TOUT CE QUE VOUS AVEZ TOUJOURS
VOULU SAVOIR SUR

Unix

SANS JAMAIS $O_S{}^E{}_R$ LE DEMANDER

Ou comment utiliser la ligne de commande quand
on n'y connaît goutte

Vincent Lozano
lozano@enise.fr

Version du 24 décembre 2010
Dernière mise à jour sur :
http://lozzone.free.fr

C'est simple... Souviens-toi...
Tu as demandé un jour au prétendu ingénieur Barnier
de créer un nouveau câbleur. Barnier fit alors une erreur [...]
Il fit tout simplement entrer le double polarisateur chromatique
en résonance avec le palpeur de mirette.
Ce qui détruisit le câbleur mais eut pour effet secondaire
et imprévu la création d'un champ d'anti-temps
dont l'épicentre mesurable se trouva être relié aux circuits chromatiques
du laboratoire de recherche sur les chrono-particules
que je dirige dans les sous-sols de l'Olog [...]

Le Garage Hermétique — Mœbius

Préface

There is more
than one way
To do it.
Larry Wall [1].

Pourquoi ce manuel ?

En 1992, un collègue du DEA Image de l'université Jean Monnet de Saint-Étienne, fait démarrer son PC en me disant : « tu vois ça c'est UNIX », il s'agissait bien sûr d'une des premières versions de *LINUX*. J'avais alors été plutôt sceptique sur cet imbroglio de messages au démarrage et sur l'aspect plutôt spartiate de l'environnement graphique. Parallèlement, nous travaillions sur des « mini-projets » (sorte de projet de fin d'étude) axés sur l'implantation d'algorithme de traitement et d'analyse d'image. À l'époque, nous utilisions un compilateur C du commerce n'exploitant pas le mode protégé du processeur Intel, et nous nous trouvâmes face au problème aberrant de la mémoire segmentée par paquet de 64 ko. Cette limitation rendait très difficile le chargement d'images numériques en mémoire. C'est finalement « grâce » à ce problème que nous sommes tombés sur un compilateur (djgpp) en ligne de commande capable d'exploiter le mode protégé des processeurs des PCs [2] et ainsi de lever cette limitation. Ce compilateur a été une révélation dans le sens où je me suis aperçu que des gens distribuaient *gratuitement* des outils qui allaient enfin me permettre de travailler confortablement.

Certains mini-projets du DEA avaient lieu à l'école des Mines de Saint-Étienne, et c'est là que j'ai eu les premiers contacts avec cette « chose » qu'est UNIX. Premiers contacts désagréables : impossible de trouver un lecteur de disquette dans ce fatras de machines, des éditeurs de texte pour programmer, dont tout le monde vantait les mérites, nécessitant 23 doigts et une mémoire d'éléphant pour arriver à insérer un caractère en fin de ligne, des utilisateurs à la limite du fanatisme, devenant même hargneux et sectaires lorsque je mentionnais mon expérience de développeur débutant en C avec Turbo C... Cependant il émanait de ce laboratoire qui utilisait les ressources de ce réseau UNIX, une certaine liberté d'action qui m'a d'emblée séduit : il était apparemment possible d'utiliser les processeurs de plusieurs machines, chaque utilisateur semblait avoir créé son propre environnement de travail, personne ne parlait de « rebooter » ou de « plantage », certains s'attelaient à des projets de logiciels ambitieux, d'autres rédigeaient des documents d'allure professionnelle, il n'était pas question de pirater un logiciel du commerce puisque tous les outils semblaient être à portée de main pour travailler.

En 1993, débutant une thèse au laboratoire Ingénierie de la Vision de la faculté

1. Créateur du logiciel Perl.
2. Et donc de permettre de joyeux `ptr=(char*)malloc(4194304);` que certes seuls les habitués du langage C comprendront...

de sciences à Saint-Étienne, j'entrepris d'installer une version de gcc (compilateur C de chez GNU) et de comprendre comment fonctionne ce fameux LaTeX (système de préparation de document qui a généré le document que vous avez sous les yeux). Le fait que l'Université soit connectée au réseau Renater [3] a bien sûr grandement aidé à mener à bien ce projet parsemé d'embûches qui m'a permis de me sensibiliser à l'environnement d'UNIX. J'ai été alors fasciné par l'immensité du système, son aspect *ouvert* donnant une sensation d'infini, effrayant de complexité. Mais cette complexité était, me semblait-il, le prix de la liberté, la liberté de pouvoir contrôler cette machine qu'est l'ordinateur.

Les années qui ont suivi, je les ai consacrées en partie à l'administration d'un système UNIX pour l'équipe de recherche et les étudiants, c'est également à cette époque que j'ai découvert et me suis investi dans l'UNIX que je pouvais emporter à la maison : *LINUX*. Je passais alors de simple utilisateur de l'informatique à acteur qui tentait de comprendre les rouages de ce meccano géant qu'est un système d'exploitation. La motivation venait bien sûr du fait que toutes les pièces de ce meccano étaient accessibles et documentées. J'avais la sensation de pouvoir potentiellement comprendre les fondements de l'informatique tout en étant conscient que cela me demanderait sans doute un temps infini...

La rencontre virtuelle via les forums de discussion, avec la communauté de « bricoleurs » de chez GNU et *LINUX* a été pratiquement une prise de conscience quasi politique, une autre vision de la démarche scientifique. Je trouvais enfin des êtres humains désirant « faire avancer le schmilblick » sans arrière pensée de capitalisation de l'information. Malgré tout, si les logiciels libres peuvent s'intégrer — et s'intègrent — dans l'économie de marché, je reste aujourd'hui admiratif vis à vis de cette idée de *diffuser les connaissances* et de s'assurer qu'elles *puissent toujours être diffusées*. C'est sans doute à ce prix que le commun des mortels, utilisateur de l'outil informatique gardera son indépendance d'esprit et sa liberté de choix.

Qu'y a-t-il dans ce manuel ?

J'ai entrepris de rédiger ce manuel lorsqu'en 1999, le responsable de maîtrise de l'IUP Vision de Saint-Étienne m'a demandé de dispenser quelques heures de cours pour présenter aux étudiants le système UNIX. Le document était alors composé d'un « petit guide de survie » présentant les fonctionnalités de base à connaître pour survivre devant un ordinateur géré par un UNIX. J'ai ensuite décidé de compléter ce document, en tentant de présenter ce qu'un *utilisateur* doit savoir pour se débrouiller sous UNIX. Ce manuel ne contient donc aucune allusion à l'*administration* d'un système. Il ne se veut pas non plus un guide de référence [4], mais plutôt une *sensibilisation* à la « philosophie » d'UNIX. On trouvera donc beaucoup de pistes à explorer et jamais de présentation systématiquement détaillée. Comme disait mon professeur de Taï Chi au sujet d'un stage qu'il avait effectué sur le Yi Qing, si UNIX était un livre, ce document serait l'équivalent de passer son doigt sur la couverture pour y enlever la poussière...

3. Le réseau Renater regroupe près d'un millier d'établissements de recherche et d'enseignement.
4. Peut-on d'ailleurs créer un guide de référence sur l'utilisation d'UNIX ?

Les informations présentées ici font partie des connaissances que j'ai acquises ces dernières années et dont je fais usage régulièrement, il s'agit donc à mon humble avis d'informations utiles et directement exploitables et non pas de fonctionnalités obscures. J'ai tenté de les présenter avec l'idée de m'adresser à un novice, dans le but de le convaincre de l'intérêt qu'il y a à apprendre UNIX. En outre, si la plupart des informations de ce manuel sont adaptées à n'importe quel UNIX, il est évident que *LINUX* est notre UNIX de « référence » de même que les outils présentés le sont dans leur version du projet GNU. Il ne s'agit pas d'un choix sectaire mais simplement de l'exploitation de la grande disponibilité de ces outils.

Ce qu'il n'y a pas dans ce manuel

L'utilisateur novice d'UNIX et de *LINUX* en particulier cherche souvent des informations pour installer ce système d'exploitation sur sa machine, pour savoir s'il faut partitionner son disque dur ou en acheter un autre, pour connaître la distribution [5] de *LINUX* qui lui conviendrait le mieux, pour savoir comment, une fois la dite distribution installée, il est possible d'avoir du son, d'utiliser son scanner, de configurer sa connexion internet, etc. Ce manuel ne traite ni de l'installation, ni de l'administration d'un système UNIX, mais donne le savoir faire nécessaire pour s'y attaquer.

D'autres interrogations concernent les équivalents des logiciels de bureautique [6], de jeux, etc. Il ne sera donc pas question dans ce manuel, ni des équivalents en logiciels libres des tableurs et autres traitements de texte, ni de l'utilisation des célèbres bureaux qu'on trouve aujourd'hui sous *LINUX* : Kde et Gnome. Nous ne pouvons que vous conseiller de vous procurer l'excellent *Simple comme Ubuntu* (Roche, 2010) pour trouver des réponses s'appuyant sur la distribution Ubuntu.

Les outils présentés ici, sont ceux que l'on peut retrouver sur n'importe quel UNIX. L'accent est donc mis sur l'usage des *commandes* et de leur utilisation interactive ainsi que dans le cadre de *scripts* ; il sera donc davantage question de clavier que de souris. Ce manuel a d'ailleurs pour objectif de convaincre le lecteur de l'intérêt de cette approche dans le cadre de l'apprentissage d'UNIX.

Il est important de noter que le titre de ce manuel est un mensonge éhonté. Le contenu de ce document ne se veut pas être une présentation d'UNIX, ou même des UNIX, ni des standards tels que Posix. L'UNIX de référence ici est GNU/*LINUX* car c'est sans doute aujourd'hui le plus accessible et le plus utilisé. Cependant, la majeure partie des outils présentés dans ce manuel peuvent être utilisés tels quels sur n'importe quel UNIX. Lorsque ça ne sera pas le cas nous tenterons d'insérer ce joli panneau dans le paragraphe.

5. Une distribution de *LINUX* est un ensemble de logiciels composé du noyau *LINUX*, d'applications (bureautique, images, sons, etc.) et surtout d'un programme d'installation et de mise à jour qui la caractérise.

6. Une des questions angoissantes pour l'utilisateur débutant ou sur le point d'utiliser UNIX est en effet de savoir s'il lui sera toujours possible d'utiliser les merveilleux outils de la suite bureautique de la célèbre entreprise dont nous tairons le nom ici...

Ce manuel est avant tout un manuel destiné aux *débutants*, il a donc pour objectif d'être *didactique* et tente donc de ne pas noyer le lecteur dans un fatras de détails techniques. Encore une fois, le titre pourrait induire le lecteur naïf que vous n'êtes pas, en erreur : vous ne saurez pas tout sur UNIX ! Aux endroits où apparaîtra ce panneau, j'ai introduit quelques concepts un peu plus « pointus » qu'il est, dans un premier temps, inutile de lire mais qui corrigent certaines imprécisions.

Comment lire ce manuel ?

Les pages se lisent de gauche à droite et de haut en bas et pour donner l'impression d'une cohérence globale, ce manuel est divisé en chapitres dont les titres sont :

UNIX et les logiciels libres : une présentation de la naissance d'UNIX, du lien avec les logiciels libres. Un chapitre non technique, assorti de considérations philosophico-politico-économico-éthiques douteuses ;

Petit guide de survie : chapitre présentant les concepts de base d'UNIX (notions d'utilisateurs, de fichiers, de processus, etc.) ;

La boîte à outils : contient une description de quelques-uns des outils du grand meccano. Ce chapitre permet d'avoir une idée de la souplesse apportées par l'homogénéité de tous ces outils ;

Communiquer ! : présente les outils axés sur le réseau, la communication entre utilisateurs, entre machines, le transfert de fichiers, l'utilisation de la messagerie et l'accès aux Web ;

Développer : contient des informations relativement précises sur le langage de commande bash (le shell de chez GNU), l'utilisation de ces étranges bêtes que sont les `makefile`s, et la façon de créer des programmes en langage C qui est le langage utilisé pour développer UNIX ;

Se mettre à l'aise : permet de comprendre comment on peut configurer son environnement de travail : personnaliser son shell, comprendre le fonctionnement des éditeurs de texte vi et emacs, configurer l'environnement graphique, et quelques pistes pour installer des logiciels sur son propre compte ;

À l'aide ! donne des pistes pour chercher de la documentation sur les commandes d'UNIX à la fois localement et en ligne.

Il est conseillé de lire ce manuel de manière linéaire au moins une fois, pour comprendre comment s'articulent les différents concepts introduits. On pourra revenir dans un deuxième temps sur des informations plus spécifiques contenues dans certains chapitres. La complexité et l'immensité du sujet font que vous trouverez beaucoup d'interconnexions entre les sujets traités.

Comment imprimer ce manuel ?

Avec une imprimante [7], en utilisant exclusivement les fichiers proposés sur `http://lozzone.free.fr`. Ce manuel est conçu pour être imprimé en deux pages logiques

7. Arf arf (comme disait Frank ZAPPA).

par page physique (*2up printing* comme disent nos amis anglosaxons). En d'autres termes, chaque page de ce document doit apparaître sur une page au format A5. Quatre solutions — et quatre fichiers associés — sont possibles :

1. une version destinée à être reliée du long coté de la feuille A4, c'est la solution pour ceux qui ne disposent pas de massicot ; le fichier portant le nom `guide-unix-0.pdf` est en outre généré pour présenter la page impaire (dite belle page) à droite (cf. figure 1a page x) ;
2. une version pour ceux qui disposent d'un massicot : dans ce cas, il faudra « plier » le document et le relier au niveau de cette pliure (voir figure 1b page x). Le fichier correspondant se nomme `guide-unix-1.pdf` ;
3. une version contenant 2 exemplaires (fichier `guide-unix-2.pdf`). On reliera ces deux exemplaires comme indiqué à la figure 1c page x ;
4. une version au format PostScript `guide-unix.ps.gz` pour ceux qui voudraient faire « à leur sauce »...

Attention, les deux derniers fichiers au format Pdf, sont prévus pour être imprimés en recto verso en demandant à l'imprimante de retourner la feuille du côté court. Il faudra donc veiller à ce que le pilote d'imprimante utilise cette option.

Que pouvez-vous faire de ce manuel ?

Nom de l'auteur : Vincent LOZANO ;

Titre : Tout ce que vous avez toujours voulu savoir sur UNIX sans jamais avoir osé le demander ;

Date : 24 décembre 2010

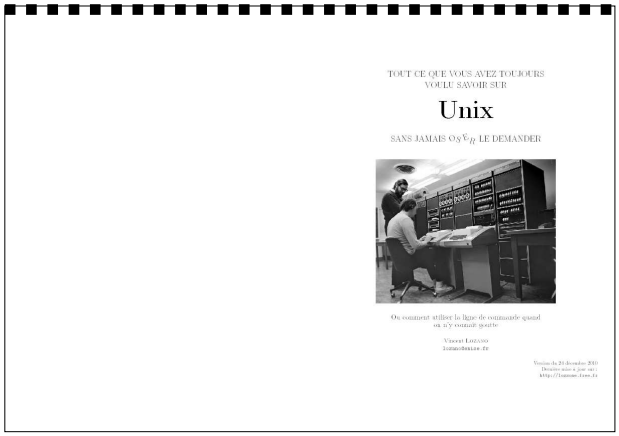
Copyleft : ce manuel est libre selon les termes de la Licence Art Libre (`http://www.artlibre.org`) (LAL).

La LAL stipule en résumé que vous pouvez copier ce manuel. Vous pouvez également le diffuser à condition :

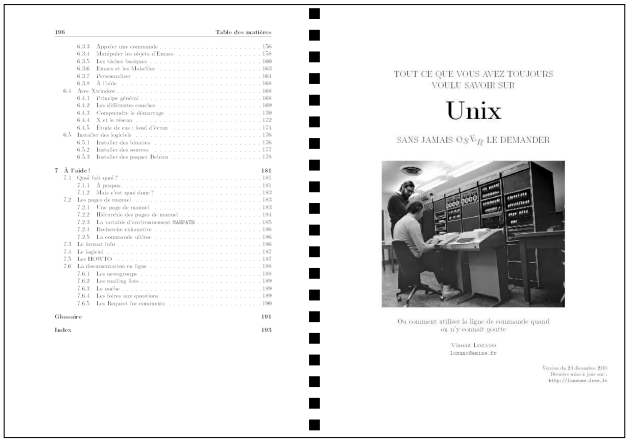
- d'indiquer qu'il est sous la LAL ;
- d'indiquer le nom de l'auteur de l'original : Vincent LOZANO et de ceux qui auraient apporté des modifications ;
- d'indiquer que les fichiers sources peuvent être téléchargés sur `http://lozzone.free.fr` ;

Enfin vous pouvez le modifier à condition :

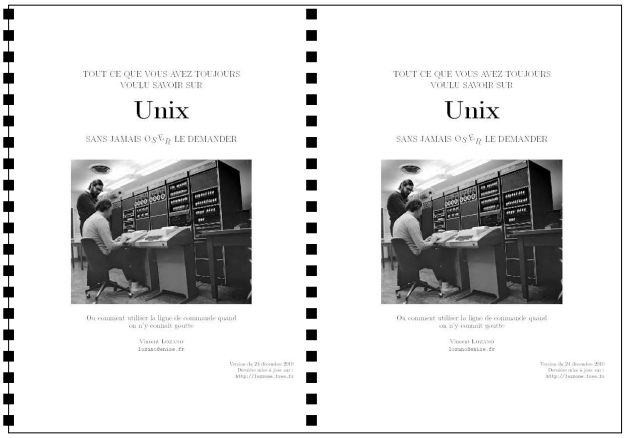
- de respecter les conditions de diffusion énoncées ci-dessus ;
- d'indiquer qu'il s'agit d'une version modifiée et si possible la nature de la modification ;
- de diffuser vos modifications sous la même licence ou sous une licence compatible.



(a) Reliure longue



(b) Reliure courte « 1 exemplaire »



(c) Reliure courte « 2 exemplaires »

FIGURE 1: Les versions imprimables

Conventions typographiques

Certaines conventions utilisées dans ce manuel nécessitent d'être quelque peu éclaircies. Les commandes UNIX qui parsèment le document apparaîtront comme ceci :

```
$ type cat
cat is /bin/cat
$
```

Certaines parties sont présentées sous forme de « notes » pour éclaircir un point sans que la lecture soit indispensable au premier abord.

Si la lecture est indispensable, on aura recours au pictogramme ci-contre pour attirer l'attention du lecteur distrait...

Les logiciels sont typographiés comme indiqués ci-avant. Les mots en anglais sont produits *like this*. Pour mettre en évidence les parties génériques d'une commande on utilisera ⟨*cette notation*⟩. Par exemple, on pourra trouver une phrase comme :

> « ...pour obtenir des informations sur les attributs d'un fichier dont le nom est ⟨*nomfic*⟩
>
> `$ ls -l` ⟨*nomfic*⟩
>
> et le tour est joué ... »

Dans la version papier apparaissent des renvois sur des chapitres ou des paragraphes, comme celui-ci dirigeant le lecteur vers la gestion des ▶processus◀ sous UNIX. § 2.4 p. 47 ◀

Merci

À l'équipe de recherche de feu l'IUP Vision de Saint-Étienne pour m'avoir soutenu dans ce travail, plus particulièrement à Serge CHASTEL qui a rédigé la première version du chapitre « À l'aide ». À Jacques LOPEZ pour avoir pu relire attentivement ce manuel et me proposer des suggestions constructives. À Laurent DEFOURS pour insister à me faire utiliser des mots du français. J'ai dû sous la pression faire :

```
for F in *.tex ; do
  sed 's/librairie/bibliothèque/g' $F |
  sed 's/Librairie/Bibliothèque/g' >  tmp.tex ;
  mv -f tmp.tex $F ;
done
```

pour rendre ce document acceptable à ses yeux[8]. À Nabil BOUKALA pour avoir trouvé exactement 138 coquilles et autres fautes dans la précédente version, ainsi que pour m'avoir indiqué l'orthographe exacte de « Massachusetts ». Aux intervenants des newsgroups autour de *LINUX* et d'UNIX pour leurs précieuses informations qu'ils m'ont indirectement apportées. À Andrea FERRARIS pour ses encouragements, à Cédric « rixed » pour ses remarques sur l'Osf, Hugues « débianiste avant tout » pour sa précision sur la naissance du projet GNU, Jacques L'HELGOUALC'H pour m'avoir suggéré qu'un *mécano* réparait des voitures et qu'un *meccano* était un jeu de construction, Paul GABORIT pour sa remarque judicieuse sur le caractère libre du package french de LaTeX, un grand merci à Géo « cherchetout » pour m'avoir transmis 149 coquilles après avoir lu le document intégrant les 138 fautes repérées par Nabil.

J'adresse mes plus profonds remerciements à Stéphane CHAZELAS pour sa lecture minutieuse de la version précédente de ce manuel. C'est suite à ses remarques constructives que ce manuel s'est enrichi de nombreuses « nota » et surtout que beaucoup d'imprécisions et d'erreurs de fond ont été corrigées...

Je tiens enfin à exprimer ma gratitude et mon grand respect pour le travail et la motivation d'Alexis KAUFMAN et Didier ROCHE qui m'ont fait confiance pour

8. Le plus terrible est qu'après avoir relu cette préface, il m'a gentiment suggéré de regarder du côté de l'option `i` de la version GNU du programme sed...

la publication de ce livre. Merci également à tous les membres du groupe de relecture des Framaboooks pour l'effort qu'ils ont fourni pour parfaire ce document : en particulier Barbara « Garburst » que l'on peut qualifier d'« œil de lynx », Laurent « lolonene », kinouchou.

Enfin, je souhaite remercier chaleureusement Christophe MASUTTI pour sa lecture attentive. Je pense qu'une bonne trentaine de virgules ainsi que bonne vingtaine d'appels de notes de bas de page ont été correctement placés grâce à lui.

À toute la communauté des hackers pour l'énergie qu'ils insufflent...

*
* *

Bonne lecture[9] !

9. Histoire de commencer un peu à mettre des notes de bas de page un peu partout, vous noterez peut-être au cours de cette lecture, que malgré tout des termes anglais apparaîtront souvent, parfois non traduits. Une honte !

Sommaire

1 Unix et les logiciels libres **1**
- 1.1 Avant-propos : la naissance d'un logiciel . . . 1
- 1.2 Unix . . . 5
- 1.3 Les logiciels libres . . . 9
- 1.4 Le cas de Gnu/Linux . . . 11
- 1.5 Quelques réflexions sur les logiciels libres . . . 13
- 1.6 Actualité et avenir des logiciels libres . . . 15

2 Petit guide de survie **21**
- 2.1 Le shell . . . 21
- 2.2 Utilisateurs . . . 28
- 2.3 Le système de fichiers . . . 30
- 2.4 Processus . . . 47
- 2.5 Quelques services . . . 55

3 La boîte à outils **61**
- 3.1 Introduction à l'expansion . . . 61
- 3.2 Redirections et tubes . . . 62
- 3.3 Les outils de base . . . 67
- 3.4 Le shell en tant que langage . . . 73
- 3.5 `grep` et la notion d'expressions régulières . . . 77
- 3.6 `awk` . . . 79
- 3.7 `sed` . . . 81
- 3.8 Études de cas . . . 83

4 Communiquer ! **87**
- 4.1 Concepts à connaître . . . 87
- 4.2 Quatre grands classiques . . . 91
- 4.3 Outils de communication d'Unix . . . 98
- 4.4 Courrier électronique . . . 101
- 4.5 Le ouèbe . . . 103

5 Développer ! **107**
- 5.1 Éditer un fichier . . . 107
- 5.2 Faire des scripts en shell . . . 109
- 5.3 Makefile . . . 124
- 5.4 Faire des projets en langage C . . . 130

6 Se mettre à l'aise ! **143**
- 6.1 Avec le shell . . . 143
- 6.2 Avec vi . . . 152
- 6.3 Avec Emacs . . . 154
- 6.4 Avec Xwindow . . . 168

6.5 Installer des logiciels 176

7 À l'aide ! 181
7.1 Quoi fait quoi ? 181
7.2 Les pages de manuel 183
7.3 Le format info 186
7.4 Le logiciel 187
7.5 Les HOWTO 187
7.6 La documentation en ligne 188

Bibliographie 191

Glossaire 193

Index 197

1 Unix et les logiciels libres

Sommaire

1.1 Avant-propos : la naissance d'un logiciel
1.2 Unix
1.3 Les logiciels libres
1.4 Le cas de Gnu/Linux
1.5 Quelques réflexions sur les logiciels libres
1.6 Actualité et avenir des logiciels libres

Les productions de génie et les moyens d'instruction
sont la propriété commune ; ils doivent être répartis
sur la surface de la France comme les réverbères dans une cité.

Grégoire (1837) [1].

AVANT d'aborder l'étude du système UNIX, nous allons, dans ce chapitre, définir quelques termes de manière à éclairer le lecteur novice. Les éclaircissements portent à la fois sur la famille de systèmes d'exploitation qu'est UNIX, et le type particulier de logiciel que sont les logiciels dits logiciels libres. De manière à positionner clairement UNIX par rapport aux logiciels libres, nous débutons ce chapitre par une présentation succincte de la naissance d'un logiciel. Vient ensuite un historique d'UNIX, qui tente de montrer que même si UNIX est aujourd'hui un ensemble de logiciels propriétaires, sa naissance et ses évolutions de jeunesse constituent sans aucun doute les prémices des logiciels libres. Nous nous livrerons en guise de deuxième partie de chapitre à une présentation de la philosophie des logiciels libres — principalement au travers de *LINUX* et du projet GNU — et de ses implications tant techniques que philosophiques.

1.1 Avant-propos : la naissance d'un logiciel

Pour se sensibiliser à la notion de logiciel libre et pour comprendre ce qu'est un système d'exploitation multi-plate-forme, il est impératif de saisir les différents outils et mécanismes qui entrent en jeu lors de la création d'un logiciel ainsi que lors de l'exécution d'un programme.

1.1.1 Du source

La conception d'un logiciel passe par différentes phases que l'on peut présenter de manière plus ou moins détaillée ; toujours est-il que ces phases s'articulent autour des étapes importantes suivantes :

1. L'abbé Grégoire est le fondateur du centre national des Arts et Métiers.

1. l'analyse du problème à résoudre ;
2. la conception d'un algorithme correspondant à cette analyse ;
3. la traduction de l'analyse dans un langage de programmation plus ou moins évolué ;
4. la compilation du programme en langage évolué, c'est-à-dire la traduction du langage évolué vers un langage moins expressif qu'est celui de la machine et plus précisément du processeur et du système d'exploitation ;
5. la phase de test qui permet de s'assurer que le programme répond aux besoins initiaux.

Le programme en langage évolué est appelé le *langage source*, le langage moins expressif le langage *cible*, et dans le cadre de la programmation avec un langage compilé, on nomme le programme cible : *binaire* ou *exécutable*. Le programme binaire est une suite d'instructions destinées à être interprétées par un processeur particulier. Un binaire est donc dédié à un processeur et un système d'exploitation d'un certain type, ce couple processeur/système est appelé *plate-forme*.

Le programme source est le programme qui reflète très exactement l'analyse et l'algorithme correspondant au problème à résoudre. On ne peut retrouver dans le binaire la complexité et la richesse de l'algorithme mis en œuvre qu'au prix d'un travail surhumain. En effet la solution serait de « décompiler » le binaire. On peut tirer du binaire une liste d'instructions issues de la couche assembleur ; c'est parce que cette liste contient des instructions peu expressives par rapport aux instructions du langage évolué, qu'il est pratiquement impossible de retrouver l'architecture initiale de l'algorithme.

Pour comprendre ce principe — qui peut ne pas être clair pour le lecteur qui n'a pas une expérience de programmeur — on peut faire plusieurs analogies. La première est celle du cuisinier qui prépare un plat avec un certain nombre d'ingrédients. En goûtant le plat, un palais averti peut détecter quelques-uns de ces ingrédients. On peut même imaginer qu'un appareil puisse faire une analyse moléculaire et établir une liste précise des éléments qui composent le plat. Cependant, il semble difficile voire impossible, à partir du plat de savoir comment le chef a procédé pour mélanger les ingrédients, le temps de cuisson, les plats et les ustensiles qu'il a utilisés, etc. Une autre analogie possible est celle de la conception d'une culasse automobile. On peut démonter une culasse, la mesurer, trouver l'alliage de métaux qui la compose, mais on ne peut à partir de ces données retrouver le modèle de thermodynamique et l'ensemble de calcul qui a mené à définir la forme particulière de cette culasse. Enfin, on peut toujours traduire un texte d'un grand philosophe en un texte composé de mots et phrases simples dans le but de le faire comprendre à un enfant de dix ans. On ne pourra cependant pas retrouver le texte original à partir du texte traduit.

Les programmes sources constituent donc
la seule information précise
concernant le fonctionnement d'un programme.

1.1.2 De la portablilité

Pour comprendre ce qu'implique la manipulation des programmes sources, nous allons expliciter dans ce paragraphe les notions :

1. d'exécution d'un programme ;

2. de compilation d'un programme source ;
3. d'interprétation d'un programme source.

Tout d'abord notons que Tanebaum (2001) a introduit un modèle de représentation d'un ordinateur qui est un modèle en couches (cf. figure 1.1 [2]). Dans ce modèle, on représente la machine physique dans la première couche dite de bas niveau puis, en faisant intervenir des langages de programmation de plus en plus évolués, on définit des couches de haut niveau. Celles-ci permettent de dialoguer avec la machine à l'aide de concepts de plus en plus abstraits et éloignés de l'architecture physique.

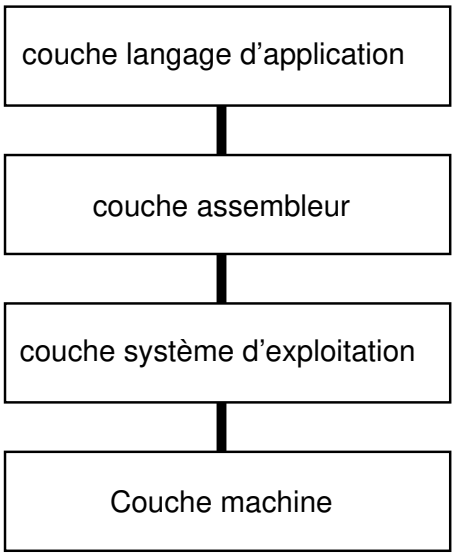


FIGURE 1.1: L'architecture en couches d'un ordinateur.

Exécution

Au niveau de la couche machine, lorsqu'un programme est en cours d'exécution, l'unité centrale passe le plus clair de son temps à transférer le code des instructions stockées en mémoire centrale pour les décoder et les exécuter. Le code de chacune de ces instructions correspond à un numéro identifiant une opération qu'est capable de réaliser le processeur utilisé. Le code des instructions est bien évidemment différent pour les processeurs fabriqués par Intel et Motorola pour ne citer que ceux-là. Par conséquent :

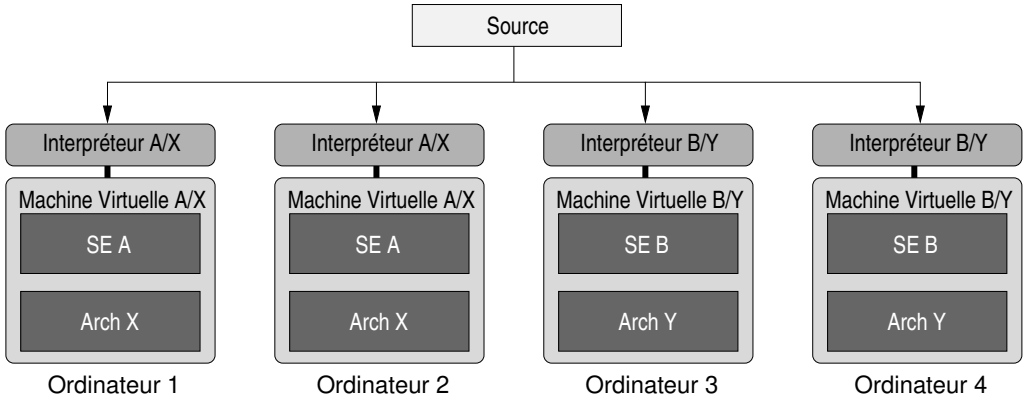
Le code en langage machine d'un programme donné
n'est pas le même selon le processeur qui doit l'exécuter.

Compilation d'un source [3]

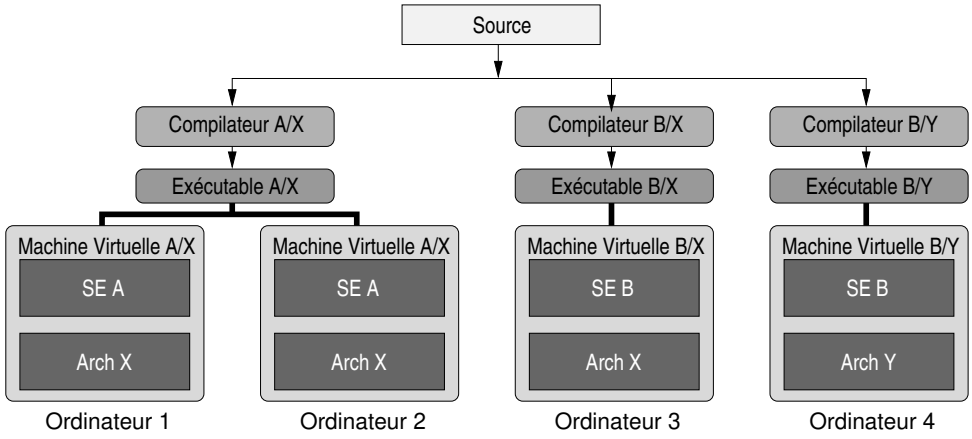
Avec un langage compilé, on dispose d'un outil logiciel particulier appelé *compilateur* pour traduire le programme source en un programme exécutable par la machine

2. La couche machine physique est en réalité composée de trois couches : machine physique, couche micro-programmée, couche machine traditionnelle.

3. Il arrive que ces fainéants d'informaticiens disent « le source » qui est une contraction de « le fichier source »...



(a) Utilisation d'un programme interprété sur différents ordinateurs.



(b) Sources compilées sur différents ordinateurs.

FIGURE 1.2: Portabilité de programmes écrits en langage compilé et interprété.

virtuelle de l'ordinateur hôte. On notera que contrairement au cas des langages interprétés, l'exécutable généré est autonome et à ce titre, peut être exécuté sur un ordinateur doté de la même machine virtuelle (c'est-à-dire doté du même système d'exploitation et de la même architecture). Dans le cas d'un ordinateur doté d'une machine virtuelle différente, l'exécutable est inutilisable, et il faudra disposer d'un compilateur pour la nouvelle plate-forme pour pouvoir produire un exécutable. À titre d'exemple, à la figure 1.2b, on notera que l'exécutable produit sur l'ordinateur 1 est utilisable sur l'ordinateur 2 doté de la même machine virtuelle. Par contre, cet exécutable n'est pas utilisable sur les ordinateurs 3 et 4, sur lesquels il est nécessaire de recompiler le source.

Interprétation d'un source

Dans le cadre des langages interprétés, le programme source est analysé par un programme particulier appelé *interpréteur* ; ce programme se charge d'exécuter chacune des instructions du programme source, les unes après les autres comme le montre la figure 1.2a. Pour pouvoir utiliser le logiciel produit sur un autre ordinateur, il faudra disposer de l'interpréteur sur cet ordinateur, et ceci, que la machine virtuelle de l'ordinateur cible soit différente ou non. En d'autres termes, pour interpréter un source en langage Perl (langage interprété), on devra disposer d'un interpréteur Perl pour Windows, *LINUX*, etc. en fonction du système installé sur l'ordinateur.

1.2 Unix

UNIX est un système d'exploitation (*operating system*), il assure donc aux utilisateurs et aux processus, la répartition des ressources fournies par la machine : calcul, stockage, impression, transfert de données, etc. Ce système d'exploitation quarantenaire, a depuis sa création les caractéristiques suivantes :

- *multi-utilisateurs* : le système identifie des personnes logiques et permet à ces personnes d'utiliser le système dans certaines limites ;
- *multi-tâches* : le système est étudié pour exécuter plusieurs programmes en même temps, grâce au concept de « temps partagé » ;
- *multi-plateforme* : on verra qu'UNIX n'est pas un système dédié à un processeur, mais que c'est une famille de systèmes que l'on retrouve sur une multitude de plates-formes.

Il faut également noter que ce système est axé sur le développement [4] ; on dispose donc d'une quantité importante d'outils, permettant de créer des programmes, rédiger des documents, administrer un système, etc.

1.2.1 Historique [5]

UNIX est un système trentenaire, multi-tâches, multi-utilisateurs et disponible sur plusieurs plate-formes. Il a pour origine un projet initié au début des années soixante, cofinancé par la société américaine Bell, le MIT (Massachusetts Institut of Technology), et le ministère de la défense américain. Le projet en question est le développement d'un système permettant à plusieurs centaines d'utilisateurs d'accéder à des ressources informatiques ; ce système devant être disséminé sur plusieurs machines pour assurer un fonctionnement continu même si une machine tombe en panne. Ce projet est baptisé Multics (pour Multiplexed Information Computer Service) et débute au début des années soixante pour être rapidement abandonné en 1969 vraisemblablement par excès d'ambition et faute de temps.

Un scientifique de chez Bell, Ken THOMPSON décide de continuer à travailler sur la partie système d'exploitation dans le but essentiel de *faire tourner des programmes* sur les machines dont il dispose. Pour plaisanter, ce système est baptisé Unics (pour Uniplexed Information ...) et devient rapidement UNIX. THOMPSON est peu après rejoint par Dennis RITCHIE qui crée le langage C en s'inspirant du langage B (langage interprété qu'avait créé THOMPSON après avoir tenté de porter le langage Fortran sur la machine PDP-7). L'intérêt est alors de pouvoir *porter* le système sur d'autres machines sans avoir à tout réécrire [6]. En 1972, les bases fondamentales d'UNIX sont prêtes et les principes de la *boîte à outils* d'UNIX sont énoncés par Doug MACILROY, l'inventeur des ▶tubes◀ [7] : Ch. 2 p. 21 ◀

1. Write programs that do one thing and do it well ;
2. Write programs that work together ;

4. Terme anglais (*to develop* pour mettre au point) passé dans le jargon des informaticiens : développer c'est exercer une activité autour de la programmation, de la création d'un logiciel.

5. Cette section est inspirée des informations disponibles dans l'ouvrage de Garfinkel et Spafford (1996) faisant lui-même référence à celui de Salus (1994)

6. Il faut imaginer qu'à l'époque chaque machine possède son propre langage, une nouvelle machine nécessite donc un nouveau programme.

7. Pour les non anglophones : « écrire des programmes qui font une seule chose et qui le font bien, écrire des programmes qui peuvent communiquer entre eux, écrire des programmes qui manipulent du texte car c'est l'interface universelle. »

FIGURE 1.3: Ken THOMPSON (assis) et Dennis RITCHIE — dans leurs versions « with luxuriant and darker hair than [they] have now[9] » — devant le Pdp11.

3. Write programs that handle textstreams because that's the universal interface.

Simson GARFINKEL et Gene SPAFFORD notent dans leur ouvrage qu'UNIX devient alors un « rêve de programmeur ». Puisque chacun pouvait alors créer ses propres outils d'une complexité croissante avec les outils élémentaires du système ; ces nouveaux outils devenant alors eux-mêmes partie intégrante du système.

En 1973 UNIX est installé sur 16 sites et à la suite d'une conférence[10] ainsi que d'une publication (courant 1974) sur les systèmes d'exploitation, des utilisateurs émettent un avis intéressé. Les demandes affluent. On comptera plus de 500 sites qui utiliseront UNIX, certains en dehors des États-Unis.

En 1978, Billy JOY, un étudiant de l'université de Californie à Berkeley, installe une bande magnétique contenant le système UNIX d'AT&T. Il apporte des modifications importantes au système et distribue ce système modifié, sous forme de sources, pour $ 40 : la Berkeley Software Distribution (BSD). Billy JOY est à l'origine de l'éditeur `vi`, d'un compilateur Pascal, du C-shell, entre autres. Ces modifications et outils font alors « le tour de la terre » puisque d'autres utilisateurs américains et européens transmettront à leur tour des améliorations à JOY. Cet échange d'information constitue sans doute les prémices du mouvement des logiciels libres. La version BSD d'UNIX est à ce titre *la* version libre d'UNIX[11].

Au début des années 80, l'importance prise par la distribution BSD, crée des

9. Comme l'indique RITCHIE lui-même sur sa page `http://cm.bell-labs.com/who/dmr`.
10. ACM Symposium on Operating Systems Principles.
11. De nombreux démêlés judiciaires entre l'université de Berkeley et la société AT&T alors « propriétaire » d'UNIX sont mentionnés par DiBona *et al.* (1999).

(a) Thompson et Ritchie (b) Ritchie et Joy

Figure 1.4: Les « papas » d'unix

tensions avec AT&T. Les utilisateurs sont alors face à *deux* unix :

- le Berkeley unix, préféré par les développeurs et porté par une société nouvellement créée par des étudiants de Berkeley : Sun ;
- AT&T unix system V : le système du propriétaire d'unix et supposé être le *standard*.

Plusieurs fabricants dont Hewlett Packard, IBM, Silicon Graphics créent des unix fondés sur la version de AT&T.

À la fin des années 80, AT&T et Sun signent une charte de développement commun, de manière à fusionner leurs versions, en prenant les avantages de chacune de ces versions pour créer un nouveau standard : System V release 4 (notée aussi SVR4). Se sentant menacés par cette alliance, HP, IBM, Digital et d'autres, créent alors l'Open Software Foundation (OSF) dont le but est de laisser unix dans les mains d'un consortium d'industriels plutôt que dans les seules mains d'AT&T.

Au début des années 90, AT&T vend unix à la société Novell, qui transfère alors la marque déposée unix à l'X/Open Consortium, et en vend les sources à la société SCO. Il y a alors cinq millions d'utilisateurs d'unix.

Aujourd'hui, la guerre des standards semble être terminée, et on a vu apparaître une norme ISO, permettant de standardiser les interfaces de programmation d'applications (API en anglais) : la norme POSIX (Portable Operating System Interface). On trouve aujourd'hui :

Des unix propriétaires

Nom	Propriétaire	Processeur
Solaris	Sun	Sparc & Intel
HPUX	HP	PA
AIX	IBM	Risc & PowerPC
Digital Unix	Digital	Alpha
Irix	SGI	

Des Unix libres

- Linux sur plate-forme Intel, Sparc, Alpha, Mac, ...
- FreeBSD sur plate-forme Intel, Alpha, PC-98 ;
- OpenBSD également multi-plate-forme.

Eric Raymond (DiBona *et al.*, 1999, chapitre 4) note à juste titre que c'est malgré tout, pendant cette lutte fratricide entre unix propriétaires que le système Windows pourtant techniquement bien plus faible, a pu s'imposer.

1.2.2 Architecture et caractéristiques

On peut décomposer un système UNIX en trois grandes entités :

Le noyau : il assure la gestion de la mémoire et des entrées sorties de bas niveau et l'enchaînement des tâches (voir figure 1.5) ;

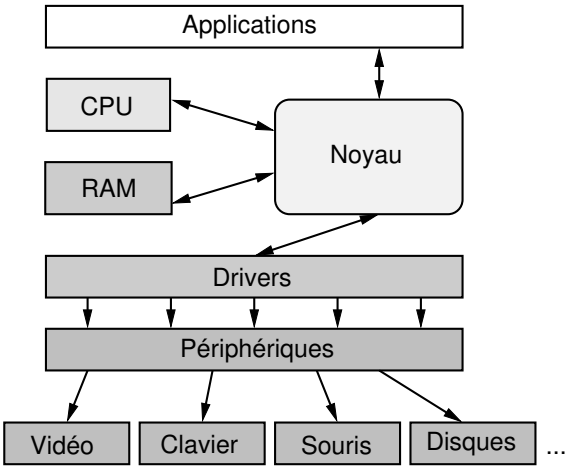


FIGURE 1.5: Modules logiciels autour d'un noyau Unix

Un ensemble d'utilitaires : dédiés à des tâches diverses :
- des interpréteurs de commande appelés *shells* permettant de soumettre des tâches au système, tâches pouvant être concurrentes et/ou communicantes ;
- des commandes de manipulation de fichiers (copie, déplacement, effacement, etc.) ;
- des commandes de gestion d'activités du système (*processus*) ;
- des commandes de communication entre utilisateurs ;
- des outils de développement (éditeurs de texte, compilateurs, débogueurs, ...)
- des outils de traitements de texte.

Une base de données système : un ensemble de fichiers contenant :
- des informations sur la configuration des différents services ;
- des scripts de changement d'état du système (démarrage, arrêt, ...).

Sur chaque système est ensuite installé un ensemble de programmes pour les besoins spécifiques de l'organisation (traitements de texte, logiciels de calculs, etc.)

Jean-Marc RIFFLET dans *La programmation sous* UNIX (Rifflet, 1995) note que les caractéristiques principales d'un système UNIX sont :
- un système hiérarchisé de processus et une « génétique des processus » : chaque ▴processus◂ hérite les caractéristiques de son parent lors de sa création ;
- des points d'accès protégés aux services offerts par le noyau. Ces points d'accès sont des routines généralement écrites en C, appelés *appels système* et font l'objet de la norme POSIX ;
- aspect multi-tâche : on a toujours « la main » ;
- des ▴langages de commande◂, véritables langages de programmation permettant l'écriture de commandes complexes pouvant elles-mêmes être utilisées comme les commandes existantes ;
- un ▴système de fichier hiérarchisé◂ (fichiers, répertoires, liens physiques et symboliques) ;

► § 2.4 p. 47
► Ch. 5 p. 107
► § 2.3 p. 30

– utilisation du concept de *filtrage* et de ►*redirections*▲. C'est une des idées fondamentales d'UNIX qui en fait toute la souplesse et la puissance. Les filtrages et les redirections constituent les principes de bases permettant d'appliquer la philosophie de la boîte à outils d'UNIX : écrire des commandes élémentaires et homogènes, pouvant communiquer entre elles. § 3.2 p. 62 ◄

1.3 Les logiciels libres

Cette section propose une présentation de ce qu'on appelle les *logiciels libres*. Dans un premier temps, on trouvera les différents types de protection de logiciels que l'on peut rencontrer ; puis dans un deuxième temps, à travers un historique du projet GNU, on découvrira le principe fondamental d'une licence de type logiciel libre.

1.3.1 Les différents types de logiciels

On rencontre plusieurs types de logiciels [12] :

Propriétaires : ces logiciels sont vendus et sont régis par une licence *restrictive* qui interdit aux utilisateurs de copier, distribuer, modifier ou vendre le programme en question ;

Shareware : un shareware (dont une laide traduction est partagiciel) est un programme généralement distribué sous forme de binaire et assujetti à une *période d'évaluation* permettant aux utilisateurs d'estimer si le logiciel leur convient ou non. Au-delà de cette période, l'utilisateur a l'obligation morale de rétribuer l'auteur s'il veut continuer à utiliser le programme et dans le cas contraire l'effacer de son disque. Des abus ayant été constatés, les concepteurs de sharewares ont aujourd'hui recours au bridage des logiciels ; bridage levé après rétribution ;

Freeware : ici *free* veut dire gratuit (graticiel, beurk); les freewares sont donc distribués gratuitement et peuvent être copiés sans restriction particulière ; ils ont le plus souvent la forme de binaires ;

Domaine public : un logiciel versé dans le domaine public, appartient à tout le monde, donc à personne. Chacun est libre de faire ce qu'il veut avec, sans aucune contrainte particulière.

Free software : ici le mot *free* s'entend comme *libre* et non gratuit. Le mot anglais étant le même pour désigner les deux sens, on trouve généralement la phrase : « free as free speech as opposed to free beer ». Les free softwares ou logiciels libres proposent une licence *permissive* qui tout en assurant la propriété intellectuelle du programme à l'auteur, autorise les utilisateurs à copier, distribuer, modifier et éventuellement vendre le logiciel ;

FIGURE 1.6: Richard STALLMAN

1.3.2 Historique du projet Gnu [13]

Dans les années 70, Richard STALLMAN travaille dans un laboratoire d'intelligence artificielle au MIT (Massachusetts Institute of Technology), et dans une ambiance de travail telle qu'il a l'habitude d'échanger ses programmes avec des collègues dans le but de les corriger et de les améliorer. Les programmes en question constituent le langage de commande d'un ordinateur [14], langage qui est écrit en commun, et partagés avec d'autres universitaires ou ingénieurs qui disposent également de cette machine.

Le premier événement qui a poussé Richard STALLMAN à prendre la décision de se lancer dans les logiciels libres est sans doute le don que fait la société Xerox au MIT au début des années 80 du fruit de leurs recherches : une imprimante Laser. Cette imprimante est dotée d'une meilleure résolution que l'imprimante du labo, mais « se plante » plusieurs fois par heure. Le problème est que la société Xerox refuse de fournir les sources du programme pilotant l'imprimante. Les membres du laboratoire se trouvent alors dans la situation paradoxale, où bien qu'ayant toutes les compétences pour modifier le pilote, ils n'ont pas le droit de le faire.

Les événements — lire à ce sujet le livre de Stallman *et al.* (2010) — qui vont faire naître le projet GNU reposent tous sur le fait qu'à un instant donné, l'accès au code source des programmes devient impossible. Ceci va à l'encontre de l'état d'esprit de communauté des développeurs du laboratoire. Richard STALLMAN décide alors d'écrire lui-même un système d'exploitation ! Il nomme ce système GNU, pour GNU is Not Unix [15]. La première pierre de ce projet est l'éditeur de texte Emacs, qui peu à peu fait l'objet de demandes de la part d'utilisateurs intéressés. Un serveur est mis en place sur Internet, et Stallman envoie également des bandes magnétiques

12. On trouvera une description équivalente des différents types de logiciels dans un document rédigé par Leclerq (1999).

13. Ce paragraphe est un résumé d'une partie du chapitre 7 de *Open Sources — Voices from the Open Source Revolution* (DiBona *et al.*, 1999). On trouvera également des informations très précises dans l'ouvrage de Stallman *et al.* (2010).

14. Le Digital PDP-10, l'un des ordinateurs les plus puissants de l'époque comme le note Stallman dans le livre de DiBona *et al.* (1999).

15. Notez l'humour informaticien sur une définition récursive sans critère d'arrêt ; une définition qui fait inexorablement déborder la pile.

contenant Emacs, aux utilisateurs ne disposant pas d'accès à Internet, moyennant une contribution.

C'est en 1983 que STALLMAN annonce officiellement sur Internet le projet GNU et crée en parallèle, la Free Software Foundation (FSF) qui a pour but de récolter des fonds en vendant les logiciels sur un support de stockage, et les manuels de ces logiciels.

Aujourd'hui, après plusieurs années de travail mené par des développeurs disséminés tout autour de la planète, le projet GNU, c'est-à-dire le système Unix libre que voulait Stallman en est au stade suivant :

1. tous les utilitaires d'un UNIX digne de ce nom sont finis : compilateur, éditeur de texte, et tous les outils de la boîte à outils d'UNIX ; ces outils ont aujourd'hui une excellente réputation et sont souvent utilisés en lieu et place des outils proposés par les UNIX propriétaires ;
2. le noyau (baptisé Hurd) est opérationnel depuis quelques années mais n'est pas utilisable sur un site de production.

C'est ici qu'intervient le noyau ▶*LINUX*◀, et à partir du début des années 90, on peut dire qu'il existe un système Unix libre, le système GNU/*LINUX*.

1.4 p. 11 ◀

1.3.3 Principe de la gpl

Le projet GNU donne naissance à une licence particulière appelée General Public Licence (GPL) qui spécifie les termes qui régissent un logiciel libre. Un logiciel « sous » GPL est un logiciel :

1. qu'on peut copier ;
2. qu'on peut distribuer ;
3. qu'on peut modifier ;
4. qu'on peut vendre ;

en respectant les contraintes suivantes :

1. on doit pouvoir en obtenir les sources (par exemple sur Internet) ;
2. il y est mentionné la ou les personnes qui en ont la propriété intellectuelle ;
3. on doit y trouver une copie de la GPL.

Cette licence garantit donc qu'un logiciel libre (sous GPL) le reste à jamais. Si une personne décide de vendre, modifier, distribuer le logiciel, il doit le faire en respectant les termes de la licence. Dans le cas de la vente, rien ne vous empêche de graver un cédérom avec les sources du noyau *LINUX*, la mention de copyright de l'auteur du noyau, et vendre le tout pour le prix qui vous semble le plus adapté !

1.4 Le cas de Gnu/Linux

1.4.1 Qu'est-ce que Linux ?

C'est un noyau UNIX.

