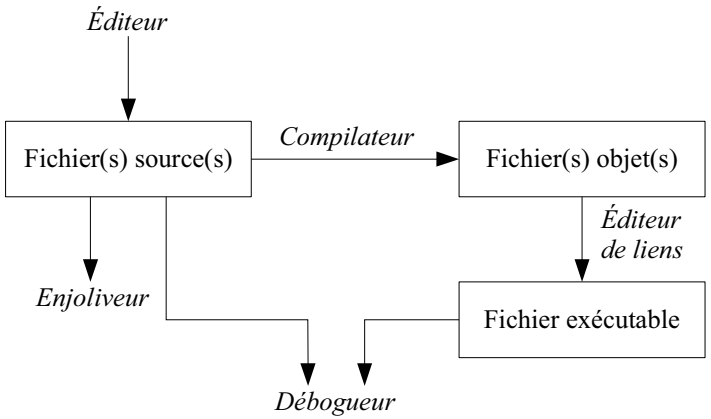
Outils de développement

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

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

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

Figure 1–1
Processus de développement en C



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

Eclipse

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

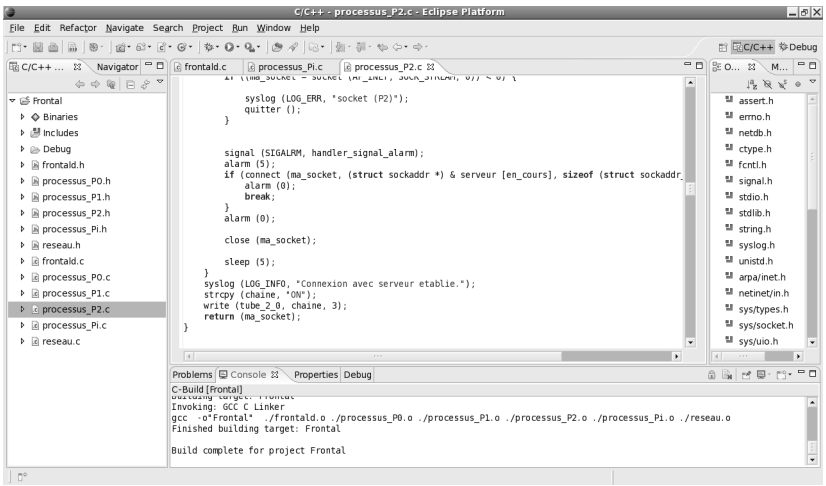


Figure 1–2
L'environnement de développement Eclipse

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

NetBeans

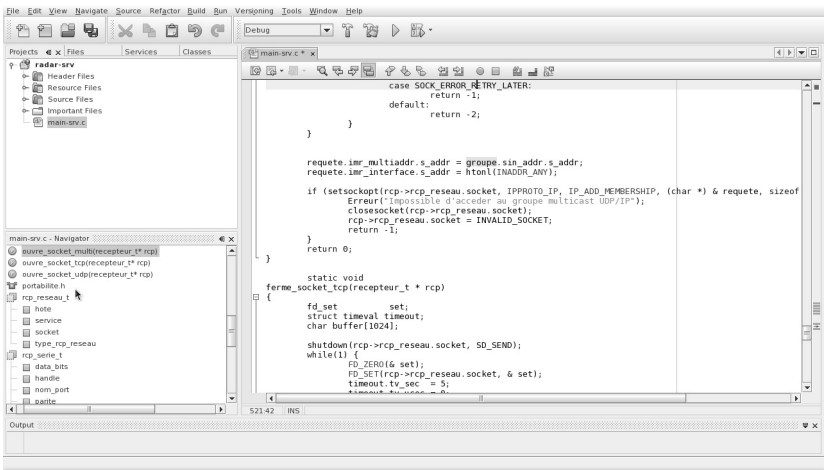


Figure 1–3
NetBeans

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

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

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

Programmation en Python

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

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

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

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

Le script est ensuite rendu exécutable avec la commande :