LINUX est un programme dont les sources constituent un ensemble de fichiers écrits principalement en C totalisant un volume de plusieurs dizaines de mégaoctets. Ces sources constituent le *noyau* d'un système UNIX, mais ne forment pas un système

FIGURE 1.7: Linus TORVALDS

d'exploitation à part entière. Même si le noyau *LINUX* est auto-suffisant dans le cadre de l'informatique embarquée, c'est la réunion des utilitaires GNU, du noyau *LINUX*, d'une bibliothèque graphique telle que XFree86 ou XOrg, qui crée le système aujourd'hui célèbre baptisé GNU/*LINUX*.

1.4.2 Historique [16]

LINUX est né en 1991, autour d'un projet de fin d'études d'un étudiant finlandais de l'université d'Helsinki, nommé Linus TORVALDS. Ce projet est une amélioration de Minix, le système d'exploitation créé par TANNENBAUM pour le processeur Intel 386. Après quelques mois de développement, TORVALDS poste un message sur Usenet, pour annoncer publiquement son projet. Plusieurs personnes se lancent dans l'aventure et une première version est diffusée au mois d'août 1991, portant le numéro de version 0.0.1. Fin octobre 91, une version officielle est annoncée, la version 0.0.2, permettant de faire fonctionner quelques utilitaires GNU.

Aujourd'hui, à l'instar des autres logiciels développés selon le modèle des free softwares, *LINUX* est maintenu et amélioré par quelques centaines de programmeurs disséminés sur la planète. C'est un noyau UNIX, répondant à la norme POSIX, et qui a été porté sur plusieurs architectures telles que les processeurs Sparc ou Alpha. Ce noyau est disponible sous forme de sources sur le site `ftp://ftp.kernel.org` et ses miroirs :

- en *version stable* 2.4.20, (chiffre de version pair) dans lequel aucune nouvelle fonctionnalité n'est insérée ;
- en *version de développement* 2.5.30 (chiffre de version impair) qui se transformera irrémédiablement en version stable 2.6.xx. **NB** : les versions de développement ne sont pas adaptées à une utilisation normale, mais destinées aux programmeurs qui désirent tester et débugger les versions « fraîches » du noyau.

16. Informations tirées entre autres de l'ouvrage de Card *et al.* (1997).

1.5 Quelques réflexions sur les logiciels libres

Les implications de ce que l'on peut qualifier de *mouvement* pour les logiciels libres sont multiples. Le fait même de le qualifier de mouvement induit une notion de politique et de philosophie. Voici les implications qui peuvent venir à l'esprit.

Un espoir...

Ce mouvement est avant tout porteur d'un espoir, et c'est en cela que certains osent prononcer le nom de révolution, l'espoir qu'il est possible pour un groupe de personnes de créer une communauté où chacun a envie de créer pour les autres en bénéficiant du travail des autres. Les logiciels libres proposent au citoyen de construire des machines avec lui, de lui laisser la liberté de concevoir ces machines, ce morceau de pierre qu'il taillait bien avant d'être citoyen. Il a donc la liberté de modifier à une échelle relative cette société, de la créer ou d'y participer. C'est peut-être sur le plan de la politique qu'on peut trouver la plupart des composants d'une révolution. Gardons bien sûr à l'esprit que ce « bouleversement » de société s'opère dans un milieu très fermé qu'est celui de la production de logiciel.

Ouverture et diffusion de savoir...

Dans un secteur certes précis de la science, la transparence — qualifiée également d'*ouverture* — est de mise, et le scientifique dispose grâce à ces logiciels d'une source de savoir qui n'est ni retenue, ni capitalisée à des fins de profits. Même si tout système finit par être perverti, ce qui se dégage de ce mouvement est la notion de *diffusion du savoir*, une des notions les plus pures qui dans un monde idéal, devrait être l'apanage du milieu de la recherche et de celui de la technologie.

Une autre vision d'Internet...

C'est le réseau Internet qui a permis et qui permet encore aujourd'hui à ce modèle de développement d'être viable. Il paraît important de souligner qu'Internet n'est pas seulement un espace sur lequel se ruent des entrepreneurs, destiné uniquement au commerce ou à la publicité, mais qu'il a été et qu'il est toujours également un lieu où un certain nombre de personnes communique et travaille en commun pour le bien d'une communauté, en fournissant à titre gracieux des informations scientifiques et techniques, sous forme de manuels et de forums de discussions.

Des logiciels alternatifs

Dans un cadre plus politique, on peut également affirmer que l'ensemble de ces logiciels constitue également une *alternative* aux géants de l'édition de logiciels dont la société Microsoft fait incontournablement partie. Cette alternative vient probablement de questions que l'on peut être amené à se poser, en tant qu'utilisateur de logiciels :

- dois-je accepter que certains logiciels « plantent » ?
- dois-je accepter de ne pas pouvoir importer des données d'un logiciel d'un fabricant vers un autre d'un fabricant différent ?
- dois-je accepter que certaines versions de mon traitement de texte préféré ne soient pas compatibles entre elles ?

- dois-je accepter de changer de version de mon système d'exploitation, à chaque nouvelle version de mon traitement de texte préféré ?
- dois-je accepter de changer d'ordinateur à chaque version de mon système d'exploitation préféré ?

On pourra ainsi se positionner contre les formats propriétaires de stockage de fichiers qui emprisonnent les utilisateurs dans un certain type de logiciels, et plus précisément dans l'utilisation de logiciels d'une marque particulière. L'alternative se justifie par le refus des logiciels dont les passerelles vers l'extérieur sont sciemment — dans un but commercial malsain — bloquées voire inexistantes. Par essence, les logiciels libres sont ouverts et peuvent théoriquement, et le plus souvent effectivement, communiquer entre eux, puisque le format de stockage est également ouvert.

Du jeu...

Une des motivations des hackers de la communauté des logiciels libres est sans aucun doute le plaisir que procure l'amusement, le jeu. De la même manière qu'un enfant trouve du plaisir à assembler des pièces de Lego ou de Meccano, et se satisfait de sa construction aussi humble qu'elle soit, un développeur de programme tire du plaisir à ce jeu qui consiste à trouver les meilleurs mots pour causer à ce `@#{&!?` d'ordinateur. Cette recherche ludique s'appelle « hacker » (cf. plus bas).

Un peu d'art...

À l'époque où Bernard Gaulle distribuait librement l'extension *french* de LaTeX, il finissait la documentation de son logiciel en écrivant : « En espérant avoir été utile à tous ». Même si on peut s'interroger sur les véritables motivations de ces milliers de bitouilleurs[17], cette motivation provient d'un profond humanisme et sans doute de l'éternelle interrogation de l'existentialiste. Si l'art est la capacité propre à l'être humain de créer en dehors de toute contingence de survie, alors on peut voir ce mouvement des logiciels libres comme un mouvement artistique. Le nom de la licence qui régit le logiciel Perl — la licence artistique — en est sans doute l'illustration.

Un peu de politique...

Un groupe d'ingénieurs, de chercheurs, d'enthousiastes, se regroupent et mettent en commun leur passion pour l'informatique et la programmation pour mener à bien un projet ambitieux. Ce projet, ou l'ensemble de ces projets, (le projet gnu, *LINUX*, LaTeX pour ne citer que ceux-là) se trouvent être des concurrents sérieux des logiciels propriétaires. Nous sommes donc aujourd'hui dans une situation intéressante où un groupe de personnes a montré que l'on peut diffuser un produit phare de la révolution informatique sans avoir recours aux mécanismes du capitalisme. Utiliser les logiciels libres peut donc aussi constituer un choix politique.

En finir avec les hackers...

Contrairement à une idée répandue par les médias, le terme de hacker désigne initialement un utilisateur enthousiaste de l'informatique et dont le but est d'améliorer les systèmes existants. Il n'y a donc pas au départ de connotation malsaine dans ce terme. Il ne faudra pas le confondre avec le terme de *cracker* — que l'on peut

17. Oui, oui certains ont traduit le mot *hacker* en « bitouilleurs », bidouilleur de bit...

traduire par pirate — individu dont l'activité (illégale) est de tenter de pénétrer sur des systèmes. STALLMAN se qualifie lui-même de *system hacker*, pour expliciter ses activités de modification de système d'exploitation de l'ordinateur PDP au sein de son labo. Le hacker est celui qui veut savoir comment les choses marchent, qui veut comprendre ce qui se cache derrière une technologie particulière — pas nécessairement l'informatique (Raymond, 2000).

1.6 Actualité et avenir des logiciels libres

Aujourd'hui [18], les logiciels libres sont en pleine effervescence et le terme de « station de travail » n'a plus de valeur. Il y a encore quelques années, on entendait par station de travail, une machine onéreuse, inaccessible pour le particulier, gérée par un système UNIX. À l'heure actuelle, les ordinateurs personnels dotés d'un système tel que *LINUX* peuvent tenir tête aux dites stations de travail, en termes de puissance et fiabilité. Il y a quelques années mon laboratoire de recherche avait constaté qu'en s'équipant de PC munis de *LINUX*, on disposait de matériels 5 fois moins chers et 7 fois plus puissants que des stations de travail vendues par un revendeur bien connu [19], dont nous tairons le nom ici.

Un aspect intéressant de *LINUX* est qu'il montre aux utilisateurs, qu'une machine n'est pas condamnée à être gérée par *un* système d'exploitation particulier. L'utilisateur a en effet le choix de coupler un processeur de son choix, avec un système d'exploitation particulier. On peut à titre indicatif, installer sur une machine munie d'un processeur Intel, le système GNU/*LINUX*, FreeBSD, BeOS, Solaris® pour ne citer qu'eux. Un PC n'est donc pas condamné à supporter Windows®.

1.6.1 Le problème des « drivers »

Il y a encore quelque temps, lorsqu'un fabricant de cartes (vidéo, adaptateur SCSI, acquisition, etc.) sortait un produit sur le marché, il fournissait un pilote logiciel (*driver*) pour des systèmes d'exploitation ciblés. Parmi ces systèmes, *LINUX* ne figurait pas. Aujourd'hui, ces fabricants sont conscients que les utilisateurs de *LINUX* représentent un potentiel d'achat de matériel non négligeable. C'est pourquoi, on voit actuellement les constructeurs adopter deux positions :

- créer eux-mêmes des pilotes pour *LINUX* (une célèbre compagnie de carte vidéo 3D fournit un driver pour ses cartes) ;
- divulguer les spécifications de leurs matériels, de manière à ce que les développeurs de la communauté puisse programmer ces pilotes. Dans ce cas précis, il y a toujours *un délai de plusieurs mois* avant qu'une carte soit opérationnelle sous *LINUX*.

Dans le pire des cas, les fabricants ne divulguent pas les spécifications de leur matériel. Il est alors impossible d'utiliser le matériel en question, à moins de procéder à l'ingénierie inverse (*reverse engineering*) qui n'est pas toujours légale dans certains pays. Il est en tous cas intéressant de noter que plusieurs fabricants ont « cédé » sous la pression de la communauté ; parmi ceux-là, les sociétés Matrox et Adaptec avaient

18. 24 décembre 2010.

19. Qui offrait du matériel plus fiable et généralement accompagné d'un contrat de maintenance, pour être tout à fait honnête dans la comparaison...

fait grand bruit lorsqu'elles avaient enfin accepté la divulgation des spécifications de leurs matériels.

1.6.2 Le problème des « virus »

Les virus informatiques se transmettent par l'intermédiaire de fichiers exécutables que les utilisateurs s'échangent. On ne dispose bien évidemment pas du code source de ces exécutables qui contiennent un virus. Dans un environnement où tous les programmes sont installés sous forme de source, si un virus existe, il est alors « visible » dans le source dans le sens où le code pernicieux peut être *a priori* localisé. Cette visibilité permet son éradication immédiate. En outre il semble que les virus ne soient pas aussi nombreux sur un système comme *LINUX*. Ceci parce que les virus, s'ils mettent en évidence une certaine connaissance de la programmation pour son auteur, montrent surtout un état d'esprit nihiliste. Dans le cadre des logiciels libres, on peut exprimer son talent en proposant une application à la communauté. Il faut cependant garder à l'esprit les considérations suivantes :

- un logiciel sous forme de source, téléchargé depuis un site se doit d'être authentifié comme ne contenant pas de virus. C'est la raison pour laquelle les sites primaires qui distribuent un package particulier fournissent également une signature électronique qui permet de vérifier que l'archive contenant le logiciel n'a pas été modifiée ;
- dans la situation où l'on ne dispose pas de signature, il faudra se poser les questions suivantes :
  1. le programme que je viens de télécharger émane-t-il d'un site ayant un caractère « officiel » ?
  2. le programme lors de son installation doit-il être exécuté avec des privilèges du super utilisateur ?
- en cas de problème, il faut souligner que la communauté a, par le biais du réseau Internet, une capacité à réagir très rapidement, en publiant des annonces ou d'éventuelles mises à jour des programmes concernés.

Aujourd'hui la tendance n'est pas aux virus, mais à *l'exploitation des trous de sécurité* des services réseau proposés par le système. L'idée est donc de détourner des défauts de programmation connus d'un logiciel réseau pour obtenir un accès privilégié sur des machines distantes. Encore une fois, Internet permet aux développeurs et aux distributeurs de ces outils de réagir rapidement face à ce genre de défaut. Un organisme centralisant les annonces de faille dans différents logiciels est le CERT (dont le nom provient de *computer emergency response team*) disposant d'un site web : `http://www.cert.org`. Cet organisme ayant pour origine un incident survenu en 1988, recense de manière officielle les vulnérabilités des logiciels communément utilisés pour la communication.

1.6.3 De l'utopie à la loi du marché : du Free Software à l'Open source

Aujourd'hui le terme *free software* est remplacé peu à peu par le terme *open source* qui lève l'ambiguïté sur le mot *free* (libre ou gratuit ?) et insiste sur l'aspect d'ouverture de ces logiciels. Il faut imaginer que lorsque STALLMAN crée le projet GNU, il est dans un état d'esprit idéaliste :

> « My work on free software is motivated by an idealistic goal: spreading freedom and cooperation. I want to encourage free software to spread, replacing proprietary software which forbids cooperation, and thus make our society better [20]. »
>
> Stallman (2009, Copyleft: pragmatism idealism).

STALLMAN fait partie des partisans des logiciels libres les plus radicaux. C'est pourquoi la GPL est très contraignante pour assurer qu'un logiciel libre le reste quoi qu'il arrive. Parmi les contraintes fortes de la GPL, l'une d'elles interdit de distribuer un logiciel propriétaire utilisant une bibliothèque libre. En d'autres termes, une société désirant vendre un logiciel non libre, ne peut utiliser une bibliothèque libre pour développer son logiciel. Cette contrainte a été levée avec une version moins restrictive : la LGPL (*lesser general public license*).

Aujourd'hui pour ne pas effrayer l'entrepreneur avec la notion de gratuité ou l'idée de communauté libre à but non lucratif, le terme *Open Source* a été créé. Ce terme dont le porte parole est sans doute Eric Raymond (cf. `http://www.opensource.org`), désigne tout logiciel régi par une licence compatible avec l'idée des logiciels libres. La définition de l'open source permet à un développeur de définir une licence compatible avec l'idée des logiciels libres. Les principales idées sont les suivantes (DiBona *et al.*, 1999, chapitre 14) :

- la libre redistribution, c'est-à-dire la possibilité de distribuer à titre gracieux ou non le logiciel protégé par la licence ;
- disponibilité du code source, soit dans le package de distribution soit par téléchargement ;
- une licence open source doit autoriser (mais n'oblige pas) de mettre les modifications sous les termes de cette licence. Ce point est très différent de ce que stipule la GPL, qui impose que les travaux dérivés soient sous GPL.

Les licences compatibles avec cette définition sont, parmi une petite vingtaine que recense le site `http://www.opensource.org` : la licence BSD, la licence X, la licence artistique (de Perl), la licence de Netscape, la GPL et la LGPL.

L'intérêt qu'ont vu les porte-paroles de l'open source réside dans le fait qu'on peut présenter les logiciels libres comme une solution économiquement viable et non comme le projet d'une communauté d'informaticiens libertaires ou communistes (termes effrayant les hommes portant des cravates (DiBona *et al.*, 1999, chapitre 14)). Il existe en fait beaucoup de sociétés qui ont adopté le modèle open source (ou logiciel libre ce qui est — encore une fois — identique malgré les dissensions qui peuvent exister entre STALLMAN et RAYMOND) ; citons parmi celles-là :

- *Cygnus Solution* qui propose un kit de développement basé sur le compilateur et le debugger de GNU ;
- *RedHat Software* qui vend une distribution *LINUX* bien connue ;
- *Netscape Communication* qui a adopté en 1998, le modèle des logiciels libres pour développer son navigateur ;
- *Corel* a créé sa propre distribution *LINUX* qui a pour but de porter ses célèbres outils de dessins, retouche d'image, etc.
- la société *Sun* (qui vient d'être rachetée par *Oracle*) s'est investie dans deux logiciels phares : MySql et OpenOffice ;

20. Traduction : « Mon travail autour des logiciels libres est motivé par un but idéaliste : diffuser la liberté et la coopération. Je veux encourager les logiciels libres à se développer, pour remplacer les logiciels propriétaires qui interdisent la coopération, et donc rendre notre société meilleure. »

– on notera également que les sociétés *IBM*, *Apple*, *HP* et *SGI* ont montré un intérêt prononcé pour le modèle open source, en sponsorisant ou choisissant d'utiliser les logiciels libres sur leur plate-forme.

Ces entreprises sont en mesure de créer du profit, car elles vendent un service aux clients. Elles bénéficient du modèle open source, dans la mesure où le client peut éventuellement modifier le produit, lequel produit est maintenu activement par une communauté de développeurs.

1.6.4 Des brevets sur les logiciels

Les brevets sur les logiciels constituent aujourd'hui une des menaces sérieuses au développement des logiciels libres. Il faut savoir qu'aujourd'hui il est possible de déposer des brevets sur les logiciels aux États Unix [21], mais que cela est interdit en Europe. La directive européenne visant à autoriser le dépôt de brevets sur des logiciels a été rejetée le 6 juillet 2005 par le parlement européen.

La création des brevets avait pour but initial de diffuser l'information scientifique et technique tout en assurant pendant un certain temps des royalties au détenteur du brevet. L'idée était donc d'inciter les créateurs à diffuser leur savoir plutôt que de le garder secret. Aujourd'hui l'utilisation intensive des brevets est une véritable guerre commerciale dont le but est d'en déposer le plus possibles et d'attaquer ensuite en justice les utilisateurs « frauduleux » d'idées qui auraient fait l'objet d'un dépôt.

Pour illustrer l'aspect pour le moins douteux de telles pratiques, on pourra mentionner la société Microsoft qui a récemment déposé un brevet sur le fait du cliquer sur une icône ou un bouton [22] et un autre sur la souris à molette [23]...

1.6.5 Quelques beaux exemples

Pour clore ce chapitre, il m'est apparu nécessaire de présenter ce que je considère comme les plus beaux projets de logiciels libres. Ces logiciels sont très utilisés aujourd'hui.

TEX et LATEX : ce système de composition de document [24], est à mon avis (partial et subjectif) un des plus beaux projets informatiques et scientifiques. TEX et LATEX permettent de rédiger des documents avec une très grande souplesse et une qualité professionnelle (lire l'excellent ouvrage de Lozano (2008) pour s'en persuader) ;

***LINUX* :** a permis et permet encore de faire découvrir aux utilisateurs curieux de l'informatique un autre système d'exploitation : UNIX. Avoir une vision alternative des systèmes assure la pluralité et évite de se diriger vers un totalitarisme culturel et technique. *LINUX* illustre également qu'à partir d'un développement quasi chaotique (Eric RAYMOND parle de *bazar*), il est sorti un noyau UNIX solide et efficace ;

GNU : une expérience de quelques années avec les outils GNU m'incite à souligner la qualité de ces outils ; pour ce qui concerne les utilitaires UNIX, ils offrent la

21. Une coquille détectée par un lecteur que je laisse en l'état...
22. Brevet *US patent* n° 6727830.
23. Brevet n° 6700564.
24. Qui a permis de composer le texte que vous avez sous les yeux.

plupart du temps beaucoup plus d'options que les versions des UNIX propriétaires. Le compilateur C et son compère l'éditeur Emacs, sont également des outils qui gagnent à être connus ;

Perl : Larry WALL le créateur de Perl note dans la préface du livre de Schwartz (1995) : « Perl a été créé pour des gens comme vous, par quelqu'un comme vous, en collaboration avec de nombreuses personnes comme vous. La magie de Perl s'est tissée collectivement, point par point, pièce par pièce, dans le cadre très particulier de votre mentalité. Si vous trouvez que Perl est un peu bizarre, cela en est peut-être la raison. ». Ce langage très puissant est un langage interprété utilisé pour l'administration système, la création de page HTML, et plein d'autres choses ; Perl fait partie de ces logiciels qui, à l'instar de LaTeX, sont composés d'un noyau de base et d'une multitude de composants créés par les utilisateurs enthousiastes ;

Gtk/Gimp : the GNU *image manipulation program*, un logiciel qui se présente comme un concurrent de Photoshop de la société Adobe. Gimp a été conçu à l'aide de la bibliothèque graphique Gtk (Gimp *toolkit*) (voir `http://www.gimp.org` et `http://www.gtk.org`);

Autour du réseau : les logiciels apache (serveur web), bind (serveur dns) et sendmail (transport de messages), sont les logiciels qui sont aujourd'hui des standards de fait. À titre d'exemple, plus de la moitié des serveurs web d'Internet sont hébergés par apache.

2 Petit guide de survie

Sommaire

2.1 Le shell
2.2 Utilisateurs
2.3 Le système de fichiers
2.4 Processus
2.5 Quelques services


```
eval $(echo \
"3chA+2A96ADB+*+8157+7AE19A^395C304B"  |\
tr "1+2*3456789ABCDE^0" "a\ b\|efijlmnorsuz@.")
```

'Tain c'est pô convivial !.

La convivialité d'un système UNIX réside dans la souplesse et la puissance des outils dont on dispose pour dialoguer avec le système. Il s'agit certes d'une convivialité à laquelle peu d'utilisateurs sont habitués ; essentiellement parce qu'elle demande un investissement sur le plan de la documentation et de l'apprentissage. Ce chapitre a donc pour but de présenter les fonctionnalités de base d'un système UNIX sur le plan de son *utilisation* (c.-à-d. qu'il ne sera pas question ici d'administration d'un système). On verra donc dans un premier temps une présentation succincte de l'interface de base à savoir le *shell*, suivront des présentations des concepts d'utilisateur, de système de fichiers, de processus. Le chapitre est clos par une explication rapide des services de base du système (impression, tâches planifiées, ...).

2.1 Le shell

Le *shell* est une interface avec le système UNIX. Il offre à l'utilisateur l'interface de base avec le système d'exploitation. L'étymologie du mot nous apprend qu'on peut l'imaginer comme une *coquille* englobant le noyau et ses composants. Le shell est également un *programme* qu'on appelle *interpréteur de commandes*. Ce programme tourne dans une fenêtre ou sur une console en mode texte. Dans une fenêtre d'un environnement graphique, il a l'allure de la figure 2.1 page suivante. On parle également de *terminal* (ou d'émulateur de terminal) pour désigner un écran ou une fenêtre dans laquelle est exécuté le shell. Il existe plusieurs shells dans le monde UNIX, les plus courants sont :

- `sh` : fait initialement référence au premier shell d'UNIX conçu par Steve *Bourne*, utilisé notamment pour les scripts système ;
- `ksh` : le Korn shell ;
- `csh` : le C shell (dont la syntaxe rappelle vaguement celle du C pour ce qui est des structures de contrôle) ;

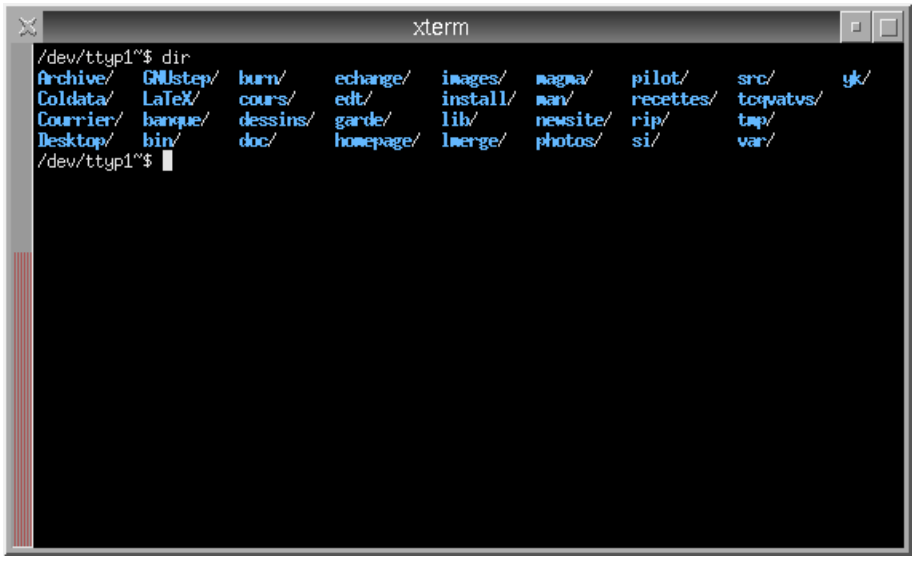


FIGURE 2.1: Un shell dans une fenêtre

- `bash` : le shell de GNU [1] qui est, comme mentionné dans la page de manuel, « trop gros et trop lent ». C'est malgré tout celui sur lequel nous nous attarderons dans ce manuel ;
- `tcsh` : le Tenex C shell contient tout ce qu'apporte `csh` avec des fonctionnalités supplémentaires notamment une édition plus aisée de la ligne de commande ;
- `zsh` le Zorn shell contenant un langage de programmation plus évolué que `bash` et des fonctionnalités de complétions avancées [2] ;
- ...

Le shell est utilisé le plus souvent de manière interactive, pour passer des *commandes* au système. Pour signifier que le shell est prêt à recevoir ces commandes, il affiche un *prompt*. Ce prompt peut contenir un nombre variable d'informations selon la ▲configuration◀, et nous y ferons référence de la manière suivante :

► § 6.1 p. 143

```
$
```

On peut illustrer le fonctionnement d'un shell, avec l'algorithme suivant :

```
Pour toujours Faire
    Afficher le prompt et attendre une commande
    Vérifier sa syntaxe
    Si la syntaxe est correcte Alors
        exécuter la commande
    Sinon
        afficher un message d'erreur
```

Notons enfin que si nous présentons ici le shell dans son utilisation interactive, il est également utilisé sous forme de ▲*scripts*◀, dans ce cas les commandes passées au

► 5.2 p. 109

1. Bash signifie « Bourne Again Shell » : un jeu de mot avec la construction anglaise « born again » qui signifie renaissance.
2. Votre serviteur n'a aucune expérience dans ce shell et vous prie de bien vouloir excuser par avance l'absence de script `zsh` dans ce manuel...

système peuvent être enregistrées dans des fichiers qui peuvent ensuite être exécutés. En cela le langage de commande peut être utilisé comme un langage interprété.

2.1.1 Qu'est-ce qu'une commande ?

Exécuter ou lancer une commande, consiste de manière synoptique en ceci :

⟨*nom-commande*⟩ ⟨*options*⟩ ⟨arg_1⟩ ⟨arg_2⟩ ... ⟨arg_n⟩ Entrée
...
résultat de la commande sur le terminal
...

⟨*nom-commande*⟩ est le nom de la commande à exécuter ; cette dernière peut accepter un certain nombre d'options dont la syntaxe est en général :

- `-`⟨*option*⟩ par exemple `-a`, ou
- `--`⟨*option*⟩ par exemple `--verbose`.

Un exemple :

```
$ ls -l guide-unix.tex  Entrée
-rw-r--r--  1 vincent users   2159 Nov  7 13:28 guide-unix.tex
$
```

Celui-ci exécute la commande `ls`[3] avec l'option `l` et l'argument `guide-unix.tex`. Le résultat de la commande `ls` est expliqué à la section 2.3.

On peut noter que commande, options et arguments[4] sont séparés par des espaces. Il faut comprendre que l'analyse de la ligne de commande par le shell est effectuée en séparant dans un premier temps les éléments (commande, options et arguments) par des espaces. Au cas où un nom de fichier contiendrait un espace, il faudra indiquer au shell que le nom est à considérer comme un seul élément en procédant comme indiqué à la section 2.1.5 page 27.

Commandes internes et commandes externes

Une commande peut être *interne* (*builtin*) au shell, ou externe. Dans le premier cas la commande est interprétée par le shell, dans l'autre il peut s'agir de n'importe quel fichier exécutable stocké dans l'arborescence. L'algorithme d'interprétation des commandes devient :

3. « Équivalent » du `dir` de MSDOS, pour afficher les propriétés d'un fichier ou le contenu d'un répertoire.
4. Du point de vue du shell une option est un argument comme un autre ayant la particularité de commencé par le caractère `-`.

```
Pour toujours Faire
    Afficher le prompt et attendre une commande
    Vérifier sa syntaxe
    Si la syntaxe est correcte Alors
        Si c'est une commande interne Alors
            l'exécuter
        Sinon
            chercher le fichier exécutable correspondant
            l'exécuter
    Sinon
        afficher un message d'erreur
```


La recherche de l'exécutable correspondant à une commande externe consiste en l'examen d'un ensemble de répertoires susceptibles de contenir l'exécutable en question. Cette liste de répertoires définie lors de la configuration du système est stockée dans une ▲variable d'environnement◀ nommée `PATH`. Dans le shell utilisé par votre serviteur :

► § 2.1.3 p. 26

```
$ type cat
cat is /bin/cat
$ type echo
echo is a shell builtin
$
```

La commande `type` nous informe que `echo` est une commande interne et que la commande `cat` ne l'est pas (*i.e.* est un programme dans le répertoire `/bin`). `type` est une commande interne du shell `bash`. L'équivalent pour la famille des C-shells est `which`.

Messages d'erreurs

Le shell et les commandes exécutées peuvent informer l'utilisateur qu'une erreur s'est produite lors de l'exécution d'une commande. Cette information concernant l'erreur se fait par un canal appelé le *flux d'erreur* qui est par défaut redirigé sur votre écran. Il existe au moins quatre situations où le shell et les commandes renvoient une erreur :

– la commande n'existe pas :

```
$ youpla
bash: youpla: command not found
$
```

– l'argument spécifié ne correspond à aucun fichier :

```
$ cp mlkj essai.tex
cp: mlkj: No such file or directory
$
```

– l'utilisateur ne possède pas les droits suffisants :

```
$ cp g.dat /
cp: cannot create file '/g.dat': Permission denied
$
```

– la commande n'est pas utilisée avec les bonnes options :

```
$ cat -l fichier.tmp
cat: invalid option -- l
Try 'cat --help' for more information.
$
```

À l'instar des boîtes de dialogue d'avertissement ou d'erreur qui surgissent dans les environnements graphiques, il est important de lire attentivement les messages pour comprendre pourquoi une commande échoue.

Composer les commandes

On peut composer deux ou plusieurs commandes à l'aide du caractère « ; » :

```
$ ls guide-unix.tex ; echo bonjour
guide-unix.tex
bonjour
$
```

Notons au passage que la commande `echo` affiche à l'écran la chaîne de caractère qui suit. Il existe deux autres opérateurs permettant de combiner deux commandes :

– l'opérateur `&&`
  ⟨*commande*$_1$⟩ `&&` ⟨*commande*$_2$⟩
  exécute ⟨*commande*$_2$⟩ si ⟨*commande*$_1$⟩ s'exécute sans erreur ;
– l'opérateur `||`
  ⟨*commande*$_1$⟩ `||` ⟨*commande*$_2$⟩
  exécute ⟨*commande*$_2$⟩ si ⟨*commande*$_1$⟩ renvoie une erreur.

Lorsqu'il est dit ci-dessus que la commande « renvoie une erreur », il s'agit plus précisément de la situation où la commande retourne une valeur différente de 0 (voir aussi § 5.2.4 page 120).

2.1.2 « Convivialité » et ergonomie

À la fin des années 80, les éditeurs de logiciels axaient leur publicité sur la « convivialité » de leurs produits. Un programme était alors dit « convivial » lorsqu'il présentait à l'utilisateur moult menus, boîtes de dialogue et autres icônes. Il s'avère cependant que pour une utilisation intensive d'un logiciel, la dite « convivialité » devient plutôt une contrainte et l'utilisateur cherche rapidement les raccourcis clavier pour une utilisation plus confortable.

D'autre part, si l'on songe que le langage de commande est un *langage* en tant que tel, il permet à l'utilisateur d'exprimer (tout) ce qu'il veut et cela de manière très souple. Ceci est à comparer avec un logiciel qui propose des fonctionnalités sous forme de menus ou de boîtes de dialogue, qui laissent finalement à l'utilisateur une marge de manœuvre souvent restreinte.

C'est pourquoi la vraie question n'est pas celle de la pseudo-convivialité — qui est plus un argument de vente auprès du grand public — mais celle de l'*ergonomie* d'un logiciel. Est-il possible d'exprimer de manière concise et rapide la tâche que l'on désire faire effectuer à la machine ? Le shell est un programme *ergonomique* et l'utilisateur qui a fait l'effort de l'*apprentissage* du langage de commande le constate

très vite[5].

Ah oui : l'utilisation du langage de commande d'UNIX repose sur l'utilisation intensive du clavier. Il est donc important de se familiariser avec ce périphérique voire d'apprendre via des méthodes adéquates à l'utiliser de manière optimale.

Dans la mesure où le pressage de touches de clavier est l'activité première de l'utilisateur UNIX, quelques « aménagements » ont été mis en place :

- *l'historique* des commandes : les touches et permettent de rappeler les commandes précédemment tapées pour éventuellement les modifier et les relancer ;
- la *complétion* des commandes : la touche (tabulation) permet de compléter les noms de commandes, les noms de fichiers et de répertoires à partir des premiers caractères de leur nom. À utiliser intensivement !

Si un jour vous vous connectez sur une « vieille » machine UNIX et/ou que vous avez à dialoguer avec un « vieux » shell ne disposant ni de la complétion ni du rappel des commandes, ce jour-là vous comprendrez que `bash` (ou le shell que vous avez l'habitude d'utiliser), malgré son caractère apparemment spartiate, est un programme vraiment *moderne*...

2.1.3 Rudiments sur les variables d'environnement

Lorsqu'un shell est exécuté par le système, un certain nombre de variables dites d'*environnement* sont instanciées. Ces variables permettent à l'utilisateur et aux programmes lancés par le shell d'obtenir plusieurs informations sur le système, la machine et l'utilisateur, entre autres. La commande `env` affiche à l'écran toutes les variables d'environnement pour le shell. En voici un extrait :

```
$ env
MAIL=/var/spool/mail/vincent
HOSTTYPE=i386
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
HOME=/home/vincent
SHELL=/bin/bash
USER=vincent
OSTYPE=Linux
$
```

On comprend aisément que ces variables d'environnement définissent respectivement, le fichier réceptacle du courrier, le type de la machine, la liste des répertoires[6] où le shell va chercher les exécutables (pour les commandes qui ne sont pas internes), le répertoire privé de l'utilisateur, le nom de l'utilisateur et le type de système d'exploitation. En préfixant le nom d'une variable par un dollar (`$`), on accède à la valeur de cette variable. Ainsi :

```
$ echo $SHELL
/bin/bash
$
```

5. On préfère rapidement presser quelques touches de clavier plutôt que de se lancer dans une vingtaine de « clic-clic » sur des demi-douzaines de fenêtres empilées les unes sur les autres.

6. Voir à ce sujet les paragraphes 6.1 page 143 et 5.4.2 page 132 qui traitent de la variable `PATH`.

est un moyen de connaître le shell qui est associé à votre utilisateur dans la base de données du système.

En réalité toutes les variables d'un shell ne sont pas des variables d'environnement, dans le sens où toutes les commandes n'y ont pas nécessairement accès. Par contre n'importe quelle variable peut devenir une variable d'environnement à l'aide de la commande interne `export`.

2.1.4 Caractères spéciaux

Un certain nombre de caractères sont dits spéciaux, car ils sont interprétés d'une manière particulière par le shell. Le plus connu de ces caractères est sans doute celui qui sépare les répertoires dans un chemin[7]. Sous UNIX, c'est le *slash* (`/`). Par exemple : `/home/users/hendrix/wahwah.dat`. Voici une liste des caractères spéciaux les plus communs du shell :

- `$` permet le mécanisme d'expansion et donc permet de faire référence à la valeur de la variable nommée après, mais aussi de ►faire des calculs◄ ou d'exécuter une autre commande (§ 3.4.2 page 75) ; § 5.2.3 p. 114 ◄
- `~` remplace le ►répertoire privé◄ de l'utilisateur ; § 2.3 p. 30 ◄
- `&` lance une commande en ►arrière-plan◄ ; § 2.4 p. 47 ◄
- `*` remplace toute chaîne de caractères ;
- `?` remplace tout caractère (voir la section suivante pour ces deux caractères) ;
- `|` pour créer un ►tube◄ ; § 3.2.2 p. 65 ◄
- `>` et `<` pour les ►redirections◄ ; § 3.2.1 p. 63 ◄
- `;` est le séparateur de commandes ;
- `#` est le caractère utilisé pour les ►commentaires◄ ; § 5.2.1 p. 109 ◄
- ...

Il est bien entendu que puisque ces caractères ont un rôle particulier, il est peu recommandé de les utiliser dans les noms de fichiers, au risque de compliquer le dialogue avec le shell (voir le paragraphe sur les mécanismes d'expansion page 61).

2.1.5 Espaces dans les noms de fichiers

Le caractère espace n'est pas un caractère spécial pour le shell, cependant son utilisation dans les noms de fichiers modifie légèrement la manipulation des commandes. Imaginons par exemple que l'on veuille effacer un fichier nommé « zeuhl wortz.txt », la commande « naïve » suivante échouera :

```
$ rm zeuhl wortz.txt
rm: cannot remove ‘zeuhl’: No such file or directory
rm: cannot remove ‘wortz.txt’: No such file or directory
$
```

car le shell tente d'effacer deux fichiers au lieu d'un seul puisqu'il délimite les mots à l'aide du caractère espace. Pour expliquer au môssieur que « zeuhl wortz.txt » désigne un seul fichier, il est nécessaire « d'échapper » le caractère espace :

```
$ rm zeuhl\ wortz.txt
```

7. Le caractère `/` n'est en réalité spécial que dans le traitement des références sur les fichiers ou répertoires, et non pour le shell lui-même.

ou d'indiquer explicitement que le nom est composé d'un seul « mot » en l'entourant de guillemets (") ou d'apostrophes (') :

```
$ rm "zeuhl wortz.txt"
```

ou :

```
$ rm 'zeuhl wortz.txt'
```

2.1.6 Caractères génériques

Il existe au moins deux moyens d'utiliser les caractères génériques (parfois dits *wildcards* ou *jokers* en anglais). Voici deux exemples en reprenant la commande `ls` :


```
$ ls guide-unix*
guide-unix.aux  guide-unix.dvi  guide-unix.log  guide-unix.tex
$
```

liste tous les fichiers du répertoire courant commençant par `guide-unix`. Et :

```
$ ls guide-unix.??x
guide-unix.aux  guide-unix.tex
$
```

liste les fichiers dont l'extension est composée de deux caractères quels qu'ils soient et terminée par un `x`. Une des étapes d'interprétation de ces commandes par le shell consiste à remplacer les jokers par leur valeur ; ainsi, la dernière commande équivaut à :

```
$ ls guide-unix.aux  guide-unix.tex
```

Ce mécanisme de remplacement des jokers par une valeur se nomme en jargon (anglais) UNIX : *pathname expansion* ou *globbing*.

Notons ici que l'on peut bien évidemment composer les caractères « * » et « ? » et qu'il existe d'autres formes de joker, comme par exemple la séquence « [...] » que nous ne présentons pas ici, et bien d'autres qui dépendent du shell utilisé. Il est pour l'instant utile de noter que ces caractères, même s'ils se rapprochent des expressions régulières présentées au paragraphe 3.5 page 77, ne doivent pas être confondues avec icelles...

2.2 Utilisateurs

UNIX a été conçu comme un système multi-utilisateur, ce qui signifie que chaque utilisateur est identifié sur le système par un utilisateur *logique* auquel correspond un certain nombre de droits ou privilèges. À un utilisateur sont, entre autres, associés :

- un numéro ou identificateur : l'uid (pour *user identifier*) ;
- une chaîne de caractère se rapprochant généralement de l'état civil, appelée *login name* ;
- le groupe auquel l'utilisateur appartient, et éventuellement d'autres groupes d'utilisateurs supplémentaires ;
- un répertoire privé (*home directory*) ;
- un shell qui sera utilisé par défaut après la connexion.

On peut savoir sous quel utilisateur logique on est connecté grâce à la commande `whoami` :

```
$ whoami
djobi
$
```

En plus de son nom, un utilisateur est identifié en interne par son *uid* ou *user identifier*. D'autre part un utilisateur appartient à un ou plusieurs groupes d'utilisateurs. Chacun de ces groupes est également identifié par son nom et un numéro appelé *gid* ou *group identifier*. On peut connaître l'identité d'un utilisateur par la commande `id` :

```
$ id lozano
uid=208(lozano) gid=200(equipe) groups=200(equipe),300(image)
$
```


l'utilisateur `lozano` appartient aux groupe `equipe` et `image`. C'est généralement l'administrateur du système qui décide de regrouper les utilisateurs dans des groupes qui reflètent la structure des personnes physiques qui utilisent les ressources informatiques. Chacun de ces groupes se voit généralement attribuer un ensemble de privilèges qui lui est propre.

Pour terminer avec les utilisateurs, sachez qu'en général un utilisateur ne peut intervenir que sur un nombre limité d'éléments du système d'exploitation. Il peut :

- créer des fichiers dans une zone particulière appelée *home directory* ou répertoire privé ou tout simplement *home*. En fonction de la configuration du système les utilisateurs pourront bien sûr écrire dans d'autres zones. Le répertoire privé est désigné dans le shell par « ~ » :

  ```
  $ echo ~
  /home/utilisateurs/djobi
  $
  ```
- influer sur le cours des programmes qu'il a lui-même lancé ;
- autoriser ou interdire la lecture ou l'écriture des fichiers qu'il a créés.

Il ne peut pas :

- effacer les fichiers d'un autre utilisateur sauf si ce dernier l'y autorise (en positionnant les droits corrects sur le répertoire contenant le fichier en question) ;
- interrompre des programmes lancés par un autre utilisateur (là aussi, sauf si il y est autorisé par l'utilisateur en question) ;
- créer des fichiers « n'importe où. »

Par conséquent une machine gérée par un système d'exploitation de type UNIX ne peut pas être compromise *par hasard ou maladresse* par un utilisateur. Puisqu'il faut bien (maintenance, mise à jour, etc.) que certains fichiers soient modifiés, il existe au moins un utilisateur possédant tous les privilèges sur le système : l'utilisateur dont l'uid est 0 portant généralement le nom de `root` (car son répertoire racine était souvent installé à la racine (*root* en anglais) du système de fichier). Cet utilisateur a la possibilité et le droit d'arrêter le système, d'effacer tous les fichiers (!), de créer des utilisateurs, et bien d'autres choses qui constituent ce que l'on nomme communément *l'administration système.* Dans une structure où le nombre de machines à administrer est conséquent, il se cache plusieurs personnes physiques derrière cet uid 0.

2.3 Le système de fichiers

Le système de fichiers d'UNIX est une vaste arborescence dont les nœuds sont des répertoires et les feuilles des fichiers. Le terme de fichier s'entend ici dans un sens très large, puisque sous UNIX, un fichier peut contenir des données, mais peut aussi être un lien sur un autre fichier, un moyen d'accès à un périphérique (mémoire, écran, disque dur, ...) ou un canal de communication entre processus. Pour embrouiller un peu plus le tout, les répertoires sont eux aussi des fichiers. Nous nous intéresserons ici aux fichiers normaux (*regular* en anglais), c'est-à-dire ceux qui contiennent des données ou sont des exécutables, ainsi qu'aux liens et aux répertoires.

2.3.1 Référencement des fichiers et des répertoires

Pour manipuler un fichier ou un répertoire d'un système de fichiers, l'utilisateur a besoin de les *nommer*, en d'autres termes il a besoin de pouvoir *désigner* un fichier particulier de l'arborescence des fichiers et ceci de manière univoque. Il existe pour ce faire, deux moyens : les *références absolues* et les *références relatives*, toutes deux décrites ci-après.

Référence absolue

Pour désigner un élément de l'arborescence en utilisant une référence absolue, on part de la racine du système de fichier et on « descend » jusqu'au fichier ou répertoire. De cette manière on désignera les fichiers `bidule.dat` et `truc.txt` de l'arborescence donnée en exemple à la figure 2.2 par :

```
/users/bidule.dat
/home/eleves/truc.txt
```

Le caractère `/` joue deux rôles :
- d'une part il désigne le *répertoire racine* (*root directory* en anglais) ;
- d'autre part il fait usage de *séparateur* de répertoires dans l'écriture des références de fichiers et répertoires.

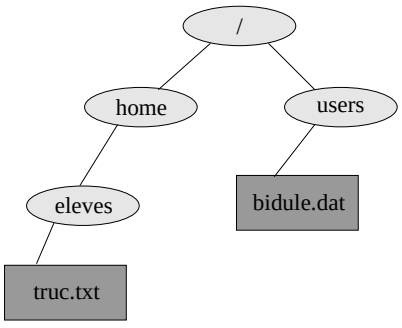


FIGURE 2.2: Une arborescence de fichiers pour l'exemple

Pour désigner un répertoire on utilisera le même principe, ainsi les répertoires `eleves` et `users` ont comme référence absolue :

```
/home/eleves
/users
```

Référence relative

Il est souvent utile de désigner les fichiers et répertoires grâce à une *référence relative*. Dans ce cas on référencera un fichier ou un répertoire *relativement* à un répertoire de base. Par exemple :

- si `/users` est le répertoire de base, alors il est possible de référencer les fichiers `bidule.dat` et `truc.txt` comme suit :
  `./bidule.dat`
  `../home/eleves/truc.txt`
- si `/home` est le répertoire de base, alors on peut les référencer comme suit :
  `../users/bidule.dat`
  `./eleves/truc.txt`

Deux répertoires particuliers existent dans chaque répertoire :

- le répertoire `.` qui désigne le répertoire courant ;
- le répertoire `..` qui désigne le répertoire parent.

On pourra noter que l'écriture `./bidule.dat` est redondante et peut être abrégée en `bidule.dat`. Enfin, le répertoire de référence dont il a été question est souvent le répertoire dit *courant* ou répertoire de travail (*working directory* en anglais). C'est le répertoire associé à chaque processus. C'est aussi le répertoire « dans lequel on se trouve » lorsqu'on lance une commande.

Le répertoire « . » a également une utilisation importante lorsqu'on veut ▶exécuter◀ un programme situé dans le répertoire courant, ce dernier ne se trouvant pas dans la liste des répertoires de recherche de la variable `PATH`.

§ 5.4.2 p. 132 ◀

Dans toutes les commandes UNIX qui manipulent des fichiers ou des répertoires, l'utilisateur est libre de désigner ces fichiers ou répertoires par le truchement des références absolues ou des références relatives. Le choix (relative ou absolue) est souvent dicté par le nombre le plus faible de touches de clavier à presser...

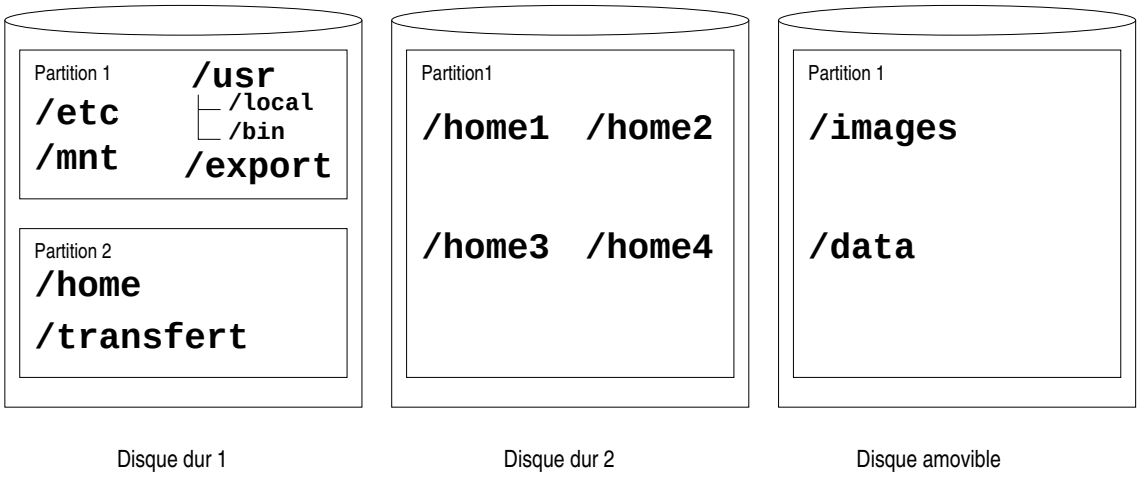
2.3.2 Arborescence

Pour comprendre le fonctionnement de l'arborescence d'un système de fichiers UNIX, supposons que la machine que nous utilisons soit composée des deux disques durs et du disque amovible de la figure 2.3b. Sous UNIX, une telle configuration de disques peut se présenter sous la forme de la figure 2.3a. En jargon UNIX, on dit que chaque partition est « montée sur » (de la commande `mount`) ou « greffée sur » un répertoire particulier. Dans notre exemple :

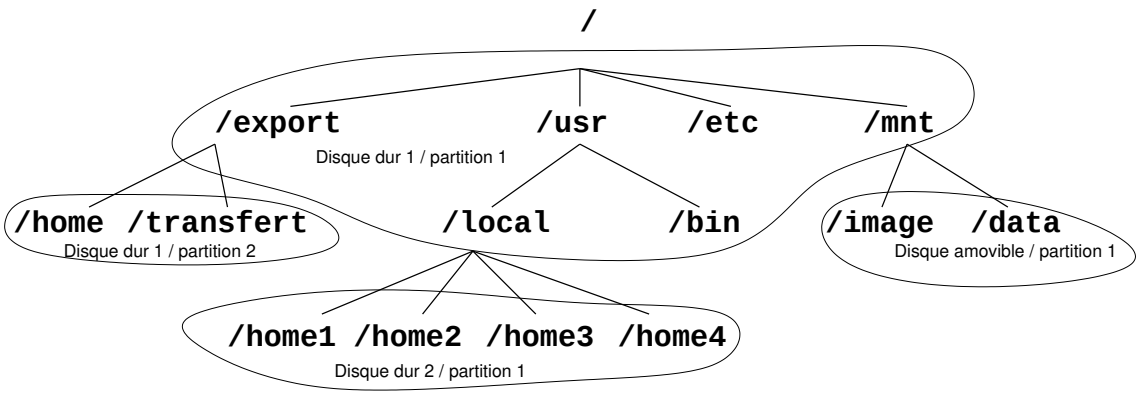
- la partition 1 du disque dur 1 est montée sur la racine (`/`) ;
- la partition 2 du disque dur 1 est montée sur `/export` ;
- la partition 1 du disque dur 2 est montée sur `/usr/local` ;
- la partition 1 du disque amovible est montée sur `/mnt`.

Ce qu'il faut retenir, c'est que quelle que soit la configuration des disques sous-jacente au système, il n'y a toujours qu'une seule racine (répertoire `/`) et donc une seule arborescence. Chaque partition est greffée en un point particulier de l'arborescence globale — on parle également de *filesystem* pour désigner la sous-arborescence greffée [8].

8. Rappelons que sous d'autres systèmes (la famille des fenêtres pour ne pas les nommer) on aurait eu dans notre étude de cas, les disques logiques `C:` et `D:` pour le premier disque dur, `E:` pour le deuxième disque dur et sans doute `F:` ou `A:` pour le support amovible ; ainsi qu'une arborescence



(a) Partionnement des disques



(b) Montages

FIGURE 2.3: Exemple de partitionnements et montages correspondant.

Notons également que différents mécanismes existent sous UNIX pour greffer des systèmes de fichiers *distants* c.-à-d. résidant sur une machine accessible via le réseau. Le plus utilisé actuellement est le système NFS (*network filesystem*) introduit par Sun.

Chaque nœud de l'arborescence est identifié en interne de manière univoque par deux nombres :

1. le numéro de la partition ;
2. le numéro du nœud appelée *inode* ou *index node*.

En d'autres termes, chaque fichier possède un numéro qui est unique sur la partition où il réside. On peut visualiser l'inode d'un fichier grâce à l'option `i` de la commande `ls` :

```
$ ls -i *.tex
65444  guide-unix.tex
$
```

ici, 65444 est l'inode du fichier `guide-unix.tex`. Sur le système où est tapé ce document, les fichiers suivants ont le même inode :

```
$ ls -i /lib/modules/2.2.12-20/misc/aedsp16.o
     19 /lib/modules/2.2.12-20/misc/aedsp16.o
$ ls -i /usr/X11R6/man/man1/identify.1
     19 /usr/X11R6/man/man1/identify.1
$
```

ce qui s'explique par le fait que le répertoire `lib` appartient à la partition `/` alors que `X11R6` appartient à une partition différente montée sur le répertoire `/usr`. Pour examiner les partitions de notre système, on peut utiliser la commande `df` avec l'option `-h` (pour *human readable*) :

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda1             194M   42M  142M  23% /
/dev/hda7             1.1G  566M  508M  53% /home
/dev/hda5             791M  606M  144M  81% /usr
/dev/sda4              96M   95M  584k  99% /mnt/zip
/dev/hdb              643M  643M     0 100% /mnt/cdrom
$
```

Cette commande affiche donc pour chaque partition le fichier *device* associé (sous la forme `/dev/xxx`) sa taille respective ainsi que son taux d'occupation. Les informations indiquant où doit être montée chacune des partitions sont généralement stockées dans un fichier nommé `/etc/fstab`.

L'option `-h` est une option de la version GNU de la commande `df`. Votre sarcastique serviteur vous invitera donc à méditer sur ces programmes censés produire des résultats « lisibles par un être humain ». Mais ne vous méprenez pas, les systèmes UNIX dont la commande `df` ne dispose pas de l'option `-h` sont quand même destinés aux humains...

propre associée à chacune de ces lettres. Rappelons en outre que ces lettres sont une séquelle du tristement célèbre Disk Operating System.

2.3.3 Privilèges

Il faut savoir que sous UNIX, on associe à chaque fichier trois types de propriétaires :

- l'utilisateur propriétaire ;
- le groupe propriétaire ;
- les autres utilisateurs.

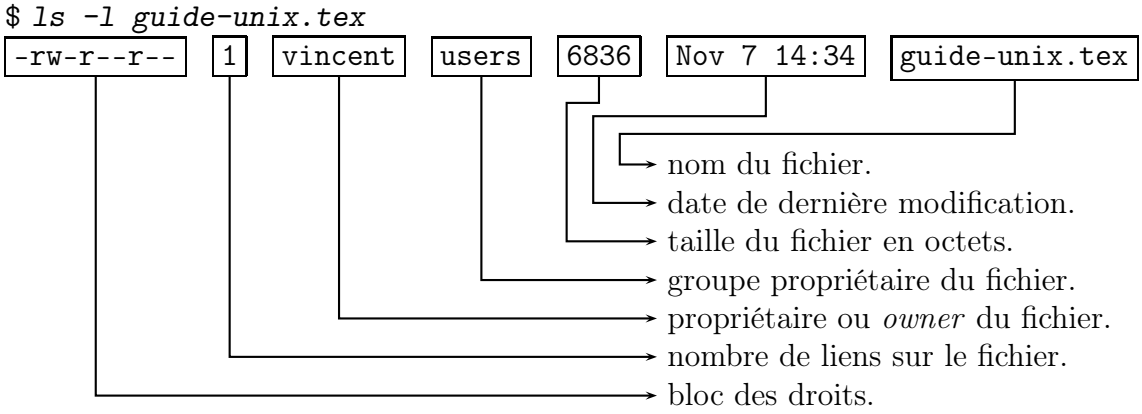
La norme POSIX apporte une extension à ce principe connue sous le nom *acces control list* (▲ACL◀), permettant de recourir si nécessaire à un réglage plus fin que les trois entités ci-dessus. Cette extension est présentée au paragraphe 2.3.10 page 44.

► § 2.3.10 p. 44

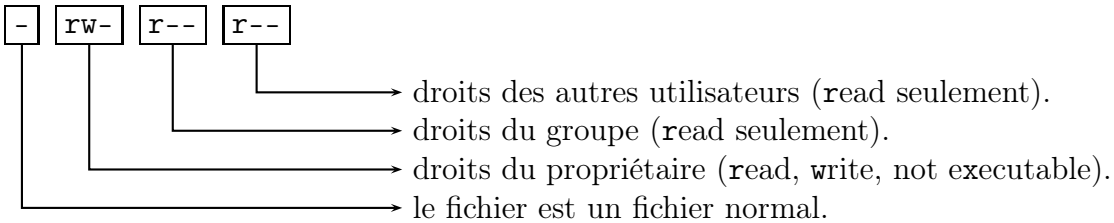
Chaque fichier possède un certain nombre d'attributs. Un sous-ensemble de ces attributs a trait aux *privilèges* ou *droits* qu'ont les utilisateurs lors de l'accès à un fichier donné. Il y a trois types de privilèges :

1. le droit de lire (read) signalé par la lettre `r` ;
2. le droit d'écrire (write) signalé par la lettre `w` ;
3. le droit d'exécuter (execute) signalé par la lettre `x`.

Ces trois types de droits associés aux trois types de propriétaires forment un ensemble d'informations associées à chaque fichier ; ensemble que l'on peut examiner grâce à la commande `ls` :



Nous verrons au paragraphe 2.3.9 page 42 la signification du terme « lien » sur un fichier. Examinons d'un peu plus près le bloc des droits :



Autre exemple :

```
$ ls -l /usr/bin/env
-rwxr-xr-x   1 root  root   6852 Aug 18 04:31 /usr/bin/env
$
```

Le fichier `/usr/bin/env` appartient à l'utilisateur `root` et au groupe `root`. Seul l'utilisateur `root` peut le modifier et tout le monde peut l'exécuter y compris les membres du groupe `root` :

- `rwx` : droits du propriétaire (`r`ead, `w`rite, e`x`écutable) ;
- `r-x` : droits du groupe (`r`ead, e`x`ecutable) ;
- `r-x` : droits des autres (`r`ead, e`x`ecutable).

Pour être plus précis, on peut dire que le fichier `/usr/bin/env` appartient à l'utilisateur dont l'uid est 0 et que la commande `ls` affiche le premier nom correspondant à cet uid. Tous les autres utilisateurs ayant le même uid sont également en mesure de le modifier.

Un dernier exemple :

```
$ ls -l ~/
drwxr-xr-x   3 vincent  users        4096 Nov  7 12:48 LaTeX
drwxr-xr-x   2 vincent  users        4096 Nov  7 14:40 bin
$
```

Cette commande liste le répertoire privé de l'utilisateur (caractère `~`) et affiche ici les droits des deux répertoires qui y résident :

- `d` : le fichier est un répertoire ;
- `rwx` : droits du propriétaire (`r`ead, `w`rite, e`x`écutable) ;
- `r-x` : droits du groupe (`r`ead et e`x`écutable) ;
- `r-x` : droits des autres (`r`ead et e`x`écutable) ;
- les nombres 3 et 2 indiquent le nombre de sous-répertoires contenus dans `LaTeX` et `bin` respectivement.

Il faut noter ici une particularité du système UNIX : un répertoire doit être exécutable pour pouvoir y pénétrer, on parle alors de *search permission* en anglais. Dans l'exemple ci-dessus, tout le monde y compris les membres du groupe `users` a le droit d'accéder aux répertoires `LaTeX` et `bin`.

2.3.4 Parcourir l'arborescence

Pour connaître le répertoire courant (*i.e.* celui où on se trouve au moment où on tape la commande) on peut taper la commande `pwd` (*print working directory*). Notez que ce répertoire courant peut toujours être désigné par le caractère « . »

```
$ pwd
/home/vincent/LaTeX/cours
$
```

À chaque utilisateur est associé un répertoire privé ou *home directory*. C'est le répertoire courant après la procédure de login. Pour se rendre dans ce répertoire, on peut taper la commande `cd` sans argument :

```
$ cd
$ pwd
/home/vincent
$
```

`/home/vincent` est donc le répertoire privé de l'utilisateur `vincent`. Si l'utilisateur est `lozano`, les commandes suivantes sont équivalentes à la commande `cd` sans argument :

```
$ cd ~
$ cd $HOME
$ cd ~lozano
```

```
$
```

On peut à tout moment changer de répertoire avec la commande `cd` (*change directory*) qui prend en argument un répertoire, par exemple :

```
$ cd /usr/local
$
```

Et, pour illustrer l'utilisation d'une référence relative :

```
$ cd /usr
$ ls -l
drwxr-xr-x   8 root     root         4096 Sep 25 00:21 X11R6
drwxr-xr-x   2 root     root        20480 Nov  5 19:16 bin
drwxr-xr-x  11 root     root         4096 Nov  5 18:56 local
$ cd local   <-------------------------- utilisation d'une référence relative
$
```

Enfin pour donner un exemple d'utilisation du répertoire « . . » désignant le répertoire *père* :

```
$ pwd
/usr/local
$ cd ..
$ pwd
/usr
$
```

2.3.5 Manipuler les fichiers

Les opérations courantes sur les fichiers sont la copie, le déplacement ou renommage et l'effacement. Commençons par la copie :

```
$ cd ~/LaTeX/cours
$ ls *.tex
truc.tex
$ cp truc.tex muche.tex
$ ls -l *.tex
-rw-r--r--  1 vincent users 1339 Nov  7 21:48 truc.tex
-rw-r--r--  1 vincent users 1339 Nov  7 21:51 muche.tex
$
```

la commande `cp` (*copy*) copie donc le fichier donné en premier argument vers le fichier donné en deuxième argument. Cette commande accepte également la copie de un ou plusieurs fichiers vers un répertoire :

```
$ cp /usr/local/bin/* .
$
```

cette commande copie tous les fichiers (sauf les sous-répertoires) contenus dans le répertoire `/usr/local/bin` dans le répertoire courant.

Pour déplacer un fichier, on dispose de la commande `mv` (*move*). De manière quelque peu analogue à la commande `cp`, la commande `mv` accepte deux formes. La première permet de *renommer* un fichier :

```
$ ls *.tex
guide-unix.tex test.tex
```

```
$ mv test.tex essai.tex
$ ls *.tex
guide-unix.tex essai.tex
$
```

L'autre forme a pour but de *déplacer* un ou plusieurs fichiers dans un répertoire :

```
$ mv essai.tex /tmp
$
```

déplace le fichier `essai.tex` dans le répertoire `/tmp`.

La commande `rm` (*remove*) efface un fichier. Sous UNIX l'effacement d'un fichier est définitif, c'est pourquoi, pour des raisons de sécurité, l'effacement d'un fichier est généralement configuré par l'administrateur en mode interactif, c'est-à-dire que le shell demande à l'utilisateur une confirmation :

```
$ rm /tmp/esssai.tex
rm: remove '/tmp/essai.tex'?yes
$
```

Ici toute autre réponse que « *yes* » ou « *y* » fait échouer la commande `rm`.

2.3.6 Et les répertoires dans tout ça ?

Et bien on peut en créer là où on a le droit, avec la commande `mkdir` (*make directory*) par exemple :

```
$ mkdir tmp
$ ls -l
...
drwxr-xr-x   2 vincent  users        4096 Nov  7 22:13 tmp
$
```

On peut également détruire un répertoire avec la commande `rmdir` (*remove directory*) à condition que le répertoire en question soit vide. On peut cependant effacer un répertoire non vide grâce à la commande `rm` et son option `-r`. Cette option permet d'effacer *récursivement* le contenu d'un répertoire :

```
$ rm -rf tmp
$
```

efface le répertoire `tmp` ; l'option `-f` (*force*) est ici, cumulée avec l'option de récursion et permet de forcer l'exécution de la commande `rm` en mode non-interactif. L'utilisateur n'est donc interrogé à aucun moment, c'est donc une forme à utiliser avec prudence...

2.3.7 Gestion des supports amovibles

Une des grandes interrogations des utilisateurs novices d'UNIX concerne l'accès aux supports amovibles (disquettes, cédéroms et autres clés USB). L'accès à de tels périphériques dépend du système installé, mais dans tous les cas il consiste à greffer (►monter◂) le système de fichiers du support amovible à un endroit précis de l'arborescence. Ce montage peut être réalisé manuellement avec la commande `mount`. Dans ce cas on trouvera dans le fichier `/etc/fstab` une ligne ressemblant à :

§ 2.3.2 p. 31 ◄

```
/dev/cdrom  /cdrom    iso9660 ro,user,noauto   0    0
```

indiquant que l'utilisateur lambda a le droit de monter un système de fichiers au format Iso9660 contenu dans un cédérom, dans le répertoire `/cdrom`. L'utilisateur pourra donc se fendre d'un :

```
$ mount /cdrom
$
```

pour avoir accès aux fichiers du dit cédérom à partir du répertoire `/cdrom`. Une fois la (ou les) opérations de lecture effectuée(s) l'utilisateur devra lancer un :

```
$ umount /cdrom
$
```

pour détacher le système de fichiers du cédérom de l'arborescence du système, et pouvoir éjecter le disque du lecteur.

Certaines versions d'UNIX utilisent un système de montage automatique des systèmes de fichiers stockés sur supports amovibles. Dans ce cas l'accès au répertoire `/cdrom`, `/mnt/cdrom`, ou quelque chose de ce genre provoquera le montage automatique du cédérom.

2.3.8 Changer les droits

Le propriétaire — et lui seul — a le droit de changer les permissions associées à un fichier. Seul l'administrateur (utilisateur `root`) a la possibilité de changer le propriétaire d'un fichier (commande `chown`). Enfin le groupe propriétaire peut être changé par le propriétaire (`chgrp`).

La commande `chmod` permet au propriétaire d'un fichier d'en changer les droits d'accès soit pour partager des données avec d'autres utilisateurs, soit au contraire pour rendre des données sensibles inaccessibles. Cette commande consiste à modifier les privilèges des trois entités propriétaires d'un fichier (utilisateur, groupe et les autres). Il existe deux syntaxes de la commande `chmod`, la première est symbolique, l'autre numérique. Nous utiliserons ici plus volontiers la première en présentant quelques « études de cas » où l'on désire modifier les droits d'un fichier. Mais avant tout, posons-nous la question suivante :

Quels sont les droits par défaut ?

Les droits par défaut sont définis par l'intermédiaire d'un *masque* de création de fichier. En jargon UNIX, ce masque est appelé *umask* pour *user file creation mask*. Ce masque est associé à un utilisateur et précise quels droits aura par défaut un fichier lors de sa création. Pour connaître son umask :

```
$ umask -S
u=rwx,g=rx,o=rx
$
```

Ce qui signifie qu'un fichier est créé avec par défaut :
- pour le propriétaire : tous les droits ;
- pour le groupe :
  - droits en lecture ;
  - droits en lecture et exécution pour les répertoires ;
- idem pour les autres.

Nous prendrons comme hypothèse pour nos études de cas que l'umask du système est celui donné par la commande `umask -S` ci-dessus. Créons un répertoire et un fichier dans ce répertoire :

```
$ mkdir echange
$ touch echange/donnees.txt
$
```

La commande `touch` change les attributs de date d'un fichier en le créant s'il n'existe pas. On peut alors examiner les droits sur ces fichiers :

```
$ ls -ld echange
drwxr-xr-x  2 vincent  users 4096 Nov 18 23:06 echange
$ ls -l echange
-rw-r--r--  1 vincent  users    6 Nov 15 21:22 donnees.txt
$
```


Attributs d'un répertoire

En considérant le répertoire comme un « catalogue » contenant une liste de fichiers, on peut voir les trois types de droits comme suit :

Lecture : compulser le catalogue, c'est-à-dire lister le contenu du répertoire avec la commande `ls` par exemple.

Écriture : modifier le catalogue, donc effacer les fichiers, les renommer, les déplacer.

Exécution : cet attribut permet à un utilisateur de *pénétrer* dans ce répertoire et d'accéder aux fichiers qui s'y trouvent.

Voyons plus précisément grâce à un exemple :

```
$ chmod g-r echange
$ ls -ld echange
drwx--xr-x  2 vincent  users  4096 Nov 20 18:03 echange
$
```

la commande `chmod` ci-dessus retire aux membres du groupe `users` le droit de lire le répertoire `echange`. C'est pourquoi lorsqu'un utilisateur autre que `vincent` (appelons-le `rene`), mais du groupe `users`, lance la commande :

```
$ ls ~vincent/echange
ls: /home/vincent/echange/: Permission denied
$
```

elle échoue, faute de privilèges suffisants. Ici c'est le droit d'examiner le *contenu* du répertoire qui n'est pas accordé. Par contre, la commande :

```
$ cat ~vincent/echange/donnees.txt
$
```

est légitime, puisque l'attribut `x` pour le groupe du répertoire `echange` est présent et permet à l'utilisateur `rene` d'accéder aux fichiers qu'il contient. C'est pourquoi, si l'on supprime aussi le droit d'exécution sur le répertoire `echange` pour le groupe `users`, comme ceci :

```
$ chmod g-x echange
$ ls -ld echange
drwx---r-x   2 vincent  users        4096 Nov 20 18:03 echange
$
```

la commande suivante, lancée par `rene` échoue faute de droits suffisants :

```
$ cat ~vincent/echange/donnees.txt
cat: /home/vincent/echange/donnees.txt: Permission denied
$
```

Il faut comprendre qu'un utilisateur ayant les droits d'écriture sur un répertoire peut effacer tous les fichiers qu'il contient quels que soient les droits et propriétaires de ces fichiers.

Autoriser un membre du groupe à lire un fichier

Avec le umask du paragraphe précédent, les membres du groupe propriétaire sont autorisés à lire un fichier nouvellement créé dans un répertoire nouvellement créé. En effet :

1. le répertoire qui contient le fichier est *exécutable* pour le groupe en question ;
2. le fichier est lisible pour le groupe.

Autoriser un membre du groupe à modifier un fichier

Pour autoriser un membre du groupe propriétaire du fichier `donnees.txt` du répertoire `echange` à modifier un fichier il suffit d'activer l'attribut `w` pour le groupe avec la commande `chmod` (en supposant que le répertoire courant est `echange`) :

```
$ chmod g+w donnees.txt
$ ls -l donnees.txt
-rw-rw-r-- 1 vincent users  40 Nov 2 18:03 donnees.txt
$
```

Interdire aux autres la lecture d'un fichier

Pour des raisons diverses on peut vouloir interdire la lecture d'un fichier aux utilisateurs n'appartenant pas au groupe du fichier. Pour ce faire, la commande :

```
$ chmod o-r donnees.txt
$ ls -l donnees.txt
-rw-rw---- 1 vincent  users   0 Nov 20 18:03 donnees.txt
$
```

interdit aux « autres » l'accès en lecture à `donnees.txt`

Autoriser au groupe la suppression d'un fichier

Lorsqu'un fichier est supprimé du système de fichier, le répertoire qui le contient est également modifié, car ce dernier contient la liste des fichiers qu'il héberge. Par conséquent, il n'est pas nécessaire d'activer l'attribut `w` sur un fichier pour pouvoir l'effacer, il faut par contre s'assurer que cet attribut est actif pour le répertoire qui le contient. Dans le cas qui nous préoccupe, la commande :

```
$ ls -ld echange
drwx---r-x   2 vincent  users       4096 Nov 20 18:03 echange
$ chmod g+rwx echange
$ ls -ld echange
```

```
drwxrwxr-x   2 vincent  users       4096 Nov 20 18:03 echange
$
```

accorde tous les droits (dont l'attribut `w`) au groupe propriétaire sur le répertoire `echange` ; ce qui permet à l'utilisateur `rene` de pouvoir effacer le fichier `donnees.txt`.

Notation symbolique et notation octale

La commande `chmod` telle qu'on l'a vue jusqu'à présent obéit à la syntaxe suivante :

`chmod` ⟨*p*⟩ ⟨*%*⟩ ⟨*a*⟩ , ... ⟨*fichier*⟩

- ⟨*a*⟩ → un ou plusieurs attributs : `r`ead, `w`rite, ou `x`ecutable.
- ⟨*%*⟩ → un opérateur : `+` ou `-` pour activer ou désactiver
- ⟨*p*⟩ → zéro ou plusieurs propriétaires : `u`ser, `g`roup, `o`ther ou `a`ll

Les arguments de la commande `chmod` peuvent être constitués de plusieurs blocs ⟨*p*⟩⟨*%*⟩⟨*a*⟩. Le cas échéant ils doivent être séparés par des virgules. Les commandes suivantes sont donc correctes :

```
$ chmod o-rwx bidule
$ chmod g+rx bidule
$ chmod -w,o+x bidule
$ chmod a+r bidule
$
```

Dans l'ordre, chacune des commandes précédentes :
- enlève tous les droits aux autres ;
- donne les droits en lecture et exécution pour les membres du groupe propriétaire ;
- enlève le droit en écriture pour tout le monde et on rajoute le droit en exécution pour les autres ;
- donne à tout le monde le droit en lecture.

On notera donc que sans indication de propriétaires, tous sont concernés et ceci en accord avec le umask. D'autre part l'option `-R` de la commande `chmod` permet d'affecter le mode spécifié de manière récursive aux fichiers contenus dans les sous-répertoires.

Mais venons-en au fait : certains puristes utilisent ce qu'on appelle la notation *octale* pour positionner les droits d'un fichier. Cette notation consiste a coder chaque bloc `rwx` du bloc de droits en une donnée numérique de la manière suivante :

2^2	2^1	2^0
r	w	x

On a alors, par exemple :
- `-rwx------` codé par 700 ;
- `-rw-r-xr-x` codé par 655 ;
- `-rw-r--r--` codé par 644.

Ce codage octal peut être utilisé en lieu et place des symboles `gor` et `rwx` avec la commande `chmod`. Il est également intéressant de noter qu'on peut afficher le `umask` en notation octale :

```
$ umask
022
$
```

Cette valeur est le complément par rapport à 777. Avec un tel masque les fichiers sont créés avec les permissions 755.

2.3.9 Liens

Une des notions un peu « troublantes » du système de fichiers d'UNIX est la notion de lien. Comme on l'a vu précédemment, en interne, le système identifie les fichiers par un numéro : l'*inode*.

Liens physiques

L'opération qui consiste à relier un fichier (un ensemble d'octets sur un disque) à un nom sur l'arborescence est précisément la création d'un *lien physique*. Soit le fichier `~/doc/unix/guide.tex` :

```
$ cd ~/doc/unix
$ ls -li guide.tex
38564 -rw-r--r-- 1 vincent users 18114 Nov 7 22:44 guide.tex
$
```

on constate que ce fichier (créé par un éditeur de texte) a pour inode 38564, et ne possède qu'un lien (`~/doc/unix/guide.tex`).

On peut créer un nouveau lien sur ce fichier éventuellement dans un répertoire différent si ce répertoire est sur la même partition que le fichier `guide.tex`, grâce à la commande `ln` (*link*) dont la syntaxe est :

`ln` ⟨*cible du lien*⟩ ⟨*nom du lien*⟩

Par exemple :

```
$ cd ~/tmp
$ ln ~/doc/unix/guide.tex unix.tex
$ ls -li unix.tex
38564 -rw-r--r-- 2 vincent users 18114 Nov 7 22:44 unix.tex
$
```

on peut alors constater (figure 2.4 page suivante) que le fichier `unix.tex` possède maintenant deux liens physiques. Dans cette situation, il y a donc *une seule* représentation des données sur le disque — ce qui est confirmé par l'inode qui est identique (38564) pour les deux liens — et *deux* noms associés à cette représentation :

1. `~/doc/unix/guide.tex`
2. `~/tmp/unix.tex`

Ce qui implique que l'on peut modifier le fichier par le biais de l'un ou l'autre des liens. Pour ce qui est de la suppression d'un lien, la règle est la suivante :

1. tant que le nombre de liens physiques sur le fichier en question est strictement supérieur à 1, la commande `rm` aura pour effet la suppression du lien et non des données.
2. lorsque le nombre de liens physiques est égal à 1, `rm` efface effectivement les données, puisqu'il n'y a plus qu'un lien unique sur celles-ci.

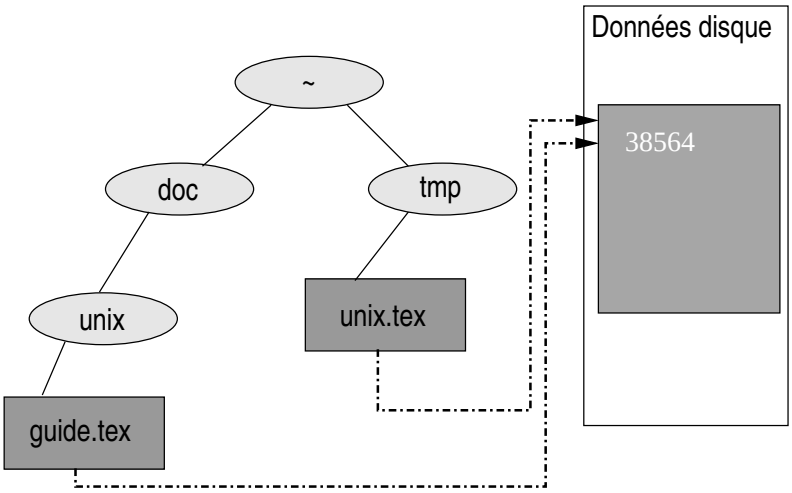


FIGURE 2.4: Exemple de liens physiques

Pour être très précis, si le fichier en question est ouvert par au moins un processus, le lien disparaîtra après la commande `rm` mais l'espace qu'occupent les données ne sera libéré que quand le dernier processus ayant ouvert le fichier se terminera.

Liens symboliques

Lorsque que l'on veut lier un fichier par l'intermédiaire d'un lien ne se trouvant pas sur la même partition que le fichier lui-même, il est nécessaire de passer par l'utilisation d'un lien dit *symbolique*. Nous allons par exemple créer un lien symbolique sur notre fichier `guide-unix.tex` depuis le répertoire `/tmp` se trouvant sur une autre partition :

```
$ cd /tmp
$ ln -s ~/doc/unix/guide.tex test.tex
$ ls -l test.tex
lrwxrwxrwx 1 vincent users 32 Nov 21 16:41 test.tex -> /hom
e/vincent/doc/unix/guide.tex
$
```

Les liens résultants sont indiqués à la figure 2.5 page suivante.

On remarquera les particularités d'un lien symbolique :

- c'est un fichier différent de celui sur lequel il pointe ; il possède son propre inode :

  ```
  $ ls -i test.tex
    23902 test.tex
  $
  ```

- le fichier contient une *référence* sur un autre fichier, ce qui explique par exemple que la taille du lien symbolique `test.tex` est de 32, qui correspond aux nombres de caractères du chemin du fichier auquel il se réfère ;
- un lien symbolique est un fichier spécial dont on ne peut changer les permissions, ce qui est indiqué par le bloc de droits :

  `lrwxrwxrwx`

La plupart des opérations sur le lien symbolique est effectuée sur le fichier sur lequel il pointe. Par contre la suppression de ce lien obéit aux règles suivantes :

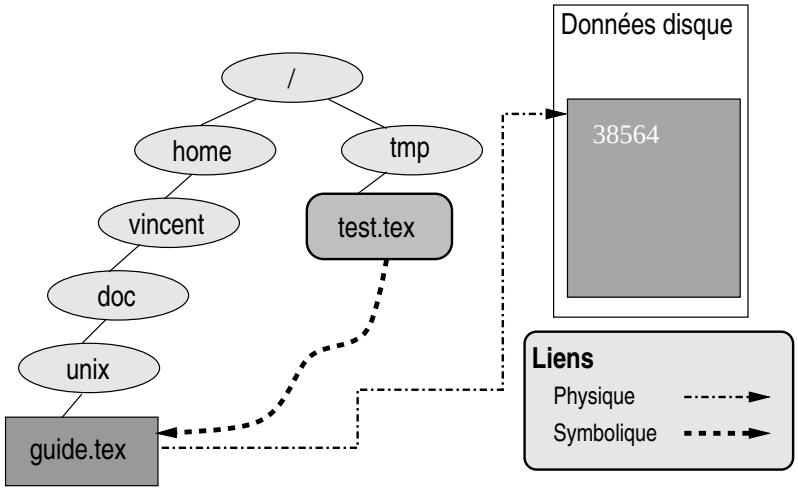


FIGURE 2.5: Exemple de lien symbolique.

1. la commande `rm` supprime le lien symbolique lui-même — qui est un fichier à part entière — et n'a pas d'influence sur le fichier auquel il se réfère ;
2. par conséquent, si on supprime le fichier, le lien symbolique existe toujours et pointe sur un fichier qui n'existe pas.

Enfin la dernière particularité du lien symbolique (qui le distingue du lien physique) provient du fait qu'il est possible de créer un tel lien sur un *répertoire*. Par exemple :

```
$ cd
$ ln -s /usr/share/texmf/tex/latex macros
$ ls -l macros
lrwxrwxrwx  1 vincent users 26 Nov 21 17:02 macros -> /usr/sh
are/texmf/tex/latex
$
```

il est alors possible d'utiliser le lien symbolique comme un raccourci vers un autre répertoire. Notons à ce sujet que la notion de lien symbolique d'UNIX, peut s'apparenter à celle de *raccourci* de Windows®. Cependant l'utilisateur attentif de ce dernier système aura remarqué que les raccourcis ne constituent juste qu'une aide à l'environnement graphique, mais qu'il est impossible de traiter le raccourci comme un véritable fichier ou répertoire le cas échéant.

Il peut être intéressant de noter que les commandes internes `cd` et `pwd` comprennent deux options :
- `-L` pour demander explicitement à suivre les liens symboliques ;
- `-P` pour demander à suivre la structure physique de l'arborescence.

2.3.10 Access control list (ACL)

Limites de la gestion des privilèges

Nous avons vu à partir de la section 2.3.3 page 34 consacrée aux privilèges sur les fichiers et répertoires que par défaut un système UNIX attache trois privilèges à un fichier :

1. les privilèges de l'utilisateur propriétaire.

2. ceux du groupe propriétaire.
3. ceux des autres.

Par conséquent la finesse du réglage des accès ne peut se faire qu'au niveau du groupe. En d'autres termes pour rendre un fichier accessible en lecture et écriture à un utilisateur λ et uniquement à lui, il n'y a pas d'autre solution que de créer un groupe γ auquel λ appartient, puis d'accorder les droits de lecture et d'écriture au groupe γ.

Imaginons que deux utilisateurs — dont les noms de connexion sont `andy` et `debrah` — veuillent collaborer sur le contenu d'un fichier `poodle.dat`. Tout d'abord, l'administrateur doit créer un groupe (nommons-le `zircon`) puis faire en sorte que `andy` et `debrah` deviennent membres de ce groupe. Ensuite l'utilisateur initialement propriétaire du fichier `poodle.dat` (par exemple `andy`) doit positionner les privilèges sur celui-ci :

```
$ ls -l poodle.dat
-rw-r--r-- 1 andy users  3100 2009-05-11 22:51 poodle.dat
$ chgrp zircon poodle.dat
$ chmod g+rw poodle.dat
$ ls -l poodle.dat
-rw-rw-r-- 1 andy zircon 3100 2009-05-11 22:51 poodle.dat
$
```

À partir de maintenant, l'utilisateur `debrah` peut lire et modifier le contenu du fichier `poodle.dat`.

Le problème ici est que la création du groupe n'est possible que pour l'administrateur du système qui seul peut ajouter un nouveau groupe d'utilisateurs. En outre si un autre utilisateur souhaitait se greffer au projet, il faudrait à nouveau lui faire une demande pour ajouter le nouveau venu au groupe. On voit donc clairement que la gestion des privilèges proposée par défaut dans UNIX a l'inconvénient d'être trop statique pour l'exemple présenté ci-dessus.

Introduction aux ACL

Nous vous proposons donc ici de découvrir quelques commandes permettant de régler les privilèges plus finement et de manière plus autonome. La gestion des ACL sous UNIX repose essentiellement sur deux commandes :

- `getfacl` examen des autorisations d'un fichier ;
- `setfacl` positionnement de ces autorisations.

Le système de fichier devra supporter les ACLs. Sous le système GNU/*LINUX* il faudra le spécifier explicitement dans les options de montage contenues dans le fichier `/etc/fstab`. Le système de votre serviteur contient par exemple :

```
/dev/sdb5    /home    ext3    acl,defaults        0       2
```

indiquant que la partition `/dev/sdb5` est montée dans le répertoire `/home` avec support pour les ACLs.

Accorder des droits à un utilisateur Reprenons l'exemple de nos deux compères `andy` et `debrah` souhaitant tous deux collaborer sur le fichier `poodle.dat` :

```
$ ls -l poodle.dat
```

```
-rw-r--r-- 1 andy users  3100 2009-05-11 22:51 poodle.dat
$
```

Pour autoriser `debrah` à lire et modifier le fichier en question, `andy` devra taper :

```
$ setfacl -m u:debrah:rw poodle.dat
$
```

La syntaxe

`setfacl` -m u:⟨*user*⟩:⟨*perm*⟩ ⟨*fichier*⟩

permet donc de *modifier* (c'est le sens de l'option `m`) les autorisations de l'utilisateur ⟨*user*⟩ (argument `u:`). Les autorisations ⟨*perm*⟩ peuvent être définies en utilisant la notation symbolique présentée au paragraphe 2.3.3 page 34.

Comme la commande `chmod`, `setfacl` autorise l'option `-R` pour accorder les droits spécifiés à tous les fichiers et sous-répertoires, récursivement.

Examiner les autorisations associées à un fichier L'option `-l` de la commande `ls` propose un affichage légèrement modifié lorsque des autorisations ACLs sont associées à un fichier :

```
$ ls -l poodle.dat
-rw-rw-r--+ 1 andy users  3100 2009-05-11 22:51 poodle.dat
$
```

Notez la présence du signe `+`. La commande `getfacl` permet de lister les permissions associées :

```
$ getfacl poodle.dat
# file: poodle.dat
# owner: lozano    <-------------------------------------- propriétaire
# group: lozano    <-------------------------------------- groupe propriétaire
user::rw-          <-------------------------------------- droits du propriétaire
user:debrah:rw-     <------------------------------------- droits de l'utilisateur debrah
group::r--         <-------------------------------------- droits du groupe propriétaire
mask::rw-          <-------------------------------------- cf. plus bas
other::r--         <-------------------------------------- droits des autres
$
```

Accorder des droits par défaut dans un répertoire Grâce à l'option `-d` de la commande `setfacl` on peut accorder des droits par défaut à un répertoire. En reprenant l'exemple précédent, la commande :

```
$ setfacl -d -m u:muffin:rw .
$
```

permet d'autoriser la lecture et l'écriture à l'utilisateur `muffin` pour tous les fichiers créés dans le répertoire courant. En d'autres termes, tous les fichiers créés par la suite seront lisibles et modifiables par l'utilisateur `muffin`.

Attention, la commande de l'exemple ci-dessus n'accorde pas à l'utilisateur `muffin` de créer des fichiers dans le répertoire courant.

Enfin, les droits par défaut peuvent être retirés avec l'option `-k`. Ici la commande :

```
$ setfacl -k .
$
```

supprime les droits par défaut sur le répertoire courant.

Révoquer des droits Pour supprimer les droits ajoutés avec la `setfacl` et son option `-m` on peut utiliser :

- l'option `-b` pour supprimer tous les droits de type ACL
- l'option `-x` pour supprimer un entrée, par exemple :

```
$ setfacl -x u:debrah poodle.dat
$
```

révoque les droits accordés sur le fichier `poodle.dat` à l'utilisateur `debrah`.

Masque ACL Lorsqu'au moins un privilège de type ACL a été accordé à un fichier, un *masque* est créé contenant l'union de toutes les autorisations. Ce masque est construit automatiquement, ainsi :

```
$ setfacl -m u:dinah:r righthereonthe.floora
$ setfacl -m u:moe:r   righthereonthe.floora
$ setfacl -m u:hum:rw  righthereonthe.floora
$
```

Le masque créé et affiché par la commande `getfacl` sera :

```
$ getfacl righthereonthe.floora
[...]
user::dinah:r--
user::moe:r--
user::hym:rw-
[...]
mask::rw-    <------------------------ union des trois permissions précédentes
[...]
$
```

L'intérêt de ce masque réside dans le fait qu'il peut être modifié pour appliquer une politique pour tous les utilisateurs. Par exemple pour ne conserver que le droit en lecture pour tous les utilisateurs ACL, on peut exécuter la commande suivante :

```
$ setfacl -m m::r righthereonthe.floora
$
```

2.4 Processus

Le système UNIX est multi-tâche et multi-utilisateur. Le noyau gère l'ensemble des processus grâce à un programme appelé *l'ordonnanceur* (*scheduler*). Ce dernier a pour but d'accorder aux processus du temps-cpu, et ceci chacun à tour de rôle, en fonction de priorités le cas échéant. Ainsi, un processus peut se trouver dans quatre états :

actif : le processus utilise le cpu

prêt : le processus attend que l'ordonnanceur lui fasse signe

mais également :

endormi : le processus attend un évènement particulier (il ne consomme pas de cpu dans cet état) ;

► § 2.4.2 p. 51 **suspendu :** le processus a été interrompu par un ▲signal◄.

Chaque processus possède un numéro qui l'identifie auprès du système. Ce numéro est appelé son *pid* ou *process identifer*. On verra plus bas que chaque processus à l'exception du tout premier créé, possède un processus père qui lui donne naissance. Ainsi l'ensemble des processus d'un système UNIX constitue un *arbre* de processus.

On notera ici qu'à la notion de processus, on associe généralement la notion de *terminal*. Un terminal est un canal de communication entre un processus et l'utilisateur. Il peut correspondre à un terminal physique comme un écran ou un clavier, mais aussi à ce qu'on appelle un émulateur de terminal comme une fenêtre XTerm. Dans les deux cas un fichier spécial dans le répertoire `/dev` (par exemple `/dev/tty1` est la première console en mode texte, sous linux) est associé au terminal, et les communications se font par le biais d'appels système sur ce fichier. La variable `TERM` informe les applications sur le comportement du terminal (entre autres choses, la manière dont les caractères de contrôle seront interprétés).

2.4.1 Examiner les processus

La commande `ps` permet d'examiner la liste des processus « tournant » sur le système. Cette commande comprend un nombre très important d'options qui sont généralement différentes selon les systèmes, il n'est donc pas question ici de les passer toutes en revue (pour ça il suffit de taper `man ps` !). Voyons tout de même quelques exemples instructifs.

Notez qu'un certain nombre d'options de la commande ps a fait l'objet de standardisation notamment au travers de la *Single Unix Specification* (SUS) portée par l'Open Group, consortium ayant pour but de standardiser ce que devrait être un système UNIX. Par exemple, la commande ps de la distribution Debian actuelle est censée respecter les recommandations de la SUS version 2. Encore une fois, vous trouverez dans ce qui suit des options ne respectant pas nécessairement tous ces standards, veuillez nous en excuser par avance.

Ceux du terminal

J'ai lancé Emacs (éditeur de texte grâce auquel lequel je tape ce document) dans une fenêtre xterm ; voici ce que me donne la commande `ps` dans cette fenêtre :

```
$ ps
 PID TTY          TIME CMD
1322 ttyp2    00:00:43 bash
1337 ttyp2    00:02:04 emacs
1338 ttyp2    00:00:00 ps
$
```

On obtient donc une liste dont la première colonne contient le *pid* de chaque processus, chacun de ces processus est associé au pseudoterminal du xterm désigné par `ttyp2`. La dernière colonne donne le nom de la commande associée au processus ; le temps donné ici est le temps-cpu utilisé depuis le lancement de la commande.

On pourra noter que puisque la commande `ps` est lancée pour obtenir la liste des processus, son pid apparaît également dans la liste (ici pid 1338).

Ceux des autres terminaux

L'option `x` de la commande `ps` permet de lister les processus de l'utilisateur qui ne sont par rattachés au terminal courant [9] :

```
$ ps x
  PID TTY      STAT   TIME COMMAND
  345 ?        S      0:00 bash /home/vincent/.xsession
  359 ?        S      0:01 fvwm2
 1254 ttyp0    S      0:00 lynx /usr/doc/vlunch/index.html
 1354 ttyp3    S      0:00 xdvi.bin -name xdvi guide-unix.dvi
29936 ttyp2    R      0:00 ps -x
$
```

La nouvelle colonne `STAT` (*status*) donne l'état de chaque processus, ici `S` indique que le processus est endormi (*sleeping*), et `R` qu'il est prêt (*runnable*).

Ceux des autres utilisateurs

La syntaxe suivante permet d'obtenir la liste des processus lancés par un utilisateur du système :

```
$ ps -u rene
  PID TTY          TIME CMD
  663 tty1     00:00:00 bash
  765 ?        00:00:00 xclock
  794 ttyp4    00:00:00 bash
  805 ttyp4    00:00:00 mutt
$
```

L'utilisateur `rene` est donc connecté sur la première console virtuelle, a lancé une horloge du système X window, et est vraisemblablement en train de lire son courrier.

Tous !

Enfin on veut parfois être en mesure de visualiser la liste de tous les processus sans critère restrictif de terminal ou d'utilisateur. Il y a encore une fois plusieurs voies pour atteindre ce but. La version de la commande `ps` de *LINUX* supporte plusieurs styles de syntaxes ; nous en donnons ci-dessous deux exemples :

```
$ ps aux
USER    PID %CPU %MEM   TTY    STAT START   TIME COMMAND
root      1  0.1  0.1   ?      S    14:11   0:03 init [5]
root      2  0.0  0.0   ?      SW   14:11   0:00 [kflushd]
root      3  0.0  0.0   ?      SW   14:11   0:00 [kupdate]
root      4  0.0  0.0   ?      SW   14:11   0:00 [kpiod]
root      5  0.0  0.0   ?      SW   14:11   0:00 [kswapd]
root    210  0.0  0.3   ?      S    14:12   0:00 syslogd -m 0
root    221  0.0  0.3   ?      S    14:12   0:00 klogd
```

9. Liste non exhaustive par souci de clarté.

```
daemon  237  0.0  0.0   ?      SW   14:12   0:00 [atd]
root    253  0.0  0.3   ?      S    14:12   0:00 crond
root    284  0.0  0.3   ttyS0  S    14:12   0:00 gpm -t ms
$
```

Cette commande liste sur le terminal, tous les processus du système — ici elle est raccourcie bien évidemment. Cette forme de listing fournit également la date de démarrage du processus (champ `START`), ainsi que le pourcentage de cpu et de mémoire utilisés (`%CPU` et `%MEM` respectivement).

Il est également intéressant de noter que l'on peut suivre la séquence de démarrage du système d'exploitation : le premier programme lancé porte le pid 1 et se nomme `init`. Les processus suivants concernent les services de base du système UNIX. Voici une autre manière d'obtenir la liste complète des processus :

```
$ ps -ef
UID        PID  PPID  STIME TTY          TIME CMD
root         1     0  14:11 ?        00:00:03 init [5]
root         2     1  14:11 ?        00:00:00 [kflushd]
root         3     1  14:11 ?        00:00:00 [kupdate]
root         4     1  14:11 ?        00:00:00 [kpiod]
root         5     1  14:11 ?        00:00:00 [kswapd]
root       210     1  14:12 ?        00:00:00 syslogd -m 0
root       221     1  14:12 ?        00:00:00 klogd
daemon     237     1  14:12 ?        00:00:00 [atd]
root       253     1  14:12 ?        00:00:00 crond
root       284     1  14:12 ttyS0    00:00:00 gpm -t ms
$
```

ici l'affichage est quelque peu différent. `STIME` est l'équivalent du `START`, et on a accès au `ppid` qui identifie le processus `père` d'un processus particulier. On remarquera dans cette liste que tous les processus sont des fils d'`init` — c'est-à-dire lancés par celui-ci. Ceci à l'exception d'`init` lui-même qui est le premier processus créé.

Personnaliser l'affichage

On peut se limiter à certaines informations en utilisant l'option `o` de la commande `ps` ; cette option permet de spécifier les champs que l'on désire voir s'afficher, par exemple :

```
$ ps -xo pid,cmd
PID CMD
346 [.xsession]
360 fvwm2 -s
363 xterm -ls -geometry 80x30+0+0
367 -bash
428 man fvwm2
$
```

n'affiche que le pid et la commande d'un processus. On a combiné ici, l'option `o` avec l'option `x` vu précédemment.

Lien de parenté

L'option `--forest` associée avec la commande précédente illustre l'arborescence des processus. On a isolé ici les processus associés à une session X ; on voit clairement grâce au « dessin » sur la partie droite, et à la correspondance entre pid et ppid, quelles sont les filiations des processus.

```
$ ps --forest -eo pid,ppid,cmd
PID  PPID CMD
324     1 [xdm]
334   324  \_ /etc/X11/X
335   324  \_ [xdm]
346   335      \_ [.xsession]
360   346          \_ fvwm2 -s
448   360              \_ xterm -ls
451   448              |   \_ -bash
636   451              |       \_ ps --forest -eo pid,ppid,cmd
576   360              \_ /usr/X11R6/lib/X11/fvwm2/FvwmAuto
577   360              \_ /usr/X11R6/lib/X11/fvwm2/FvwmPager
$
```

2.4.2 Modifier le déroulement d'un processus

Il existe au moins trois manières de modifier le déroulement normal d'un processus :

1. changer la priorité d'un processus avec la commande `nice` ;
2. utiliser la commande `kill`. Cette commande permet à partir du pid d'un processus, d'envoyer ce qu'on appelle en jargon UNIX, un *signal* à ce processus. En tant qu'utilisateur du système, on envoie généralement des signaux pour interrompre, arrêter ou reprendre l'exécution d'un programme ;
3. utiliser le contrôle de tâche (*job control*) depuis le shell. Ce mécanisme permet de modifier le déroulement des programmes s'exécutant dans un terminal donné.

Priorité d'un processus

Par défaut un processus lancé par un utilisateur a la même priorité que celui d'un autre utilisateur. Dans la situation où l'on désire lancer un processus « gourmand » sans gêner les autres utilisateurs du système, on peut utiliser la commande `nice` :

```
$ nice -n ⟨priorité⟩ ./groscalcul
```

où $-20 \leq \langle priorité \rangle \leq 19$, 19 désignant la plus petite priorité (processus « laissant sa place » aux autres) et -20 la plus grande. Par défaut, les utilisateurs n'ont pas le droit d'utiliser une priorité négative, par conséquent tous les utilisateurs sont égaux face à l'utilisation des ressources de la machine.

Notons l'existence de la commande `renice` permettant de baisser la priorité d'un processus une fois qu'il est lancé. Par exemple, avec la commande :

```
$ renice -10 -p 4358
4358: old priority 0, new priority 10
$
```

on change la priorité du processus de pid 4358 en lui affectant la valeur 10.

La notion de signal

Lorsqu'on veut interrompre ou suspendre un processus, l'intervention revient toujours à envoyer ce qu'on appelle un *signal* au processus en question. Lorsqu'un processus reçoit un signal il interrompt le cours normal de son exécution et peut :

- soit s'arrêter, c'est le comportement par défaut ;
- soit exécuter une routine particulière que le concepteur de la commande aura eu soin de définir et reprendre son cours normal.

Selon la mouture de l'UNIX et de la plate-forme sous-jacente, on dispose d'une trentaine de signaux dont on peut avoir la liste avec la commande `kill` et l'option `-l`.

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
$
```

Tous ces signaux peuvent être détournés de manière à installer une routine à exécuter à la réception de ce signal ; ceci à l'exception du signal `KILL` (n° 9) et du signal `STOP` (n° 19) qui ne peuvent être détournés et donc ont toujours pour effet d'arrêter (respectivement d'interrompre) le processus.

La commande `kill`

Nous passerons ici en revue quelques utilisations de la commande `kill`. Dans un cadre très pratique, il y a trois situations vitales où il est nécessaire de connaître cette commande :

1. un programme « s'emballe », c'est-à-dire accapare les ressources mémoire ou cpu du système, il est alors nécessaire d'en stopper l'exécution ;
2. un programme utilise trop de ressources pour le confort des autres utilisateurs, il est nécessaire de l'interrompre momentanément ;
3. un programme ne répond plus aux sollicitations de l'utilisateur, il n'y a pas d'autres moyen que de l'interrompre en lui envoyant un signal de fin d'exécution.

Passons en revue ces trois situations en supposant que sur notre terminal la commande `ps` nous renvoie :

```
$ ps -o pid,pcpu,pmem,state,cmd
PID %CPU %MEM S CMD
2177  0.0  2.1 S -bash
2243 99.9 99.9 R winword-2005
2244  0.0  1.7 R ps -o pid,pcpu,pmem,state,cmd
$
```

Anti-emballement On décide que le programme `winword-2005` prend beaucoup trop de ressources — probablement à cause d'un bug — on intervient en décidant d'éliminer purement et simplement ce processus du système :

```
$ kill 2243
$
```

ce qui a pour effet d'envoyer le signal `TERM` au processus de pid 2243. Il se peut que l'envoi de ce signal soit sans effet si le programme en question a été conçu pour détourner ce signal. Le processus de pid 2243 apparaîtra toujours dans la liste des processus et il faudra lui envoyer un signal `KILL` :

```
$ kill -KILL 2243
$
```

qui comme on l'a vu précédemment ne peut être détourné ; le processus de pid 2243 est donc irrémédiablement arrêté.

Assurer le confort des autres On considère ici que le fameux programme `winword-2005` que tout le monde utilise sans vraiment savoir pourquoi, n'est pas en train de s'emballer, mais simplement prend un peu trop de ressources. La tâche effectuée est lourde, mais importante pour l'utilisateur qui décide donc d'*interrompre* le programme pour le reprendre plus tard :

```
$ kill -STOP 2243
$
```

On peut constater que le programme interrompu est effectivement dans cet état en examinant le champ `STATE` :

```
$ ps -o pid,pcpu,pmem,state,cmd
PID %CPU %MEM S CMD
2177  0.0  2.1 S -bash
2243  0.0 25.3 T winword-2005
2244  0.0  1.7 R ps -o pid,pcpu,pmem,state,cmd
$
```

Ce champ est positionné à `T` qui indique que le processus de pid 2243 est interrompu. L'utilisateur soucieux d'achever son travail, pourra alors lancer la commande en fin de journée pour ne pas gêner ses collègues :

```
$ kill -CONT 2243
$
```

qui indique au processus de pid 2243 de reprendre son exécution.

Contrôle de tâches

Le contrôle de tâches ou *job control* permet de manipuler les processus tournant sur un terminal donné. Sur chaque terminal, on distingue deux types de processus :

1. le processus en *avant-plan*, c'est celui qui peut utiliser le terminal comme canal de communication avec l'utilisateur par le biais du clavier ; on dit également qu'on « a la main » sur ce type de processus ;
2. les processus en *arrière-plan* (ou tâche de fond) ; ceux-là peuvent utiliser le terminal comme canal de sortie (pour afficher des informations par exemple) mais ne peuvent lire les données que leur fournirait l'utilisateur.

Ce qui est écrit ci-dessus est le comportement qu'on rencontre généralement sur un terminal avec la plupart des applications. Il n'est par contre pas inutile de noter que selon qu'une application intercepte ou ignore les deux signaux particuliers `TTOU` et `TTIN` (respectivement écriture ou lecture d'un processus en arrière-plan), le comportement pourra être différent.

Les shells supportant le contrôle de tâches [10] offrent un certain nombre de commandes et de combinaisons de touches qui permettent entre autres :

- d'interrompre ou d'arrêter un programme ;
- de passer un programme en arrière ou en avant-plan ;
- de lister les processus en arrière-plan.

Nous vous proposons de passer en revue ces fonctionnalités en se plaçant dans le cadre d'un exemple simple : supposons que dans un terminal donné, un utilisateur lance l'éditeur Emacs :

```
$ emacs
```

Une fois Emacs lancé, l'utilisateur n'a plus la main sur le terminal [11], et tout caractère saisi reste en attente, jusqu'à la fin de l'exécution d'Emacs :

```
$ emacs
qdqsqmlskdjf
  ←————————————————————————————————————— fin du progamme emacs
$ qdqsqmlskdjf
bash: qdqsqmlskdjf: command not found
$
```

Arrêter le programme

Pour arrêter un programme en avant-plan, on peut utiliser la célèbre combinaison de touche Ctrl c (`Ctrl-c`) :

```
$ emacs
      ←————————————————————————————————————— Control + c
$
```

`Ctrl-C` envoie le signal `INT` au processus en avant-plan dans le terminal.

Passage en arrière-plan

Pour passer Emacs (le programme de notre exemple) en arrière-plan, il faut procéder en deux temps : appuyer sur la touche Control puis la touche z en même temps (on note souvent `Ctrl-z`) [12], ce qui a pour effet d'envoyer le signal `STOP` au programme s'exécutant en avant-plan sur le terminal.

```
$ emacs
  ←————————————————————————————————————— Control + z
[1]+  Stopped                 emacs
$
```

10. Ce qui est le cas de la plupart des shells modernes.
11. Sauf si Emacs est lancé en mode texte dans le terminal...
12. Cette combinaison est généralement celle que l'on trouve sur un système UNIX, mais notez que cela peut être différent sur certains systèmes.