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

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

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

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

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

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

Éditeurs de texte

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

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

Vi et Emacs

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

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

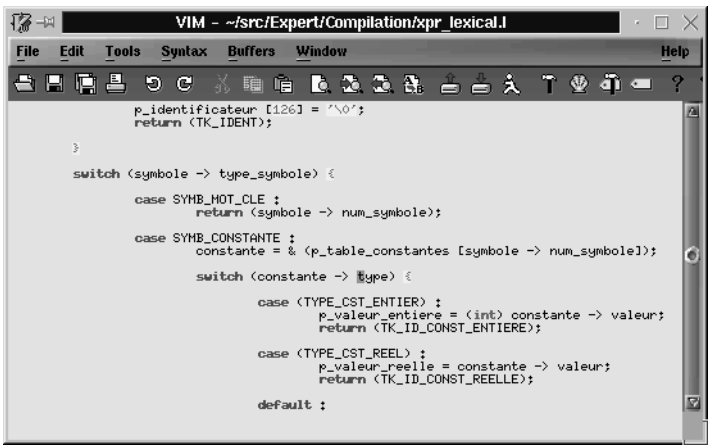
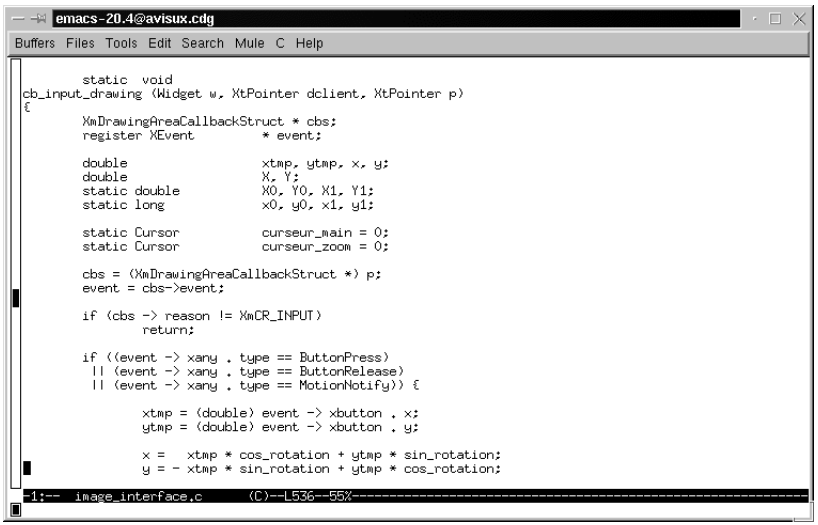


Figure 1–4
Vi sous X11

Figure 1–5
Emacs sous X11



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

Éditeurs Gnome ou KDE

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

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

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

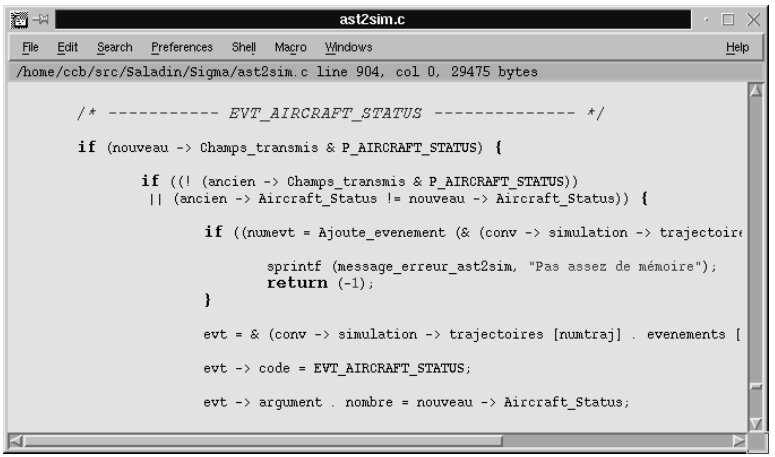
On les utilisera donc plutôt comme des outils d'appoint mais rarement pour travailler longuement sur une application.