Il faut ensuite passer le programme en tâche de fond grâce à la commande interne `bg` (*background*) :

```
$ bg
[1]+ emacs &
$
```

Le numéro entre crochet (`[1]`) est le numéro de la tâche dans le terminal courant. On parle aussi de *job*. On peut également lancer directement une commande ou programme en tâche de fond, en postfixant la commande par le caractère `&` :

```
$ emacs&
[1] 1332
$
```

Ici le système attribue à Emacs le job `[1]`, et le shell affiche sur le terminal le pid correspondant (1332).

Lister les tâches

La commande `jobs` permet d'obtenir une liste des tâches en arrière-plan, ou interrompues, sur le terminal. Par exemple :

```
$ xclock & xlogo & xbiff &
[1] 1346
[2] 1347
[3] 1348
$
```

lance trois programmes donnant lieu à la création d'une fenêtre graphique. On peut lister ces tâches :

```
$ jobs
[1]   Running                 xclock &
[2]-  Running                 xlogo &
[3]+  Running                 xbiff &
$
```

Passage en avant-plan

Il est possible de faire référence à une tâche par son numéro grâce à la notation `%`⟨*numéro_de_tâche*⟩. Ceci est valable pour les commandes `bg` et `kill` vues précédemment, mais aussi pour la commande permettant de passer une tâche en avant-plan : la commande `fg` (*foreground*). Ainsi par exemple pour passer `xlogo` en avant-plan :

```
$ fg %2
xlogo
  ←————————————————————————————————— maintenant on n'a plus la main.
```

2.5 Quelques services

Un système d'exploitation peut être vu comme une couche logicielle faisant l'interface entre la machine et l'utilisateur. Cette interface peut elle-même être appré-

hendée comme un ensemble de services fournis à l'utilisateur. Outre les services vus jusqu'ici permettant de lancer des programmes et de stocker des données [13], il existe — entre autres — des services liés à l'impression et la planification.

Les services d'UNIX répondent généralement au principe client/serveur, dans lequel le client est l'application vue par l'utilisateur, et le serveur un *daemon* (disk and execution monitor), c'est-à-dire un programme attendant en tâche de fond, pour répondre aux éventuelles requêtes.

2.5.1 Impression

Sur les réseaux locaux, l'impression sous UNIX se fait généralement sur des imprimantes PostScript connectées à un réseau. Le PostScript est un langage d'impression conçu par la société Adobe, c'est un langage de description de page. Ce langage est *interprété*, et les imprimantes dites PostScript disposent d'un interpréteur implanté sur une carte électronique. Le principe pour imprimer est donc d'envoyer un fichier au format PostScript sur une imprimante. En fonction du format du fichier initial, il faudra procéder à une conversion avec les logiciels suivants :

- a2ps convertit un fichier texte en fichier PostScript ;
- convert permet de convertir à peu près n'importe quel type d'image en PostScript ;
- dvips permet de convertir la sortie de TEX (format `dvi`) en PostScript.

Si l'on ne dispose pas d'imprimante PostScript, il faut générer le PostScript comme indiqué précédemment et utiliser un *interpréteur logiciel* pour transformer ce PostScript en fichier compatible avec l'imprimante en question (voir figure 2.6). L'interpréteur PostScript couramment utilisé sous UNIX est Ghostscript (`http://www.cs.wisc.edu/~ghost`). Les commandes de base disponibles pour l'utilisateur sont :

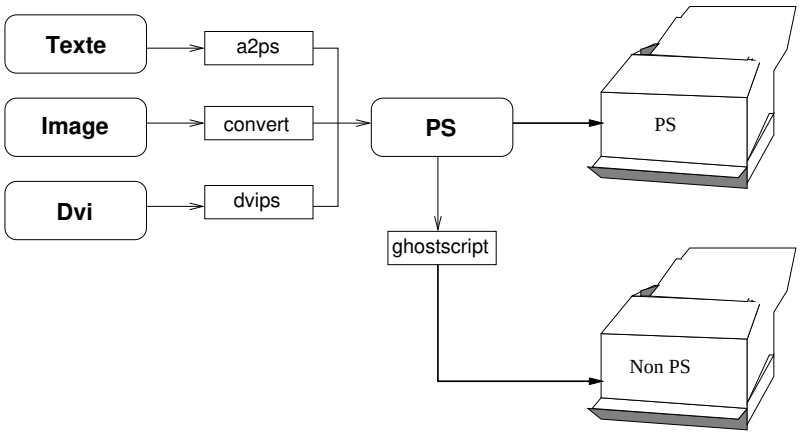


FIGURE 2.6: Impression sous Unix.

- `lpr` : pour soumettre un fichier à imprimer ;
- `lpq` : pour lister la liste des « jobs » d'impression en attente ;
- `lprm` : pour enlever un job de la liste en attente.

Selon le système que vous utiliserez seront accessibles plusieurs files d'impression en fonction des imprimantes disponibles. Seront également accessibles des options

13. Seront étudiés plus loin dans ce manuel les services liés à la communication entres utilisateurs, et le téléchargement de fichiers.

dépendantes des caractéristiques de l'imprimante (alimentation manuelle, bac A3, recto/verso, etc.). Renseignez-vous auprès du môssieur qui vous sert d'administrateur système.

Il existe deux familles de commandes héritées de deux versions d'UNIX. Historiquement, la branche BSD proposait les commandes `lp`, `lpq` et `lprm`, et les commandes correspondantes de la branche SysV étaient `lp`, `lpstat` et `cancel`. Aujourd'hui les programmes gérant l'impression sur un système UNIX peuvent utiliser l'une ou l'autre de ces familles, voire les deux.

2.5.2 Le service `at`

La commande `at` permet de lancer une commande à (*at*) une heure particulière. Sa syntaxe est la suivante :

`at` ⟨*date*⟩

On pourra par exemple lancer :

```
$ at 23:00
at> groscalculdelamort
at>   ←———————————————————————————————————— Ctrl-D
$
```

pour lancer à 23 heures le programme `groscalculdelamort` qui utilise beaucoup de ressources. La sortie du programme sera envoyée par mail à l'utilisateur après son exécution. D'autres syntaxes existent pour spécifier la date de lancement :

```
$ at now + 2hours
$
```

pour lancer un programme dans deux heures. Et :

```
$ at 16:20 tomorrow
$
```

pour demain à 16:20,

```
$ at  13:14 10/23/02
$
```

pour lancer un programme le 23 octobre 2002 à 13 heures 14. On peut également lister les «jobs» prévus par la commande `at` grâce à la commande `atq` :

```
$ atq
1       2001-03-14 18:27 a
2       2001-03-15 16:20 a
3       2002-10-23 13:14 a
$
```

La commande `atrm` quant à elle, permet de supprimer une tâche prévue, en donnant son identificateur (le numéro indiqué dans la première colonne des informations renvoyées par `atq`) :

```
$ atrm 1
$
```

On peut également préparer les commandes dans un fichier texte et utiliser ce fichier — notons-le `commandes.at` — comme argument de la commande `at` :

```
$ at -f commandes.at 15:00 tomorrow
$
```

2.5.3 Le service `cron`

`cron` est un service équivalent à ce que le système d'exploitation édité par la société Microsoft appelle les « tâches planifiées ». L'interface utilisateur est la commande `crontab`. Pour savoir quelles sont vos tâches planifiées (dans le jargon UNIX, on dit inspecter « sa crontab »), on lancera la commande :

```
$ crontab -l
no crontab for djobi
$
```

Pour effacer toutes vos tâches planifiées, la commande suivante suffira :

```
$ crontab -r
$
```

Pour définir une ou plusieurs tâches planifiées, la commande :

```
$ crontab -e
```

► 6.2 p. 152 lance un éditeur (qui peut être par défaut ▲vi◀) qui vous demande de saisir des informations dans un format particulier. Chaque ligne ce celui-ci devra contenir les informations suivantes :

⟨*minutes*⟩ ⟨*heures*⟩ ⟨*jour du mois*⟩ ⟨*mois*⟩ ⟨*jour de la semaine*⟩ ⟨*commande*⟩

où :

– ⟨*minutes*⟩ : prend des valeurs de 0 à 59 ;
– ⟨*heures*⟩ : de 0 à 23 ;
– ⟨*jour du mois*⟩ : de 1 à 31 ;
– ⟨*mois*⟩ : de 1 à 12 ;
– ⟨*jour de la semaine*⟩ : de 0 à 7 où 0 et 7 représentent le dimanche.

Par exemple la ligne :

```
10 0 * * 0 rm -rf ~/.wastebasket/*
```

efface tous les dimanches à minuit passé de dix minutes, le contenu du répertoire « poubelle » `~/.wastebasket`. La ligne :

```
0 12 30 12 * echo "bon anniversaire" | mail pote@truc.net
```

► § 4.4 p. 101 envoie un ▲mail◀ à votre pote tous les 30 décembre à midi. La ligne :

```
0 8-18 * * * echo "ça gaze ?" | mail pote@truc.net
```

envoie un mail toutes les heures pile de (8 heures à 18 heures) pour vous rappeler à son bon souvenir. La ligne :

```
0 8-18/4 * * 1,3,5 rm -f ~/tmp/*
```

vidange le répertoire `tmp` de l'utilisateur le lundi, mercredi et le jeudi, et ceci de 8 heures à 18 heures, toutes les 4 heures. Notez enfin que le service `cron` vous informe par mail du résultat de l'exécution de votre tâche. Voir la page de manuel de `crontab` pour de plus amples informations, ainsi que l'exemple du chapitre 6 paragraphe 6.4.5 page 175 illustrant une utilisation du service cron pour mettre des images tirées au hasard en fond d'écran.

2.5.4 L'utilitaire nohup

nohup permet de lancer une commande qui ne sera pas arrêtée lorsque l'utilisateur se déconnectera du système[14]. Lors de la déconnexion, tous les programmes reçoivent en effet le signal HUP, qui a par défaut l'effet de les arrêter[15]. Pour lancer la commande ⟨*cmd*⟩ « en nohup », on tapera :

nohup ⟨*cmd*⟩

Par exemple pour lancer un téléchargement qui va durer plusieurs heures, on peut utiliser ▶wget▴ avec nohup : § 4.5.2 p. 104 ◀

```
$ nohup wget ftp://bidule.chouette.org/pub/truc-1.2.tar.gz
nohup: appending output to 'nohup.out'
  ←——————————————————————————————————————————————— Ctrl-z
[1]+  Stopped                 wget
$ bg
[1]+ wget &
$
```

Le système vous informe alors que la sortie du programme — en l'occurrence wget — est ▶redirigée▴ dans le fichier nohup.out. Fichier qu'on pourra consulter lors de la prochaine connexion au système. § 3.2.1 p. 63 ◀

Que faire alors si le programme lancé avec nohup est un programme interactif qui attend des données au clavier ? Une solution simple est de visualiser le contenu du fichier nohup.out dans un autre terminal avec la commande ▶tail -f nohup.out▴ et répondre aux questions dans le terminal où est lancé la commande en nohup. Par exemple si on ne dispose pas de wget mais uniquement de l'utilitaire ftp, il faudra procéder, avant de lancer le transfert proprement dit, à un série d'interaction avec le programme. § 3.3.6 p. 69 ◀

Conclusion

Vous êtes maintenant armés pour découvrir le chapitre suivant. Celui-ci va vous présenter les utilitaires composant la « boîte à outils » d'un système UNIX. Le bon maniement de ces outils présuppose que vous avez correctement digéré le présent chapitre. C'est par contre celui qui vous fera prendre conscience de toute la puissance de la ligne de commande[16].

14. Il ne s'agit donc par réellement d'un service de planification, mais d'une commande utile dans le cadre de la gestion des tâches.
15. Notons cependant que certains logiciels sont programmés pour détourner ce signal.
16. Le suspense est tout bonnement insoutenable...

3 La boîte à outils

Sommaire

3.1 Introduction à l'expansion
3.2 Redirections et tubes
3.3 Les outils de base
3.4 Le shell en tant que langage
3.5 `grep` et la notion d'expressions régulières
3.6 `awk`
3.7 `sed`
3.8 Études de cas

Cut prints selected parts of lines from each FILE to standard output. With no FILE, or when FILE is -, read standard input.

`man cut`.

L'IDÉE conductrice des développeurs d'UNIX de la première heure était de concevoir des outils élémentaires destinés à une tâche bien spécifique ; chacun de ces outils devait être conçu de telle manière à ce qu'il puisse communiquer avec les autres. En tant qu'utilisateur d'UNIX on retrouve cette philosophie à tout moment : chaque utilitaire, à l'instar d'une pièce de meccano®, est destiné à effectuer une tâche simple et la connaissance de quelques pièces permet de construire ses propres outils. Ce chapitre a pour objet de présenter les mécanismes qui permettent de faire communiquer ces programmes entre eux, puis de présenter les pièces de la boîte à outils les plus communément utilisées sous UNIX.

3.1 Introduction à l'expansion

Avant d'aborder l'étude de quelques outils UNIX, il convient de comprendre quelques-uns des mécanismes entrant en jeu dans l'interprétation des commandes. Comme nous l'avons vu précédemment, certains ►caractères◄ ont une signification particulière pour le shell. Lorsqu'on veut utiliser ces caractères autrement que pour leur signification spéciale, il faut généralement utiliser le caractère *backslash* (\) :

§ 2.1.4 p. 27 ◄

```
$ echo \*
*
$
```

Sans le backslash, le caractère * aurait été remplacé par tous les fichiers du répertoire courant. Un moyen couramment utilisé avec le shell pour protéger les caractères spéciaux est d'utiliser les caractères *quote* ' et *double quote* ". Le premier évite l'expansion de tout caractère :

```
$ echo '$USER #'
$USER #
$
```

Le deuxième permet au caractère $ de conserver sa signification :

```
$ echo "$USER #"
lozano #
$
```

Il sera souvent question de protéger les caractères spéciaux lors de l'utilisation des outils que nous présentons dans ce chapitre.

3.2 Redirections et tubes

Par défaut, tout processus a accès à trois canaux de communications (figure 3.1):

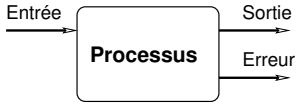


FIGURE 3.1: Les trois flux associés à un processus

1. le flux de *sortie* (*standard output*) par défaut dirigé vers le terminal courant ;
2. le flux d'*erreur* (*standard error*) également dirigé par défaut vers le terminal courant ;
3. le flux d'*entrée* (*standard input*) sur lequel transitent par défaut les données provenant du clavier.

Ainsi, la commande :

```
$ echo bonjour
bonjour
$
```

affiche « bonjour » sur le flux standard. La commande :

```
$ ls amlkjg
ls: amlkj: No such file or directory
$
```

affiche le message sur le flux d'erreur. Et la commande (du shell `bash`) :

```
$ read A
hop là  <------------------------------------------ saisie par l'utilisateur
$
```

attend la saisie d'une ligne sur le flux d'entrée, ligne qui sera stockée dans la variable `A`.

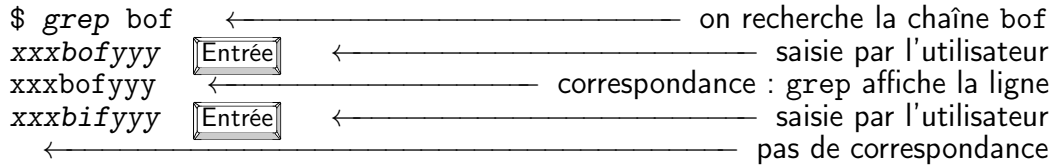
Ce qu'il est intéressant de comprendre, c'est que la grande majorité des commandes et utilitaires d'UNIX suivent le modèle de ces trois flux standard, qu'on pourrait définir de la manière suivante :

- tout affichage est envoyé par défaut sur le flux de sortie ;
- tout message d'erreur est envoyé sur le flux d'erreur ;

– toute donnée peut être lue depuis le flux d'entrée.

À titre d'exemple, la commande `grep` suit ce modèle. Cette commande est destinée à ne récupérer que les lignes d'un fichier contenant une ►chaîne de caractères◄ : 3.5 p. 77 ◄

1. `grep` affiche le résultat sur le flux de sortie ;
2. les messages d'erreurs éventuels sont affichés sur le flux d'erreur ;
3. `grep` peut attendre les données à filtrer sur son flux d'entrée :



On peut indiquer à une commande qui attend des données sur son flux d'entrée, la fin des données à l'aide du caractère `EOF` saisi par la combinaison de touches Ctrl d.

Nous verrons par la suite que c'est précisément parce que la plupart des commandes UNIX suivent ce modèle des trois flux, qu'on peut *composer* les utilitaires avec une grande souplesse. Cette « composabilité » qui fait la grande force d'UNIX est possible grâce à ce qu'on appelle les *redirections*.

3.2.1 Redirections

Il est possible de *rediriger* les trois flux présentés au paragraphe précédent, grâce à une syntaxe particulière qui varie très légèrement selon le shell utilisé.

Redirection du flux de sortie

On peut rediriger le flux de sortie vers un fichier :

```
$ echo bonjour > test.txt
$ ls -l test.txt
-rw-r--r--   1 vincent  users 8 Dec 11 16:19 test.txt
$
```

Le caractère `>` permet la redirection dans le fichier `test.txt`. On peut noter que le choix du caractère évoque la direction du flux « vers » le fichier. Puis :

```
$ more test.txt
bonjour
$
```

La ►commande `more`◄ qui a pour but d'afficher le contenu d'un fichier à l'écran, permet ici de comprendre que `echo` a réalisé son affichage dans le fichier `test.txt` et non à l'écran. § 3.3.1 p. 67 ◄

Redirection du flux d'erreur

On peut dans certaines situations vouloir récupérer les informations envoyées par un programme sur le flux d'erreur. Le principe est le même que pour le flux de sortie, seul l'opérateur est différent :

```
$ ls qsmlkjf 2> erreur.txt
```

```
$ cat erreur.txt
ls: qsmlkjf: No such file or directory
$
```

On peut également coupler les deux redirections de la manière suivante :

```
unprogramme > sortie.txt 2> erreur.txt
```

Pour rediriger les deux flux (sortie et erreur) dans un même fichier, on pourra utiliser :

```
unprogramme > sortie-erreur 2>&1
```

Ou plus simple :

```
unprogramme >& sortie-erreur
```


Redirection en mode ajout

L'opérateur > crée ou écrase le fichier destinataire de la redirection. On peut cependant utiliser le mode « ajout » (*append*) pour ne pas écraser un fichier déjà existant :

```
$ echo bonjour > donnees.txt
$ echo salut >> donnees.txt
$ cat donnees.txt
bonjour
salut
$
```

Redirection du flux d'entrée

Lorsqu'un programme attend des données depuis le clavier, on peut lui fournir directement ces données en utilisant le contenu d'un fichier. On dit alors qu'on redirige le flux d'entrée du programme en question. Le premier exemple que nous proposons utilise la commande `bc` qui est d'après la page de manuel, *an arbitrary precision calculator language.* Une session avec `bc` se présente comme suit :

```
$ bc -q
2+3*5   Entrée
17
quit
$
```

Si on stocke l'expression du calcul (`2+3*5`) dans un fichier, on peut indiquer à `bc` que les données proviennent de ce fichier :

```
$ cat calcul.dat
2+3*5
$ bc < calcul.dat
17
$
```

On dit qu'on a redirigé le flux d'entrée de `bc` depuis le fichier `calcul.dat`.

Le second exemple concerne la commande `mail`. Pour envoyer un mail en utilisant cette commande, le principe est le suivant :

```
$ mail -s "Des redirections" monpote
Bon ben alors...
A+
.
EOT
$
```

l'option `-s` permet de spécifier le sujet. L'argument qui suit (`monpote`) est l'adresse électronique du destinataire du mail. Une fois la commande lancée, on peut taper le texte du message ; le caractère « . » sur une seule ligne permet de finir le texte et d'envoyer le message.

La version originelle de `mail` ne dispose pas de l'option `-s`, cette option apparaît dans le programme standard `mailx`. Sur le système *LINUX* de votre serviteur `mailx` et `mail` font référence au même programme. Attention par contre, d'autres systèmes peuvent être configurés différemment.

Ce qu'il faut comprendre ici, c'est que `mail` attend le texte du message sur le flux d'entrée. On peut alors rediriger cette entrée « depuis » un fichier qui contiendrait le corps du message :

```
$ cat msg.txt     <------------------------- msg.txt contient le message
alors voilà
A+
$ mail -s "redirections" monpote < msg.txt
$
```

Donc en supposant que l'on ait saisi le message dans le fichier `msg.txt`, la dernière commande `mail` peut se traduire par « envoyer un mail en lisant les données depuis le fichier `msg.txt`. » On dit alors qu'on a redirigé le flux d'entrée.

Le trou noir

On trouve généralement sur les systèmes UNIX un fichier particulier dont le nom est `/dev/null` : c'est un « trou noir » qui absorbe toute redirection sans broncher ; ce fichier est un fichier spécial, il ne grossit pas lorsque qu'on y redirige des données. Il peut être utile pour ne pas être dérangé par les messages d'erreur par exemple :

```
id ⟨user⟩ > test 2> /dev/null
```

cette commande écrit dans le fichier `test` les numéros identifiant l'utilisateur ⟨*user*⟩. Si l'utilisateur n'existe pas le message d'erreur n'est pas affiché puisqu'envoyé dans `/dev/null`.

3.2.2 Les tubes (*pipes*)

Les tubes ou *pipes* offrent un mécanisme permettant de composer plusieurs commandes en connectant le flux de sortie de l'une avec le flux d'entrée de la suivante (figure 3.2).

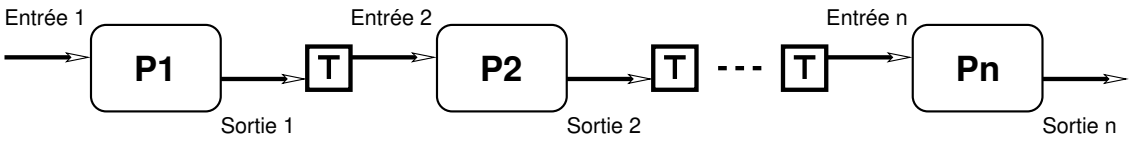


FIGURE 3.2: Principe des tubes

Reprenons l'exemple de l'envoi du mail du paragraphe précédent. Supposons que nous disposions de deux fichiers de données contenant :

```
$ more data-I.txt
machin
chose
$
```

et :

```
$ more data-II.txt
bidule
chouette
$
```

On veut envoyer ces deux fichiers au sieur `monpote`. Pour ce faire :

```
$ cat data-I.txt data-II.txt | mail -s "some data" monpote
$
```

La *sortie* de la commande `cat` — qui réalise la concaténation des données des fichiers `data-I.txt` et `data-II.txt` — est utilisée comme *entrée* de la commande `mail` (corps du message). On peut composer à l'« infini » les tubes comme le montre l'exemple suivant : on veut également envoyer une copie des données triées au gars `monpote`. Le tri peut être fait par la commande ▲`sort`◀ :

► 3.3.2 p. 67

```
$ sort data-I.txt data-II.txt
bidule
chose
chouette
machin
$
```

On peut donc exploiter le fait que `sort` peut trier les données arrivant sur le flux d'entrée, pour envoyer le mail avec *deux* tubes :

```
$ cat data-I.txt data-II.txt | sort | mail -s "some data" monpote
$
```

donc ici, la sortie de `cat` est utilisée comme entrée pour `sort` qui envoie à `mail` le corps du message sur le flux d'entrée. Notez enfin qu'on aurait pu écrire :

```
$ sort data-I.txt data-II.txt | mail -s "some data" monpote
$
```

car `sort` est capable de trier les fichiers qu'on lui passe en arguments. L'exemple avait ici pour but de combiner plusieurs tubes.

Un point important à noter est que lors de l'utilisation de plusieurs commandes par l'intermédiaire de tubes, chacune des commandes est lancée *en même temps* par

le système. Bien que l'on imagine visuellement un flux d'information transitant de gauche à droite, les commandes sont lancées de manière concurrente.

Toute utilisation « sérieuse » du shell d'UNIX passe par une utilisation fréquente des tubes et redirections. Les utilitaires présentés dans la suite de ce chapitre peuvent se voir comme des *filtres*, c'est-à-dire que l'on peut considérer qu'ils filtrent le flux de sortie des programmes auxquels ils s'appliquent. En d'autres termes, un filtre peut être utilisé avec un tube puisqu'il attend systématiquement des données sur son flux d'entrée et envoie le résultat du « filtrage » sur son flux de sortie.

3.3 Les outils de base

Nous donnons ici quelques outils fréquemment utilisés sous UNIX dont il faut connaître l'existence. Chacun de ces outils est présenté avec quelques options, également les plus communes. Pour une utilisation avancée il faudra se référer aux ▶pages de manuel◀. Nous utiliserons pour les exemples le fichier suivant : Ch. 7 p. 15

```
$ cat fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
Janis Joplin 1969
$
```

3.3.1 Afficher

Le rôle de la commande `cat` est de concaténer (*concatenate*) plusieurs fichiers. Le résultat de cette concaténation étant envoyé sur le flux de sortie. Ainsi :

`cat` ⟨*fichier*$_1$⟩ ⟨*fichier*$_2$⟩ ... ⟨*fichier*$_n$⟩

concatène les fichiers de 1 à n et les affiche sur le flux standard. Le filtre `more` peut être utilisé lorsque l'affichage est trop long pour le terminal. Les fichiers sont alors affichés page par page :

`cat` ⟨*fichier*$_1$⟩ ⟨*fichier*$_2$⟩ `| more`

Selon les systèmes on peut utiliser la commande `less`[1] (plus ergonomique) à la place de `more`.

3.3.2 Trier

C'est la commande `sort` :

```
$ sort fichier.dat
Janis Joplin 1969
Jimi Hendrix 1970
Jim Morrison 1971
$
```

1. Encore un jeu de mots d'informaticien : le remplaçant de l'ancêtre `more` est `less`, ah-a ah-a, assez ri... J'apprends à l'instant que la nouvelle évolution de `more` s'appelle `most`, arf, arf, arf... hum, un peu de sérieux.

Par défaut les lignes sont classées selon la première colonne. On peut classer selon les noms :

```
$ sort -k2 fichier.dat
Jimi Hendrix 1970
Janis Joplin 1969
Jim Morrison 1971
$
```

L'option `-k` pour *key* en anglais spécifie la clef du tri. Pour trier selon les dates de :

```
$ sort -n -k3 fichier.dat
Janis Joplin 1969
Jimi Hendrix 1970
Jim Morrison 1971
$
```

Notez l'utilisation de l'option `-n` qui permet d'utiliser l'ordre numérique (par défaut c'est l'ordre lexicographique). L'option `-u` peut quant à elle être utilisée pour supprimer les doublons.

3.3.3 Découper en colonnes

La commande `cut` permet d'afficher une ou plusieurs colonnes d'un flux de données :

```
$ cut -d' ' -f1,3 fichier.dat
Jimi 1970
Jim 1971
Janis 1969
$
```

Le séparateur de colonnes est par défaut la tabulation. On peut en spécifier un autre avec l'option `-d` (pour délimiteur). Ici on a utilisé le caractère espace comme délimiteur. On a ensuite spécifié les champs (option `-f` pour *field*) numéro 1 et numéro 3 (respectivement prénom et date).

Voici pour s'amuser un exemple d'utilisation des tubes :

```
$ cut -d' ' -f1,3 fichier.dat | sort -k2 -n
Janis 1969
Jimi 1970
Jim 1971
$
```

Ce qui permet de trier par année le résultat du `cut` précédent...

3.3.4 Recoller les colonnes

Supposons que l'on dispose d'un fichier :

```
$ cat groups.dat
The Experience
The Doors
Kosmik Band
$
```

On peut alors fusionner ce fichier avec notre fichier d'étoiles filantes :

```
$ paste -d' ' fichier.dat groups.dat
Jimi Hendrix 1970 The Experience
Jim Morrison 1971 The Doors
Janis Joplin 1969 Kosmik Band
$
```

On utilise ici aussi le caractère espace à la place de celui par défaut (tabulation). On pourra également se référer à la commande `join` qui elle, fusionne deux fichiers selon un champ commun.

3.3.5 Compter

La commande `wc` (pour *word count*) permet de compter les caractères, les mots et les lignes d'un flux de données :

```
$ wc fichier.dat
    3       9      54
$
```

Ce qui signifie que le fichier en question possède 3 lignes, 9 mots et 54 octets (les espaces et sauts de ligne sont comptabilisés). On peut également afficher uniquement l'un de ces nombres, par exemple le nombre de lignes :

```
$ wc -l fichier.dat
   3
$
```

les options pour le nombre de mots et de caractères sont respectivement `w` (*word*) et `c` (*byte*).

Attention, même si aujourd'hui un caractère est encore codé sur un octet, on devrait trouver à l'avenir de plus en plus de fichiers texte dont les caractères sont codés sur plusieurs octets. Dans ce cas on pourra utiliser l'option `-m` (pour *multibyte*) pour compter le nombre de caractères et non d'octets.

3.3.6 Tête-à-queue

Les commandes `head` et `tail` comme leur nom l'indique, permettent d'afficher la tête ou la queue d'un fichier ou du flux d'entrée. Par exemple :

```
$ head -n 2 fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

affiche les deux premières lignes de notre fichier. L'option `-c` permet d'afficher les n premiers octets :

```
$ head -c 7 fichier.dat
Jimi He$
```

De manière analogue `tail` affiche la fin d'un flux :

```
$ tail -n 1 fichier.dat
Janis Joplin 1969
$
```

L'option `-c` est également disponible. La nuance avec la commande `tail` est qu'en utilisant un nombre de lignes n (ou d'octets) commençant par `+`, `tail` affiche les données à partir de la n^e ligne (ou n^e caractère). Ainsi :

```
$ tail -n +2 fichier.dat
Jim Morrison 1971
Janis Joplin 1969
$
```

affiche le fichier `fichier.dat` à partir de la deuxième ligne. Une autre utilisation de la commande `tail` est la surveillance d'un fichier qui grossit. En considérant qu'un fichier `data.log` reçoit régulièrement des données, on peut surveiller les dernières données en utilisant l'option `-f` de `tail` :

```
$ tail -f data.log
189
123
234
```

On interrompt l'affichage avec Ctrl c.

3.3.7 Utilitaires disques et fichiers

Chercher

Pour chercher un fichier ou un répertoire on peut utiliser la commande `find`. Cette commande est très puissante et permet de chercher des fichiers avec de très nombreux critères (type de fichier, nom, taille, permission, propriétaires, date de modifications, etc.). Chacun de ces critères peut être combiné par des « ou » ou des « et ». Nous ne vous proposerons ici que quelques exemples :

```
$ find ~ -name core
/home/equipe/lozano/LaTeX/test/core
/home/equipe/lozano/LaTeX/these/core
/home/equipe/lozano/LaTeX/these/src/core
/home/equipe/lozano/src/pas/core
/home/equipe/lozano/install/ummstdod/core
$
```

Cette commande cherche les fichiers dont le nom est `core` à partir du répertoire privé.

```
$ find . -name "*.tex"
./guide-unix.tex
$
```

Celle-ci trouve les fichiers dont l'extension est `.tex` à partir du répertoire courant. Et :

```
$ find ~/cours -type d -a  -name "po*"
/home/equipe/lozano/cours/pov
/home/equipe/lozano/cours/pov.bak
/home/equipe/lozano/cours/images/pov-hof
$
```

trouve tous les répertoires dont le nom commence par `po`. L'option `-a` couple les critères par un « et ». Cette option est implicite et peut donc être omise.

La commande `find` peut-être particulièrement coûteuse lorsqu'on doit scanner toute l'arborescence. C'est pour cette raison que certains UNIX proposent la commande `locate` qui utilise un fichier de données contenant l'emplacement de tous les fichiers sur le système. Ce fichier de données doit être mis à jour régulièrement (généralement une ou deux fois par jour) par `cron`. L'avantage de cette méthode est bien entendu la rapidité de la recherche, l'inconvénient réside dans le fait que le fichier de données n'est pas à jour en temps réel et ne contient pas d'autres informations que les noms des fichiers (et pas leur contenu).

Obtenir des informations

La commande `du` donne la taille qu'occupe un répertoire ou un fichier.

```
$ du -b guide-unix.tex
71680   guide-unix.tex
$
```


L'option `-b` affiche en octets (*byte*). Si l'argument de cette commande est un répertoire le calcul d'occupation disque est fait récursivement sur tous les fichiers et sous-répertoires :

```
$ du -k ~/cours
590     /home/equipe/lozano/cours/unix
380     /home/equipe/lozano/cours/c++
1755    /home/equipe/lozano/cours/images
7736    /home/equipe/lozano/cours/pov
$
```

L'option `-k` est utilisée ici pour afficher en kilo-octets. On peut obtenir le total de chaque sous-répertoire grâce à l'option `-s` (*summarize*) :

```
$ du -sh ~/cours
10M    /home/equipe/lozano/cours
$
```

La commande `df` donne le taux d'encombrement des partitions relatives aux ▶disques locaux◀ et réseaux. L'option `-h` (*human readable*) de cette commande donne l'occupation des partitions en méga-octets, giga-octets, etc.

§ 2.3.2 p. 31 ◀

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda5             197M   56M  131M  30% /tmp
/dev/sda3             972M  471M  450M  51% /usr
/dev/sda6             197M  9.8M  177M   5% /var
count:/export/home0   2.8G  1.7G 1015M  63% /mnt/count/home0
count:/export/home5   969M  247M  672M  27% /mnt/count/home5
count:/export/softs   969M  238M  681M  26% /mnt/count/softs
count:/export/home6   969M  614M  305M  67% /mnt/count/home6
mailhost:/var/spool/mail
                      972M   53M  869M   6% /var/spool/mail
$
```

Archivage

Avertissement : cette section fera sans doute faire des bonds aux puristes qui diront à juste titre que les seules commandes standardisées par Posix sont `pax` et `compress`. J'ai choisi ici de présenter `tar` et `gzip` (et même `bzip2`) car il s'agit de commandes très répandues et également parce que j'ai l'habitude de les utiliser !

L'archivage est une tâche très fréquente lorsqu'on manipule des fichiers. UNIX distingue deux opérations dans l'archivage :

1. l'archivage proprement dit qui consiste à rassembler plusieurs fichiers dans un seul, qui constitue l'*archive* ;
2. la compression de l'archive pour en optimiser la taille. Cette opération doit être effectuée sauf dans de rares cas où les fichiers archivés sont déjà compressés (tas de fichiers Jpeg ou Pdf par exemple).

La première opération est réalisée par la commande `tar` (*tape archiver*, utilisée historiquement pour archiver sur bande magnétique), la deuxième par `gzip`[2]. Nous donnerons ici un exemple très classique d'archivage : l'archivage des fichiers contenus dans un répertoire :

```
$ pwd
/home/equipe/lozano/cours/unix
$ cd ..
$ tar cvf ~/transfert/unix.tar unix   <-------------- création de l'archive
unix
unix/arborescence.eps
unix/arborescence.fig
unix/flux.fig
... etc ...

$ ls -l ~/transfert/unix.tar
-rw-r----- 1 lozano equipe 768000 Dec 13 14:49 unix.tar
$ gzip ~/transfert/unix.tar   <-------------------- compression de l'archive
$ ls -l ~/transfert/unix.tar.gz
-rw-r----- 1 lozano equipe 239039 Dec 13 14:49 unix.tar.gz
$
```

Voici quelques explications :

- l'option `c` de `tar` *crée* l'archive (*create*) ;
- l'option `f` précise que l'archive à créer est un fichier (*file*) (et non un enregistrement sur une bande magnétique) ;
- l'option `v` (*verbose*) permet d'afficher les fichiers (et leur chemin) en cours d'archivage ;
- le nom de l'archive suit (`unix.tar`) puis le nom du répertoire contenant les fichiers à archiver ;
- la commande `gzip` compresse l'archive en la *renommant* avec le suffixe `.gz`.

2. `gzip` est devenu le standard *de facto* de compression sous UNIX ; malgré tout, on ne trouve sous certains systèmes que la commande `compress` qui suit peu ou prou la même syntaxe que `gzip`. Notons également que `gzip` sera à moyen terme probablement supplanté par `bzip2` qui permet d'obtenir un meilleur taux de compression.

L'opération de « désarchivage » se déroule en utilisant les commandes `gzip` et `tar` dans l'ordre inverse. Par exemple, pour décompacter l'archive précédente dans le répertoire `~/tmp` :

```
$ cd ~/tmp
$ gzip -d ~/transfert/unix.tar.gz    <-------------------- décompression
$ tar xvf ~/transfert/unix.tar    <---------------------- décompactage
unix/
unix/arborescence.eps
unix/arborescence.fig
unix/flux.fig
... etc ...

$ ls unix
Makefile    disques.fig   groups.dat      guide-unix.tex~
Makefile~   ecoute.sh     guide-unix.aux  guide-unix.toc
patati.eps  fichier.dat   guide-unix.dvi  missfont.log
... etc ...
$
```

Notez l'option `-d` de `gzip` pour décompresser, et l'option `x` de `tar` pour extraire (*extract*).

La version GNU de `tar` permet d'appeler `gzip` automatiquement grâce à l'option `z`, ce qui permet de réaliser l'archivage et la compression sur la même ligne de commande :

```
$ tar xfz ~/transfert/unix.tar.gz
$
```

On pourra également appeler `bzip2` pour les archives compressées avec cet utilitaire, grâce à l'option `j` :

```
$ tar xfj ~/transfert/unix.tar.bz2
$
```

Puisque nous avons vu les tubes, les puristes rétorqueront que même sans la version GNU de `tar` il est possible de n'utiliser qu'une seule ligne de commande ; ceci en tirant partie du fait que `gzip` et `tar` suivent le modèle des « trois flux » :

```
$ tar cf - unix | gzip -c > ~/transfert/unix.tar.gz
$
```

Petites explications :

- le caractère `-` indique à `tar` que l'archive doit être envoyé sur le flux de sortie ;
- `gzip` compresse les données provenant du flux d'entrée et les envoie sur le flux de sortie grâce à l'option `-c` ;
- le résultat est redirigé dans le fichier `~/transfert/unix.tar.gz`.

La décompression sur une ligne de commande peut être faite comme suit :

```
$ gzip -dc ~/transfert/unix.tar.gz | tar xvf -
```

3.4 Le shell en tant que langage

Jusqu'à maintenant nous avons vu le shell comme un interpréteur de commandes qui se « contente » d'exécuter les commandes qu'on lui passe. En réalité le shell est un

langage de programmation assez évolué permettant d'automatiser un grand nombre de tâches quotidiennes. Le shell (et nous parlerons ici essentiellement du shell `bash`) dispose donc de variables, de tableaux, de structures de contrôle (if, while, for), et également de la notion de fonctions. Toutes ces fonctionnalités sont détaillées au chapitre 5.

Nous vous présenterons ici par le biais d'une « étude de cas » deux fonctionnalités intéressantes du shell : la *substitution de commande* et la boucle « pour ». Le problème est le suivant : suite à une erreur lors d'un transfert de fichier, on a dans un répertoire une liste de fichiers dont les noms sont en majuscules. On veut transformer ces noms en minuscules. L'algorithme que nous allons implémenter en langage de commande pourrait s'écrire :

```
Pour chaque fichier f dont l'extension est JPG Faire
    renommer f en minuscule
```

3.4.1 Afficher des informations avec `printf`

La commande `printf` est un « grand classique » qu'on retrouve dans plusieurs langages de programmation (C, Php pour ne citer que ceux-là). L'idée de cette commande est de réaliser un affichage *formaté* (c'est le « `f` » de `printf`). Cette commande remplace souvent avantageusement la commande `echo` bien plus limitée.

Pour ce qui est de son utilisation, `printf` attend une *chaîne de format* précisant l'allure de l'affichage, suivie d'un nombre variable d'arguments. Par exemple :

```
$ printf "%d est un nombre pseudo-aléatoire\n" $RANDOM
14234 est un nombre pseudo-aléatoire
$
```

Ce premier exemple peut être traduit en français par : je souhaite afficher un entier (`%d`) en base 10, suivi de la phrase « est un nombre pseudo-aléatoire », suivie d'un saut de ligne (`\n`). Dans l'exemple suivant la chaîne de format est suivie par 2 arguments :

```
$ printf "En voici 2 autres : %d et %d\n" $RANDOM $RANDOM
En voici 2 autres : 23943 et 83442
$
```

On comprend donc qu'à chaque `%d` contenu dans la chaîne de format doit correspondre un argument. `printf` reconnaît un certain nombre de type de données, entre autres :

- `%d` pour afficher un entier en base décimale (on peut également utiliser `%o` pour la base octale et `%x` pour la base héxadécimale) ;
- `%f` pour afficher un nombre à virgule flottante ;
- `%s` pour afficher une chaîne ;

Par ailleurs, chacun de ces types de données accepte un certain nombre de modificateurs. En voici quelques-uns :

```
$ printf "==[%010d]==\n" $RANDOM
==[0000007052]==
$
```

pour afficher un entier sur 10 caractères comblés avec des zéros si nécessaire.

```
$ printf "==[%-20s]==\n" $USER
==[lozano               ]==
$
```

pour une chaîne produite sur 20 caractères alignée à gauche (signe « - »). D'autres exemples devraient vous venir à l'esprit après avoir lu le « fantastique manuel ».

3.4.2 Substitution de commande

La substitution de commande est un mécanisme permettant d'exploiter la sortie d'une commande dans l'appel d'une autre commande. La syntaxe permettant de réaliser la substitution est `$(...)`. Pour comprendre comment cela se passe, soit la commande suivante, affichant le nombre de ligne du fichier `fichier.dat` :

```
$ wc -l fichier.dat
3 fichier.dat
$
```

On se débarrasse pour l'exemple du nom de fichier :

```
$ wc -l fichier.dat | cut -f1 -d' '
3
$
```

On peut également s'en débarasser en utilisant une redirection :

```
$ wc -l < fichier.dat
3
$
```

Si on voulait maintenant utiliser cette valeur dans une commande :

```
$ echo Il y a $(wc -l < fichier.dat) musiciens
Il y a 3 musiciens
$
```

Ici, dans un premier lieu, ce qui se trouve entre les caractères `$(...)` est remplacé par ce qu'affiche la commande `wc -l ...`, c'est-à-dire 3, puis la commande `echo` réalise l'affichage. Avec `printf` on écrira :

```
$ printf 'Il y a %d musiciens' "$(wc -l < fichier.dat)"
$
```

Un autre exemple : sachant que la commande `date` affiche la date sous un format que l'on peut définir :

```
$ date +%h%d
Dec15
$
```

Il est possible de sauvegarder un fichier comme suit :

```
$ cp source.cc source.cc.$(date +%h%d)
$
```

La commande `cp` ci-dessus copie le fichier `source.cc` en un fichier dont le nom est `source.cc.` concaténé avec ce que renvoie la commande `date`.

```
$ ls source.cc*
source.cc     source.cc.Dec15
$
```

Il existe une autre syntaxe pour la substitution de commande. En lieu et place de la forme `$(`⟨*commande*⟩`)`, certains vieux shells ne disposent que de la syntaxe utilisant le *backquote* (‘) :

‘⟨*commande*⟩‘

Qu'on se le dise...

3.4.3 La structure `for` de `bash`

Le shell `bash` propose une structure de contrôle nommée `for` que l'on peut traduire par « pour chaque », la syntaxe en est la suivante :

`for` ⟨*variable*⟩ `in` ⟨*liste*⟩ `;` `do` ⟨*com₁*⟩ `;` ⟨*com₂*⟩ `;` `...` `;` `done`

On peut donc écrire quelque chose de la forme :

```
$ for F in *.fig ; do echo "$F.tmp" ; done
arborescence.fig.tmp
disques.fig.tmp
flux.fig.tmp
pipes.fig.tmp
$
```

On affiche ici, pour `F` prenant les valeurs de l'expansion de `*.fig`, la variable `F` suivie de la chaîne `.tmp`, ce qui a, je vous l'accorde, peu d'intérêt.

Notez ici les guillemets dans la syntaxe `echo "$F.tmp"` qui a pour but de ne pas considérer `$F.tmp` comme une liste si jamais il existe un fichier dont le nom contient un espace.

On peut imaginer un autre exemple, qui utilise l'expansion de commande :

```
$ for F in $(find . -name '*.jpg') ; do xzgv "$F" & ; done
$
```

qui lance `xzgv` (un visualiseur d'image) en tâche de fond, pour chaque fichier portant l'extension `jpg` dans le répertoire courant et ses sous-répertoires.

L'exemple ci-dessus possède une limitation : il ne fonctionne pas si le nom des fichiers portant l'extension `jpg` contient un espace (ou un saut de ligne, ce qui est plutôt rare...). Nous vous renvoyons au paragraphe sur la variable ▲IFS◀ pour pallier ce problème.

► § 5.2.6 p. 122

3.4.4 Revenons à nos moutons

« Nos moutons », c'est notre problème de transformation de noms de fichiers majuscules en minuscules. La commande `tr` (*translate*) a la capacité de transformer une chaîne de caractères :

```
$ echo SALUT | tr A-Z a-z
salut
$
```

Donc pour renommer nos fichiers on peut imaginer la commande suivante, qui rassemble les concepts vus dans ce paragraphe :

```
$ for F in $(find . -name '*JPG') ; do    Entrée
```

```
et?> mv "$F" "$(echo $F | tr A-Z a-z)"; done
$
```

Ce qui peut se traduire en français par : tous les fichiers finissant par JPG doivent être renommer en utilisant le résultat de l'affichage du nom original filtré par tr (c'est-à-dire en minuscule).

Cette commande présente l'inconvénient de ne pas fonctionner si les images JPG en question se trouvent dans des sous-répertoires dont les noms contiennent des majuscules. Elle possède en outre les mêmes limitations que l'exemple précédent (noms de fichier contenant un espace ou un saut de ligne). Par ailleurs, il est bon de savoir que ce genre de tâches (copie/déplacement de fichiers en salve) peut être réalisé de manière sûre (en limitant les risques d'écrasement de fichiers) grâce à l'utilitaire mmv disponible dans toutes les bonnes crémeries.

Dans la commande précédente, l'utilisateur a appuyé sur la touche Entrée avant que la commande find ne soit complète. Le shell affiche alors un autre prompt qui se distingue du prompt habituel, pour informer l'utilisateur que la commande est incomplète. La manière d'afficher ce prompt est définie par la ▶variable PS2◀. Dans la suite de ce document nous n'indiquerons plus la pression sur la touche Entrée dans les exemples.

§ 6.1.1 p. 144 ◀

3.5 grep et la notion d'expressions régulières

La commande grep est une commande « célèbre » d'UNIX et permet d'afficher les lignes d'un flux correspondant à un *motif* donné. Par exemple :

```
$ grep Hendrix fichier.dat
Jimi Hendrix 1970
$
```

ou :

```
$ grep Jim fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

Les motifs en question (Hendrix et Jim) dans les deux exemples précédents se nomment en jargon UNIX[3] une *expression régulière* (ou une *expression rationnelle*[4]) (*regular expression* parfois abrégé en *regexp*). L'étude de la théorie des expressions régulières pourrait faire l'objet d'un document de la taille de ce guide de survie. Voici cependant sous forme d'un tableau à quoi correspondent quelques-uns des motifs d'une expression régulière.

3. Et plus généralement dans le domaine de l'étude des langages informatiques.
4. J'utilise pour ma part la première forme, ne pas taper svp...

`.`	désigne n'importe quel caractère	B
`*`	zéro ou plusieurs fois l'élément précédent	B
`?`	zéro ou une fois l'élément précédent	E
`+`	une ou plusieurs fois l'élément précédent	E
`^`	correspond au début de la ligne	B
`$`	correspond à la fin de la ligne	B
`[abc]`	un caractère parmi `abc`	B
`[^abc]`	tout caractère sauf `a`, `b` ou `c`	B
`{`$\langle n \rangle$`}`	exactement $\langle n \rangle$ fois l'élément précédent	E
`{`$\langle n \rangle$`,`$\langle m \rangle$`}`	au moins $\langle n \rangle$ fois et au plus $\langle m \rangle$ fois l'élément précédent[5].	E

La version de grep utilisée pour l'exemple est celle de chez GNU. Cette commande comprend trois ensembles d'expressions régulières (basique, étendue et Perl). Dans le tableau précédent les deux premiers ensembles sont indiqués respectivement par B et E dans la colonne de droite.

ε

Voyons quelles sont les applications de ces motifs. La commande :

```
$ grep '197.$' fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

récupère les lignes finissant par la chaîne « 197 » suivi d'un caractère quel qu'il soit.

Notez qu'il est sage de « quoter » (entourer par des apostrophes) le motif car beaucoup de caractères composant les expressions régulières sont des caractères spéciaux pour le shell.

La commande :

```
$ grep '^Ji' fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

récupère les lignes commençant par la chaîne « Ji ». La commande :

```
$ grep '^J.*1970' fichier.dat
Jimi Hendrix 1970
$
```

récupère les lignes commençant par « J » et contenant « 1970 ». La commande :

```
$ grep -E 'Jimi?' fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

récupère les lignes contenant « Jimi » ou « Jim ». Il est ici nécessaire d'utiliser l'option `-E` (*extended regexp*) pour pouvoir utiliser le caractère `?`. Parmi les options intéressantes de `grep`, en voici trois :

– l'option `-w` (*word*) permet de restreindre le résultat de la correspondance à un mot entier :

5. Dans cette expression, $\langle n \rangle$ ou $\langle m \rangle$ peuvent être omis.

```
$ grep Jim fichier.dat
Jimi Hendrix 1970
Jim Morrison 1971
$
```

mais :

```
$ grep -w Jim fichier.dat
Jim Morrison 1971
$
```

Pour le Gnu grep qui dispose de l'option `-w`, un « mot » est défini comme un ensemble de caractères parmi les lettres, les chiffres et le caractère souligné « _ » (*underscore*).

– l'option `-i` (*ignore case*) permet d'ignorer la casse (minuscule ou majuscule) dans la correspondance avec le motif :

```
$ grep -i jimi fichier.dat
Jimi Hendrix 1970
$
```

– l'option `-v` (*revert match*) inverse le sens de la correspondance et n'affiche que les lignes qui ne correspondent pas à l'expression régulière :

```
$ grep -vi jim fichier.dat
Janis Joplin 1969
$
```

La commande `grep` a été utilisée pour les exemples avec un fichier comme argument, mais elle peut bien évidemment faire la correspondance sur son flux d'entrée et être composée avec des tubes. Par exemple, la commande :

```
$ ps -ef | grep '^lozano'
lozano   11396     1  0 14:53 ?        00:00:00 xterm -ls
lozano   11398 11396  0 14:53 pts/2    00:00:00 -bash
lozano   11512 11398  0 14:58 pts/2    00:00:00 ps -ef
lozano   11513 11398  0 14:58 pts/2    00:00:00 grep lozano
...
$
```

filtre le résultat de la commande `ps` en ne renvoyant que les lignes commençant par la chaîne `lozano`. Ce qui permet d'avoir la liste des processus dont `lozano` est le propriétaire. Notez enfin que la commande `ps -fu lozano` aurait donné un résultat identique...

3.6 awk

On peut voir `awk`[6] comme un filtre permettant d'effectuer des modifications sur les lignes d'un flux ; ces lignes étant sélectionnées par l'intermédiaire d'une expression régulière (dans le même esprit que `grep`) ou d'une condition quelconque. Cependant `awk` dispose d'un langage de programmation spécial (dont la syntaxe pourrait se rapprocher de celle du C) permettant de manipuler et modifier les lignes

6. Du nom de ses auteurs initiaux en 1977, Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan

sélectionnées comme un ensemble de champs (à l'instar de `cut`). La syntaxe générale d'un programme `awk` est une suite de *règles* définies comme un couple de *motifs* et d'*actions* :

- les motifs permettent de sélectionner des lignes dans le flux et sont :
  - soit des expressions régulières et doivent être délimités par des / (par exemple `/abc/`) ;
  - soit des expressions simples (égalité, inégalité,...) ;
- les actions sont exécutées pour chaque motif et sont délimitées par des `{` et `}` (comme par exemple dans l'expression `{print "bonjour\n"}`).

► § 3.1 p. 61 Ces caractères ayant une signification particulière pour le shell, il faut les ▲protéger◄ par des cotes ' .

Voici un petit programme `awk` :

```
$ awk '{print $1,$3}' fichier.dat
Jimi 1970
Jim 1971
Janis 1969
$
```

Dans cet exemple, on sélectionne toutes les lignes (pas de motif spécifié) et on affiche les champs numéro 1 (`$1`) et 3 (`$3`) — correspondant aux prénoms et dates — de chacune de ces lignes. `awk` délimite les champs par un nombre quelconque d'espaces ou tabulations[7]. On peut utiliser une expression régulière comme ceci :

```
$ awk '/^Jim/{print $1,$3}' fichier.dat
Jimi 1970
Jim 1971
$
```

où l'expression régulière « ^Jim » spécifie « ligne commençant par `Jim` ». Voici un autre exemple :

```
$ awk '$3==1970{print "En "$3" "$2"..."}' fichier.dat
En 1970 Hendrix...
$
```

Ici on a sélectionné les lignes à l'aide d'une expression simple : un test d'égalité sur le 3e champ. On peut en outre effectuer un appariement de motif uniquement sur un champ particulier, grâce à l'opérateur ~ de `awk` :

```
$ awk '$2 ~ /lin$/ {print "Cette chère "$1"..."}' fichier.dat
Cette chère Janis...
$
```

L'opérateur ~ permet en effet de filtrer le flux avec une correspondance sur un champ (ici le 2e doit finir par « `lin` »).

On peut utiliser des variables en leur donnant un nom. Leur manipulation (affectation, comparaison,...) s'apparente à celle du langage C. On peut par exemple faire la moyenne des années comme suit :

```
$ awk '{s+=$3} END{print "moyenne : "s/NR}' fichier.dat
moyenne : 1970
$
```

7. Comportement qui peut être changé par l'option `-F`.

on a ajouté pour chaque ligne la valeur du 3e champ à la variable `s`. Et dans la clause `END` (qui n'est exécutée qu'à la fin), on divise par `NR` le nombre d'enregistrements (lignes) du flux (*number of record*). La commande `print` peut également être remplacée par la commande `printf`.

`awk` dispose d'un langage complet, possédant des structures de contrôle (boucle for, boucle while, if,...), un ensemble d'outils pour le traitement des chaînes de caractères, ainsi que des fonctions mathématiques (racine carré, log, fonctions trigonométriques, etc.).

3.7 sed

`sed` (pour *stream editor*) est, comme son nom l'indique, un éditeur de flux. Il permet de faire des transformations sur des chaînes de caractères. Nous donnerons un exemple simple qui consiste à remplacer une chaîne de caractères dans un fichier. La syntaxe est alors :

`sed 's/`⟨*recherche*⟩`/`⟨*remplace*⟩`/'` ⟨*fichier*⟩

ou

`sed 's/`⟨*recherche*⟩`/`⟨*remplace*⟩`/' <` ⟨*fichier*⟩

où ⟨*recherche*⟩ est la chaîne recherchée et ⟨*remplace*⟩ la chaîne qui servira de remplacement. Par exemple, pour changer le nom d'une variable dans un source C :

```
$ sed 's/iterateur/compteur/g' < fichier.c > nouveau.c
$
```

Ici `sed` lit le contenu de `fichier.c` et réalise le remplacement (commande `s` pour *substitute*) de la chaîne « `iterateur` » par la chaîne « `compteur` ». La commande `g` (*global*) assure que la chaîne sera remplacée plusieurs fois par ligne le cas échéant. Enfin notons que, comme `awk`, `sed` renvoie son résultat sur le flux de sortie.

En imaginant que l'on veuille faire la même opération sur plusieurs fichiers, on peut utiliser la commande `for` vue précédemment :

```
$ for F in *.c ; do
et?> sed 's/iterateur/compteur/g' < "$F" > tmp &&
et?> mv -f tmp "$F"; done
$
```

La commande `s` de `sed` dispose également de « registres » qui permettent de réutiliser des parties de la chaîne qui correspondent aux motifs spécifiés. Considérons l'exemple suivant : à partir de notre célèbre fichier `fichier.dat` contenant :

fichier.dat

```
Jimi Hendrix 1970
Jim Morrison 1971
Janis Joplin 1969
```

on veut construire la liste :

```
Hen1970
Mor1971
Jop1969
```

Soit l'expression régulière suivante :

```
^[A-Za-z]* [A-Z][a-z]{2}[a-z]* [0-9]{4}
```

cette expression est capable de reconnaître :

- un début de ligne suivi d'une suite de caractères en minuscules ou en majuscules (`[A-Za-z]*`) ;
- suivie d'un espace ;
- suivi d'une majuscule (`[A-Z]`), suivie de deux minuscules (`[a-z]{2}`), suivies d'un nombre quelconque de lettres en minuscules (`[a-z]*`) ;
- suivi d'un espace ;
- suivi d'exactement quatre chiffres (`[0-9]{4}`).

De manière à utiliser cette expression avec `sed`, il est nécessaire d'utiliser l'option `-r` forçant l'utilisation des expressions régulières étendues (*extended regexp*) :

```
$ sed -r 's/^[A-Za-z]* [A-Z][a-z]{2}[a-z]* [0-9]{4}/xxx/' fichier.dat
xxx
xxx
xxx
$
```

Toutes les lignes correspondent à l'expression régulière, elles sont donc remplacées par « `xxx` ». De manière à réutiliser des parties de l'expression régulière dans le remplacement, on les délimite par les caractères `(` et `)`, les registres ainsi formés sont nommés `\1`, `\2`, etc. Dans notre exemple nous allons récupérer les trois premières lettres du nom et l'année :

```
$ sed -r 's/^[A-Za-z]* ([A-Z][a-z]{2})[a-z]* ([0-9]{4})/\1\2/' \
          fichier.dat
Hen1970
Mor1971
Jop1969
$
```

Notez la position des couples de caractères `(` et `)`. Le registre `\1` contient le nom, et le registre `\2` contient l'année.

Sans la version GNU de `sed` qui propose l'option `-r`, on doit utiliser les caractères `\(` et `\)` pour délimiter les motifs à réutiliser. D'autre part on utilise l'expression régulière pour préciser qu'un motif est répété n fois doit être `\{`n`\}` et non `{`n`}`. Ce qui donne :

```
^[A-Za-z]* \([A-Z][a-z]\{2\}\)[a-z]* \([0-9]\{4\}\)
```

pour la partie gauche de la commande `s`...

Notons, juste pour rire, que la solution à ce problème en utilisant `awk` est la suivante :

```
$ awk '{print substr($2,0,3)$3}' fichier.dat
Hen1970
Mor1971
Jop1969
$
```

Mais gardons à l'esprit que cet exemple vise surtout à illustrer la puissance des expressions régulières.

3.8 Études de cas

Nous vous présentons dans cette section trois « études de cas » mettant en application la plupart des outils standard d'UNIX.

3.8.1 Manipuler la liste des utilisateurs

Sur un système UNIX, on peut obtenir la liste des utilisateurs potentiels du système grâce à la commande `getent passwd`[8]. Cette commande renvoie le fichier contenant les caractéristiques de chaque utilisateur :

```
$ getent passwd
merle_g:x:1321:1300:MERLE Gerard/home/4annee/merle_g:/bin/tcsh
talji_y:x:966:900:TALJI Yacine:/home/1/talji_y:/bin/tcsh
maurin_r:x:945:900:MAURIN Raphaelle:/home/1/maurin_r:/bin/tcsh
sergen_c:x:1415:1400:SERGENT Chris:/home/5/sergen_c:/bin/tcsh
arnaud_l:x:1416:1400:ARNAUD Lionel:/home/5/arnaud_l:/bin/tcsh
lepeti_c:x:1408:1400:LEPETIT Claire:/home/5/lepeti_c:/bin/tcsh
 ... etc ...
$
```

Le flux renvoyé a la structure suivante, de gauche à droite :

- le *login_ name* de l'utilisateur ;
- un « `x` » pour le champ mot de passe ;
- le numéro (`uid` pour *user identifier*) de l'utilisateur ;
- le numéro (`gid` pour *group identifier*) de l'utilisateur ;
- son nom en clair ;
- son répertoire racine ;
- son shell de connexion.

On peut noter que ces champs sont séparés par des « : ». Le nombre d'utilisateurs potentiels est donc le nombre de lignes renvoyées par la commande `getent` :

```
$ getent passwd | wc -l
   277
$
```

Pour obtenir la liste des utilisateurs dont le groupe principal est un groupe particulier, par exemple le groupe numéro 1300, on peut agir comme suit :

```
$ getent passwd | grep -w 1300 | cut -d':' -f5
MERLE Geraldine
ESCOFFIER Carole
GACON-LOCATELLI Anne
BOYAUD Xavier
SIGUIER Vincent
LASNIER Christophe
PEROCHE Jerome
... etc ...
$
```

on fait une recherche de toutes les lignes contenant la chaîne 1300 (ce qui n'est pas exactement ce que l'on cherche, je vous l'accorde, mais lisez la suite), et pour ces

8. Sur votre système il peut s'agir d'une autre commande...

lignes, on affiche le 5e champ (en précisant que le séparateur est « : »). Pour classer par ordre alphabétique, il suffit de rajouter un tube avec `sort` :

```
$ getent passwd | grep -w 1300 | cut -d':' -f5 | sort
ALBERT-GONDRAND Johanne
BARNEOUD Julien
BAROQUE Philippe
ARTOS Romain
...
$
```

Enfin si on voulait afficher le résultat sous la forme prénom-nom, on pourrait rajouter un « petit » `awk` :

```
$ getent passwd | grep 1300 | cut -d':' -f5 | sort |
et?> awk '{print $2" "$1}'
Johanne ALBERT-GONDRAND
Julien BARNEOUD
Philippe BAROQUE
Romain BARTOS
...
$
```

Les lecteurs attentifs auront sans doute noté que le filtre `grep 1300` pourra sélectionner une ligne contenant la chaîne 1300, même s'il ne s'agit pas du numéro de groupe. Pour remédier à ce problème on pourrait utiliser à la place du `grep 1300` :

```
gawk -F: '$4==1300{print $5}'
```

qui aurait tout de même quelque peu surchargé notre exemple à vocation pédagogique (l'option `F` indique à `awk` le séparateur de champ des données en entrée, la variable `OFS` pour *output field separator*)...

3.8.2 Envoyer des mails

Pour faire ce qu'on appelle du *mailing* (envoyer le même message à plusieurs utilisateurs), on peut créer deux fichiers :

- `destinataires` contenant les adresses électroniques des destinataires, et :
- `message` contenant le message à envoyer.

On peut créer `destinataires` manuellement ; on peut également le créer automatiquement, par exemple les destinataires « groupe 1300 » pourraient être construits comme suit :

```
$ getent passwd | awk -F: '$4 == 1300' { print $1 }' > destinataires
$
```

L'envoi des messages est réalisé grâce à une boucle `for` :

```
$ for U in $(cat destinataires); do\
et?> mail -s "salut" $U < messages
et?> done
$
```

Si la commande `mail` de votre système est capable d'envoyer des messages à plusieurs utilisateurs à la fois, vous pourriez également écrire :

```
$ mail -s "salut" $(getent passwd | awk -F: '$4 == 1300'{print $1})
$
```

3.8.3 Estimer l'occupation de certains fichiers

Pour estimer la taille d'un certain type de fichier (comme les images, les fichiers PostScript, ...) on peut procéder comme suit : tout d'abord rechercher ces fichiers avec la commande `find` :

```
$ find ~/ -name "*.ps"
/home/equipe/lozano/LaTeX/test/test.ps
/home/equipe/lozano/LaTeX/local/local.ps
/home/equipe/lozano/LaTeX/local/local.2up.ps
...
$
```

qui donne la liste de tous les fichiers PostScript (extension `ps`) dans la zone de l'utilisateur. La version GNU de `find` permet également d'afficher les caractéristiques des fichiers listés grâce à une option tout à fait analogue au `printf` du C :

```
$ find ~/ -name "*.ps" -printf "%f %k\n"
/home/equipe/lozano/LaTeX/test/test.ps 6
/home/equipe/lozano/LaTeX/local/local.ps 1850
/home/equipe/lozano/LaTeX/local/local.2up.ps 1880
...
$
```

la chaîne de format du `printf` utilise `%p` pour le nom du fichier, et `%k` pour sa taille arrondie en kilo-octets (voir l'aide de la commande `find`). Grâce à un tube avec `awk` on peut afficher les fichiers dont la taille est supérieure à 1 méga-octet :

```
$ find ~/ -name "*.ps" -printf "%p %k\n" |
et?> awk '$2 > 1024 {print $1" "$2}'
/home/equipe/lozano/LaTeX/local/local.ps 1850
/home/equipe/lozano/LaTeX/local/local.2up.ps 1881
/home/equipe/lozano/LaTeX/util/local.ps 1855
/home/equipe/lozano/install/lettre/doc/letdoc2.ps 1385
...
$
```

On peut également obtenir la taille totale de ces fichiers :

```
$ find ~/ -name "*.ps" -printf "%k\n" |
et?> awk '{taille+=$1}END{print taille}'
52202
$
```

Pour finir si on voulait effacer tous ces fichiers, ou seulement ceux dont la taille est supérieure à un seuil donné, il suffirait de procéder comme suit :

```
$ find ~/ -name "*.ps" | xargs rm -f
$
```

Sans argument particulier, `find` affiche la liste des fichiers répondant au critère. La commande `xargs` passe en argument de la commande qui la suit (ici `rm -f`) le flux

qu'elle reçoit en entrée. Donc ici on va exécuter `rm -f` avec comme argument tous les fichiers renvoyés par la commande `find`.

Voici un autre exemple d'utilisation de la commande `xargs` : supposons qu'on veuille chercher dans quel fichier *include* (les fichiers entête du langage C) est défini l'appel système `gethostbyname`[9]. Pour ce faire on va chercher tous les fichiers dont l'extension est `.h` du répertoire `/usr/include` et de ses sous-répertoires :

```
$ find /usr/include -name "*.h"
/usr/include/utempter.h
/usr/include/apm.h
/usr/include/ansidecl.h
/usr/include/bfd.h
...
$
```

Pour l'ensemble de ces fichiers on va utiliser `grep` pour chercher la chaîne de caractère `gethostbyname` :

```
$ find /usr/include -name "*.h" | xargs grep -w gethostbyname
/usr/include/netdb.h:extern struct hostent *gethostbyname __P
((__const char *__name));
/usr/include/tcpd.h:#define gethostbyname fix_gethostbyname
$
```

Cette idée peut être utilisée pour chercher une chaîne dans un ensemble de fichiers. Un autre forme juste pour confirmer l'épigraphe de la préface :

```
$ grep --include='*.h' -rw gethostbyname /usr/include
$
```

demande à grep de faire une recherche récursive (`-r`) dans le répertoire spécifié en dernier argument (`/usr/include/`) en n'examinant que les fichiers dont le nom finit par `.h` (ceci est une « exclusivité » GNU)...

Conclusion

Vous êtes maintenant armés pour étudier les chapitres suivants respectivement consacrés à la communication via le réseau, au développement de programmes et la configuration de son environnement. Le chapitre traitant du développement inclut une partie non négligeable présentant le shell en tant que langage de programmation en mode non interactif. De manière générale le présent chapitre donne un aperçu de la « philosophie » d'UNIX et doit vous permettre d'appréhender différents outils et logiciels que vous serez susceptible de rencontrer.

9. Encore un exemple farfelu lorsqu'on sait que `man gethostbyname` donne l'information recherchée.

4 Communiquer !

Sommaire

4.1 Concepts à connaître
4.2 Quatre grands classiques
4.3 Outils de communication d'Unix
4.4 Courrier électronique
4.5 Le ouèbe

Report bugs to bug-groff@prep.ai.mit.edu.
Include a complete, self-contained example
that will allow the bug to be reproduced,
and say which version of groff you are using.

Extrait du manuel de groff.

Le protocole réseau TCP/IP a été intégré au début des années 80, dans la branche BSD d'UNIX. On trouve donc parmi les utilitaires standard des outils axés sur la communication entre utilisateurs, le transfert de fichiers, l'accès à des machines distantes, etc. Ce chapitre propose tout d'abord un aperçu du concept d'adresse IP et du système de nommage des machines sur Internet (DNS). Il présente ensuite les outils classiques de communication que sont `ftp`, `telnet`, `rlogin` et `ssh`. Après que les outils réseaux propres à UNIX ont été passés en revue, le chapitre est clos par la présentation des utilitaires de base pour envoyer des mails et pour accéder à des ressources localisées sur le Web.

4.1 Concepts à connaître

Pour comprendre de quoi il est question dans ce chapitre, il est nécessaire de connaître quelques concepts liés à la technologie d'un réseau.

4.1.1 Notion d'adresse IP

L'adresse IP (pour *Internet Protocol*) est un ensemble de quatre nombres destiné à identifier de manière univoque une machine sur un réseau local et sur Internet. Cette adresse se compose de quatre octets de la forme :

$$\langle xxx \rangle.\langle yyy \rangle.\langle zzz \rangle.\langle ttt \rangle$$

Par exemple : 172.30.4.78 est une adresse IP. On peut donc potentiellement référencer $256 \times 256 \times 256 \times 256 = 256^4 = 4294967296$ soit plus de quatre milliards de machines [1].

1. En réalité une partie des ces adresses sont dites « privées » c'est-à-dire qu'elles sont réservées à un usage interne et ne sont donc pas visibles de l'extérieur.

Aujourd'hui les réseaux utilisent IP version 4 pour laquelle les adresses sont codées sur 4 octets comme expliqué ci-dessus. Cependant, bien que quatre milliards d'adresses peuvent paraître suffisants pour identifier chacune des messages, on arrive aujourd'hui à une pénurie. C'est la principale raison pour laquelle le protocole IP dispose aujourd'hui d'une version 6 dans laquelle les adresses sont codées sur 16 octets. On peut donc adresser :

$$256^{16} = 340282366920938463463374607431768211456 \text{ machines.}$$

4.1.2 Notion de DNS

De manière à éviter aux utilisateurs d'un réseau de se souvenir des adresses IP des machines, un système de nommage a été mis en place depuis presque vingt ans. Ce système s'appelle le *Domain Name System* ou DNS. Il utilise un protocole qui assure la traduction d'une adresse IP en un nom de machine (`ftp.lip6.fr` par exemple) et inversement. Le protocole repose sur les idées suivantes :

- les machines du réseau sont regroupées en *domaine* ;
- ces domaines sont organisés en arborescence ;
- chaque sous-domaine est géré par un serveur responsable des machines qu'il contient. On est donc en présence d'une base de données décentralisée ;
- un système de cache est utilisé pour limiter le trafic réseau.

Le domaine « racine » est noté « . » ; le point sert également de séparateur de sous-domaine (à l'instar du / pour les répertoires). Sous le domaine racine existe un certain nombre de sous-domaines :

- un domaine pour chaque pays, par exemple `fr` pour la France, `de` pour l'Allemagne, `es` pour l'Espagne, etc.
- le domaine `org` pour les organisations à but non lucratif (par exemple le célèbre `www.gnu.org`) ;
- le domaine `edu` regroupant initialement les universités américaines (par exemple, le serveur web de l'université de Chicago est `www.uchicago.edu`) ;
- le domaine `net` regroupant les machines dont le contenu est axé sur le réseau ;
- le domaine `com` destiné à héberger les sites dont le but est de vous faire acheter des marchandises virtuelles ou non [2] ;
- ...

Chacun de ces domaines peut ensuite contenir des sous-domaines, pouvant également contenir des sous-domaines, etc. L'ensemble forme donc une arborescence dont la racine est le domaine « . », chaque nœud est un sous-domaine, et chaque feuille est une machine [3] ayant une adresse IP donnée. Par exemple, sur la figure 4.1, la machine `ayler` appartient au domaine `freejazz`, lui-même sous-domaine du domaine `fr`.

Lors d'une requête de connexion sur une machine un système complexe d'interrogation de l'arborescence des serveurs de noms permet d'obtenir la traduction d'un nom vers une adresse IP et inversement (pour les lecteurs intéressés se référer au manuel de Albitz et Liu (1998)).

2. Que penser alors d'un slogan d'une célèbre compagnie de téléphone française qui disait il y a quelques années : « bienvenue dans un monde .com » ?

3. Ou plus rigoureusement l'interface d'une machine.

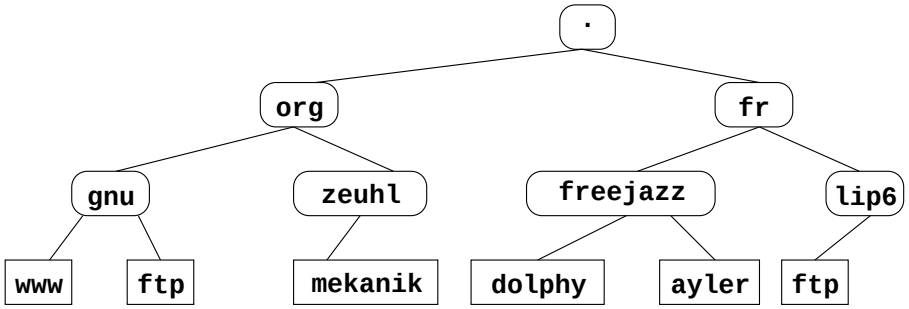


FIGURE 4.1: Arborescence de domaine DNS.

4.1.3 Quelques utilitaires réseau

Suivant les droits que vous auront donnés le ou les administrateurs du site sur lequel vous travaillez, vous aurez la possibilité d'obtenir des informations ayant trait aux adresses et aux noms des machines qui vous entourent.

ping

La commande `ping` permet, à l'aide d'un protocole particulier[4], d'obtenir un écho d'une machine distante, et ainsi de savoir si elle est susceptible de pouvoir échanger des informations avec la vôtre. Sa syntaxe est la suivante :

`ping` ⟨*machine_distante*⟩

Par exemple

```
$ ping ftp.lip6.fr
ftp.lip6.fr is alive
$
```

host

`host` permet d'obtenir l'adresse IP d'une machine d'après son nom et inversement. La syntaxe est :

`host` ⟨*nom_machine* ou *adr_IP_machine*⟩

Par exemple pour la résolution directe :

```
$ host ayler.freejazz.fr
ayler.freejazz.fr has address 145.23.6.5
$
```

Et pour la résolution inverse :

```
$ host 145.23.6.5
5.6.23.145. IN-ADDR.ARPA domain name pointer ayler.freejazz.fr
$
```

On trouvera dans les commandes `nslookup` et `dig` des fonctionnalités plus évoluées. La dernière est plus simple d'utilisation que la première et permet une meilleure intégration dans les scripts shell.

4. Le protocole ICMP pour *Internet Control Message Protocol*.

traceroute

`traceroute` est un utilitaire permettant d'obtenir la liste des machines relais (passerelles) qui interviennent sur le trajet des informations qui transitent de votre machine vers une machine distante. Sa syntaxe est :

`traceroute` ⟨*machine_distante*⟩

Par exemple :

```
$ traceroute www.mekanik.org
1  ghost-lan (182.16.1.254)  1.15 ms *  0.91 ms
2  cisco (183.40.200.65)  2.96 ms  2.90 ms  2.85 ms
3  all.jazz.net (183.38.56.245)  152.64 ms  21.48 ms  21.20 ms
4  jazz.net (183.38.56.105)  25.12 ms  103.28 ms  247.98 ms
5  jazz.net (183.38.56.105)  78.73 ms  25.23 ms  149.62 ms
6  musik.fr (185.230.88.13)  192.93 ms 25.23 ms  149.62 ms
7  intercon.musik.fr (124.57.254.123)  364.57 ms *  56.35 ms
8  all.mekanik.org  (185.83.118.1)  144.20 ms  75.32 ms *
9  www.mekanik.org (185.83.118.1) 52.02 ms 45.06 ms  49.07 ms
$
```

nslookup et dig

La commande `nslookup` permet d'interroger un serveur de nom de manière plus souple que la commande `host`. Un exemple succinct de session avec `nslookup` serait :

```
$ nslookup
Default Server: dolphy.freejazz.fr     <-------------- serveur de nom utilisé
Address:  145.23.65.13
> ayler   <------------------------------------------------- qui est ayler ?
Default Server: dolphy.freejazz.fr
Address:  145.23.65.13

Name:    ayler.freejazz.fr
Address:  145.23.65.57
$
```

Un aspect de `nslookup` qui peut être utile est qu'on peut obtenir la liste des machines qui s'occupent du courrier pour un domaine donné :

```
> set query=MX
> zeuhl.org   <------------------------------ demande d'info sur un domaine
Default Server: dolphy.freejazz.fr
Address:  145.23.65.13

Non-authoritative answer:
zeuhl.org      mail exchanger = zain.zeuhl.org

Authoritative answers can be found from:
zeuhl.org              nameserver = wortz.zeuhl.org
wortz.zeuhl.org        internet address = 185.83.118.2
$
```

Ici on demande à `nslookup` de renvoyer des informations concernant le *mail exchanger* (MX). La réponse illustre le fait que le DNS utilise un système de cache : il y a en effet une réponse « qui ne fait pas autorité » (*non authoritative*), celle de `dolphy`, qui est dans le cache du serveur du domaine local. Il est d'autre part mentionné que l'on peut avoir une réponse « qui fait autorité » (c'est-à-dire digne de confiance) en utilisant expressément le serveur de nom du domaine `zeuhl.org` dont l'adresse est indiquée.

Le même type de requêtes peut être effectué à l'aide de la commande `dig`. Par exemple :

```
$ dig dolphy.freejazz.fr
[...snipsnip...]
;; ANSWER SECTION:
nephtys.lip6.fr.        21184   IN      A       195.83.118.1
;; Query time: 42 msec
;; SERVER: 212.27.53.252#53(212.27.53.252)
[...snipsnip]
$
```

Pour rendre `dig` moins bavarde on peut utiliser l'option `+short` :

```
$ dig +short dolpy.freejazz.fr
145.23.65.13
$
```

La résolution inverse se fait à l'aide de l'option `-x` :

```
$ dig +short -x 185.83.118.2
wortz.zeuhl.org
$
```

4.2 Quatre grands classiques

Comme toutes les applications réseau, les quatre grands classiques que nous présentons dans cette section (ftp, telnet, rlogin et ssh) utilisent des protocoles d'échange de données entre deux machines distantes, basés sur l'architecture suivante :

- le client est un programme tournant sur la machine locale. C'est l'utilitaire manipulé par l'utilisateur lui permettant d'effectuer des transferts de données ;
- le serveur est un programme qui tourne sur la machine distante dont l'existence est généralement ignorée par l'utilisateur.

4.2.1 ftp

ftp (*file transfert protocol*) permet d'échanger des fichiers entre deux machines du réseau. L'application cliente permet de se *connecter* sur la machine distante et d'y déposer (*upload*) ou d'y récupérer (*download*) des fichiers. Il existe des clients graphiques pour effectuer les transferts, néanmoins il est bon de connaître l'interface basique qui est une interface texte. Sa syntaxe est la suivante :

`ftp` ⟨*machine_distante*⟩

Supposons qu'on dispose d'un compte `ftp` dont le login est `albert` avec un mot de passe donné :

```
$ ftp ayler.freejazz.fr
Connected to ayler.freejazz.fr
220- Salut garçon !
Name (albert.freejazz.fr:stundher): albert   <-------- saisie du login
331 Password required for ayler.
Password:  <------------------------------------------ saisie du mot de passe
230 User albert logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
$
```

Dans l'échange entre le client et le serveur on notera qu'à chaque requête le serveur répond avec un code défini par le protocole (220 : serveur prêt pour un nouvel utilisateur, 331 : utilisateur ok, mot de passe nécessaire, 230 : utilisateur connecté, etc.)

Une fois connecté on peut se déplacer dans l'arborescence du serveur avec des commandes analogues à celle d'un shell UNIX (`cd`, `ls`, `pwd`, etc.[5]) :

```
ftp> cd transfert   <-------------------------- changement de répertoire
250 CWD command successful.
ftp> dir   <----------------------------------------- liste de ce répertoire
200 PORT command successful.
150 Opening ASCII mode data connection for directory listing.
-rw-r-----   1 208      200       20241 Nov 20  2000 data.gz
-rw-r-----   1 208      200        2946 Apr  5  2000 data.txt
226 Transfer complete.
$
```

Le transfert de fichier peut se faire dans deux modes :

- *ascii* utilisé pour transférer des fichiers « texte » (ne contenant que des caractères Ascii) ; **attention** c'est généralement le mode par défaut des clients `ftp` en ligne de commande ;
- *binaire* utilisé pour transférer tout autre type de fichier, dans le doute on utilisera ce mode.

Les deux modes de transfert (binaire ou Ascii) permettent de transférer correctement un fichier texte UNIX vers un système Windows ou MacOs, et inversement. En effet, sur ces trois familles de système (UNIX, MacOs et Windows) le caractère de saut de ligne est codé différemment : le caractère 13 pour UNIX, le caractère 10 pour MacOs, et le couple 13 et 10 pour Windows. Le mode Ascii tient compte de ces différences, tandis que le mode binaire récupère chacun des octets sans traitement particulier.

On peut donc transférer un fichier en mode *ascii* comme suit :

```
ftp> get data.txt   <-------------------- récupération du fichier data.txt
local: data.txt remote: data.txt
150 Opening ASCII mode connection for data.txt (2946 bytes).
226 Transfer complete.
2946 bytes received in 2.9 seconds (0.98 Kbytes/s)
$
```

ou en mode *binaire* :

5. Faire `help` pour avoir la liste des commandes disponibles.

```
ftp> bin    <-------------------------------------- passage en mode binaire
200 Type set to I.
ftp> get data.gz
local: data.gz remote: data.gz
200 PORT command successful.
150 Opening BINARY mode connection for data.gz (20241 bytes)
226 Transfer complete.
20241 bytes received in 8.4 seconds (2.35 Kbytes/s)
$
```

On peut alors mettre poliment[6] fin à la connexion :

```
ftp> bye
221 Goodbye.
$
```

Les clients `ftp` disposent généralement des commandes suivantes :

– `mget` : pour récupérer plusieurs fichiers en même temps ;

– `prompt` désactive les confirmations demandées à l'utilisateur lors d'un `mget` ;

– `hash` : affiche un « # » tous les kilo-octets pour indiquer l'avancement du transfert ;

– `!`⟨*cmd*⟩ exécute ⟨*cmd*⟩ sur la machine locale ;

– ...

Notez enfin que toutes ces commandes peuvent être abrégées (*e.g.* `prom` à la place de `prompt`).

Il est courant que des serveurs proposent des fichiers de manière publique. On appelle souvent ces serveurs des « serveurs ftp anonymes », car il n'est pas nécessaire d'avoir un compte pour se connecter sur ces machines. Tous les serveurs distribuant des logiciels libres proposent un accès anonyme. Dans ce cas :
– le login est : `ftp` ou `anonymous`
– le mot de passe : votre adresse électronique de manière à ce que l'administrateur du site puisse faire des statistiques.

4.2.2 telnet et rlogin

telnet et rlogin sont également des protocoles client/serveur et proposent un service de connexion pour utiliser les ressources (processeurs, mémoire, disque, etc.) d'une machine distante. Cette dernière doit exécuter un serveur `telnet` ou `rlogin` pour pouvoir proposer le service. Une session se présente alors sous cette forme :

```
$ telnet mekanik.zeuhl.org    <------------------ nom de la machine distante
Trying 123.45.67.89...
Connected to mekanik.zeuhl.org (123.45.67.89).
Escape character is '^]'.
Green Hat Linux release 16.0 (Klimt)
Kernel 5.2.5-15smp on an i986
login: stundehr
passwd:    <-------------------------------------- saisie du mot de passe
mekanik:~>    <----------------------------------- prompt de la machine distante
```

6. Certains serveurs peuvent d'ailleurs vous reprocher votre manque de politesse, si vous utilisez une autre commande...

Notez qu'il faut bien sûr avoir un compte utilisateur sur la machine distante. Une fois connecté on dispose d'un terminal exécutant un shell particulier ; on peut donc passer des commandes au système distant. Pour effectuer des connexions en mode graphique on se reportera au paragraphe 6.4.4.

rlogin fonctionne de manière identique à telnet à la différence près qu'il considère que par défaut le login sur la machine distante est le même que celui de la machine locale. Il permet en outre d'utiliser un mécanisme de « confiance » permettant d'éviter de saisir le mot de passe. Ainsi, si le répertoire privé de l'utilisateur `stundehr` sur `ayler.freejazz.org` (la machine distante), contient un fichier `.rhosts` :

~/.rhosts
```
# stundehr de mekanik.zeuhl.org est le bienvenu
mekanik.zeuhl.org stundehr
```

on pourra alors se connecter sur `ayler` depuis `mekanik` en tant qu'utilisateur `stundehr` sans avoir à saisir ni login name, ni mot de passe :

```
stundehr@mekanik:~> rlogin ayler.freejazz.org
stundehr@ayler:~>
```

Notez que l'on peut préciser un utilisateur différent pour `rlogin` avec l'option `-l`. Ainsi en étant `albert` sur `ayler.freejazz.org`, on peut exécuter :

```
$ rlogin -l stundehr mekanik.zeuhl.org
```

pour préciser qu'on tente de se connecter en tant qu'utilisateur `stundehr`.

4.2.3 Secure shell (ssh)

Les deux commandes précédentes présentent un inconvénient majeur : les transactions entre le serveur et le client ne sont pas chiffrées ; il est ainsi possible à l'aide d'outils spécialisés de voir passer en clair sur le réseau les commandes tapées par l'utilisateur, et, nettement plus inquitétant, le mot de passe saisi lors de l'initialisation de la connexion. C'est pourquoi on utilise aujourd'hui la commande `ssh` permettant de mettre en œuvre des transactions cryptées. La syntaxe est identique à celle de `rlogin`. La commande suivante, par exemple, permet de se connecter en tant qu'utilisateur `stundehr`.

```
$ ssh -l stundehr mekanik.zeuhl.org
Password:
```

L'autre forme est :

```
$ ssh stundehr@mekanik.zeuhl.org
Password:
```

Si, pour différentes raisons, vous acceptez le fait de saisir un mot de passe pour vous connecter sur une machine distante, vous pouvez passer directement à la section suivante (4.3 page 98). Si vous souhaitez utiliser un mécanisme de confiance avec `ssh` et ainsi *éviter d'avoir à saisir systématiquement un mot de passe*, on aura recours au principe de chiffrement[7] par clé publique.

7. Le chiffrement d'un message est une technique de cryptographie permettant de le « brouiller » pour le rendre illisible. La *cryptographie*, quant à elle, est la science de l'échange des messages protégés.

Dans cette technique de cryptographie, on génère deux clés : une *publique* accessible par tous et une *privée* qui doit être gardée secrète. On peut alors procéder à deux opérations génériques :

Le chiffrement d'un message : pour envoyer un message à un destinataire D on le chiffre avec la clé publique de D. Ce dernier et lui seul pourra déchiffrer le message avec sa clé privée.

La signature électronique : l'expéditeur d'un message peut le signer avec sa clé privée. N'importe quel destinataire pourra s'assurer de l'origine de ce message grâce à la clé publique de celui qui prétend être l'expéditeur.

Pour mettre en application le chiffrement par clé publique dans le cadre de `ssh`, on procède comme suit :

1. création du couple de clés privée et publique ;
2. diffusion de la clé publique vers les machines sur lesquelles on veut se connecter.

Par la suite, une connexion sur une machine distante sera établie après les étapes suivantes :

1. le client contacte le serveur en envoyant la clé ; publique[8] ;
2. le serveur examine si cette clé fait partie des clés autorisées. Dans l'affirmative il génère un nombre aléatoire $\mathcal{A}$ qu'il chiffre avec la clé publique et qu'il envoie au client ;
3. étant donné que ce dernier est en possession de la clé privée il est le seul à pouvoir déchiffrer le message contenant le nombre $\mathcal{A}$. L'accès à la clé privée est généralement verrouillée par une phrase secrète (cf. plus bas) connue par son propriétaire. Elle pourra lui être demandée lors de cette étape. Finalement le client renvoie ce qu'il a déchiffré (qui doit correspondre au nombre $\mathcal{A}$) au serveur ;
4. le serveur compare la réponse du client[9] au nombre qu'il a lui-même généré. Si et seulement si les deux informations sont identiques l'accès est autorisé.

Création du couple de clés

On crée la clé privée et la clé publique grâce à la commande suivante :

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/lozano/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/lozano/.ssh/id_rsa.
Your public key has been saved in /home/lozano/.ssh/id_rsa.pub.
The key fingerprint is:
e2:bb:3b:82:56:da:34:02:d4:85:f3:b3:4b:b9:7b:1e stundehr@mekanik
[...]
$
```

8. Rappelez-vous qu'on a déposé la clé publique sur ledit serveur...
9. En réalité ce sont les signatures MD5 qui sont comparées.

Dans la séquence ci-dessus, on doit saisir une « pass phrase » qui vous sera demandée à l'étape 3 pour que le client accède à la clé privée. Il est possible ici de laisser cette phrase vide ; le client `ssh` aura alors accès à votre clé privée sans qu'on vous demande de mot de passe. Si une phrase secrète est saisie, il sera nécessaire de passer par un « agent ssh » (cf. 4.2.3 page ci-contre) qui vous évitera de saisir cette phrase à chaque connexion.

Votre répertoire contient désormais deux fichiers (la clé privée ne doit bien évidemment être lisible que par vous) :

```
$ ls -1 ~/.ssh/id*
/home/stundehr/.ssh/id_rsa        <-------------------------- la clé privée
/home/stundehr/.ssh/id_rsa.pub    <-------------------------- la clé publique
$
```

On peut bien sûr créer plusieurs couples de clés en précisant le nom du fichier au moment de l'exécution de la commande `ssh-keygen`.

Diffusion de la clé publique

La diffusion de la clé publique consiste à déposer le contenu du fichier correspondant dans le fichier `~/.ssh/authorized_keys` du serveur sur lequel on veut se connecter. Il existe plusieurs méthodes pour faire cette opération. En supposant la configuration suivante :

machine locale	free.jazz.org
utilisateur	ayler

machine distante	cool.jazz.org
utilisateur	miles

voici une méthode[10] qui fonctionne si le fichier n'existe pas encore sur la machine `cool.jazz.org` :

```
$ scp ~/.ssh/id_rsa.pub miles@cool.jazz.org:~/clef.pub
miles@cool.jazz.org's password :
id_rsa.pub                    100%   396     1KB/s   00:00
$
```

Puis *après s'être connecté sur le serveur* `cool.jazz.org`, il faut créer le répertoire `~/.ssh` et un fichier `authorized_keys`[11] :

```
$ mkdir .ssh
$ chmod go-rwx .ssh
$ mv clef.pub .ssh/authorized_keys
$
```

Si le fichier existe déjà, on remplacera la dernière commande par :

```
$ cat clef.pub >> .ssh/authorized_keys
$ rm clef.pub
$
```

qui *ajoute* au fichier `authorized_keys` le contenu de la clé publique copiée sur le serveur. Pour les « r3b3lz », on peut aussi exécuter la commande depuis le client :

```
$ ssh miles@... < ~/.ssh/id_rsa.pub "cat - >> ~/.ssh/autorized_keys"
$
```

10. La commande `scp` est présentée plus loin à la section 4.3.6 page 100.
11. On suppose ici, par souci de clarté, que `~/` est le répertoire de travail.

Cette commande (dans laquelle j'ai remplacé le nom de la machine distante par « [...] ») exécute sur le serveur distant la commande :

```
cat - >> ~/.ssh/authorized_keys
```

qui elle, attend sur son entrée des données qui seront ajoutées au fichier situé à droite de l'opérateur « append » >>. Ces données attendues sont envoyées sur l'entrée de la commande `ssh` qui les transmet au serveur.

Au moins sur la distribution Ubuntu, toutes ces solutions peuvent être automatisées grâce à un script shell connu sous le nom `ssh-copy-id` qui prend en argument le login et la machine distante sur laquelle on veut copier la clé publique.

Déverouillage de la phrase secrète

Si vous avez suivi jusqu'ici, que vous *avez saisi une « pass phrase » non vide* au moment de la création des clés, alors lors d'une connexion `ssh` sur la machine distante vous avez dû obtenir :

```
$ ssh miles@cool.jazz.org
Enter passphrase for key '/home/ayler/.ssh/id_rsa':
$
```

Ce qui, je vous entends le dire *in petto*, ne nous avance guère... C'est vrai : nous étions censés simplifier la connexion ssh et « voilatipa » qu'on nous demande un mot de passe ! C'est ici qu'intervient la notion d'« agent ssh » un logiciel qui garde trace des clés utilisées pendant une session.

Je ne rentrerai pas ici dans les détails de l'installation ou du lancement d'un agent ssh. La plupart des distributions en lance un (sous le nom `ssh-agent`) au moment du lancement de la session X. L'environnement Gnome pour ne pas le citer, lance même son propre gestionnaire de clés (gnome-keyring-daemon).

Lorsque l'agent ssh est actif, on peut lui demander gentiment d'ajouter dans son « trousseau » notre clé privée :

```
$ ssh-add
Enter passphrase for /home/ayler/.ssh/id_rsa:
Identity added: /home/ayler/.ssh/id_rsa (/home/ayler/.ssh/id_rsa)
$
```

Sans argument cette commande demande le déverrouillage de la clé correspondant au fichier `~/.ssh/id_rsa`.

À partir de maintenant et jusqu'à la fin de la session,
vous devriez pouvoir vous connecter sur votre serveur
sans saisir de mot de passe (ouf...)

Quelques petites choses que vous aimeriez faire :

- lister les clés gérées par l'agent : `ssh-add -l` ;
- effacer toutes les clés mémorisées par l'agent : `ssh-add -D` ;
- changer la phrase secrète associée à une clé :
  `ssh-keygen -p -P`⟨*ancienne*⟩ `-N`⟨*nouvelle*⟩ `-f` ⟨*fichier*⟩
  remplace l'⟨*ancienne*⟩ phrase par une ⟨*nouvelle*⟩ pour la clé privée stockée dans le ⟨*fichier*⟩.

Pour avoir accès à vos serveurs préférés par `ssh` sans avoir à saisir de mot de passe, il vous faudra :

1. distribuer votre clé publique sur chacun des serveurs ;
2. déverrouiller votre clé secrète en début de chaque session.

Certains agents vous permettent de ne faire qu'une fois pour toute la deuxième opération, d'autres vous obligent à modifier le démarrage de votre session pour automatiser ce déverrouillage.

4.3 Outils de communication d'Unix

Voici quelques outils qu'on retrouve sur un système UNIX. Dans la mesure où ces outils autorisent les utilisateurs à agir sur des machines distantes, il est possible que certains de ces services ne soit pas disponibles, pour des raisons de sécurité, sur les systèmes où le lecteur les testera.

4.3.1 who

`who` permet d'obtenir la liste des utilisateurs connectés sur le système. Il s'agit des utilisateurs connectés sur la console (l'écran situé physiquement à coté de la machine), ou par telnet, ou par rlogin, etc.

```
$ who
albert       console      Mar 20 17:04
albert       pts/0        Mar 20 17:04
stundehr     ttyp1        Mar 20 17:28
$
```

La deuxième colonne indique les terminaux où sont connectés les utilisateurs `albert` et `stundehr`. Cette commande peut aussi indiquer si l'utilisateur est connecté à distance sur le système et le cas échéant la machine depuis laquelle il est connecté.

On trouve parfois la commande `rwho` qui indique quels utilisateurs sont connectés sur les machines UNIX du réseau local. L'avantage de cette commande réside dans le fait qu'il n'est pas nécessaire de se connecter sur une machine pour savoir quels autres utilisateurs l'utilisent.

4.3.2 Changer d'identité

La commande `su` permet de devenir un autre utilisateur. Il faut bien sûr connaître son mot de passe :

`su` ⟨*user*⟩

Par exemple en étant connecté en tant qu'utilisateur `albert`, on peut faire :

```
$ su stundehr
Password:   <-------------------------- saisie du mot de passe de stundehr
$
```

on « devient » alors l'utilisateur `stundehr`. C'est-à-dire qu'on lance un shell en tant que `stundehr`. Notez qu'en faisant :

```
$ su - stundehr
```

```
Password:
$
```

le shell lancé est alors un ►shell de connexion◄ (*login shell*). § 6.1.6 p. 148 ◄

4.3.3 write

`write` envoie un message particulier sur le terminal d'un utilisateur :

```
$ write stundehr
bonjour mon gars
  ←———————————————————————————————— Ctrl-d pour finir le message
$
```

Un utilisateur peut être spécifié sous la forme ⟨*user*⟩@⟨*machine*⟩, s'il est connecté sur une machine distante. En outre tout utilisateur peut désactiver la réception de message avec la commande :

```
$ mesg n
$
```

et :

```
$ mesg y
$
```

pour la réactiver.

4.3.4 finger

`finger` renvoie les informations de connexion d'un utilisateur donné sur une machine donnée. Ces informations concernent l'utilisateur lui-même (nom d'état civil, répertoire de travail, shell de connexion), ses dernières connexions, s'il a lu son mail, et plein d'autres choses. La syntaxe est :

`finger` ⟨*utilisateur*⟩`@`⟨*machine*⟩

Si ⟨*utilisateur*⟩ dispose dans son répertoire racine sur ⟨*machine*⟩, des fichiers `.plan` ou `.project`, leur contenu est alors affiché par la commande `finger`. On peut donc mettre dans ces fichiers des informations qu'on désire diffuser (comme ses coordonnées par exemple). Si l'utilisateur utilise un fichier ►`.forward`◄, son contenu est également affiché par `finger`. Voici un exemple : § 4.4.3 p. 103 ◄

```
$ finger bidule@machine.distante.fr
Login: bidule                     Name: Machin Bidule
Directory: /home/bidule           Shell: /bin/bash
On since Mon Apr 30 10:32 (CEST) on pts/1 from :0.0
   22 minutes 10 seconds idle
On since Mon Apr 30 09:23 (CEST) on pts/2 from :0.0
On since Mon Apr 30 10:55 (CEST) on pts/3 from :0.0
   2 minutes 50 seconds idle
No mail.
No Plan.
$
```

Étant notoirement faible sur le plan de la sécurité, le serveur `finger` est souvent désactivé sur les machines.

4.3.5 talk

`talk` — comme son nom l'indique — permet de « parler » avec un autre utilisateur sur la même machine ou sur une machine distante. Une fois la commande lancée, la fenêtre du terminal est divisée en deux parties, et chaque utilisateur voit ce que tape l'autre en temps réel [12]. Pour initier une discussion, il est nécessaire que les deux protagonistes lancent la commande `talk`, dont la syntaxe est :

`talk` ⟨*utilisateur*⟩`@`⟨*machine*⟩

Ainsi :
- l'un (`stundehr`) lance : `talk albert@ayler.freejazz.org`
- l'autre (`albert`) : `talk stundehr@mekanik.zeuhl.org`

Après quelques secondes, une fois la connexion établie, chacun des utilisateurs a devant les yeux un écran où chacun voit le texte de l'autre dans la partie inférieure. :

Utilisateur : `albert` — Machine : `ayler.freejazz.fr`

```
Salut stundehr !

------------------------------
```

Utilisateur : `stundehr` — Machine : `mekanik.zeuhl.org`

```
------------------------------
Salut stundehr !
```

`stundehr` peut alors répondre à son tour :

Utilisateur : `albert` — Machine : `ayler.freejazz.fr`

```
Salut stundehr !

------------------------------
Salut albert !
```

Utilisateur : `stundehr` — Machine : `mekanik.zeuhl.org`

```
Salut albert !