Nedit

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

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

Figure 1–6
Nedit



Compilateur, éditeur de liens

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Débogueur, profileur

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

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

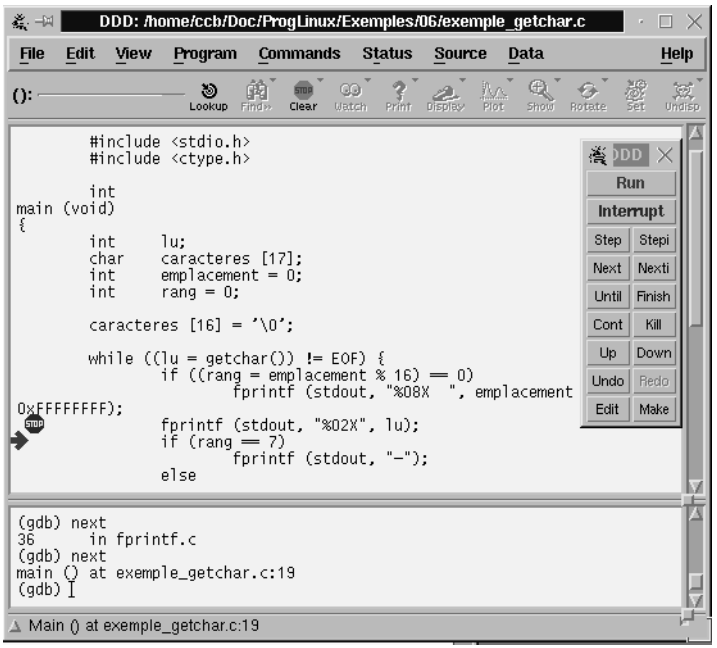


Figure 1–7
Utilisation de ddd

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

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

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

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

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

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

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

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

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

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

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

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

Nous commençons par demander un aperçu du listing du programme :

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

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

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

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

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

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

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

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

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

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

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

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

Nous quittons à présent `gdb` :

```
(gdb) quit
$
```

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

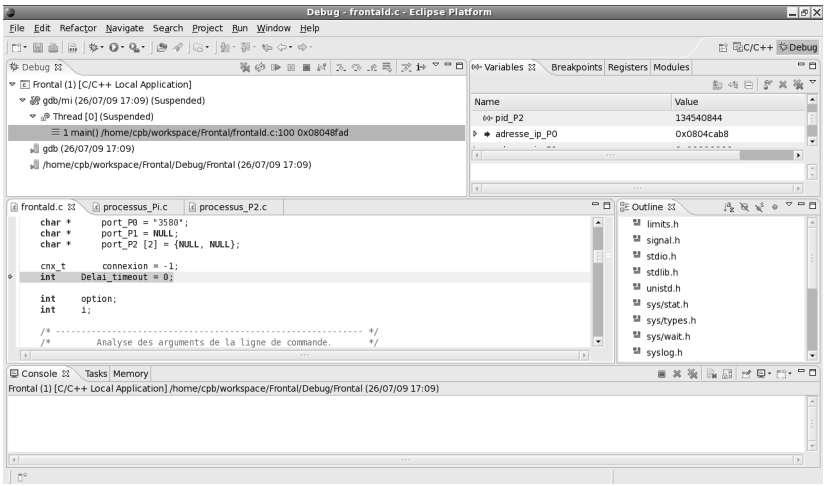


Figure 1–8
Débogage sous Eclipse

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

Valgrind

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

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

Gprof

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

On compile donc le programme à profiler ainsi :

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

On l'exécute alors normalement :

```
$ ./programme
$
```

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

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

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

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

```
$ strace ./programme
```

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

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

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

Traitement du code source

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

Vérificateur de code

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

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

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

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

Mise en forme

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

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

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

hello.c :

```
#include <stdio.h>

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

Nous demandons une mise en forme dans le style Gnu :

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

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

Voyons la conversion en style Kernighan et Ritchie :

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

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

Et finalement le style Berkeley original :

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

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

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

Utilitaires divers

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

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

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

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

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

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

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

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

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

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

Construction d'application

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

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

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

```
cible : dépendances
    commandes
```

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

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

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

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

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

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

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

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

Distribution du logiciel

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

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

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

Archive classique

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

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

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

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

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

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

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

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

Paquetage à la manière Red Hat

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

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

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

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

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

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

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

- Suppression d'un paquetage :

```
$ rpm -e paquet
```

- Recherche du paquetage contenant un fichier donné :

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

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

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

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

Bibliothèques supplémentaires pour le développement

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

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

Interface utilisateur en mode texte

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

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

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

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

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

Développement sous X-Window

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

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

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

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

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

Les environnements KDE et Gnome

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

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

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

Conclusion

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

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