------------------------------
Salut stundehr !
```

Le caractère `EOF` (généralement `Ctrl-d`) met fin à la session, et la combinaison `Ctrl-l` rafraîchit l'écran, ce qui est parfois utile.

4.3.6 « remote » commandes

Nous avons déjà vu une commande de la famille « `r` » (`r` pour *remote* c'est-à-dire « à distance ») : `rlogin`. Il existe deux autres commandes de ce type : `rcp`

12. Plus précisément, à la vitesse autorisée par le débit du réseau.

permettant de copier un fichier sur (ou depuis) une machine et `rsh` pour lancer un shell distant.

Aujourd'hui il est vivement déconseillé d'utiliser ces commandes pour des raisons de sécurité car toutes les informations transitent en clair sur le réseau. Nous présenterons donc leurs équivalents sécurisés à savoir `scp` et `ssh`.

- `scp` permet de copier un fichier d'une machine à une autre :
  `scp` ⟨*utilisateur*⟩`@`⟨*machine*⟩`:`⟨*fichier*⟩ ⟨*fichier_dest*⟩
  Par exemple :

```
$ rcp ayler@mekanik.zeuhl.org:/home/ayler/ghost.mp3 .
$
```

- `ssh` permet — en plus de la connexion présentée au § 4.2.3 — de lancer une commande sur une machine distante. La syntaxe est :
  `ssh` ⟨*machine*⟩ ⟨*commande*⟩
  Par exemple :

```
$ ssh mekanik.zeuhl.org ls -l
-rw-r-----   1 albert  sax    199 Sep  1  2000 ghost.mp3
-rwxr-----   1 albert  sax     61 May  4  2000
-rwxr-----   1 albert  sax     77 May  4  2000
$
```

On peut noter qu'à l'instar de la commande `rlogin`, les commandes `ssh` et `scp` acceptent l'option `-l` permettant de spécifier un autre utilisateur pour la connexion.

4.4 Le courrier électronique [13]

Bien qu'il existe de nombreux logiciels de lecture de mail sous UNIX [14], il est bon de connaître les outils de base que propose UNIX pour pouvoir lire son courrier simplement en lançant une session `ssh` sur la machine sur laquelle vous recevez votre courrier.

4.4.1 Format des adresses

Une boîte aux lettres électronique est identifiée par une adresse de la forme :

⟨*nom*⟩`@`⟨*domaine*⟩

La partie ⟨*domaine*⟩ peut désigner une machine existante, mais la plupart du temps désigne un domaine générique, comme par exemple dans :

```
bart@uchicago.edu
```

`uchicago` est un nom de domaine et non de machine. De manière analogue, la partie ⟨*nom*⟩ de l'adresse électronique ne correspond pas nécessairement à un utilisateur du système distant, ainsi l'utilisateur `zappa` qui a un compte UNIX sur une machine du domaine `uncleme.at` peut avoir l'adresse suivante :

```
frank@uncleme.at
```

13. Le mél comme disent les académiciens, le courriel comme disent les Québécois, l'e-mail comme disent les anglophones, le mail comme disent les Français.

14. Pour s'en assurer, on peut faire une recherche sur le site `http://sourceforge.net`.

On peut trouver également la forme :

```
Frank Zappa <franck@uncleme.at>
```

forme permettant de rajouter des fantaisies autour d'une adresse existante, tant que les fantaisies en question se limitent à l'extérieur des < et >.

4.4.2 Mail user agents

Parmi l'ensemble des logiciels utilisés sous UNIX pour acheminer le courrier existent deux familles :

- les *mail transport agents* (MTA) : les logiciels chargés d'acheminer le courrier d'une machine à une autre ; on peut faire l'analogie avec le facteur ou la poste qui délivre les courriers dans les boîtes ;
- les *mail user agents* (MUA) : logiciels destinés à l'utilisateur permettant à la fois de consulter sa boîte aux lettres, et d'envoyer des courriers (au mail transport agent qui fait suivre).

En tant qu'utilisateur UNIX, on est confronté à la deuxième catégorie, la première étant réservée aux tâches d'administration système. Il existe sous UNIX une multitude de logiciels de messagerie utilisateur, le plus commun est `mail`. Ce `mail` existe également sous d'autres variantes `Mail` et `mailx`. Tous ces utilitaires ont des syntaxes généralement légèrement différentes selon les systèmes [15]. Toujours est-il que le principe est souvent le suivant, pour envoyer un mail à `franck@uncleme.at` :

```
$ mail franck@uncleme.at
Subject: Dental floss
Hi.
.    ←———————————— Un point seul sur une ligne pour envoyer le message
$
```

En ayant déjà rédigé le message dans un éditeur de texte (Emacs pour ne pas le nommer), on peut directement utiliser les redirections [16] :

► § 3.2 p. 62

```
$ cat tofranck
Hi.
Jazz is not dead,
It just smells funny.
$ mail -s "Dental floss" franck@uncleme.at < tofranck
$
```

Certains logiciels proposent l'option `-c` pour positionner le champ `CC:` (copie carbone), et bien d'autres encore. Par exemple `mutt` (`http://www.mutt.org`) utilise l'option `-a` pour attacher un fichier. Et :

```
$ mutt -a arf.jpg -s "Blow j." franck@uncleme.at < tofranck
$
```

où `arf.jpg` est une image au format Jpeg qui sera envoyée sous forme d'attachement.

15. Nous vous invitons donc à compulser les pages de manuels traitant de ces utilitaires.

16. Comme indiqué précédemment la commande `mail` utilisée ici accepte l'option `-s` ce qui n'est pas toujours le cas sur un système autre que *LINUX*. On s'orientera dans ce cas vers la commande `mailx`.

4.4.3 Faire suivre son courrier

Il arrive souvent d'avoir des comptes sur plusieurs machines UNIX. Dans ce cas on peut vouloir ne relever qu'une boîte aux lettres. Pour ce faire il est nécessaire de faire suivre le courrier destiné aux autres boîtes sur la seule boîte que l'on veut consulter. On utilise alors un fichier nommé `.forward` qui doit se situer dans le répertoire privé. Par exemple :

~/.forward
```
albert@ayler.freejazz.fr
```

permet de faire suivre tout courrier arrivant sur la machine concernée à l'adresse `albert@ayler.freejazz.fr`. Si l'utilisateur (`stundher`) utilisant ce `.forward` souhaite également conserver une copie sur la machine, il peut ajouter une ligne à son fichier :

~/.forward
```
\stundehr
albert@ayler.freejazz.fr
```

Notez l'utilisation du \ pour éviter une récursion sans fin.

Tout ce qui est raconté ici au sujet de l'acheminement du courrier à l'aide du fichier `.forward` ne sera effectif que si la machine UNIX du réseau local sur laquelle on crée le `.forward` est effectivement la machine qui reçoit le courrier et que le MTA de cette machine utilise le mécanisme du `.forward`.

4.5 Le ouèbe

De manière analogue à ce qu'il a été dit précédemment sur le courrier électronique, il existe de nombreux navigateurs web très intuitifs comme Netscape, Mozilla, Opera, pour ne citer que les plus connus. Le but de ce paragraphe est la présentation d'outils particuliers ne présentant pas d'interface graphique, mais ayant un intérêt évident.

4.5.1 Format des adresses

Le standard pour localiser une ressource sur le réseau est l'*uniform resource locator* ou URL, ou encore « localiseur uniforme de ressources », sa forme la plus simple est toujours :

⟨*protocole*⟩`://`⟨*machine*⟩`/`⟨*fichier*⟩

Par exemple dans l'url :

`http://www.freejazz.fr/dolphy/horns/bassclarinet.jpg`

le protocole est `http` (*hypertext transfert protocol*, le protocole de transfert hypertexte utilisé sur le Web), la machine est `www.freejazz.fr`, et le fichier désigne une image nommée `bassclarinet.jpg` dans le répertoire `dolpy/horns`. De même :

`ftp://ftp.lip6.fr/pub/Linux/Readme`

désigne le fichier `Readme` stocké dans le répertoire `pub/Linux` de la machine dont le nom est `ftp.lip6.fr` et accessible via `ftp`.

4.5.2 Wget l'aspirateur

wget est un utilitaire du projet GNU permettant le téléchargement de fichiers en utilisant le protocole `ftp` ou `http`. wget effectue ces téléchargements sans intervention de l'utilisateur et en suivant éventuellement des liens ou les sous-répertoires d'un serveur web ou ftp :

```
$ wget http://www.freejazz.fr/dolphy/bassclarinet.jpg
```

téléchargera le fichier `bassclarinet.jpg`. Mais cet utilitaire est particulièrement utile pour télécharger plusieurs fichiers. Par exemple pour aspirer tout ou partie d'un site ouèbe, on pourra écrire :

```
$ wget -r -l1 http://www.bidule.fr/machin/chose
```

qui sauvera :

1. le fichier `index.html` du répertoire `machin/chose` ;
2. tous les liens apparaissant dans ce fichier `index.html` ; c'est la signification des options `-r` (récursif) et `-l1` lien de « niveau 1 ».

En lançant :

```
$ wget -r -l2 http://www.bidule.fr/machin/chose
```

on télécharge également les liens apparaissant dans les pages mentionnées dans le fichier `index.html` (liens de « niveau 2 »).

Lors de l'utilisation de l'option `-r` pour suivre récursivement les liens jusqu'à un niveau donné, il est souvent utile de demander à wget de ne pas remonter au répertoire parent. Dans l'exemple précédent, ceci permet de se cantonner au répertoire `machin/chose` et ses descendants. On formule ce souhait à l'aide de l'option `-np` ou `--no-parent`.

Pour finir avec cette courte introduction à wget, il n'est pas inutile de noter l'usage qui suit. Supposons que la page désignée par :

```
http://www.truc.fr/machin/chose.html
```

contiennent des liens sur des images au format Jpeg portant l'extension `jpg`, alors la commande :

```
$ wget -r -l1 -Ajpg http://www.truc.fr/machin/chose.html
$
```

récupérera uniquement ces fichiers à l'exception de tous les autres liens de la page nommée `chose.html`.

Notons que pour les commandes transférant plusieurs fichiers, wget créera de lui-même sur la machine locale, un répertoire pour la machine hôte contenant le cas échéant, d'autres répertoires apparaissant dans l'url. Pour éviter cela, on pourra faire appel aux options :

- `-nH` ou `--no-host` pour ne pas créer de répertoire correspondant à la machine hôte ;
- `-nd` ou `--no-directories` pour ne pas créer de répertoire du tout ;
- `--cut-dirs`=⟨*nombre*⟩ pour éviter de créer ⟨*nombre*⟩ niveaux dans l'arborescence de répertoires.

Par exemple :

```
$ wget -nH --cut-dirs=1 -r -l1
   http://www.freejazz.fr/dolphy/horns/
```

créera le répertoire `horns` dans le répertoire courant pour y sauver les fichiers aspirés.

Comme la plupart de ces utilitaires « ne faisant qu'une chose mais le faisant bien », on trouvera une foultitude d'options diverses et variées permettant d'arriver à ses fins quelle que soit l'exigence de la requête.

D'autres utilitaires effectuant le même type de tâche existent. Parmi eux les logiciels curl et lftp disponibles chez votre crémier le plus proche.

4.5.3 Lynx l'extraterrestre

lynx (jeu de mots d'informaticiens anglophones sur *links*, les liens) est un butineur de la toile (*web browser*) en mode texte. Le propos de lynx est de compulser des pages au format html. Ce browser peut être utile dans les situations où l'utilisateur veut éviter de lancer les logiciels habituels gourmands en ressources, ou simplement lorsque l'environnement courant ne permet pas d'utiliser le mode graphique. L'utilisation de lynx est simple :

- « flêche bas » : lien suivant dans la page ;
- « flêche haut » : lien précédent ;
- « flêche droit » : suivi du lien ;
- « flêche gauche » : dernier lien dans l'historique ;

En outre les touches :

- d : permet de télécharger (*download*) un fichier désigné par le lien courant ;
- z : permet d'interrompre le téléchargement ;
- g : permet d'aller (*go*) vers une autre url ;
- q : permet de quitter.

lynx possède une aide en ligne très complète qu'on peut consulter en pressant la touche ?, et un site web `www.lynx.org`. Ainsi pour consulter le site `www.opensource.org`, on tapera :

```
$ lynx www.opensource.org
```

On peut également se connecter par ftp sur une machine en utilisant le format particulier des url :

```
$ lynx ftp://stundehr@mekanik.zeuhl.org
```

dans l'hypothèse où l'on dispose d'un compte dont le login est `stundehr` sur la machine `mekanik.zeuhl.org`.

Stéphane CHAZELAS me souffle que lynx est dépassé et lourd. D'autres programmes existent permettant de naviguer en mode texte sur le ouèbe, comme w3m et plus particulièrement elinks ou links2...

Conclusion

Ce (court) chapitre a eu pour but de présenter les utilitaires ayant trait à la communication sous UNIX. Il est intéressant de noter que tous les outils autour du

transfert de données, de messagerie instantanée ou différée, d'accés aux systèmes à distance, etc. existent depuis longtemps et sont utilisables en mode texte. Ce deuxième aspect qui pourrait paraître curieux à l'heure d'aujourd'hui, assure à l'utilisateur la possibilité de manipuler des outils réseaux légers s'il le souhaite, et surtout lui permet *d'automatiser* ses requêtes ce qui est l'objet du chapitre suivant.

5 Développer !

Sommaire

5.1 Éditer un fichier
5.2 Faire des scripts en shell
5.3 Makefile
5.4 Faire des projets en langage C


```
% Les 5 premières lignes du source TeX
\chapter{Développer !}
\label{chap-developper}
\begin{epigraphe}{\recursion}
\verb&\verb@\verb|\verb#\verb°\verb/\verb+% Les 5 premières+\\/°#|@&
```

\recursion.

DÉVELOPPER de *to develop*, « mettre au point » en anglais, est peut être l'activité qu'il est idéal de réaliser sous un système UNIX. Il dispose en effet d'une palette impressionnante ►d'outils◄ — dédiés à une tâche spécifique — qu'on peut voir comme des composants logiciels pouvant communiquer entre eux de manière élégante et homogène. Nous verrons en ouverture de ce chapitre une présentation détaillée des concepts fondamentaux de la programmation en shell ; la partie suivante est consacrée à l'étude de l'utilitaire make, outil puissant permettant de gérer un projet de développement. Ce chapitre est clos par la présentation de l'utilisation d'un compilateur C sur un système UNIX. Pour développer sous UNIX, il faut en outre savoir comment *éditer* un fichier, ce pré-requis fait l'objet de la première section de ce chapitre.

Chap. 2 p. 21 ◄

Certaines sections de ce chapitre exigent que vous ayez quelques notions de programmation ou d'algorithmique : savoir ce qu'est une variable et une boucle ne seront pas inutile pour comprendre la section 5.2 sur les scripts en shell. La section 5.4 sur les projets en langage C présuppose que vous connaissez ce langage, sa lecture est donc très certainement inutile si ça n'est pas le cas.

5.1 Éditer un fichier

Vous entendrez sans doute un jour quelqu'un vous dire : « sous UNIX, tout est fichier [1] ». En outre, comme nous avons vu que les commandes communiquent entre elles à l'aide de flots de texte, la tâche qui consiste à éditer (c'est-à-dire créer/-modifier) un fichier contenant du texte est une tâche extrêmement courante sous UNIX.

1. Si vous ne l'avez jamais entendu, tentez de la placer lors d'une soirée entre amis, cela fait toujours de l'effet.

5.1.1 Sans éditeur

Même si cela reste rare, il peut arriver d'avoir à créer un fichier contenant du texte sans éditeur de texte. Dans ce cas on peut utiliser les ▴redirections◂, par exemple :

► § 3.2.1 p. 63

```
$ echo bonjour > fichier.txt
$
```

stocke la chaîne « `bonjour` » dans le fichier `fichier.txt`. On peut même, éventuellement ajouter des sauts de lignes (caractère « `\n` »), ou des tabulations (caractères « `\t` ») en utilisant l'option `-e` de `echo` :

```
$ echo -e "a\nb\nc\td" > bonjour.txt
$ cat bonjour.txt
a
b
c       d
$
```

Pour être sûr d'arriver à ses fins dans ce genre de situation il est souvent préférable d'utiliser la commande `printf` plutôt que la commande `echo`, en écrivant :

```
$ printf "a\nb\nc\td\n" > bonjour.txt
$
```

Ceci parce que la commande `echo` de votre système peut ne pas réagir à l'option `-t`.

Une autre manière de faire est d'utiliser la commande `cat` avec une redirection :

```
$ cat > bonjour.txt      ←——— attente de données destinées à bonjour.txt
a  [Entrée]
b  [Entrée]
 [Ctrl] [d]      ←——————————————— caractère de fin de fichier
$
```

Ces méthodes ne permettent évidemment pas de modifier le contenu du fichier créé, autrement qu'en l'écrasant avec de nouvelles données.

5.1.2 Avec un éditeur

Deux éditeurs sont très répandus sur les systèmes UNIX, il s'agit de vi (prononcer « vi aïe ») et Emacs (prononcer « et max » ou « i max »). Les paragraphes 6.2 page 152 et 6.3 page 154 présentent respectivement ces deux éditeurs. Dans l'immédiat voici ce qu'il faut savoir pour éditer un fichier avec Emacs :

```
$ emacs bonjour.txt
```

tapez alors tranquillement votre texte, puis pour le sauvegarder, appuyez sur les touches [Ctrl] [x] [c] (c'est-à-dire [x] puis [c] tout en maintenant [Ctrl] enfoncée). Pour charger un fichier existant, utilisez la commande [Ctrl] [x] [s] Les menus `Files/Save Buffer` et `Files/Open File...` vous permettront d'effectuer également ces opérations.

5.2 Faire des scripts en shell

Cette section a pour but d'exposer les rudiments de la programmation en shell. Le shell utilisé est bash. C'est le shell développé par le projet GNU. Il incorpore les fonctionnalités du shell sh ainsi que certaines tirées des shells csh et ksh. La modeste expérience de l'auteur a montré qu'il est nécessaire d'expérimenter pour parvenir à ses fins avec le shell ; il permet cependant, une fois la phase d'apprentissage passée [2], d'automatiser certaines des tâches quotidiennes d'un utilisateur de système UNIX et pour cette raison il est très utile d'en connaître les principes de base. La lecture des chapitres 2 et 3 est un pré-requis indispensable à la lecture de cette section.

Si vous utilisez un shell de connexion de type `tcsh` ou `csh`, rien ne vous empêche de programmer avec `sh` ou `bash`. C'est pourquoi nous focaliserons ici notre attention sur le shell `bash` de la famille `sh` qui, encore une fois, ne vous engage pas à changer vos habitudes « interactives » si vous utilisez un autre shell de connexion.

Enfin, pour vous donner une idée de la complexité du programme bash, la commande :

```
$ PAGER=cat man bash | wc -lw
Reformatting bash(1), please wait...
 4522   33934
$
```

que vous êtes pratiquement capable de comprendre si vous avez lu jusqu'ici, nous indique que la page de manuel de bash sur le système de votre serviteur, contient 33934 mots soit environ 4500 lignes sur un terminal d'environ 90 caractères de large. Pour info :

```
$ pdftotext guide-unix.pdf - | wc -w
68837
$
```

`guide-unix.pdf` est le document que vous avez sous les yeux...

5.2.1 Commentaires

Tout langage de programmation dispose de symboles particuliers pour insérer des commentaires dans le code ; c'est-à-dire des portions de texte qui seront ignorées par l'interpréteur ou le compilateur le cas échéant. Ce caractère en langage de commande d'UNIX est le caractère `#`. Tout le texte suivant ce caractère jusqu'à la fin de la ligne sera ignoré. Notons toutefois que s'il est suivi du caractères `!`, ce qui suit est interprété d'une manière particulière par le système comme expliqué au paragraphe suivant.

5.2.2 Choisir l'interpréteur

Un script en shell n'est qu'un fichier texte contenant une liste de commandes. Par exemple, un fichier `bidule` contenant la commande :

bidule

```
echo bonjour
```

2. Passe-t-elle un jour ?

est un script shell ! Pour exécuter ce script on lance l'interpréteur de commande avec le fichier comme argument :

```
$ bash bidule
bonjour
$
```

De manière à simplifier l'utilisation du script `bidule` on peut préciser quel sera l'interpréteur du programme contenu dans le fichier. Pour cela on place en tête du fichier les deux caractères `#!` suivis de la référence absolue de l'interpréteur utilisé :

bidule

```
#!/bin/bash
echo bonjour
```

il faudra ensuite rendre exécutable le script :

```
$ chmod +x bidule
$
```

Ces deux opérations permettent d'utiliser le script en l'appelant directement comme une commande :

```
$ ./bidule
bonjour
$
```

Notons, qu'on peut généraliser cette idée à n'importe quel interpréteur, par exemple, le fichier `test.awk` :

test.awk

```
#!/usr/bin/awk -f
/^Jim/{print $1,$3}
```

Une fois rendu exécutable, il donne bien ce qu'on attend du premier exemple de la section 3.6 page 79 :

```
$ ./test.awk fichier.dat
Jimi 1970
Jim 1971
$
```

ici c'est donc la commande `awk -f` qui est chargée d'interpréter les commandes contenues dans `test.awk`.

5.2.3 Variables

Les variables du shell sont des symboles auxquels on affecte des valeurs. Ces variables ne sont pas ou très faiblement typées comme nous le verrons un peu plus bas. L'affectation et la lecture des variables se fait grâce à la syntaxe suivante (voir aussi 2.1.3 page 26) :

testvar.sh

```
#!/bin/bash
N=4
NOM=trucmuche
echo "le nom est $NOM et la variable N vaut $N"
```

qui donne à l'exécution :

```
$ ./testvar.sh
le nom est trucmuche et la variable N vaut 4
$
```

Dans certaines situations il est nécessaire d'utiliser les accolades en plus du dollar pour lire le contenu de la variable. La syntaxe est alors `${N}`.

```
#!/bin/bash
BOF=loz
echo mon nom est ${BOF}ano
```

Sans les accolades, le shell aurait cherché la valeur de la variable `$BOFano`, variable qui n'existe pas. En shell, les variables ne sont pas *déclarées* au sens d'un langage de programmation compilé. En fait toutes les variables existent potentiellement. Par conséquent, le shell n'émet pas de message d'erreur lorsqu'on tente d'accéder à une variable à laquelle on n'a jamais affecté de valeur. Cette variable est considérée comme étant « vide » à l'expansion :

varvide.sh

```
#!/bin/bash
BOF=loz
echo mon nom est $BOFano
```


donne à l'exécution :

```
$ ./varvide.sh
mon nom est
$
```

ceci rend le débuggage de scripts parfois délicat. Pour pallier ce problème on peut utiliser la commande :

```
$ set -u
$
```

pour demander au shell de générer une erreur lorsqu'on veut faire référence à une variable non intialisée :

```
$ set -u
$ echo $variablenonintialisee
-bash: variablenonintialisee: unbound variable
$
```

Arguments de la ligne de commande

Les arguments passés en ligne de commandes — c'est-à-dire lors de l'appel de la commande que constitue le script — sont stockés dans les variables `$0`, `$1`, `$2`, etc. Ainsi :

testarg.sh

```
#!/bin/sh
echo Arguments "[$0] [$1] [$2] [$3] [$4]"
```

donne :

```
$ ./testarg.sh -t bonjour "les amis" 34
Arguments [./testarg.sh] [-t] [bonjour] [les amis] [34]
$
```

On peut noter qu'à l'instar du langage C, l'argument numéro 0 est le programme à exécuter, ici `./testarg.sh`.

Notez en outre l'utilisation des guillemets en ligne de commande. Lors de l'exécution de `testarg.sh` on utilise ces guillemets pour regrouper les mots `les` et `amis` en une seule chaîne de caractères, on peut alors les considérer comme un seul argument.

Pour le traitement des paramètres de la ligne de commande, on dispose également de quelques variables prédéfinies :

- la variable `$#` contient le nombre d'arguments de la ligne de commande sans compter la commande elle même — les arguments sont donc comptés à partir de `$1`;
- les variables `$*` et `$@` contiennent toutes les deux l'ensemble des arguments à partir de `$1` mais ont une signification différente lorsqu'elles sont utilisées entre guillemets :

  testarg2.sh
  ```
  #!/bin/sh
  for ARG in "$*" ; do echo $ARG ; done
  ```

  donne :

  ```
  $ ./testarg2.sh a b c
  a b c
  $
  ```

  dans ce cas l'ensemble des arguments est considéré comme une seule chaîne de caractères et :

  testarg3.sh
  ```
  #!/bin/sh
  for ARG in "$@" ; do echo $ARG ; done
  ```

  donne :

  ```
  $ ./testarg3.sh a b c
  a
  b
  c
  $
  ```

  ici chaque argument est considéré comme une chaîne de caractères à part entière.

Modifications

Il existe plusieurs mécanismes permettant d'agir sur les variables :

1. l'aide à l'instanciation de variables ;
2. le traitement du contenu.

Chacun de ces mécanismes suit une syntaxe particulière (la plupart du temps assez difficilement mémorisable !). Voici donc à titre d'exemple quelques-uns des outils [3]

3. Se reporter au `man` de `bash` pour des informations précises.

correspondant :

- `${N:-4}` renvoie la valeur de `N` si on lui en a affecté une, 4 sinon ; ce qui permet d'utiliser une valeur par défaut ;
- `${N:?`⟨*msg*⟩`}` renvoie le message d'erreur ⟨*msg*⟩ si `N` n'a pas été instancié et quitte le script ; ceci peut être utile pour tester les arguments de la ligne de commande.

Voici un exemple d'utilisation de ces outils :

varmod.sh

```
#!/bin/sh
NOM=${1:?"vous devez fournir un nom"}
PRENOM=${2:-"djobi"}
echo Qui : $PRENOM $NOM
```

ce qui peut donner à l'exécution :

```
$ ./varmod.sh
./varmod.sh: 1: vous devez fournir un nom
$
```

le texte `./varmod.sh: 1:` doit se comprendre comme : « l'argument n° 1 du script `varmod.sh` est manquant »

```
$ ./varmod.sh djoba
Qui : djobi djoba
$ ./varmod.sh goulbi goulba
Qui : goulba goulbi
$
```

Pour ce qui est des outils de traitement de contenu la syntaxe est :

- `${N%`⟨*motif*⟩`}` et `${N%%`⟨*motif*⟩`}` suppriment respectivement la plus petite et la plus longue chaîne répondant à l'expression régulière ⟨*motif*⟩, à la *fin* du contenu de la variable `N` ;
- `${N#`⟨*motif*⟩`}` et `${N##`⟨*motif*⟩`}` suppriment respectivement la plus petite et la plus longue chaîne répondant à l'expression régulière ⟨*motif*⟩, au *début* du contenu de la variable `N`.

et voici un exemple très instructif inspiré par Newham et Rosenblatt (1998) :

```
$ P=/home/local/etc/crashrc.conf.old
$ echo ${P%%.*}    ←——— supprime la plus grande chaîne commençant par '.'
/home/local/etc/crashrc
$ echo ${P%.*}    ←——— supprime la plus petite chaîne commençant par '.'
/home/local/etc/crashrc.conf
$ echo ${P##/*/}    ←——— supprime la plus grande chaîne entourée de '/'
crashrc.conf.old
$ echo ${P#/*/}    ←——— supprime la plus petite chaîne entourée de '/'
local/etc/crashrc.conf.old
$
```

Une application de ce type d'outils serait, par exemple, la conversion en salve, de plusieurs fichiers JPEG en TIFF. En supposant que :

- les fichiers portent l'extension `.jpg` ;
- sont dans le répertoire courant ;

– on dispose d'un utilitaire que l'on nommera `convert` qui est capable d'effectuer cette conversion ;

on peut alors écrire le script (voir § 3.4.3 page 76 pour l'utilisation de la boucle `for`) :

```
#!/bin/sh
for F in *.jpg ; do
  convert "$F" "${F%.jpg}.tif"
done
```

Dans ce script, si la variable `F` vaut à une itération `bidule.1.jpg` alors :

– `${F%.jpg}` vaut `bidule.1`, c'est-à-dire le nom du fichier auquel on a supprimé la chaîne `.jpg` à la fin ;
– `{F%.jpg}.tif` vaut `bidule.1.tif`, c'est-à-dire, `bidule.1` concaténé avec la chaîne `.tif`.

Finalement, à cette itération on exécute la commande :

```
convert bidule.1.jpg bidule.1.tif
```

Ce principe est très utile dans beaucoup de situations où l'on veut traiter plusieurs fichiers. Par exemple le très classique cas où l'on veut renommer un ensemble de fichiers :

```
$ ls *.wav
audio1.wav audio2.wav audio3.wav
audio4.wav audio5.wav audio6.wav
audio7.wav audio8.wav audio9.wav
$
```

Imaginons que l'on veuille renommer ces fichiers sous la forme suivante :

`crescent_`x`.wav`

On peut alors écrire :

```
#!/bin/sh
for F in audio*.wav ; do
  mv $F crescent_${F#audio}
done
```

Un moyen mnémotechnique de se souvenir de la syntaxe de ces opérateurs est la suivante : # désigne en anglais le numéro (*number*) et on dit généralement « numéro 5 » ; # supprime donc en *début* de chaîne ; inversement on dit généralement « 5 % » , % supprime donc en *fin* de chaîne. Le nombre de caractères # ou % rappelle si on supprime la plus *petite* (1 caractère) ou la plus *longue* (2 caractères) chaîne correspondant au motif. 'Tain c'est pô convivial ce bazar !

Arithmétique

Bien que les variables du shell ne soient pas typées, on peut cependant évaluer des expressions arithmétiques ; ces expressions n'auront un sens que si les contenus des variables sont des valeurs numériques, bien entendu. La syntaxe pour évaluer une expression arithmétique est :

`$[`⟨*expression_arithmétique*⟩`]`
ou
`$((`⟨*expression_arithmétique*⟩`))` (à préférer)

Par exemple :

```
$ N=3
$ echo $((N+1))
4
$ N=$[N+4]
$ echo $N
7
$
```

Pour illustrer l'utilisation de l'arithmétique sur les variables, considérons par exemple que l'on dispose d'un ensemble de fichiers dont les noms ont la forme suivante :

```
$ ls *.dat
2002-qsdfff.dat 2002-sdhjlk.dat  2002-yuiqso.dat
2003-azerzz.dat 2003-sddfg.dat   2003-qsdfrtuy.dat
2003-ertyf.dat  2004-ersssty.dat 2004-sdf.dat
...
$
```


c'est-à-dire :

⟨*année sur 4 chiffres*⟩`-`⟨*n caractères*⟩`.dat`

suite à une erreur de manipulation, on désire remplacer toutes les années a par $a-1$ (donc 2002 par 2001, 2003 par 2002, etc.). Si l'on suppose qu'une variable `F` prenne comme valeur le nom d'un de ces fichiers :

```
2002-ertyf.dat
```

alors on peut extraire l'année (`2002`) comme suit :

```
annee=${F%-*}
```

maintenant la variable `annee` contient le nom du fichier sans ce qui se trouve après le caractère `-`. On peut alors décrémenter cette variable :

```
annee=$((annee-1))
```

le nouveau nom du fichier sera composé de cette variable `annee` concaténée avec ce qui se trouve après le caractère `-`, soit :

```
$annee-${F#*-}
```

D'où le script de conversion utilisant une boucle ▶`for`◀ § 3.4.3 p. 76 ◀

arithmetique.sh

```
#!/bin/sh
for F in ????-*.dat ; do
  annee=${F%-*}
  annee=$((annee-1))
  mv "$F" $annee-${F#*-}
done
```

On aurait également pu extraire les deux parties du nom de fichier (année et le reste) à l'aide de ▲sed◀ comme le montre le programme suivant : ► 3.7 p. 81

arithmetique-2.sh

```
#!/bin/sh
for F in ????-*.dat ; do
  annee=$(echo "$F" | sed -r 's/([0-9]{4})-.*/\1/')
  reste=$(echo "$F" | sed -r 's/[0-9]{4}-(.*)/\1/')
  annee=$((annee-1))
  echo mv "$F" $annee-$reste
done
```

Pour information l'expression régulière :

```
[0-9]{4}-.*
```

reconnaîtra une chaîne de caractères commençant par 4 chiffres, suivis d'un tiret, suivi de ce qui vous passe par la tête. On notera également l'utilisation de la ▲substitution de commande◀ et d'un tube pour stocker dans une variable l'application de la commande `sed` à la valeur `$F`. ► § 3.4.2 p. 75

5.2.4 Structure de contrôle et tests

Les scripts en shell prennent leur intérêt lorsque l'on peut y insérer des boucles et autres structures conditionnelles et itératives. Chacune de ces structures de contrôle inclut généralement l'évaluation d'une expression booléenne.

On verra un peu plus loin qu'en réalité les valeurs des expressions booléennes correspondent toujours à un ▲code de retour◀ d'une commande. Et que la valeur de ce code est interprétée comme « vrai » ou « faux ». ► § 5.2.4 p. 120

Tests

La syntaxe pour effectuer un test — que nous reprendrons plus loin avec les structures de contrôle — est la suivante :

`test` ⟨*expression_booléenne*⟩
ou
`[` ⟨*expression_booléenne*⟩ `]`

Plusieurs tests sont disponibles, ayant trait au contrôle de caractéristiques de fichiers (existence, exécutabilité, etc.) et à la comparaison de chaîne de caractères, ou d'expressions arithmétiques. Parmi celles-ci :

test	renvoie « vrai » si...
`-f` ⟨*fichier*⟩	⟨*fichier*⟩ existe [4]
`-x` ⟨*fichier*⟩	⟨*fichier*⟩ est exécutable [5]
`-d` ⟨*fichier*⟩	⟨*fichier*⟩ est un répertoire [6]
⟨*chaîne*$_1$⟩`=`⟨*chaîne*$_2$⟩	⟨*chaîne*$_1$⟩ et ⟨*chaîne*$_2$⟩ sont identiques
⟨*chaîne*$_1$⟩`!=`⟨*chaîne*$_2$⟩	⟨*chaîne*$_1$⟩ et ⟨*chaîne*$_2$⟩ sont différentes
`-z` ⟨*chaîne*⟩	⟨*chaîne*⟩ est vide

Les expressions booléennes effectuant des tests arithmétiques peuvent quant à elles, être réalisées sous la forme :

⟨*expr*$_1$⟩ ⟨*opérateur*⟩ ⟨*expr*$_2$⟩

où ⟨*opérateur*⟩ peut prendre les valeurs du tableau ci-dessous :

`-eq`	égal (*equal*)
`-ne`	différent (*not equal*)
`-lt`	inférieur (*less than*)
`-gt`	supérieur (*greater than*)
`-le`	inférieur ou égal (*less than or equal*)
`-ge`	supérieur ou égal (*greater than or equal*)

On peut également combiner les expressions booléennes :

test	renvoie « vrai » si...
`!` ⟨*expr*⟩	⟨*expr*⟩ renvoie « faux »
⟨*expr*$_1$⟩ `-a` ⟨*expr*$_2$⟩	⟨*expr*$_1$⟩ et (*and*) ⟨*expr*$_2$⟩ renvoient « vrai »
⟨*expr*$_1$⟩ `-o` ⟨*expr*$_2$⟩	⟨*expr*$_1$⟩ ou (*or*) ⟨*expr*$_2$⟩ renvoient « vrai »

Le shell — notamment `bash` — dispose de bien d'autres outils de tests et nous vous invitons vivement à compulser la page de manuel ou vous procurer l'ouvrage de Newham et Rosenblatt (1998) pour de plus amples informations.

Structures de contrôle

Voici trois structures de contrôle disponibles dans `bash`. Ce sont les trois structures classiques d'un langage de programmation :

- le « if then else » pour faire un choix à partir d'une expression booléenne ;
- le « case » permettant de faire un choix multiple en fonction d'une expression ;
- le « tant que » dont le but est de réitérer des commandes tant qu'une expression est vraie.

Le shell dispose de quelques autres structures que nous passerons sous silence. On peut également se référer au paragraphe 3.4.3 page 76 pour la structure de contrôle de type « for ». Toujours dans un souci de pragmatisme nous illustrons ici l'usage de ces trois structures de contrôle grâce à des exemples. Examinons tout d'abord le `if` :

testif.sh

```
#!/bin/sh
A=$1
B=$2
if [ $A -lt $B ]; then
   echo "l'argument 1 est plus petit que l'argument 2"
else
   echo "l'argument 1 est plus grand que l'argument 2"
fi
```

4. et est un fichier « régulier » (*regular file*) un fichier normal, quoi, pas un lien symbolique, ni un répertoire, ni ...

5. ou compulsable pour un répertoire.

6. ou un lien symbolique vers un répertoire.

On peut noter l'utilisation du test `[...]` et du mot-clef `fi` qui termine la clause `if then else`. Il ne faut pas omettre le point-virgule qui clôt l'expression test. À l'exécution :

```
$ ./testif.sh 4 6
l'argument 1 est plus petit que l'argument 2
$ ./testif.sh 45 6
l'argument 1 est plus grand que l'argument 2
$
```

Pour illustrer l'utilisation du `while` écrivons une petite boucle qui compte de 1 à n (un nombre donné par l'utilisateur) :

testwhile.sh

```
#!/bin/sh
LAST=$1
n=1
while [ "$n" -le "$LAST" ]; do
  echo -n "$n"
  n=$((n+1))
done
echo et voilou
```

qui donne par exemple :

```
$ ./testwhile.sh 10
1 2 3 4 5 6 7 8 9 10 et voilou
$
```

Notez le mot-clef `done` qui clôt la clause `while`. Ici l'option `-n` de la commande `echo` ne crée pas de saut de ligne à la fin de l'affichage. Histoire de confirmer l'assertion de Larry Wall[7], on peut utiliser une boucle « pour » pour générer et afficher une séquence. On fera appel à la commande `seq` dont voici un exemple d'utilisation :

```
$ seq 1 5
1
2
3
4
5
$
```

ce petit utilitaire a donc pour tâche d'afficher les entiers de 1 à 5. Si vous êtes curieux, la page de manuel vous indiquera que l'on peut également changer l'incrément et deux ou trois autres choses. On peut donc écrire le script suivant :

compter.sh

```
#!/bin/sh
for I in $(seq 1 10); do
   printf "%d " "$I"
done
echo et voilou.
```

qui donne le même résultat que le script précédent.

7. There's more than one way to do it.

Les puristes auront noté que la commande seq n'est disponible que pour les utilisateurs travaillant sur des systèmes où le paquet coreutils de chez M. GNU est installé.

Pour ce qui est du `case`, on peut imaginer que l'on veuille écrire un script permettant d'identifier l'origine des fichiers en fonction de leur extension[8].

testcase-I.sh

```
#!/bin/sh

NOM=${1:?"Donnez un nom de fichier svp"}
EXT=${NOM##*.}

case $EXT in
  tex) echo on dirait un fichier TeX ;;
  dvi) echo on dirait un fichier device independant ;;
  ps)  echo on dirait un fichier Postscript ;;
  *)   echo chai pô ;;
esac
```

On utilise ici le modificateur de variable `##` pour supprimer la plus grande chaîne finissant par le caractère « . » et ainsi obtenir l'extension. Le mot-clef `esac` (case à l'envers) clôt la clause `case`. Le choix par défaut est indiqué par « * » (n'importe quelle chaîne de caractères). À chaque choix correspond une série de commandes à exécuter (ici une seule) ; série qui doit être terminée par deux points-virgules « ;; ». On teste :

```
$ ./testcase-I.sh ~/LaTeX/guide/guide-unix.tex
on dirait un fichier TeX
$ ./testcase-I.sh /etc/lpd.conf
chai pô
$
```

La chaîne de caractères précédant la parenthèse dans chaque clause de case peut être un ▶joker◀ (*wildcard*), comme le montre la dernière clause qui utilise le caractère * pour indiquer « toute (autre) chaîne de caractère ».

§ 2.1.6 p. 28 ◀

Une limitation du script précédent est que les fichiers :

```
tex
.tex
```

seront pris pour un fichier TEX. Ceci est dû au fait que si :

```
$ NOM=tex
$
```

alors l'expression `${NOM##*.}` renvoie `tex` car aucune substitution n'a pu être réalisée. On peut imaginer la solution suivante qui utilise les caractères génériques du shell :

8. Notez que l'utilitaire `file` fait ce travail beaucoup mieux...

testcase-II.sh

```
#!/bin/bash
NOM=${1:?"Donnez un nom de fichier svp"}
case $NOM in
  *[^/].tex) echo on dirait un fichier TeX ;;
  *[^/].dvi) echo on dirait un fichier device independant ;;
  *[^/].ps)  echo on dirait un fichier Postscript ;;
  *)   echo chai pô ;;
esac
```

Dans ce script, l'expression `*[^/].tex` veut dire toute chaîne de caractères composée de :

- `*` : toute chaîne même vide ;
- `[^/]` : un caractère différent de `/` ;
- `.tex`

Les commandes renvoient une valeur !

Les expressions booléennes utilisées par exemple dans la structure `if` sont en réalité la valeur de retour d'une commande. En effet toute commande UNIX renvoie une valeur en fonction du succès de son exécution. Il existe en outre une variable particulière contenant le code retour de la dernière commande exécutée : la variable `$?`. Voici un exemple avec la commande `id` qui donne des informations sur un utilisateur particulier :

```
$ id lozano
uid=1135(lozano) gid=200(users) groups=200(users)
$ echo $?
0   ←———————————————————————————————— tout s'est bien passé.
$ id djobi
id: djobi: No such user
$ id $?
1   ←———————————————————————————————— Il y a eu un problème.
$
```

On peut alors utiliser cette valeur directement dans une structure de type `if` comme suit :

testtest.sh

```
#!/bin/bash
if id $1 2> /dev/null 1>&2; then
  echo c\'est un utilisateur
else
  echo ce n\'est pas un utilisateur
fi
```

Il est important de noter que dans la construction :

```
if ⟨commande test⟩ ; then
   ⟨commandes⟩
fi
```

⟨*commandes*⟩ seront exécutées si et seulement si ⟨*commande test*⟩ renvoie la valeur 0 (indépendamment de ce qu'elle peut afficher). Dans notre exemple on a donc :

```
$ ./testtest.sh lozano
c'est un utilisateur
$ ./testtest.sh djobi
ce n'est pas un utilisateur
$
```

On peut noter l'utilisation de ►redirections◄ `2>` pour rediriger le flux d'erreur (2) et `1>&2` pour rediriger le flux standard (1) sur le flux d'erreur (2). § 3.2 p. 62 ◄

Un script renvoie la valeur retournée par la dernière commande exécutée. On peut cependant forcer le renvoi d'une valeur particulière en utilisant la commande `exit` suivie d'une valeur.

Pour finir ce paragraphe et boucler la boucle des « expressions booléennes », supposons qu'il existe dans le répertoire courant un fichier `guide-unix.tex` et tapons :

```
$ [ -f guide-unix.tex ]
$ echo $?
0   ←———————————————————————————————— le fichier existe
$
```

Puis :

```
$  [ -f guide-unix.texitoi ]
$ echo $?
1   ←———————————————————————————————— le fichier n'existe de pas
$
```

Où l'on apprend donc que `[` est une commande qui, comme beaucoup de ses copines, renvoie 0 si tout est ok, et 1 sinon...

5.2.5 Fonctions

Une des caractéristiques des shells de la famille `sh` est la possibilité de créer des fonctions. Voici un exemple succinct qui illustre les syntaxes de déclaration et d'appel, ainsi que la visibilité des variables :

testfunc.sh

```
#!/bin/sh

N=4
function mafunc()
{
  I=7
  echo $1 $2 $N
}
mafunc a b # appel à la fonction avec 2 arguments
echo et : $I
```

qui donne à l'exécution :

```
$ ./testfunc.sh
```

```
a b 4
et : 7
$
```

on notera donc qu'à l'intérieur du corps de la fonction, les arguments de celle-ci sont notés `$1`, `$2`, etc. En outre :

- les variables instanciées dans la fonction sont visibles en dehors (c'est le cas de `$I` dans notre exemple) ;
- les variables instanciées avant la définition de la fonction sont visibles dans celle-ci (c'est le cas de `$N` dans notre exemple).

Enfin, à l'instar d'un script, on peut faire renvoyer une valeur à une fonction avec la commande `return` suivie d'une valeur. Voici un exemple d'utilisation de fonction : on reprend le script `arithmetique.sh` du paragraphe 5.2.3 page 114 et on utilise une fonction pour faire le changement de nom :

arithmetique-3.sh

```
#!/bin/sh

function changernom()
{
  annee=${1%-*}
  annee=$[annee-1]
  mv "$1" "$annee-${1#*-}"
}
for F in ????-????.dat ; do
  changernom $F
done
```

5.2.6 Input field separator

Un jour ou l'autre le développeur de script shell est confronté au problème suivant. Avec notre fichier :

```
Jimi Hendrix 1970
Jim Morrison 1971
Janis Joplin 1969
```

on écrit naïvement le script ayant pour but d'effectuer un traitement par ligne des fichiers passés en paramètre :

naif.sh

```
#!/bin/sh
for ligne in $(cat "$@") ; do
  echo "Une ligne : $ligne"
done
```

On aura une certaine déception à l'exécution :

```
$ ./naif.sh fichier.dat
Une ligne : Jimi
Une ligne : Hendrix
```

```
Une ligne : 1970
Une ligne : Jim
Une ligne : Morrison
Une ligne : 1971
Une ligne : Janis
Une ligne : Joplin
Une ligne : 1969
$
```

Ceci vient du fait que les éléments constituant la liste $\langle \ell \rangle$ de :

`for ligne in` $\langle \ell \rangle$ `; do ...`

sont délimités par les caractères contenus dans la variable `IFS`. Cette variable contient par défaut :

- le caractère espace,
- la tabulation,
- le saut de ligne.

Or nous souhaiterions ici clairement que les éléments de la liste soient délimités par le saut de ligne. Il suffit pour cela de modifier le script :

naif.sh

```
#!/bin/sh
# c'est le saut de ligne qui fait office de séparateur pour
# chaque élément d'une liste
IFS="
"
for ligne in $(cat "$@") ; do
  echo "Une ligne : $ligne"
done
```

qui donnera ce qu'on attend, c'est-à-dire :

```
$ ./naif.sh fichier.dat
Une ligne : Jimi Hendrix 1970
Une ligne : Jim Morrison 1971
Une ligne : Janis Joplin 1969
$
```

5.2.7 Fiabilité des scripts

Pour rendre fiables les scripts que vous écrirez, il faudra qu'ils puissent traiter correctement les fichiers aux noms « bizarroïdes ». Le premier cas est le cas des noms de fichiers contenant des espaces. Supposons que dans un répertoire :

```
$ ls -1
bidule.dat
le machin.dat
truc.dat
$
```

et que l'on écrive le script suivant [9] :

9. Qui n'a comme seul intérêt l'exemple, puisque `rm *.dat` fera aussi bien l'affaire.

```
for f in *.dat ; do
  echo "je vais effacer $f..."
  rm $f
done
```

Celui-ci affichera, au moment de traiter le fichier `le machin.dat` :

```
Je vais effacer le machin.dat
rm: cannot lstat ‘le’: No such file or directory
rm: cannot lstat ‘machin.dat’: No such file or directory
```

C'est-à-dire que la commande `rm` tente d'effacer deux fichiers :
- le fichier `le`
- et le fichier `machin.dat`

Pour faire comprendre à la commande `rm` que le nom du fichier est composé des deux mots, on doit écrire :

```
for f in *.dat ; do
  echo "je vais effacer $f..."
  rm "$f"   <-------------------------------- on quote la variable
done
```

Autre situation « tordue » : lorsque le nom d'un fichier commence par le caractère tiret (-) par exemple `-biduletruc.txt`. Dans ce cas :

```
$ ls -biduletruc.txt
ls: invalid option -- e
Try ‘ls --help’ for more information.
$ rm -biduletruc.txt
rm: invalid option -- b
Try ‘rm --help’ for more information.
$
```

ici `e` est la première option connue à la commande `ls`. Pour `rm` c'est l'option `b`. La solution [10] pour contourner ce problème est d'insérer dans la ligne de commande le couple `--` pour indiquer que ce qui suit ne contiendra plus d'option. Ainsi :

```
$ ls -- -biduletruc.txt
-biduletruc.txt
$
```

On pourra faire de même pour la commande `rm`.

5.3 Makefile

L'utilitaire `make` est un outil destiné, au départ, à gérer la compilation d'un projet incluant plusieurs fichiers. Grâce à un fichier — le `makefile` — on précise quels sont les fichiers à recompiler lorsqu'on apporte une modification à un ou plusieurs fichiers composant le projet. Dans la pratique, `make` peut être utilisé dans le cadre

10. Une autre solution consiste à préfixer le nom du fichier par `./`

d'un projet dont la gestion ne nécessite pas nécessairement de compilation ; en fait tout projet dans lequel on a besoin à un moment donné de *construire* (ou *faire*) un fichier à partir d'un ou plusieurs autres, peut faire appel à `make`. On distinguera donc deux concepts :

1. l'utilitaire `make` lui-même qui est un programme, dont les versions varient selon le système ; nous utiliserons ici le `make` de GNU ;
2. le fichier `makefile`, qui peut porter n'importe quel nom, et qui contient un ensemble de règles pour définir :
   (a) à partir de quels fichiers on construit le projet ;
   (b) quels sont les fichiers à « refaire » en fonction des modifications apportées au projet.

   Ce fichier est interprété par l'utilitaire `make`.

5.3.1 Principes de base

Nous exposons ici les concepts de base de l'utilisation de `make` en nous basant sur un exemple simple : supposons que nous utilisions le logiciel `xfig`[11] pour créer des dessins et que nous ayons besoin de les insérer dans un document sous la forme d'un fichier au format PostScript encapsulé[12]. Une fois le fichier `test.fig` sauvé au format de `xfig`, on désire automatiser l'exportation en PostScript, avec la commande[13] :

```
$ fig2dev -L eps test.fig > test.eps
$
```

Notion de cible

Comme nous le notions un peu plus haut, `make` permet de *fabriquer* des fichiers. Le fichier que l'on veut construire s'appelle en jargon `make`, la *cible*. Supposons que le fichier à exporter s'appelle `test.fig`, le fichier à fabriquer se nommera `test.eps`. On crée alors un fichier que l'on nomme `Makefile` contenant :

```
test.eps :
⟼ fig2dev -L eps test.fig > test.eps
```

le caractère ⟼ indique l'insertion d'une *tabulation*. Sa présence est *indispensable* pour que l'utilitaire `make` analyse les règles du fichier `makefile`.

Par défaut, `make` cherche à exécuter la première cible qu'il rencontre. Par convention, on définit une cible `all` pour « tout faire » :

11. Qui est à la fois laid et pratique, et dispose de fonctionnalités étonnantes comme son fameux « open compound ».
12. Le PostScript encapsulé diffère du Postscript dans la mesure où il est destiné à être intégré dans un autre document. Le fichier contient d'ailleurs à cette fin une « boîte englobante » (*bounding box* permettant d'obtenir son encombrement.
13. L'exportation peut certes être réalisée interactivement avec la souris, depuis le logiciel `xfig`, mais tel n'est pas notre propos.

```
all : test.eps

test.eps :
⟼ fig2dev -L eps test.fig > test.eps
```

Pour construire le fichier `test.eps` on peut alors au choix lancer :

```
$ make
fig2dev -L eps  test.fig > test.eps
$
```

ou :

```
$ make test.eps
fig2dev -L eps  test.fig > test.eps
$
```

On peut d'ores et déjà remarquer que la commande à exécuter est préalablement affichée à l'écran ; ceci peut être désactivé (voir 5.3.5 page 129).

Notion de dépendance

À chaque modification du fichier `test.fig`, il est nécessaire d'exporter à nouveau en `eps`. C'est typiquement le genre de tâche que l'on peut confier à `make`. On insère donc dans le fichier `Makefile` une *dépendance* qui spécifie que la construction du fichier `test.eps` dépend de la date du fichier `test.fig` :

```
test.eps : test.fig
⟼ fig2dev -L eps test.fig > test.eps
```

La signification de cette dépendance est la suivante :

1. si le fichier `test.fig` est plus récent que le fichier `test.eps` la prochaine invocation de `make` construira à nouveau `test.eps` ;
2. sinon une invocation de `make` donnera :

   ```
   $ make
   make: 'test.eps' is up to date.
   $
   ```

qui est tout simplement la traduction de « est à jour ».

L'utilitaire `make` gère les dépendances en utilisant les *dates* associées aux fichiers ; il est donc impératif que l'horloge du système fonctionne correctement et en particulier, dans le cas d'un système en réseau, que l'heure du système stockant les fichiers soit synchrone avec l'heure du système exécutant l'utilitaire lui-même.

Notion de règle

L'ensemble formé par une cible, une ou des dépendances, et les commandes s'appelle une *règle*. La syntaxe générale est la suivante :

⟨*cible*⟩ : ⟨*dépendances*⟩
⟼⟨*commande*$_1$⟩
⟼⟨*commande*$_2$⟩
⟼...
⟼⟨*commande*$_n$⟩

Nous verrons par la suite que l'on peut définir des règles relativement subtiles permettant d'automatiser la gestion de fichiers de projets complexes.

5.3.2 Variables

Dans un fichier `Makefile` on peut utiliser des variables, par exemple :

```
FICHIER = test
FIGOPTS=-Leps
all : $(FICHIER).eps
$(FICHIER).eps : $(FICHIER).fig
⟼ fig2dev $(FIGOPTS) $(FICHIER).fig > $(FICHIER).eps
```

La syntaxe s'apparente donc à celle du shell :
- affectation avec l'opérateur `=` ;
- accès avec l'opérateur `$` et les parenthèses.

5.3.3 Règles implicites et variables prédéfinies

La règle que l'on a construite jusqu'ici est adaptée au fichier `test.fig` et à lui seul. Un moyen de généraliser cette règle à d'autres fichiers est d'utiliser ce qu'on nomme une *règle implicite*, permettant de définir de manière générale, comment construire un fichier `.eps` à partir d'un fichier `.fig`. Un fichier `Makefile` répondant à ce problème peut être le suivant :

```
1 EPSFILE = test.eps
2 all : $(EPSFILE)
3 %.eps : %.fig
4 ⟼ fig2dev -L eps $< > $@
```

On peut expliquer ce `Makefile` quelque peu cryptique de la manière suivante :
- la ligne 3 contient :
  - la cible `%.eps` : c.-à-d. n'importe quel fichier dont l'extension est `eps` ;
  - la dépendance `%.fig` : un fichier dont l'extension est `.fig` ;
- la ligne 4 contient la commande proprement dite qui utilise des variables prédéfinies du GNU `make` :
  - la variable `$<` pour le premier fichier de la liste des dépendances (`test.fig` dans notre exemple);
  - la variable `$@` qui désigne le fichier cible (`test.eps`).

Le mécanisme de création du fichier `test.eps` est donc le suivant :

1. on doit faire `test.eps` (cible `all`) ;
2. pour ce faire, on dispose d'une règle permettant de faire un `.eps` à partir d'un `.fig` ;

3. il existe un fichier `test.fig` dans le répertoire ;
4. on peut donc faire `test.eps`.

Il existe bien entendu d'autres variables automatiques, parmi celles-ci : `$^` qui contient toutes les dépendances de la cible séparées par des espaces. Voir le manuel info de GNU `make` pour plus de précisions.

5.3.4 Outils sur les variables

On peut également, grâce à certaines des fonctionnalités du GNU `make`, automatiser l'instanciation des variables. Dans l'exemple qui nous concerne, on peut dans un premier temps mettre dans une variable `FIGFILE` tous les fichiers du répertoire de travail portant l'extension `.fig` :

```
FIGFILE=$(wildcard *.fig)
```

le mot-clef `wildcard` que l'on peut traduire par *joker* ou caractère générique permet donc ici d'arriver à nos fins. Une deuxième étape consiste à créer la variable `EPSFILE` à partir de `FIGFILE` en remplaçant dans le contenu de cette dernière, la chaîne `.fig` par la chaîne `.eps` :

```
EPSFILE=$(patsubst %.fig, %eps,$(FIGFILE))
```

Le mot-clef `patsubst` signifie ici *pattern substitution*. Notre `Makefile` permettant de gérer l'exportation de nos dessins est maintenant capable de générer les `.eps` à partir de tous les fichiers `.fig` du répertoire courant. On peut également y rajouter une cible de « nettoyage » permettant d'effacer les fichiers `.eps` lorsqu'ils ne sont plus nécessaires :

```
FIGFILE=$(wildcard *.fig)
EPSFILE=$(patsubst %.fig, %eps,$(FIGFILE))
all : $(EPSFILE)
clean :
⟼ rm -f $(EPSFILE)
%.eps : %.fig
⟼ fig2dev -L eps $< > $@
```

Un raccourci de `patsubst` pour la substitution précédente est :

```
EPSFILE=$(FIGFILE:.fig=.eps)
```

qui remplace la chaîne `.fig` par la chaîne `.eps`, à la fin des mots. La forme `patsubst` est donc en quelque sorte une généralisation de la précédente.

Plus généralement, on peut instancier une variable en utilisant le mot clef `shell` suivi d'une ou plusieurs commandes interprétées par le shell. Un exemple d'une telle construction peut être :

```
# récupère le dernier fichier TeX sauvé
TMP=$(shell ls -1tr *.tex | tail -1)
# enlève l'extension
LASTTEXFILE=$(shell echo $(TMP) | sed s/\.tex$$// )
```

Il existe bien d'autres outils de traitement sur les chaînes de caractères dans les variables, parmi ceux-ci de nombreux opérateurs de type « rechercher/remplacer », des opérateurs propres aux noms de fichiers (suppression d'extension, du répertoire, etc.), des boucles de type « pour chaque », voir à ce sujet le manuel au format `info` du GNU `make`.

5.3.5 Options et fonctionnalités diverses

Options de make

Comme toute commande UNIX, `make` peut prendre quelques options et paramètres :

- les paramètres doivent correspondre au nom d'une cible du `Makefile`. Dans notre exemple, elle peut être :

```
$ make bidule.eps
...
$
```

  ou :

```
$ make clean
rm -f arborescence.eps disques.eps flux.eps pipes.eps
$
```

  si le fichier `bidule.fig` existe. Si make ne trouve pas de règle pour fabriquer la cible, un message d'erreur est renvoyé :

```
$ make djobi
make: Nothing to be done for ‘djobi’.
$
```

  Si aucun paramètre n'est précisé, c'est la première cible rencontrée qui est exécutée.
- par défaut `make` cherche les règles et les cibles parmi quelques fichiers dont les noms sont prédéfinis (`Makefile`, `makefile`, etc.), l'option `-f` permet d'utiliser un autre nom pour le fichier :

```
$ make -f bidule.mak
...
$
```

- l'option `-n` demande à `make` d'afficher les commandes prévues pour la cible en question, sans les exécuter ; ce qui peut être utile pour débuguer les `Makefile` ;
- notez enfin que si la commande d'une règle est préfixée par le caractère `@`, cette commande n'est pas affichée avant d'être exécutée comme elle l'est par défaut :

```
cible1 :
⟼ echo six bleus 1
cible2 :
⟼ @echo six bleus 2
```

```
$ make cible1 cible2
echo six bleus 1   <------------------------- echo de la commande
six bleus 1
six bleus 2   <------------------------------ pas d'echo pour cible2
$
```

Règles prédéfinies

Comme nous aurons l'occasion de le voir au paragraphe 5.4 les utilitaires `make`, et notamment le GNU `make`, définissent certaines règles dite *implicites* pour fabriquer des fichiers « standards ». À titre d'information, le GNU `make` dispose d'un catalogue de règles pour, entre autres :

- compiler du C, du C++, du Pascal ;
- faire des éditions de liens ;
- créer des fichiers `dvi` à partir des sources TEX ;
- ...

Nous exploiterons ces règles prédéfinies dans la section suivante dédiée à la compilation d'un programme en langage C.

5.4 Faire des projets en langage C

Nous supposons ici que le lecteur connaît le langage C, et présentons les mécanismes de la création d'un exécutable UNIX à partir d'un source écrit en C. La création d'un programme passe par différentes phases d'analyse dont on peut trouver des « méthodes » dans les livres de génie logiciel et d'algorithmique. Nous nous attarderons ici sur la phase « technique » et supposons que le lecteur a un niveau correct en langage C.

5.4.1 Remarques préliminaires

Voici un petit glossaire qui permettra de clarifier notre vocabulaire. Les différents types de fichiers que nous rencontrerons seront :

Source : c'est le fichier texte qui contient le programme en langage C ;

Exécutable : c'est le fichier binaire qui contient des instructions destinées au processeur (code machine) et qui peut être chargé en mémoire par le système d'exploitation ;

Objet : c'est un fichier binaire résultant de la *compilation*. Ce fichier contient le code machine de certaines fonctions définies par le programmeur et des appels à des fonctions définies autre part (dans un autre fichier objet ou dans une *bibliothèque*) ;

Bibliothèque : c'est un fichier binaire s'apparentant à un fichier objet et contenant le code d'un certain nombre de fonctions rendues publiques ;

Bibliothèque dynamique : il s'agit d'une bibliothèque dont le code machine n'est pas inséré dans l'exécutable, mais qui est chargé à la demande lors de l'*exécution*.

Les actions que nous serons amenés à effectuer pour la gestion de notre projet seront :

Compilation : c'est la phase permettant de faire une analyse syntaxique, grammaticale et sémantique du fichier source et de créer, si cette première phase

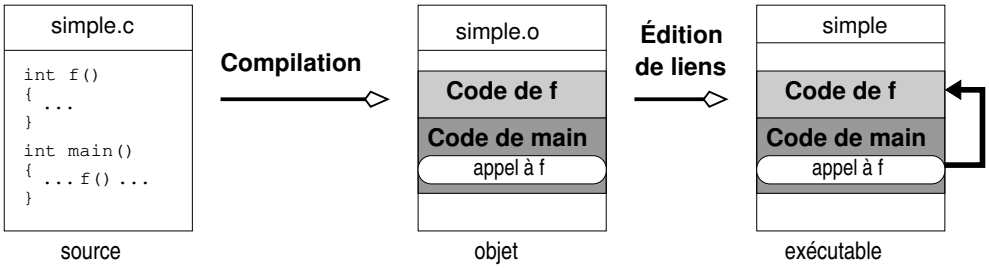


FIGURE 5.1: Création d'un exécutable à partir d'un seul fichier source

se déroule sans heurt, un fichier objet ; elle se décompose en *preprocessing*, compilation et assemblage ;

Édition de liens : c'est la phase qui regroupe les fichiers objets et les bibliothèques, et qui après avoir vérifié que toutes les fonctions appelées dans les fichiers objets pointent bien vers un code objet, crée un exécutable. L'édition de liens en anglais se dit *to link*, et l'utilitaire permettant de l'effectuer est le *linker*.

5.4.2 Étude du cas simple : un seul fichier source

Supposons que l'on ait à créer un exécutable à partir du fichier suivant (voir aussi figure 5.1) :

simple.c

```
int f()
{
  return 4;
}
int main()
{
  int i=f();
  printf("Bonjour tout le monde.\n");
  return 0;
}
```

Création de l'exécutable

On peut dans le cas simple d'un projet se réduisant à un seul fichier source, lancer une commande qui crée directement l'exécutable :

```
$ gcc -o simple simple.c
$
```

C'est l'option `-o` ⟨*nom*⟩ qui permet de créer un exécutable dont le nom est ⟨*nom*⟩ [14]. Ce nom peut être choisi librement, aucune extension particulière n'est nécessaire. Ce sont les *attributs* du fichier qui le rendent exécutable et non son nom.

Ici les phases de compilation et d'édition de liens sont réalisées automatiquement sans que l'utilisateur n'ait à le demander explicitement. Dans ce cas, l'éditeur de liens trouve le code de la fonction `f` dans le fichier `simple.c` compilé.

14. Par défaut ce nom est généralement `a.out`, le nom que THOMPSON avait utilisé pour le premier assembleur créé sur les toutes premières versions d'UNIX.

Exécution

On exécute le fichier résultant de l'édition de liens en l'appelant par son nom :

```
$ ./simple
Bonjour tout le monde.
$
```

Tant que le répertoire où réside l'exécutable n'est pas listé dans le contenu de la variable `PATH`, il est nécessaire de préfixer l'exécutable par le chemin pour pouvoir l'exécuter : c'est ici la signification du « `./` » dans `./test`. Dans le cas contraire, deux cas peuvent se produire :

1. le système ne trouve pas votre programme, par exemple si votre programme s'appelle `bidule` et que vous tentez de l'exécuter comme suit :

   ```
   $ bidule
   bash: bidule: command not found
   $
   ```

2. le système n'appelle pas *votre* programme. Par exemple si vous avez la mauvaise idée de nommer votre exécutable `test` :

   ```
   $ test   <-------------------------------- Il semble ne rien se passer
   $ type test
   test is a shell builtin
   $
   ```

   il existe donc une autre commande nommée `test` et c'est elle qui est exécutée à la place de la votre.

Certains objecteront à la lecture de ce paragraphe : « Mais pourquoi ne pas rajouter le répertoire courant dans la variable `PATH`[15] ? » La réponse est que pour des raisons de sécurité il n'est pas conseillé de rajouter « . » (le répertoire courant) dans cette variable (lire à ce sujet Garfinkel et Spafford (1996)). Il est donc bon de prendre d'ores et déjà l'habitude d'utiliser le fameux « `./` ». On pourra également lire le paragraphe 6.1.4 page 147 qui propose une autre solution.

Préprocesseur

Avant d'être compilé, un source en langage C est passé à la moulinette du *préprocesseur*. Ce programme va traiter toutes les directives de type `#...` et les remplacer par du code source. Il supprimera également tous les commentaires. Le résultat de ce « traitement de texte » sera présenté au compilateur. Sous UNIX, le préprocesseur porte le doux nom de `cpp` (*C preprocessor*). On peut voir `cpp` en action en utilisant l'option `-E` de `gcc`, sur un fichier exemple :

testcpp.c

```
#define N 4
#define carre(a) ((a)*(a))

int main(int argc, char** argv)
{
  int i=N,j,k;
  j=2*N;
```

15. Si vous n'avez pas l'intention d'objecter, continuez donc à lire tranquillement en sautant ce qui suit...

```
  k=carre(i+j);
  return 0;
}
```

puis :

```
$ gcc -E testcpp.c
int main(int argc, char** argv)
{
  int i= 4 ,j,k;
  j=2* 4 ;
  k= (( i+j )*( i+j ));
  return 0;
}
$
```

On peut également noter que le préprocesseur de `gcc` définit plusieurs symboles qui identifient le système sous-jacent. On peut utiliser ces symboles lorsqu'on veut créer du code C portable d'un système (UNIX) à un autre :

```
$ cpp -dM  testcpp.c
 ...
#define __linux__ 1
#define __gnu_linux__ 1
#define __unix__ 1
#define __i386__ 1
#define __INT_MAX__ 2147483647
#define __LONG_LONG_MAX__ 9223372036854775807LL
#define __DBL_MIN__ 2.2250738585072014e-308
#define N 4
$
```

Ces symboles peuvent ensuite être exploités par des directives de type `#ifdef` :

```
#if defined __linux__ && defined __i386__
int fonction_de_la_mort(int,int)
#else
int fonction_de_la_mort(double,double)
#endif
```

Répertoires de recherche

Lorsque `cpp` se met en branle, et qu'il rencontre une directive `#include`, il va chercher dans une liste de répertoires le fichier spécifié. On peut voir la liste de recherche par défaut en utilisant à nouveau l'option `-v` de `gcc` :

```
$ gcc -v -E fantome.c
 ...
gcc version 3.3.5 (Debian 1:3.3.5-13)
  /usr/lib/gcc-lib/i486-linux/3.3.5/cc1 -E -quiet -v src/testcpp.c
#include "..." search starts here:
#include <...> search starts here:
/usr/local/include
```

```
/usr/i386-redhat-linux/include
/usr/lib/gcc-lib/i486-linux/3.3.5/include
/usr/include
End of search list.
$
```

On notera que :

- les fichiers inclus par une directive du type `#include "⟨fichier⟩"` sont recherchés à partir du *répertoire courant* ;
- on peut influer sur la liste des répertoires dans lesquels une directive `#include` recherche les fichiers, grâce à l'option `-I` de `gcc`. Si par exemple, on désire inclure le fichier `bidule.h` se trouvant dans `~/mes_includes`, on spécifiera en ligne de commande :

  ```
  $ gcc -I ~/mes_includes -c monfichier.c
  ```

  avec :

  monfichier.c

  ```
  #include <bidule.h> /* ou "bidule.h" */
  ...
  ```

Voir le code assembleur

Sur les systèmes UNIX utilisant l'environnement de développement de chez GNU, la plupart des langages de programmation compilés (C, C++, Pascal, Prolog) sont d'abord traduits en langage d'assemblage, on fait ensuite appel à l'outil assembleur pour produire le fichier objet. Par simple curiosité on peut voir le code assembleur produit pour le fichier suivant :

testas.c

```
int mafonction(double d)
{
  return 2.3*d-7.4;
}
```

On peut demander à `gcc` de s'arrêter à la phase d'assemblage (c'est-à-dire ne pas produire le fichier objet) avec la commande suivante :

```
$ gcc -S testas.c
$ ls -l testas.*
-rw-r--r--  1 lozano lozano  57 Nov 11 18:30 testas.c
-rw-r--r--  1 lozano lozano 640 Nov 11 18:31 testas.s
$
```

On peut alors — toujours dans le cadre d'une curiosité malsaine — examiner le contenant du code assembleur généré, avec la commande `cat` :

```
$ cat testas.s
...
mafonction:
      pushl   %ebp
      movl    %esp, %ebp
      subl    $16, %esp
...
$
```

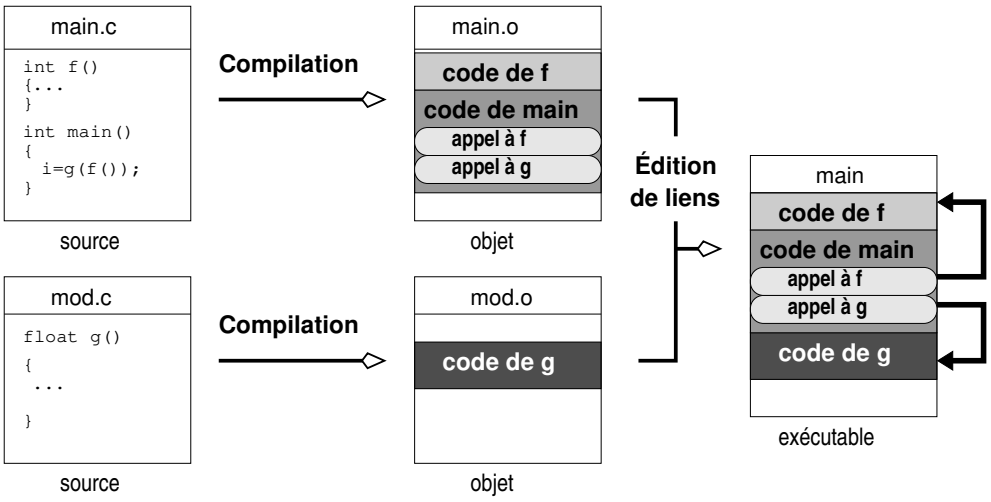


FIGURE 5.2: Compilation séparée

On peut ensuite traduire ce code en appelant explicitement l'outil assembleur :

```
$ as testas.s -o testas.o
$
```

5

Autres options intéressantes

`gcc` dispose de moult options que nous ne détaillerons bien évidemment pas ici. Voici cependant quelques options utiles à connaître pour mener à bien vos projets :

- l'option `-O`⟨*nombre*⟩ [16] est met en route différentes méthodes d'optimisation pour produire du code plus rapide ou utilisant moins de mémoire. En ce qui concerne ⟨*nombre*⟩, il peut prendre des valeurs de 0 à 3 (0 : pas d'optimisation, 3 : optimisation max) ;
- l'option `-Wall` affiche tous les *warnings* de compilation ; c'est une option incontournable qui permet d'attirer l'attention du programmeur sur des portions de programmes « douteuses » (variables non initialisées, problème de transtypage, ...) ;
- `-D`⟨*macro*⟩ est une option passée au préprocesseur qui produit le même résultat que l'inclusion dans le source d'une ligne `#define` ⟨*macro*⟩ 1 ;

5.4.3 Compilation séparée

Dans le contexte de la compilation séparée on a à compiler plusieurs modules source. Examinons, en guise d'exemple, la situation (figure 5.2) dans laquelle un fichier contient le `main` :

main.c

```
#include <stdio.h>
#include "mod.h"

int f() { return 4; }
int main(int argc, char** argv)
{
```

16. Ici il s'agit d'un « O » majuscule, pour « optimisation. »

```
  int i=g(f());
  return 0;
}
```

qui fait appel à une fonction g dont la déclaration se trouve dans `mod.h` et la définition dans `mod.c` :

mod.h

```
float g(float);
```

mod.c

```
float g(float f)
{
  return 2*f;
}
```

Pour compiler ce projet :

1. On doit compiler séparément `main.c` et `mod.c` ;
2. Faire l'édition de liens à partir des fichiers objets créés.

Compilation en ligne de commande

Pour créer un fichier objet à partir d'un source en langage C — c'est-à-dire procéder à la phase de compilation — on peut lancer la commande :

```
$ gcc -c main.c -o main.o   <-------------- compilation du premier module
$ gcc -c mod.c -o mod.o     <-------------- compilation du deuxième module
$
```

Il faut noter ici l'option `-c` du compilateur qui précise qu'on n'effectue que la compilation et l'option `-o` (*output*) précise que les fichiers objet doivent être stockés dans les fichiers `main.o` et `mod.o`.

Édition de liens en ligne de commande

Pour passer à la phase suivante qui est la création de l'exécutable par le biais de l'édition de liens, on utilise la commande suivante :

```
$ gcc -o main main.o mod.o
$
```

Pour effectuer l'édition de liens, on précise donc le nom de l'exécutable avec l'option `-o`, et on spécifie quels sont les fichiers objet à lier.

5.4.4 Bibliothèques

Lier un exécutable avec une bibliothèque

Pour ce qui concerne Linux — et c'est le cas de la plupart des systèmes UNIX — les exécutables créés sont des exécutables *dynamiques*, c'est-à-dire que le code qu'ils contiennent fait appel à des bibliothèques dynamiques. Soit le source suivant :

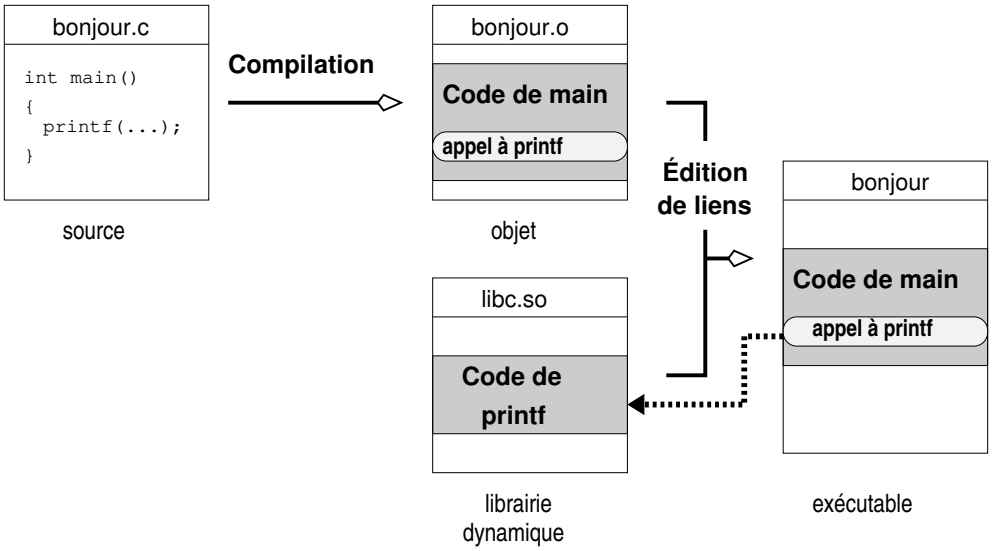


FIGURE 5.3: Édition de liens dynamique.

bonjour.c

```
#include <stdio.h>
int main()
{
  printf("bonjour tout le monde\n");
  return 0;
}
```

et la commande suivante pour créer un exécutable :

```
$ gcc -o bonjour bonjour.c
$
```

On peut examiner l'exécutable créé avec la commande `ldd` pour obtenir les bibliothèques dont il dépend :

```
$ ldd bonjour
    libc.so.6 => /lib/libc.so.6 (0x4001a000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

On voit donc ici que l'exécutable `bonjour` est lié dynamiquement avec la bibliothèque standard, dite « libC » (voir figure 5.3). Ces deux bibliothèques sont automatiquement ajoutées à l'édition de liens, sur le système Linux de l'auteur. La première contient le code de la bibliothèque standard du C (et donc celui de la fonction `printf`), la deuxième contient les instructions permettant de charger dynamiquement les bibliothèques.

Dans certaines situations, il faut préciser explicitement que le code fait appel à des fonctions définies dans une bibliothèque spécifique. Par exemple pour compiler le fichier en langage C suivant faisant appel à une routine de la bibliothèque mathématique :

testmath.c

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

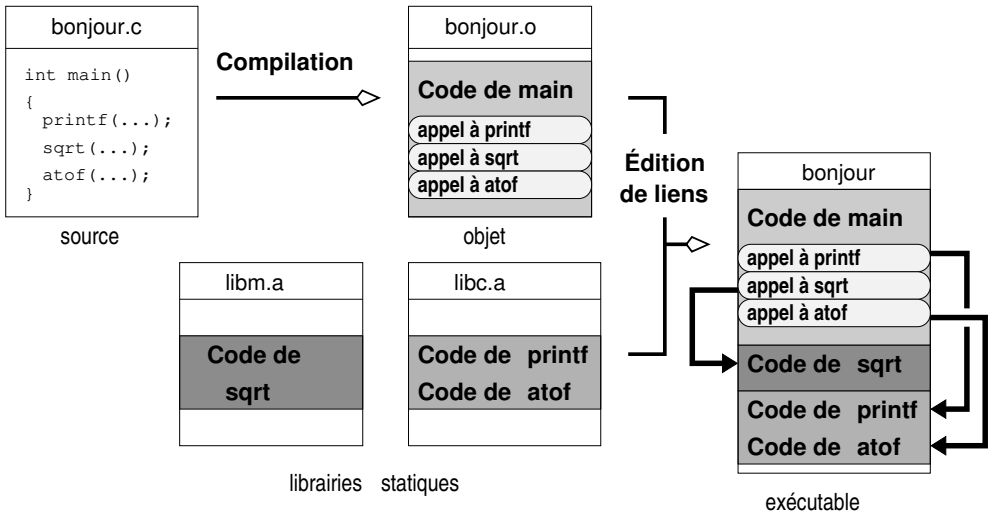


FIGURE 5.4: Édition de liens statique.

```
#include <math.h>

int main(int argc, char** argv)
{
  if (argc==2)
    /* renvoie la racine de l'argument 1 */
    printf("%f\n",sqrt(atof(argv[1])));
  return 0;
}
```

on procédera comme suit :

```
$ gcc -c testmath.c -o testmath.o
$ gcc -o testmath  testmath.o -lm
$
```

L'option `-l` précise que l'on veut ajouter la bibliothèque mathématique du C à l'édition de liens. On a alors :

```
$ ldd testmath
libm.so.6 => /lib/libm.so.6 (0x40021000)
libc.so.6 => /lib/libc.so.6 (0x40043000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

on constate donc que l'exécutable `testmath` est lié dynamiquement avec la bibliothèque mathématique stockée dans le fichier `/lib/libm.so.6`. Il est possible de forcer la création d'un exécutable statique. Pour cela on impose alors au *linker* d'insérer dans l'exécutable le code des fonctions provenant de ou des bibliothèques nécessaires :

```
$ gcc -static -o testmath.stat testmath.o -lm
$ ls -lh testmath*
-rwxr-xr-x  1 lozano lozano  12K Nov 11 17:54 testmath
-rw-r--r--  1 lozano lozano  223 Nov 11 17:54 testmath.c
-rw-r--r--  1 lozano lozano  948 Nov 11 17:54 testmath.o
```

```
-rwxr-xr-x  1 lozano lozano 471K Nov 11 17:58 testmath.stat
$
```

On pourra noter la différence de taille des exécutables créés. Ceci confirme que dans le cas « statique » le code des fonctions issu des bibliothèques est inséré dans l'exécutable (voir figure 5.4 page ci-contre).

Où sont les bibliothèques ?

Lorsqu'on précise, lors de l'édition de liens, l'option `-l` suivie de `m`, le *linker* cherche dans une série de répertoires prédéterminée, un fichier nommé `libm.so` pour la bibliothèque dynamique et `libm.a` pour la bibliothèque statique. Dans le cas où la bibliothèque à inclure n'est pas dans la liste de répertoires standard, il est nécessaire d'utiliser l'option `-L` suivie du répertoire où se trouve la bibliothèque en question. Par exemple, on écrira :

```
$ gcc -L ~/lib -o djobi djobi.o -lmabib
$
```

si la bibliothèque `libmabib.so` ou `libmabib.a` se trouve dans le répertoire `~/lib`. S'il s'agit d'une bibliothèque dynamique (`libmabib.so`), pour pouvoir *exécuter* le fichier `djobi`, il faudra impérativement indiquer au programme qui charge les bibliothèques dynamiques où se trouve `libmabib.so`. On peut réaliser cela en positionnant la variable d'environnement `LD_LIBRARY_PATH` avec la liste des répertoires susceptibles de contenir des bibliothèques dynamiques :

`export LD_LIBRARY_PATH=`⟨*dir*$_1$⟩`:`⟨*dir*$_2$⟩`: $LD_LIBRARY_PATH`

Dans notre exemple on écrira :

```
$ export LD_LIBRARY_PATH=~/lib:$LD_LIBRARY_PATH
$
```

À titre indicatif, sous le système HPUX (l'UNIX de Hewlett Packard) cette variable se nomme `SHLIB_PATH`.

Créer une bibliothèque

On peut créer des bibliothèques qui seront liées statiquement ou dynamiquement lors de l'édition de liens. La création d'une bibliothèque consiste à rassembler un certain nombre de fichiers objets dans un seul fichier. On supposera, à titre d'exemple, que l'on dispose de trois fichiers : `mod1.o` `mod2.o` et `mod3.o` résultant de la compilation de trois fichiers écrits en langage C.

Statique Pour créer une bibliothèque statique, on pourra utiliser la commande :

```
$ ar ruv libbidule.a mod1.o mod2.o mod3.o
ar: création de libbidule.a
a - mod1.o
a - mod2.o
a - mod3.o
$
```

La commande `ar` crée un fichier bibliothèque `libbidule.a`. L'option `ru` permet d'insérer en remplaçant chaque objet dans l'archive, l'option `v` permet de rendre la commande `ar` plus bavarde qu'elle ne l'est par défaut. On se reportera à la page de manuel pour plus de détail. Une fois la bibliothèque statique créée on peut l'utiliser lors d'une édition de liens comme suit :

```
$ gcc -L. -o monappli monmain.o -lbidule
$
```

On notera :

1. l'option `-L` pour indiquer qu'il faut chercher la bibliothèque dans le répertoire courant ;
2. la syntaxe `-lbidule` qui cherche une bibliothèque nommée `libbidule.a`

L'option `-s` de la commande `ar` ou l'utilisation de la commande `ranlib` en lieu et place de `ar` permet de créer un index des symboles et ainsi d'accélerer les éditions de liens que l'on réalisera avec la bibliothèque.

Dynamique Pour créer une bibliothèque dynamique on utilisera la syntaxe suivante :

```
$ gcc -shared -o libbidule.so mod1.o mod2.o mod3.o
$
```

Pour utiliser une telle bibliothèque à l'édition de liens :

```
$ gcc -L. -o monappli monmain.o -lbidule
$
```

Il faudra s'assurer que la variable d'environnement `LD_LIBRARY_PATH` contienne le répertoire où se trouve `libbidule.so` pour que cette dernière puisse être chargée au moment de l'exécution de `monappli` (cf. § 5.4.4 page précédente).

5.4.5 Se simplifier la vie avec `make`

Encore une fois, nous supposons que le lecteur a à sa disposition le GNU `make`. Ce dernier définit deux règles liées à la gestion de projet en C et C++. La première règle est la suivante :

```
%.o : %.c
⟼$(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

qui indique comment construire le fichier objet à partir du fichier C. Les variables `CPPFLAGS` et `CFLAGS` permettent respectivement de passer des options au préprocesseur et au compilateur [17]. On pourra donc utiliser par exemple le `makefile` suivant :

makefile.test.1

```
CFLAGS=-Wall
CPPFLAGS=-I/usr/local/machin-chose

test.o : test.c
```

17. En C++, `CFLAGS` devient `CXXFLAGS`.

ce qui donnera :

```
$ make -f makefile.test.1
cc -Wall -I/usr/local/machin-chose  -c test.c -o test.o
$
```

Le GNU `make` positionne par défaut la variable `$(CC)` à la valeur `cc`. En outre, il définit un ensemble de règles pour l'édition de liens :

- s'il n'y a qu'un seul fichier objet, la règle est la suivante :
  `% : %.o`
  `⟼$(CC) $(LDFLAGS) $< $(LOADLIBES)`
  `LDFLAGS` permet de passer des options au linker, et `LOADLIBES` permet d'indiquer quelles sont les bibliothèques nécessaires à l'édition de liens.
- s'il s'agit d'une compilation séparée, une règle du type :
  `main : main.o fichier1.o fichier2.o`
  lancera la compilation des fichiers `main.c`, `fichier1.c` et `fichier2.c`, puis l'édition de liens avec les trois fichiers objets générés pour créer l'exécutable `main`. Dans le cas de la compilation séparée, il est nécessaire qu'un des fichiers source porte le nom de la cible (ici c'est `main.c`).

Voyons deux exemples, le premier pour le cas d'un seul fichier objet :

makefile.test.2

```
CFLAGS=-Wall
CPPFLAGS=-I/usr/local/machin-chose
LDFLAGS=-L/usr/local/bidule-truc
LOADLIBES=-lbidule

test : test.o
```

qui donne :

```
$ make -f makefile.test.2 test
cc -Wall -I/usr/local/machin-chose  -c test.c -o test.o
cc -L/usr/local/bidule-truc  test.o -lbidule  -o test
$
```

Dans le cas de la compilation séparée, on pourra écrire le `makefile` suivant qui permet de créer l'exécutable de l'exemple du paragraphe 5.4.3 page 135 :

makefile.test.3

```
CFLAGS=-Wall
CPPFLAGS=-I/usr/local/machin-chose
LDFLAGS=-L/usr/local/bidule-truc
LOADLIBES=-lbidule

main : main.o mod.o
```

qui donne :

```
$ make -f makefile.test.3
cc -Wall -I/usr/local/machin-chose  -c main.c -o main.o
cc -Wall -I/usr/local/machin-chose  -c mod.c -o mod.o
cc -L/usr/local/bidule-truc main.o mod.o -lbidule -o main
$
```

Pour finir cette section sur l'utilisation de make dans le cadre d'un projet en langage C, nous allons voir comment on peut demander à gcc de générer pour nous les dépendances du projet. Dans notre exemple de compilation séparée, ces dépendances sont les suivantes :

- `mod.o` dépend uniquement de `mod.c` : on doit reconstruire le premier si le dernier a été modifié ;
- `main.o` dépend de `main.c` et de `mod.h` à cause de la présence de la directive `#include "mod.h"` : si `mod.h` ou `main.c` a été modifié il faut reconstruire `main.o`.

Cette liste peut être générée automatiquement par gcc grâce à l'option `-MM` qui affiche les dépendances liées aux fichiers inclus avec la directive du préprocesseur `#include"`⟨*fichier*⟩`"`. Ainsi :

```
$ gcc -MM main.c mod.c
main.o: main.c mod.h
mod.o: mod.c
$
```

On peut utiliser cette option de `-MM` directement dans le `makefile` lui-même :

```
CFLAGS=-Wall
CPPFLAGS=-I/usr/local/machin-chose
LDFLAGS=-L/usr/local/bidule-chouette
LOADLIBES=-lbidule
OBJFILES=main.o est_f.o
CFILES=$(OBJFILES:.o=.c)

main : $(OBJFILES)
depend.dat :
⟼ $(CC) -MM $(CPPFLAGS) $(CFLAGS) $(CFILES) > depend.dat
-include depend.dat
```

Dans ce `makefile`, on a défini une cible nommée `depend.dat` dont le but est de construire le fichier de dépendances `depend.dat`. La directive `-include` permet ensuite d'inclure le fichier `depend.dat` même si celui-ci n'existe pas (c'est la signification du tiret qui préfixe `include`). On pourra également remarquer l'utilisation des variables `$(OBJFILES)` et `$(CFILES)`, créées ici pour « rationaliser » le `makefile` (cf. § 5.3.4 page 128).

Conclusion

La « digestion » des informations fournies dans ce chapitre devrait vous donner les bases nécessaires pour créer des scripts divers et variés, lesquels, à l'instar de ceux de l'auteur, iront en se complexifiant et s'améliorant. La connaissance des outils « standard » d'UNIX (`awk`, `sed`, etc.), couplée avec celles des scripts shell et du fonctionnement de `make`, est un atout majeur pour entamer un projet de développement ambitieux. C'est également un atout pour se lancer dans les tâches d'administration système. La connaissance des principes de base de la compilation sous UNIX est quant à elle importante, car elle permet de comprendre les mécanismes qui entrent en jeu dans la création d'un exécutable.

6 Se mettre à l'aise !

Sommaire

6.1 Avec le shell
6.2 Avec vi
6.3 Avec Emacs
6.4 Avec Xwindow
6.5 Installer des logiciels

On login (subject to the `-noprofile` *option):*
if `/etc/profile` *exists, source it.*
if `~/.bash_profile` *exists, source it,*
On exit: if `~/.bash_logout` *exists, source it.*

man bash.

DÉVELOPPER un projet c'est aussi être dans un environnement agréable et correspondant à ses propres besoins. Ce chapitre a pour but de présenter la configuration de l'environnement de travail pour que le programmeur puisse se focaliser sur son travail. Ce chapitre n'est pas un inventaire de ce que l'on peut configurer sous UNIX, mais consiste en une présentation des mécanismes de configuration proprement dits. Nous présentons donc ici, la configuration du shell, l'utilisation d'`emacs` et `vi`, le principe de la configuration d'un environnement graphique, ainsi que l'installation de logiciels pour un utilisateur sans privilège. Cette dernière section est terminée par une présentation de la gestion des paquets Debian.

6.1 Avec le shell

Le shell est l'environnement de travail de base sous UNIX. Grâce au shell on peut dialoguer avec le système par le biais de commandes, de scripts, en lançant des logiciels, etc. Il est donc important que cet environnement soit configuré le mieux possible par rapport au besoin de l'utilisateur. La plupart des configurations effectuées au niveau du shell se font :

- soit à l'aide de variables dites ►*variables d'environnement*◄ ; § 2.1.3 p. 26 ◄
- soit en positionnement des variables propres au shell utilisé.

Les possibilités de configuration présentées dans cette section ne prennent leur intérêt que couplées avec les ►fichiers de démarrage◄ : la configuration d'un environnement utilisateur passe par le positionnement de variables dans ces fichiers de démarrage de manière à rendre actif l'environnement personnalisé à chaque connexion. § 6.1.6 p. 148 ◄

Les variables d'environnement sont des variables qui sont « exportées » — c'est-à-dire transmises — aux ►processus◄ fils du shell, par l'une des syntaxes suivantes : § 2.4 p. 47 ◄

- `export` ⟨*variable*⟩=⟨*valeur*⟩ pour la famille `sh`, ou :

– `setenv` ⟨*variable*⟩ ⟨*valeur*⟩, pour la famille `csh`.

Voyons sur un exemple comment sont exportées ces variables en considérant le script suivant :

testexport.sh

```
#!/bin/bash
echo X=$X
echo Y=$Y
```

```
$ X=4
$ export Y=5
$ ./testexport.sh
X=
Y=5
$
```

on constate ici que le script `testexport.sh` « connaît » la valeur de la variable `Y` mais `X`, elle, est inconnue car uniquement définie pour le shell.

6.1.1 Le prompt

On peut changer l'allure du prompt en modifiant le contenu de la variable `PS1`. Par exemple :

```
$ PS1="vasyfaitpter# "
vasyfaitpter# date
Wed Jun  7 11:55:21 CEST 2000
vasyfaitpter#
```

On peut avoir un prompt un peu plus évolué dans lequel il est possible d'ajouter entre autres (ce qui suit est spécifique à bash) :

– le nom de l'utilisateur : `\u` (*user*) ;
– le nom de la machine : `\h` (*host name*) ;
– le répertoire de travail : `\w` ou `\W` pour une forme abrégée (*working directory*) ;
– la date et l'heure : `\d` et `\t` (*date* et *time*).

Nous avons un petit faible pour le prompt suivant :

```
$ PS1="\u@\h:\w>"
lozano@coltrane:~/tcqvatvs/guide>
```

ou plus succinctement :

```
$ PS1="\h\w>"
coltrane~/tcqvatvs/guide>
```

Les shells utilisent également un prompt particulier pour indiquer qu'une commande n'est pas complète bien que l'utilisateur ait pressé la touche Entrée . Ce prompt s'affiche sous bash en fonction de la valeur de la variable `PS2`. Par défaut cette variable contient la chaîne « > », on peut y mettre par exemple :

```
$ PS2="et?> "
$
```

pour attirer l'attention. Ainsi :

```
$ for x in 1 2 3; do
et?> echo $[x+1]
et?> done
2
3
4
$
```

Les autres shells disposent également de variables permettant de spécifier l'allure du prompt ; la syntaxe est légèrement différente mais l'idée générale est la même.

6.1.2 Historique des commandes

Comme nous avons pu le noter, les shells d'UNIX proposent généralement un mécanisme d'historique permettant aux vaillants utilisateurs de rappeler les commandes déjà saisies, pour éventuellement les corriger ou simplement les ré-exécuter. L'utilisateur peut compulser cet historique avec la commande `history` :

```
$ history
...  ...
510  xdvi guide-unix.dvi &
511  pdflatex guide-unix
512  acroread guide-unix.pdf &
513  xfig figs/lien_symbolique.fig&
514  history
$
```

Par défaut le shell bash stocke cet historique, à chaque fois qu'on quitte un shell, dans un fichier dont le nom est stocké dans la variable `HISTFILE`. Par exemple sur la machine de votre serviteur :

```
$ echo $HISTFILE
/home0/vincent/.bash_history
$
```

Deux autres variables sont bonnes à connaître :

- `HISTFILESIZE` le nombre de lignes à mémoriser pour l'historique ;
- `HISTCONTROL` qu'on peut positionner à la valeur :
  - `ignoredups` pour ne pas stocker dans l'historique deux commandes identiques ;
  - `ignorespace` pour ne pas stocker les lignes commençant par un espace ;
  - `ignoreboth` pour cumuler les deux effets précédents.

Pour info, sur mon système :

```
$ echo $HISTFILESIZE $HISTCONTROL
500 ignoreboth
$
```

6.1.3 Alias et fonctions

Il est souvent intéressant d'utiliser les *alias* pour créer ce qu'on pourrait appeler des « raccourcis » de commandes. Par exemple, au lieu d'écrire :

```
$ telnet -l lozano coltrane.freejazz.org
Trying xx.xx.xx.xx...
Connected to coltrane.freejazz.org
Escape character is '^]'.
...
$
```

On peut définir un alias :

```
$ alias trane='telnet -l lozano coltrane.freejazz.org'
$ trane
Trying xx.xx.xx.xx...
Connected to coltrane.freejazz.org.
Escape character is '^]'.
$
```

Contrairement aux alias de `tcsh`, ceux de `bash` « se contentent » d'effectuer le remplacement de l'alias par sa définition. Mais cette définition ne peut contenir de référence aux arguments de la ligne de commande. Il faut pour cela utiliser des ▲*fonctions*◀ de `bash`.

► § 5.2.5 p. 121

Voici un exemple simple de fonction : supposons que l'on se connecte régulièrement sur la machine `ayler.freejazz.org` mais sous des utilisateurs différents. On pourrait avoir l'idée de génie d'écrire :

```
function sshayler()
{
  ${1?:="donnez moi un login plize..."}
  ssh -X -l $1 ayler.freejazz.org
}
```

► § 6.1.6 p. 148

Une fois cette fonction stockée dans un ▲fichier de démarrage◀, on pourra simplement taper :

```
sshayler vincent
$
```

pour se connecter en tant qu'utilisateur `vincent` sur `ayler.freejazz.org`, quelle joie... Notons enfin qu'il y a un ordre de priorité dans l'exécution des commandes, alias et autres fonctions ; cet ordre est le suivant :

1. les alias ;
2. les fonctions ;
3. les commandes internes ;
4. les commandes externes.

Ainsi, si par exemple on définit un alias :

```
$ alias xdvi='xdvi -keep'
$
```

Alors la commande :

```
$ xdvi bidule.dvi &
$
```

fera d'abord appel à l'alias qui lui, fera appel à la commande externe. L'option `-a` de la commande `type` peut donner des informations à ce sujet :

```
$ type -a xdvi
xdvi is aliased to 'xdvi -keep'    <-------------------------------- alias
xdvi is /usr/bin/xdvi    <-------------------------------- commande externe
$
```

6.1.4 Environnement de développement

Lorsqu'on développe des applications il arrive qu'on veuille les stocker dans un sous-répertoire du répertoire privé tel que, par exemple, `~/bin`. Sans modification particulière de l'environnement les programmes stockés dans ce répertoire ne pourront être exécutés sans spécifier le nom complet du répertoire de stockage :

`~/bin/`⟨*mon-exécutable*⟩

Pour remédier à cet inconvénient on peut modifier la variable d'environnement `PATH`. Cette variable contient une liste de répertoires séparés par des « : » dans lesquels le shell va chercher les exécutables appelés en ligne de commande. Par exemple, sur notre système :

```
$ echo $PATH
/usr/local/Acrobat5/bin:/usr/local/bin:
/bin:/usr/bin:/usr/X11R6/bin
$
```

Par conséquent, pour rajouter `~/bin` dans cette liste, on peut écrire :

```
$ export PATH=~/bin:$PATH
$
```

Ceci permet d'ajouter en début de liste notre répertoire de stockage des exécutables « maison ». Lorsque ces exécutables sont liés dynamiquement avec des bibliothèques « maison », on pourra stocker ces bibliothèques dans un répertoire, par exemple `~/lib` ; et pour faire en sorte que leur chargement se déroule correctement il faudra mettre à jour la variable d'environnement `LD_LIBRARY_PATH` :

```
$ export LD_LIBRARY_PATH=~/lib:$LD_LIBRARY_PATH
$
```

On pourra aussi se reporter au paragraphe 5.4.4 page 139 pour plus d'informations au sujet des bibliothèques dynamiques.

6.1.5 Interaction avec les logiciels

Certains logiciels font appel à des variables d'environnement pour fonctionner. On ne peut évidemment pas dresser une liste exhaustive de ces variables. Voici cependant quelques exemples qui permettront de comprendre le mécanisme global : beaucoup de logiciels sont conçus pour avoir un comportement par défaut que l'on peut surcharger en positionnant une variable d'environnement.

– certains programmes utilisent la variable `EDITOR` au moment de lancer l'édition d'un texte ; ils supposent généralement que l'éditeur par défaut est `vi` ; si pour une raison ou une autre vous désirez utiliser un autre éditeur :

```
$ export EDITOR=⟨mon_éditeur⟩
$
```

fera l'affaire ;

– la variable `TERM` indique au système le type de terminal utilisé. La valeur « passe-partout » est le `vt100` compatible avec la plupart des terminaux et qui suffit à la plupart des applications. On pourra donc écrire :

```
$ export TERM=vt100
$ reset
...
$
```

en cas de liaison difficile avec une machine distante possédant un système n'identifiant pas votre type de terminal.

► § 7.2 p. 183

– la variable `PAGER` définit le programme chargé d'afficher les pages de manuel avec la commande `man` ;

– ...

En règle générale, les pages de manuel contiennent une rubrique baptisée « Environnement » qui décrit les variables d'environnement ayant une influence sur le programme en question.

6.1.6 Sauvegarder ses configurations grâce aux fichiers de démarrage

Lorsque le système lance un processus de shell, ce dernier va lire — suivant sa famille `sh` ou `csh` — dans un ordre défini, un certain nombre de fichiers pouvant contenir des commandes ou des initialisations de variables d'environnement. Ces fichiers sont appelés *fichiers de démarrage* et portent généralement le doux nom, de `login`, `profile`, et autre `monshellrc`... Pour comprendre le fonctionnement des fichiers de démarrage, il faut d'abord distinguer les trois types de lancement d'un shell :

login shell : ou *shell de connexion*, il est lancé lors de la connexion (ou *login*) sur le système ;

non login interactive shell : il s'agit d'un shell lancé autrement que par une procédure de connexion, par exemple par l'intermédiaire d'une fenêtre `xterm` ;

non interactive : shell lancé pour interpréter un fichier de commande ;

Dans le cas du shell `bash`, les fichiers lus sont les suivants :

login shell : `~/.bash_profile` ;

non login shell : `~/.bashrc` ;

non interactif : le fichier dont le nom est dans la variable `ENV`.

Voici un exemple de configuration classique des fichiers de démarrage. De manière à tenir compte à la fois de la phase de connexion (login shell) et des shells interactifs (non login interactive shell), une solution est d'articuler la lecture des fichiers `~/bash_profile` et `~/.bashrc` comme suit :

`~/.bash_profile`

```
if [ -f ~/.bashrc ]; then
   . ~/.bashrc
fi
# exportation de variables :
...
```

et :

~/.bashrc
```
# définition d'alias et fonctions
...
```

Ainsi, les variables utiles à la configuration de l'environnement sont initialisées une fois pour toute à la connexion. Ce qui suit est un extrait des fichiers de configuration de l'auteur :

~/.bash_profile
```
if [ -f ~/.bashrc ]; then
   . ~/.bashrc
fi
export PATH=~/bin:$PATH
export MANPATH=~/man:$MANPATH
export POVINI=~/povray3.6/povray.ini
export TEXINPUTS=~/LaTeX/myinputs:
export LD_LIBRARY_PATH=~/lib:~/src/lib:~/lib/GL
PS2="et?> "
HISTCONTROL=ignoreboth
umask 027
```

et :

~/.bashrc
```
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias up='cd ..'
alias uup='cd ../..'
alias ou='find . -name '
alias ltr='ls -ltr'

function cleantex()
{
  /bin/rm -f $1.aux $1.log $1.bbl $1.blg $1.mtc* $1.?dx\
  $1.?lg $1.?nd $1.toc
}
```

Pour ajouter un répertoire personnel à la liste des répertoires de recherche, le shell bash pris comme support dans ce manuel permet d'écrire :

```
export PATH=~/bin:$PATH
```

Cependant certains shells ne procèdent pas à l'expansion du ~, il faudra alors écrire :

```
PATH=~/bin:$PATH export PATH
```

6.1.7 Étude(s) de cas

Un autre LaTeX

Lorsqu'on utilise ce merveilleux et monstrueux logiciel qu'est LaTeX pour produire des documents, on fait appel à deux programmes distincts :

– on lance le visualiseur de document `xdvi` avec la commande :
```
xdvi madoc.dvi &
```

– puis, à intervalles réguliers, on lance la compilation du source :
```
latex madoc.tex
```

Une fois la compilation terminée, il faut rendre active la fenêtre de logiciel xdvi pour visualiser les changements dans le document. Or, en lisant la documentation de `xdvi` on peut tomber sur :

```
SIGNALS
   When xdvi receives a USR1 signal, it rereads the dvi file.
```

L'idée qui germe donc dans l'esprit du lecteur d'esprit vif est la suivante : il serait intéressant de pouvoir envoyer automatiquement ce signal au programme xdvi à chaque compilation avec LaTeX, et ce pour éviter d'avoir à cliquer sur la fenêtre du visualiseur pour la rendre active... Voici l'algorithme proposé :

```
Début
  lancer la compilation du document D.tex
  trouver le pid p du xdvi qui visualise D.dvi
  Si p existe  Alors
    envoyer le signal USR1 au programme de pid p
  FinSi
Fin
```

Et voici comment on peut implémenter cet algorithme en shell :

```
function latex()
{
  # on enlève l'éventuelle extension
  j=${1%.*}
  # appel du  vrai  LaTeX
  command latex $j
  # récupération du pid de(s) xdvi(s)
  pids=$(ps -ef | grep xdvi.bin | grep $j.dvi \
                | awk '{print $2}')
  # envoi du signal USR1 pour chacun d'eux
  for p in $pids ; do
    kill -USR1 $p
  done
}
```

On notera que dans la fonction, on fait appel à `command latex` pour éviter un appel récursif à la fonction. Le reste de la fonction fait appel à des outils (`ps`, `grep`, `awk`, boucle `for` du shell) présentés dans les chapitres précédents (en particulier les chapitres 2 et 3).

Il est intéressant de noter que la commande `fuser` permettant d'obtenir la liste des processus utilisant un fichier particulier, nous offre une autre solution au problème. En effet si on tape :

```
$ fuser guide-unix.dvi
guide-unix.dvi:       2587
$
```

alors le système nous indique que le processus de pid 2587 a ouvert le fichier en question. La version de `fuser` de votre serviteur permet en outre à l'aide de l'option `-k` d'envoyer un signal particulier au(x) processus détecté(s). Ainsi :

```
$ fuser -k -USR1 guide-unix.dvi
$
```

permet d'envoyer le signal `USR1` aux processus utilisant `guide-unix.dvi`...

Une poubelle

Sous UNIX, l'utilisateur n'a généralement pas la possibilité de récupérer un fichier qu'il a effacé. La solution adoptée par les systèmes d'exploitation consiste à proposer à l'utilisateur une « corbeille » dans laquelle on déplace les fichiers plutôt que de les effacer réellement [1]. Lorsque la corbeille est vidée, les fichiers sont effectivement effacés. Il est tout à fait possible d'envisager de configurer l'environnement de travail pour mettre en place un tel mécanisme. Voici quelques ébauches que le lecteur pourra lui-même faire évoluer selon ses besoins :

corbeille

```
CORBEILLE=~/.corbeille
#
# nouvelle commande rm
#
function rm()
{
  # si la corbeille n'existe pas, on la crée
  if [ ! -d $CORBEILLE ]; then
     mkdir $CORBEILLE
  fi
  # on déplace tous les fichiers de la ligne de commande
  # dans la corbeille
  mv -f "$@" $CORBEILLE
}
```

Cette nouvelle commande `rm` peut être utilisée aussi bien pour un ensemble de fichiers que pour un ensemble de répertoires. On peut également créer une commande de vidange de la corbeille :

corbeille

```
function vidange()
{
  rm -rf $CORBEILLE
}
```

Cette nouvelle version de la commande `rm` est limitée par le fait que tous les fichiers et répertoires effacés sont déplacés dans le même répertoire (celui faisant office de corbeille). Par conséquent :

1. Ceci sans parler des sauvegardes journalières des données que tout bon administrateur système doit assurer...

- il n'y aucune trace dans la corbeille du répertoire dans lequel était initialement stocké le fichier ou répertoire effacé ;
- si on efface deux fichiers portant le même nom, seul le dernier sera sauvegardé dans la poubelle ;
- si on efface un répertoire portant le nom d'un fichier déjà effacé le shell nous enverra sur les roses avec le message :
  `mv: cannot overwrite non-directory ‘ /.corbeille/`⟨*dir*⟩`’ with directory ‘`⟨*dir*⟩`/’`

Enfin, cette fonction ne traite pas le cas des options éventuellement passées en ligne de commande. Pour éviter d'écraser un fichier déjà dans la corbeille, on peut utiliser l'option `backup` de la commande `mv` du projet GNU :

```
mv -f --backup=numbered $* $CORBEILLE
```

Cette option permettera de numéroter les fichiers dupliqués en faisant appel à la syntaxe :

⟨*nom du fichier*⟩`.~`⟨*numéro*⟩`~`

Par exemple, toutes les sauvegardes du fichier `bidule.txt`, seront nommées :

```
bidule.txt bidule.txt.~1~ bidule.txt.~2~ bidule.txt.~3~ ...
```

ə

6.2 Avec vi

`vi` — prononcer vi aïe — est l'éditeur de référence des systèmes UNIX; c'est en quelque sorte le « dénominateur commun » des éditeurs de texte dans la mesure où la majorité des systèmes UNIX possède au moins cet éditeur. C'est une des raisons pour lesquelles il est bon de connaître le principe de son fonctionnement. Le concept le plus important de `vi` est qu'il fonctionne selon deux modes :

Mode commande : dans ce mode on peut passer des ordres d'édition à `vi` (effacement d'une ligne, déplacement du curseur, etc.).

Mode insertion : dans ce mode on peut saisir du texte.

Ce principe quelque peu archaïque[2] peut surprendre au premier abord, c'est pourquoi nous avons dressé une liste des « pourquoi » utiliser `vi` :

- parce que dans certaines situations, seul `vi` est disponible : par exemple lorsqu'on se connecte sur une machine distante pour modifier un fichier ;
- parce que c'est « puissant », en effet derrière un aspect simpliste[3] et rudimentaire[4] se cache une multitude de fonctionnalités surprenantes pour l'édition de texte ;
- parce que c'est « léger », donc le lancement est très rapide, et peu de ressources mémoire ou calcul sont grevées ;
- et enfin parce que ça fait snob `;-)`[5]

2. Ne pas taper svp. Je m'adresse aux djeuns qui ne connaissent que des éditeurs où il y a de jolis onglets/boutons/menus/icones/...

3. cf. note 2.

4. cf. note 2.

5. Parmi les querelles de clochers du monde UNIX, le débat Emacs *versus* vi est très célèbre. Cette remarque douteuse n'arrange d'ailleurs pas les choses.

Après cette introduction, voici quelques commandes de `vi` à connaître, ces commandes suffisent à survivre dans la majorité des cas :

`i` passe en mode insertion (on peut alors « taper » du texte) ;

`ESC` revient en mode commande (quitte le mode insertion) ;

`:w` sauve le fichier ;

`:q` quitte vi. Cette commande et la précédente peuvent être utilisées en même temps (`:wq`) et dans le cas où l'on doit forcer la sauvegarde ou la sortie il est nécessaire d'ajouter un `!`, par exemple (`:q!`), pour quitter `vi` sans sauver même si le fichier a été modifié ;

`dd` ou ⟨*n*⟩ `dd` efface une ou ⟨*n*⟩ lignes ;

`a` passe en mode insertion après le curseur ;

`^` ou `$` va en début ou fin de ligne (ces deux commandes rappellent la syntaxe des expressions régulières) ;

`/` ⟨*motif*⟩ cherche la chaîne ⟨*motif*⟩ dans le texte ;

`n` cherche l'occurrence suivante ;

`.` répète la dernière action ;

`cw` ⟨*texte*⟩ `ESC` remplace par ⟨*texte*⟩ la portion comprise entre le curseur et la fin du mot courant (*change word*) ;

`w` ou `b` avance ou recule d'un mot ;

`o` ou `O` insère une ligne vierge et commence l'insertion en dessous ou au dessus du curseur ;

`Y` ou ⟨*n*⟩ `Y` copie une ou ⟨*n*⟩ lignes (*yank*)

`p` ou `P` colle le texte copié (ou coupé) par n'importe quelle commande qui copie (par exemple `Y`) ou qui coupe (par exemple `d`) en dessous ou au dessus de curseur ;

Voici un exemple de « rechercher/remplacer » dans `vi` [6] :

```
$ vi bidule
...
$
```

Puis une fois dans l'éditeur taper les commandes suivantes pour remplacer la chaîne « `machin` » par la chaîne « `chose` » :

1. `/machin` Entrée ⟵ cherche la première occurrence
2. `cw chose` Esc ⟵ la remplace
3. `n` ⟵ cherche la suivante
4. `.` ⟵ la remplace à nouveau
5. répéter à partir de 3 tant que c'est nécessaire.

Le même remplacment peut se faire en mode commande en tapant :

```
:%s/machin/chose/g
```

6. Accrochez-vous...

Sur certains systèmes, le vi disponible est vim pour vi *improved* (soit vi « amélioré », ce qui en dit long sur l'état du premier[7]). Cette version est moins légère que la version originale mais apporte beaucoup de fonctionnalités et des facilités pour sa configuration.

6.3 Avec Emacs[8]

Présenter Emacs en quelques pages est une tâche difficile. Donner envie au lecteur d'utiliser ce logiciel extraordinaire en est une autre. La première chose que l'on peut dire au sujet de cet « éditeur de texte à tout faire » est que son nom est une contraction de *Editing MACroS* et que son origine remonte à 1976 date à laquelle Richard M. Stallman, fondateur du projet GNU, commença le développement d'Emacs.

Emacs fait partie des « stars » des logiciels libres, dans le sens où c'est sans doute un modèle de logiciel *ouvert*. À l'instar de LaTeX, les utilisateurs enthousiastes ont montré que grâce à cette ouverture on pouvait ajouter de nombreuses fonctionnalités à ce logiciel fascinant. Emacs est essentiellement utilisé pour les facilités qu'il apporte à l'édition de fichier source au sens large du terme ; l'auteur — qui en a une expérience somme toute assez maigre — a pu éditer les fichiers suivants : scripts shell, langage C, C++, Lisp, Pascal, Perl, PostScript, script Matlab, fichiers LaTeX et HTML, des Makefiles, script de Persistence of Vision, et tout ceci en bénéficiant de l'insertion de commentaires, de l'indentation automatique et de la mise en évidence des mots clefs en couleur, entre autres choses. Mais bien que son rôle premier soit l'édition de fichiers texte, il est aussi capable de jouer aux tours de hanoï (`M-x hanoi`), au morpion (`M-x gomoku`), à Tétris (`M-x tetris`), de vous psychanalyser (`M-x doctor`), de lire les mails et les news, de transférer des fichiers par ftp, de compulser un calendrier (`M-x calendar`) permettant notamment de savoir que la date du jour (4 juillet 2000) correspondant au septidi de Messidor de l'année 209 de la Révolution française, éditer des fichiers en hexadécimal (`M-x hexl-mode`), j'en passe et des pires...

Cette section est une introduction aux concepts fondamentaux d'Emacs ainsi qu'aux commandes à connaître pour y prendre goût. Nous donnons en fin de ce paragraphe quelques idées pour chercher de l'aide pour les explorations futures.

6.3.1 Concepts de base

Pour commencer, il est important de noter qu'Emacs a la particularité de pouvoir s'exécuter aussi bien en mode texte dans un terminal qu'en mode graphique (ou fenêtre) en s'appuyant sur un serveur X. Dans l'un ou l'autre des modes, on pourra utiliser les commandes présentées dans cette section.

Les objets graphiques qui composent l'éditeur Emacs sont les suivants :

Frame : Emacs est multi-fenêtre, et nomme chacune d'elles *frame*[9]. La figure 6.1 montre une session avec deux *frames* ;

7. cf. note 5.

8. Cette section est beaucoup plus longue que celle consacrée à l'éditeur vi. Je suis un utilisateur quotidien et enthousiaste d'Emacs, c'est pourquoi de manière totalement subjective et partiale, je m'étendrais quelque peu sur le sujet...

9. Nous conserverons ici à dessein le jargon anglais d'Emacs en utilisant une fonte *particulière*.

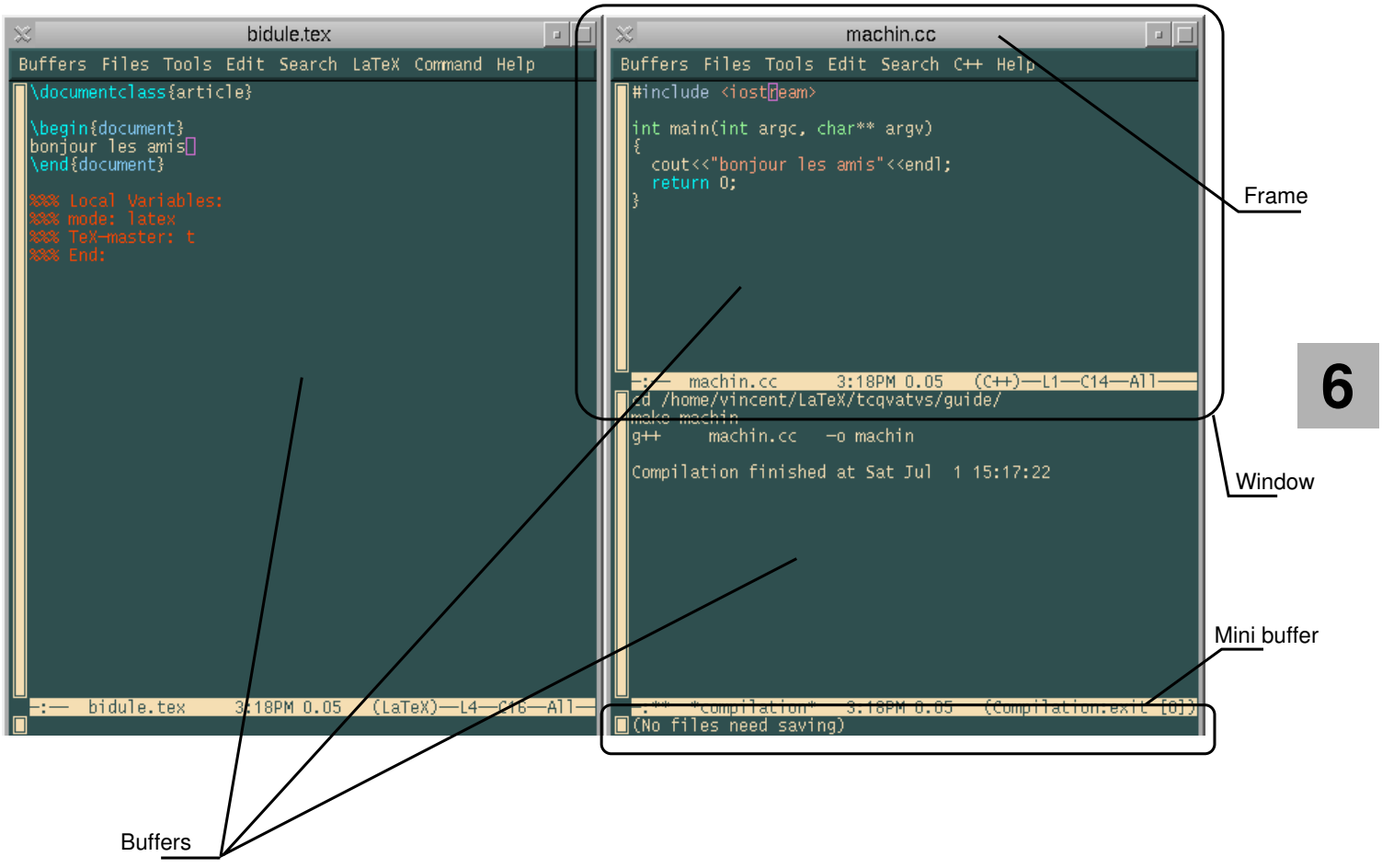


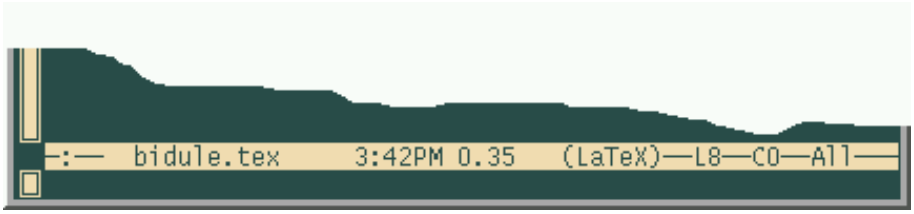
FIGURE 6.1: Emacs dans un environnement graphique

Window : les *frames* contiennent une ou plusieurs *windows*. La figure 6.1 montre trois *windows* : une dans la frame de gauche et deux dans la frame de droite ;

Buffer : les données qui sont affichées dans les *windows* sont des zones mémoires appelées *buffers*. Ces zones contiennent du texte à éditer. Un *buffer* peut être sauvegardé sur disque, créé à partir d'un fichier, etc. En fait sous Emacs, « tout est buffer » ;

Minibuffer : buffer particulier composé d'une ligne tout en bas de chaque *frame*. C'est dans cette zone qu'a lieu la majeure partie des interactions avec l'utilisateur ;

Status bar : ou barre d'état, située en bas de chaque *window*, elle indique différentes informations ; ici : le nom du fichier, l'heure, la position du curseur, le mode d'édition (cf. plus bas), etc. :



Menu : chaque *frame* dispose d'un menu permettant de lancer diverses fonctions.

Les deux autres concepts fondamentaux sont les suivants :

Commandes : toute action exécutée dans l'éditeur, quelle qu'elle soit, est le résultat de l'interprétation d'une fonction Lisp[10]. Cette fonction peut être appelée par son nom, par une combinaison de touches, ou par le biais du menu. Il faut noter que toutes les fonctions n'ont pas nécessairement de combinaison de touches ou d'entrée dans le menu ;

Mode : chaque buffer est « piloté » par Emacs dans un *mode* particulier. Le mode en question correspond à la manière dont les commandes vont être interprétées. À titre d'exemple, les fonctions d'indentation, de mise en couleur, de mise en commentaire, sont interprétées différemment selon qu'Emacs est en mode LaTeX, ou en mode C pour ne citer que ceux-là.

6.3.2 Notations utilisées

Dans les diverses documentations que l'on trouve traitant d'Emacs, et dans emacs lui-même, on retrouve les notations suivantes au sujet des combinaisons de touches :

- `C-`⟨t⟩ : pour une pression sur [Ctrl] et la touche ⟨t⟩ ;
- `M-`⟨t⟩ : désigne l'utilisation de la touche « Meta » :
  - une pression sur [Alt] et la touche ⟨t⟩, ou à défaut :
  - une pression sur [Esc] **puis** la touche ⟨t⟩.
- `ESC` : la touche [Esc] ;
- `RET` : la touche [Entrée] ;
- `SPC` : la barre d'espacement ;

Ainsi :

- `M-x` signifie [Alt] et [x] ou [Esc] puis [x] ;

10. Le Lisp — dont le nom vient de l'anglais *list processing* — est le language utilisé pour concevoir les différentes extensions d'Emacs. La configuration de celui-ci demande d'ailleurs de connaître les rudiments de ce langage.

– `C-x C-c` signifie Ctrl et x et Ctrl et c , que l'on peut obtenir en maintenant Ctrl enfoncée tout en tapant dans l'ordre x puis c .

6.3.3 Appeler une commande

À partir de l'exemple de la commande d'ouverture d'un fichier, nous allons passer en revue quelques-unes des notions importantes d'Emacs concernant l'appel des commandes. On peut appeler la commande d'ouverture de fichier (`find-file`) de quatre manières :

– par son nom à l'aide de la combinaison : `M-x find-file` ;
– à l'aide de la combinaison : `C-x C-f` ;
– à l'aide d'un éventuel raccourci clavier défini sur votre installation ;
– à l'aide de l'entrée `File/Open File` dans la barre de menu.

Le minibuffer

Dans les trois cas le lancement de cette commande donne le focus au *minibuffer*, et l'utilisateur doit entrer le nom du fichier qu'il veut ouvrir ou un nom quelconque pour créer un nouveau buffer.

`M-x find-file` Entrée `monfichier.txt` Entrée

Dans le minibuffer la touche « Espace » ( ) ou « Tab » (⟺) permet de compléter automatiquement un nom de commande ou un nom de fichier. En cas d'équivoque une pression sur ces touches crée une nouvelle *window* avec un *buffer* contenant les différentes alternatives possibles à la complétion. Par exemple si on tape :

`M-x find`

on fait apparaître une fenêtre contenant le buffer suivant :

```
Click mouse-2 on a completion to select it.
In this buffer, type RET to select the completion near point.

Possible completions are:
find-alternate-file               find-alternate-file-other-window
find-dired                        find-file
find-file-at-point                find-file-literally
find-file-other-frame             find-file-other-window
find-file-read-only               find-file-read-only-other-frame
find-file-read-only-other-window  find-function
find-function-at-point            find-function-on-key
find-function-other-frame         find-function-other-window
find-grep-dired                   find-name-dired
find-tag                          find-tag-noselect
find-tag-other-frame              find-tag-other-window
find-tag-regexp                   find-variable
find-variable-at-point            find-variable-other-frame
find-variable-other-window
```

qui montre donc toutes les commandes dont le nom commence par « `find` ». Donc en tapant les deux premiers caractères :

`M-x find` `fi` Entrée

on complète jusqu'à `find-file` et il est alors possible de saisir le nom du fichier que l'on veut charger. On notera que la complétion automatique fonctionne aussi sur les noms de fichiers. Enfin, la fenêtre d'information vous indique également que vous pouvez utiliser le bouton du milieu de votre souris[11] pour choisir le fichier parmi la sélection.

Stop !

Pour interrompre une action entreprise dans le *minibuffer*, il faut utiliser la combinaison `C-g` qui appelle la fonction `keyboard-quit`. Cette fonction est également utile pour interrompre l'exécution d'une commande.

Lorsqu'on débute sous Emacs, on a tendance à « laisser traîner » son curseur dans le *minibuffer* ; ce qui peut entraîner un comportement étrange de l'éditeur qui vous narguera par des :

```
Command attempted to use minibuffer while in minibuffer
```

dans ce cas une ou deux pressions sur `C-g` remet Emacs dans le droit chemin.

Historique

La combinaison `C-x ESC ESC` rappelle la dernière commande exécutée. Les touches ↑ et ↓ dans le *minibuffer* permettent alors de choisir et éventuellement modifier l'une des dernières commandes exécutées :

```
Redo: (find-file "~/tcqvatvs/guide/chap-configurer.tex" 1)
```

cet historique permet en outre d'avoir une première idée de la syntaxe d'appel des commandes Lisp.

Arguments et répétition

Certaines commandes d'Emacs attendent un argument numérique. On peut passer cet argument en préfixant l'appel de la commande par `C-u` ⟨*n*⟩, pour passer la valeur ⟨*n*⟩. Si la commande en question n'attend pas d'argument elle sera simplement exécutée ⟨*n*⟩ fois. Par exemple la séquence : `C-u 24 a`, insère :

```
aaaaaaaaaaaaaaaaaaaaaaaa
```

dans le *buffer* courant.

6.3.4 Manipuler les objets d'Emacs

Les objets d'Emacs— *frame*, *window*, *buffer*, etc. — peuvent être manipulés via un certain nombre de fonctions qu'il est souhaitable de connaître.

11. Si votre souris n'a qu'un bouton, c'est que vous avez un Machinetoque, changer alors d'ordinateur ; si votre souris n'a que deux boutons, vous vous êtes fait avoir par le protocole « micro logiciel », changez de souris.

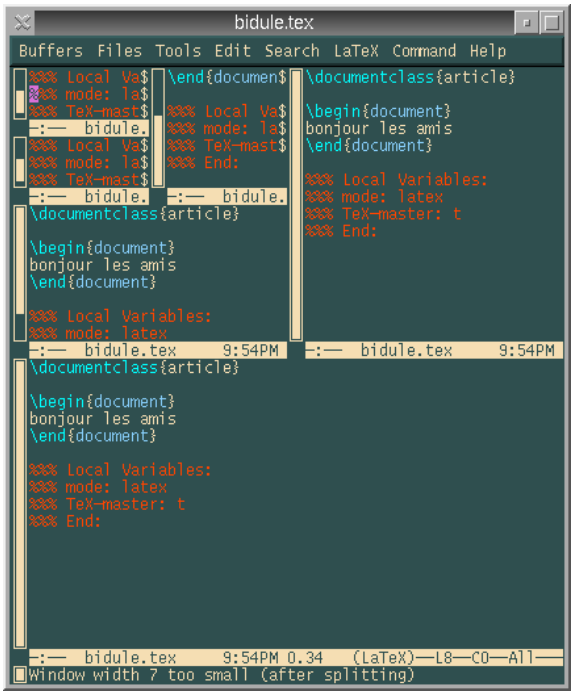


FIGURE 6.2: The Emacs quad tree

Frame les tâches courantes autour des *frames* sont la création et la destruction. On crée une nouvelle *frame* avec :

- `C-x 5 2` ou :
- `M-x make-frame-command`
- par le menu `File/Make New Frame`

on détruit celle sur laquelle on se trouve avec :

- `C-x 5 0` ou :
- `M-x delete-frame`
- en cliquant sauvagement sur le bouton « delete window » de son gestionnaire de fenêtres préféré.

Il est bien évident que les *frames* d'Emacs ne sont disponibles que dans un environnement permettant le « multi-fenêtrage » comme X window...

Window on peut créer de nouvelles fenêtres dans une *frame* à partir de la première :

- `C-x 3` « divise horizontalement » la *window* actuelle ;
- `C-x 2` « divise verticalement » la *window* actuelle ;
- `C-x 1` ne conserve que la *window* contenant le curseur

ces combinaisons de touches correspondent respectivement aux fonctions :

- `M-x split-window-horizontaly` ;
- `M-x split-window-verticaly` ;
- `M-x delete-other-windows`.

Il est particulièrement intéressant de noter qu'après une série de `C-x 2` et `C-x 3` — dans cet ordre — on obtient un *quad tree* à la mode Emacs (voir figure 6.2). Mais ne nous égarons pas dans des considérations douteuses, puisqu'on peut aussi redimensionner une fenêtre coupée en deux horizontalement (après un `C-x 2`) à

l'aide de :

- `C-x ^` ou `M-x enlarge-window` ;
- en déplaçant la *status bar* à l'aide du bouton du milieu de votre souris.

Buffer on peut « tuer » un buffer, lorsqu'on a fini d'éditer un fichier, ou simplement parce que le buffer en question n'est plus utile lors de la session, avec :

- `C-x C-k` ou
- `M-x kill-buffer`, ou :
- le menu `File/Kill Current Buffer`

► § 6.3.5 p. 161 Notez bien que si le buffer a été modifié depuis la dernière ▲sauvegarde◄, vous devrez répondre à quelques questions qui vous seront gentiment posées par le truchement du *minibuffer*. Il est également possible d'avoir la liste de tous les buffers ouverts depuis le début de la session avec :

- `C-x C-b` ou `M-x list-buffers`, ou :
- par le menu `Buffers/List All Buffers`

Une partie de cette liste est visible dans les premières entrées du menu `Buffers`.

6.3.5 Les tâches basiques

ə

Se déplacer / effacer

Malgré ce que dit le tutorial d'Emacs, qui préconise l'usage des combinaisons (`C-f`, `C-b`, `C-n` et `C-b`) pour déplacer le curseur, il semble raisonnable de penser qu'en ce début de siècle, on peut utiliser les flèches du clavier. Il est cependant intéressant de noter que ces raccourcis existent — ils sont facilement mémorisables : forward, backward, next, previous — pour les utiliser en cas d'urgence. Les autres combinaisons utiles pour se déplacer sont :

- `C-e`/`C-a` va en fin/début de ligne ;
- `C-` ← et `C-` → va un mot vers la gauche et vers la droite respectivement (on pourra également utiliser `M-f` (*forward*) et `M-b` (*backward*) ;
- `C-` ↑ et `C-` ↓ va un paragraphe vers le haut et vers le bas respectivement ;
- `C-l` centre verticalement le curseur au milieu de la *window* courante en déplaçant le texte le cas échéant ;
- `M-<` et `M->` permettent d'aller ou début et à la fin du buffer, ces fonctions sont également disponibles avec les touches Home et Fin si votre clavier en dispose ;

Pour effacer du texte, les touches « delete » et « backspace » ont l'effet escompté. On peut en outre effacer à partir du curseur jusqu'à la fin de la ligne avec `C-k`. Cette opération s'apparente à un « couper » puisque le texte effacé est stocké pour éventuellement être collé. Notez également que `M-` ⟵ efface le mot avant le curseur.

Sélectionner

Pour sélectionner une zone, il suffit de se placer au début de la zone que l'on veut traiter et appuyer sur `C-SPC`. En fonction de la configuration d'Emacs, lorsqu'on déplace le curseur, on voit apparaître la zone sélectionnée en « inverse vidéo ». La souris permet également de sélectionner une zone, bien entendu.

Copier, Couper, Coller...

Une fois la zone sélectionnée, on peut procéder aux trois opérations classiques :
- `C-w` : couper ;
- `C-y` : coller ;
- `M-w` : copier.

Le « coller » d'Emacs apporte beaucoup de souplesse, puisqu'une fois la séquence `C-y` entrée, on peut éventuellement choisir le texte à coller grâce à la combinaison `M-y` qui permet de choisir une selection parmi les zones copiées ou coupées jusqu'à présent.

Défaire, refaire

Emacs dispose d'un « undo » permettant de « remonter » jusqu'aux premières modifications apportées lors du démarrage de la session. Le « undo » s'obtient grâce à la combinaison `C-_` ou `C-x u`[12].

Une propriété intéressante du « undo » d'Emacs est qu'il peut aisément se transformer en « redo ». Pour cela il suffit de taper le fameux `C-g`. Si on peut voir le « undo » comme une « remontée vers le passé », un `C-g` permet de faire en sorte que l'utilisation du `C-_` suivant « revienne vers le futur » :

	`abcdef`	
`C-_`	`abcde`	undo
`C-_`	`abcd`	undo
`C-g`	`abcd`	glurps !
`C-_`	`abcde`	redo

Manipulation de fichiers

Les trois opérations qui sont indubitablement utiles sont :

La sauvegarde : qui sauve le contenu du *buffer* dans un fichier est obtenue grâce à :
- `C-x C-s` ou `M-x save-buffer`, ou :
- menu `File/Save Buffer`.

Le chargement : permet d'insérer le contenu d'un fichier dans un *buffer* tout neuf. Si le fichier n'existe pas, un *buffer* est créé portant le nom spécifié et la sauvegarde suivante créera le fichier sur le disque. Nous avons vu précédemment que le chargement pouvait être réalisé grâce :
- la commande `M-x find-file`, ou
- le raccourci `C-x C-f`, ou
- le menu `File/Open File`.

La mise à jour : il arrive que le fichier dont on a un buffer dans Emacs soit modifié par une source extérieure (un autre programme). Dans ce cas la « copie » dont on dispose dans le *buffer* n'est plus à jour par rapport au contenu du fichier, il faut donc utiliser la commande :
- `M-x revert-buffer` ou :
- menu `File/Revert Buffer`

qui recharge le contenu du fichier dans le buffer courant. Dans cette situation, Emacs avertit l'utilisateur par plusieurs messages dans le *minibuffer*.

12. le premier est plus simple sur un clavier français.

D'autres opérations peuvent être utiles à connaître comme le célèbre « sauver sous » (`C-x C-w` ou `M-x write-file`) qui sauve le buffer dans un fichier portant un autre nom. Est également intéressante `M-x insert-file` (ou `C-x i`) qui insère à l'endroit où est le curseur le contenu d'un fichier.

Rechercher / Remplacer

Voici les deux fonctions à connaître pour rechercher des chaînes de caractères dans un *buffer* et éventuellement les remplacer.

Juste chercher... La première est la fonction `M-x isearch-forward` dont un raccourci pratique est `C-s`. Dès que la combinaison `C-s` est pressée, Emacs attend par l'intermédiaire du *minibuffer* les lettres composant la chaîne de caractères à rechercher. À chaque lettre entrée, le curseur se déplace sur la première partie du texte correspondant à la chaîne formée jusque là. On a alors plusieurs alternatives :

- la chaîne est introuvable (message `Failing I-search`) ;
- la chaîne a été trouvée : en pressant à nouveau sur `C-s` le curseur se positionne sur l'occurrence suivante ;
- si on presse `C-g` le curseur se repositionne là où il se trouvait au début de la recherche et celle-ci s'arrête ;
- si on presse Entrée , le curseur reste dans sa position actuelle et la recherche s'arrête.

Le pendant de `C-s` est `C-r` qui effectue le même travail, mais en cherchant « vers le haut » (`M-x isearch-backward`). On peut d'ailleurs passer du « backward » au « forward » sans problème lors d'une recherche.

Chercher et remplacer La fonction `M-x query-replace`, dont le raccourci est `M-%`, permet, comme son nom l'indique, de chercher une chaîne et de la remplacer par une autre. Le *minibuffer* vous demandera donc deux chaînes pour effectuer les remplacements. Si une chaîne est trouvée, Emacs vous demandera ce que vous désirez faire, vous pourrez alors répondre :

- `y` pour remplacer ;
- `b` pour ne pas remplacer ;
- `!` pour remplacer partout ;
- `q` pour quitter ;
- `h` pour voir les autres possibilités !

Une particularité de cette commande est qu'elle peut agir uniquement sur une zone si elle est préalablement sélectionnée. Notez aussi qu'Emacs est généralement configuré pour préserver les majuscules dans le remplacement, c'est-à-dire que dans un texte où l'on demande de remplacer « `a`« par « `b` », « `A` » sera remplacé par « `B` ».

Il existe une version améliorée de la commande `M-x search-replace` : la commande `query-replace-regexp` (dont le raccourci est `M-C-%`) qui permet de chercher et remplacer des chaînes spécifiées à partir ▲d'expressions régulières◀. Par exemple :

► § 3.5 p. 77

`M-C-%` Entrée `^b.*e$` Entrée `hoplà` Entrée

dans le texte de gauche, donne le texte de droite :

bcde	hoplà
bbd	bbd
cde	cde
bqmlksdjfe	hoplà

Nous rappelons que `^b.*e$` signifie : une ligne qui commence par « `b` » suivi d'une série de caractères et finissant par « `e` ».

Indenter / Commenter

Un des plaisirs que l'on peut tirer de l'utilisation de ce gros patapouf d'Emacs[13] est l'indentation automatique des sources et l'insertion de commentaires[14]. Il n'y a rien de difficile à apprendre dans ce paragraphe puisque tout est fait automatiquement en fonction du *mode* utilisé. Prenons l'exemple d'un fichier C++. À chaque pression sur la touche ⇆ le curseur se mettra à la « bonne place », c'est-à-dire en fonction de l'imbrication des diverses boucles et autres blocs. Si le curseur ne se met pas à une position correcte c'est que le source comporte une erreur de syntaxe. Il y a trois commandes à connaître :

- `M-x indent-region` qui indente la zone sélectionnée en fonction du mode utilisé ;
- `M-x comment-region` qui commente la région sélectionnée en fonction de la syntaxe du langage correspondant au mode d'Emacs pour le buffer ;
- `C-u M-x comment-region` qui retire les commentaires de la zone sélectionnée.

Lorsqu'Emacs est en mode langage C, on peut commenter une région préalablement sélectionnée avec le raccourci `C-c C-c`. Pour décommenter on pourra utiliser le raccourci `C-u C-c C-c`.

6.3.6 Emacs et les Makefiles

La commande `M-x compile` permet d'exécuter une commande dans un shell fils d'Emacs. Ainsi lorsqu'on lance cette fonction, le *minibuffer* attend une commande à exécuter :

```
Compile command: make
```

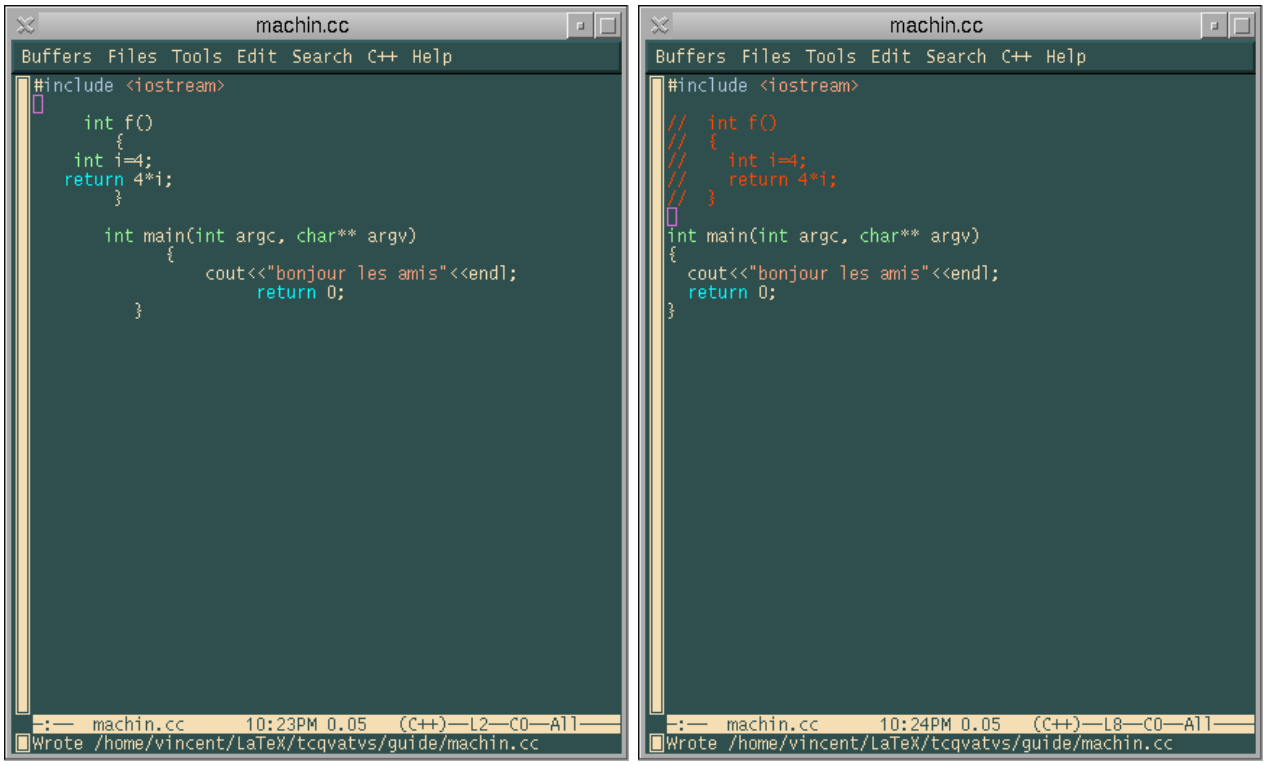
On peut alors remplacer `make` par la commande de son choix, par exemple `echo salut les coins coins` et presser la touche Entrée. La commande est alors exécutée dans un shell et le résultat est affiché dans un *buffer* nommé `*compilation*` :

```
cd /home/vincent/LaTeX/tcqvatvs/guide/
echo salut les coinscoins
salut les coinscoins

Compilation finished at Wed Jul  5 23:17:32
```

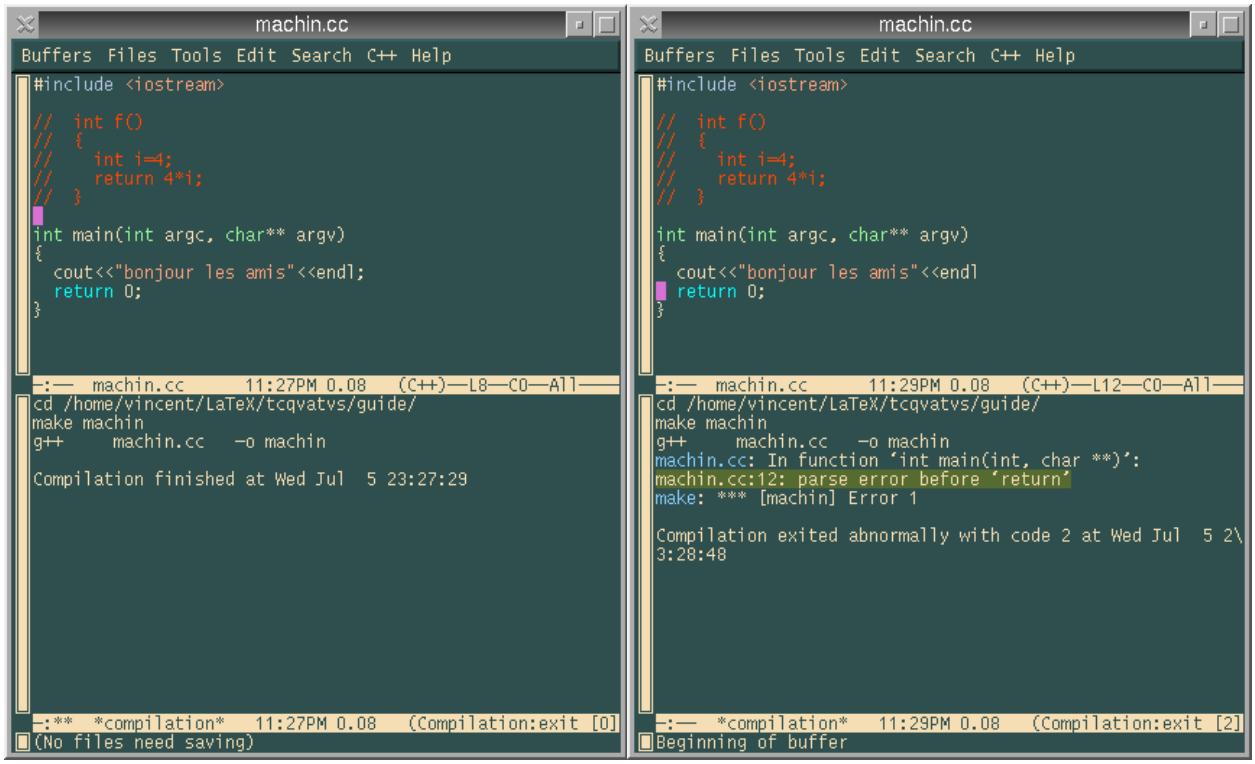
Ce qui n'est certes pas passionnant. Cependant si en lieu et place d'un bonjour aux canards, on invoque make, cela devient plus intéressant. En effet, en éditant un fichier C (par exemple), on peut lancer la compilation dans Emacs. L'éditeur devient alors un environnement de développement qui se suffit à lui même. Le buffer `*compilation*` peut en effet être utilisé pour cerner les erreurs de compilation. La

13. Des mauvaises langues disent que Emacs est l'acronyme de « Emacs makes a computer slow ».
14. Notez bien que d'autres éditeurs comme vim pour ne pas le citer, sont également capables de ce genre de prouesses.



(a) (b)

FIGURE 6.3: Commentaires et indentation automatique : pour passer de la figure (a) à la figure (b), on a appelé la fonction `M-x indent-region` après avoir sélectionné le buffer entier ; puis après avoir sélectionné le corps de la fonction `f()`, on a fait appel à la commande `M-x comment-region` pour commenter son code.



(a) (b)

FIGURE 6.4: Compilation dans Emacs

figure 6.4b montre le contenu de ce *buffer* lorsque le source contient une erreur. Un clic de souris, avec le bouton du milieu sur la ligne contenant l'erreur positionne le curseur à la ligne du programme source correspondant.

6.3.7 Personnaliser

Macros

Si on reprend l'étymologie du mot Emacs on pense à « macro ». Les macros sont des séquences de touches que l'utilisateur peut enregistrer et réutiliser plus tard. Le principe d'enregistrement d'une macro est le suivant :

1. commencer l'enregistrement ;
2. tapoter sur le clavier la séquence de touches qu'on veut mémoriser ;
3. finir l'enregistrement ;
4. donner un nom à la macro définie ;
5. stocker éventuellement le « code » de la macro.

Le commencement et la fin de l'enregistrement sont stipulés par les commandes `M-x start-kbd-macro` et `M-x end-kbd-macro` dont les raccourcis élégants sont « `C-x (` » et « `C-x )` ». Toute touche tapée entre le début et la fin de l'enregistrement est stockée dans la macro courante. On peut ensuite donner un nom à la macro définie avec `M-x name-last-kbd-macro`, et enfin insérer le code Lisp de cette macro dans le fichier de démarrage à l'aide de la commande `M-x insert-named-kbd-macro`. Voyons un exemple : c'est une pratique courante en langage C, d'insérer dans les fichiers include la directive de compilation suivante :

```
#ifndef __BIDULE_H
#define __BIDULE_H
```

(...contenu du fichier include...)

```
#endif
```

Pour éviter d'insérer systématiquement ces directives, à chaque création d'un fichier include, on peut envisager de créer une macro de la manière suivante :

1. top départ : `C-x (` ←——— à partir d'ici tout ce qu'on tape est enregistré
2. enregistrement de la macro :
   - `M-x <` ←——— aller au début du buffer
   - `#ifndef ___H` Entrée
   - `#define ___H` Entrée
   - `M-x >` ←——— aller à la fin du buffer
   - Entrée `#endif`
3. `C-x )` : fin d'enregistrement.
4. donner un nom à la macro avec `M-x name-last-kbd-macro` : dans le *minibuffer* on répond par exemple `insert-ifndef` qui est ma foi un nom très seyant pour cette macro.

On peut maintenant utiliser cette macro comme une autre commande. Il manque à notre exemple l'association d'une combinaison de touche à cette macro et la sauvegarde de cette macro pour les sessions suivantes ; c'est l'objet des paragraphes suivants.

Raccourcis clavier

On peut associer une combinaison de touches à une fonction en utilisant la commande `M-x global-set-key`. Cette commande demande à l'utilisateur la combinaison puis la fonction à y associer. On pourra par exemple associer la combinaison de touches `F5` (ou `F10`, ou `C-x C-...`)[15] à la fonction qui commente une région :

`M-x global-set-key` Entrée F5 Entrée `comment-region` Entrée

Le fichier `.emacs`

On peut stocker des configurations personnelles — telles que des raccourcis clavier par exemple — dans un fichier qui est chargé à chaque lancement d'Emacs. Ce fichier doit porter le nom `.emacs` et doit résider dans le répertoire privé de l'utilisateur. La « petite » difficulté réside dans le fait que le contenu de ce fichier doit être en Lisp. La composition d'un fichier `.emacs` dépasse le cadre de ce manuel[16] c'est pourquoi nous donnerons ici quelques possibilités :

```
; pour preserver les liens lors d'une sauvegarde :
(setq backup-by-copying-when-linked t)
; remplace une selection par ce que l'on tape
```

15. On peut utiliser la combinaison `C-h k` pour vérifier si la combinaison choisie n'est pas déjà utilisée par Emacs.

16. Et surtout les compétences de l'auteur...

```
(delete-selection-mode t)
; numero de colonne
(setq column-number-mode t)
; bip visible et non sonore
(setq visible-bell t)
```

Notez tout de même, qu'un moyen possible pour construire son propre fichier `.emacs` peut consister à :

1. exécuter la commande que l'on veut sauvegarder dans le `.emacs`, par exemple `M-x standard-display-european` pour afficher les caractères accentués ;
2. faire apparaître le code Lisp d'appel dans le minibuffer avec l'historique `C-x ESC ESC` ;
3. copier ce code dans son `.emacs` : `(standard-display-european nil)` en utilisant le mécanisme de copier/coller d'Emacs ;
4. recharger le `.emacs` avec la commande `M-x load-file`

Cette méthode est bien entendu limitée à l'ajout de fonctionnalités simples.

Francisation

Pour forcer Emacs à afficher les caractères accentués, il peut être nécessaire de lancer la commande `M-x standard-display-european`. Ainsi la phrase qui par défaut s'affichait comme ceci :

```
cet apr\350s midi d'\351t\351 \340 la campagne...
```

s'affichera comme cela :

```
cet après midi d'été à la campagne...
```

Modes

Comme on l'a entr'aperçu un peu plus haut, chaque *buffer* est piloté par un *mode* particulier qui gère l'indentation, la mise en couleur, etc. Ce mode est indiqué dans la barre d'état.

À chaque chargement de fichier, Emacs se place dans un mode d'édition particulier en fonction de l'extension du fichier. Cette reconnaissance peut échouer dans certains cas. On peut alors basculer manuellement dans le mode désiré avec les commandes `M-x c++-mode`, `sh-mode`, `latex-mode`, etc.

On peut également charger le mode désiré de manière automatique. Un exemple : lors du chargement d'un fichier dont l'extension est `.h`, le mode C est lancé ; si l'on préfère le mode C++, il faudra insérer dans son `.emacs` :

```
(setq auto-mode-alist
   (cons '("\\.h\\'" . c++-mode) auto-mode-alist))
```

Certains modes sont dits *mineurs* par opposition aux modes *majeurs* (*minor and major modes*) dans la mesure où ils sont chargés en plus du mode majeur. C'est le cas par exemple des modes :

– `M-x outline-minor-mode` qui permet d'utiliser un mode « plan » dans les fichiers LaTeX offrant la possibilité de masquer certaines sections (paragraphes, sous paragraphes, etc.) ;

– `M-x auto-fill-mode` qui permet de passer à la ligne automatiquement dans les *buffers* en fonction d'un nombre de caractères défini pour chaque ligne.

On peut charger ces modes mineurs grâces aux commandes introduites ci-dessus, ou automatiquement en utilisant la notion de *hook*. Un hook associé à un mode donné est, entre autres, un moyen d'exécuter des commandes lorsque le mode en question est chargé. Les deux lignes suivantes dans votre `.emacs` :

```
(add-hook 'LaTeX-mode-hook
   (function ( lambda() (auto-fill-mode))))
(add-hook 'LaTeX-mode-hook
   (function ( lambda() (outline-minor-mode))))
```

chargent automatiquement les modes `outline` et `auto-fill` lorsque un fichier LaTeX est chargé avec le `LaTeX-mode`.

Le « bureau »

Lors d'une session Emacs, on ouvre une quantité de *buffers* plus ou moins importante. Il est agréable de pouvoir recharger ces *buffers* lors de la session suivante. Pour ce faire il faut procéder en deux temps :

1. ajouter la ligne :

   ```
   (desktop-read)          ; chargement du bureau
   ```

   dans son `.emacs` (le caractère « `;` » est le commentaire du Lisp) ;
2. exécuter la commande `M-x desktop-save` (une seule fois suffit). Précisez que vous voulez sauver votre bureau dans votre répertoire racine, et le tour est joué.

À chaque lancement d'Emacs, vos *buffers* seront chargés. Si vous êtes curieux vous pouvez jeter un œil dans le fichier `.emacs.desktop` qui contient une description des fichiers à charger. Notez que la sauvegarde du bureau ne tient pas compte des *frames*.

6.3.8 À l'aide

La touche magique pour obtenir de l'aide à partir d'emacs est `C-h`. Cette touche est un préfixe aisément mémorisable qui, associé avec d'autres combinaisons fournit les informations suivantes :

– `C-h C-k` donne la signification de la combinaison de touches que vous taperez (k pour *key*) ; c'est aussi utile pour savoir si une combinaison de touche est active ;
– `C-h C-f` décrit la fonction dont vous donnerez le nom ;
– `C-h C-t` lance le très instructif *tutorial* d'Emacs ;
– `C-h C-i` lance l'ensemble des manuels au format ◂info◂ installés sur votre ordinateur ; il y a bien sûr le manuel d'Emacs.

► § 7.3 p. 186

6.4 Avec Xwindow

Mais qu'est-ce donc que cet étrange X, ou X window ou Xlib ? Derrière ce joli nom se cachent plusieurs concepts réunis en un seul : une interface graphique, un

protocole réseau et une bibliothèque de développement ! On peut voir X comme la couche intermédiaire entre *le matériel* et les applications *graphiques*. Pour l'instant on peut se contenter de dire que la couche X est la couche nécessaire pour pouvoir utiliser UNIX en mode graphique, c'est-à-dire dans un environnement multi-fenêtres, avec des jolis menus, la souris et tout.

6.4.1 Principe général

X est basé sur le principe de *client-serveur*. Les rôles de l'un et l'autre des acteurs sont les suivants :

Serveur : c'est le logiciel qui a pour but de piloter la carte graphique et l'écran de la machine destinée à afficher, ainsi que répondre au mouvement de la souris et aux pressions des touches du clavier. Il faut qu'il y ait un serveur X en attente sur une machine pour pouvoir accéder à la couche graphique.

Le client : c'est l'*application* qui après s'être *connectée* au serveur, l'utilise pour effectuer entre autres les affichages. Par exemple, si vous utilisez un navigateur `pdf` ou un visualiseur `ps` pour consulter ce document, les affichages sont gérés par le serveur X, les évènements de la souris et du clavier sont interceptés par le serveur et gérés par le client par le truchement d'une file d'attente d'événements.

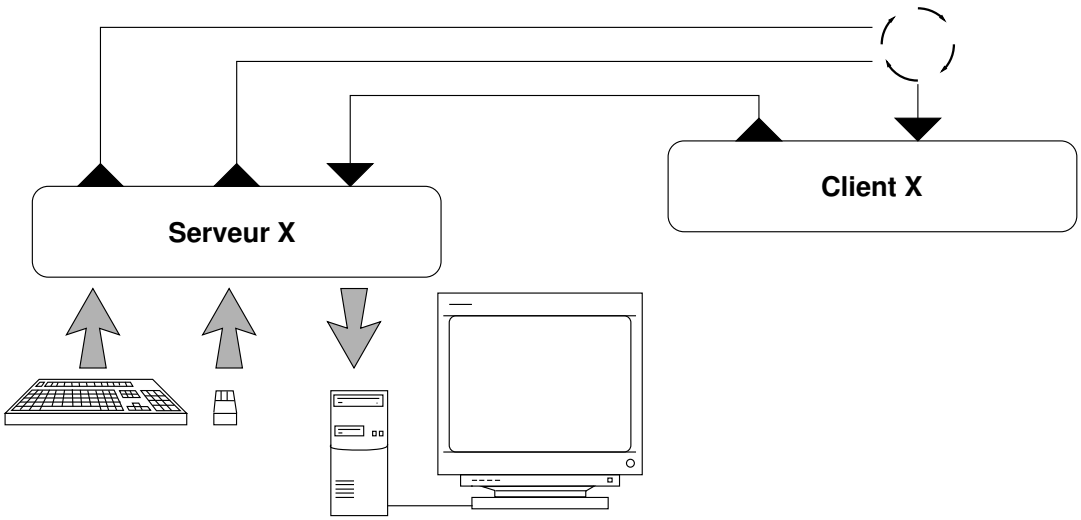


FIGURE 6.5: Principe de fonctionnement du protocole X : les requêtes sont issues du matériel (clavier/souris). Elles sont interceptées par le serveur, et sont gérées par le client par l'intermédiaire d'une file d'attente d'évènements. Le traitement de ces requêtes correspond à un affichage particulier.

En s'appuyant sur la figure 6.5 voyons quel est le principe du mode client-serveur de X :

1. au lancement, le client (c'est-à-dire l'application) va se connecter au serveur ; le résultat de la connexion consiste en l'établissement d'un canal de communication entre l'application et les périphériques d'affichage et de saisie de la machine pilotée par le serveur ; ce canal en jargon X est appelé un *display* ;

2. une fois connecté, le client va demander au serveur d'afficher un certain nombre de choses, qui correspondent à l'application elle-même ;
3. le client se met alors en attente d'évènements ;
4. lorsque l'utilisateur presse une touche de clavier ou manipule la souris, le serveur avertit le client de la position de la souris, du bouton qui a été pressé, etc. Le client réagit alors en fonction de ces informations et envoie un ordre au serveur (par exemple si la zone où survient un clic de souris est un bouton, le client va demander au serveur d'afficher un bouton enfoncé).

6.4.2 Les différentes couches

Lorsqu'on est face à un environnement graphique sous UNIX, il faut distinguer plusieurs couches différentes :

X : c'est la couche de base, c'est *in fine* toujours X qui dessine à l'écran ; comme on peut le constater à la figure 6.6a page ci-contre, une application est exécutée dans une fenêtre qui n'est qu'un rectangle à l'écran ;

Le gestionnaire de fenêtres : c'est le programme qui « habille » les fenêtres avec des barres de titres, ascenseurs et autres fonctionnalités qui permettent à l'utilisateur de les redimensionner, déplacer, iconifier, etc. Chacun de ces programmes (appelés en anglais *window managers*) apporte un « look and feel » particulier à l'environnement : l'affichage (le look) et la manière de manipuler les fenêtres (le feel) dépendent du gestionnaire de fenêtres. Les plus connus sont fvwm (figure 6.6b), windowmaker (figure 6.6d), afterstep (figure 6.6c), mwm ;

Les widgets : ce sont les différents objets qui composent une application (boutons, ascenseurs, menus, boîtes de dialogue, etc.). Ces objets sont construits par le biais de bibliothèques dont les plus connues sont Xt, Xaw, Gtk, Qt, Motif, lequelles sont bâties à partir des bibliothèques X de base ;

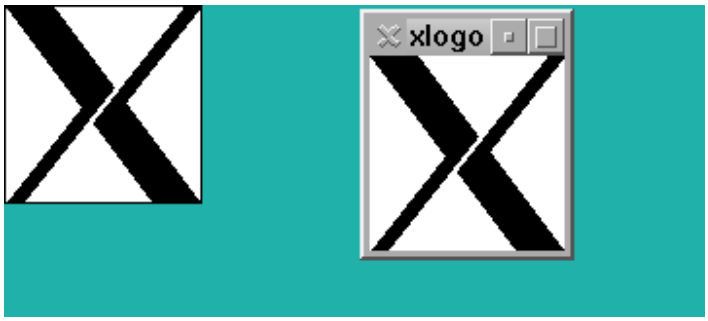
Les environnements intégrés : il y a quelques années, devant la multitude de choix possibles quant aux gestionnaires de fenêtres et la relative difficulté qu'il y a à configurer l'environnement graphique, plusieurs vendeurs d'UNIX (Sun, HP, IBM) ont décidé de créer le *Common Desktop Environnement* (dit CDE) destiné à homogénéiser l'interface graphique des systèmes UNIX. Dans le monde des logiciels libres, l'idée a également séduit et deux environnements intégrés sont nés : Gnome et Kde. Ces environnements ne seront pas traités dans ce manuel, ceci dans un souci d'ouverture. Nous préférons en effet présenter les couches inférieures pour une meilleure compréhension globale [17].

6.4.3 Comprendre le démarrage

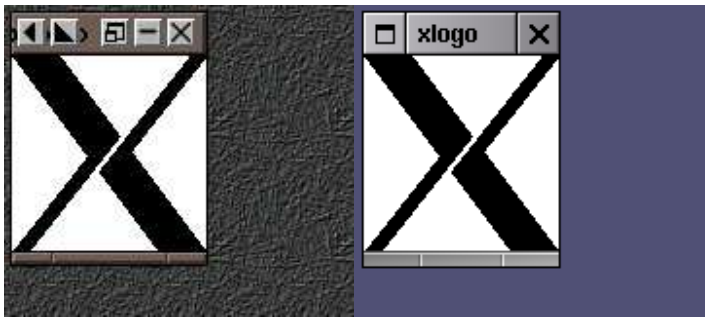
Un système UNIX peut passer en mode graphique par deux moyens :

1. directement au démarrage, dans ce cas la fenêtre de login est une fenêtre graphique et le fichier `~/.xsession` de l'utilisateur est exécuté ;

17. Aussi parce qu'à mon humble avis, bien que ces environnements permettent au novice de s'initier à UNIX, ils tendent à reproduire ce qu'on reproche à Windows : à savoir la dissimulation des informations...



(a) Xlogo tout nu (b) Avec fvwm



(c) Avec afterstep (d) Avec windowmaker

FIGURE 6.6: Différents gestionnaires de fenêtres

2. manuellement : l'utilisateur se connecte sur une console texte, et exécute un utilitaire généralement appelé `startx` ou quelque chose d'approchant, qui permet de lancer l'environnement graphique ; dans ce cas précis c'est le fichier `~/.xinitrc` de l'utilisateur qui est exécuté.

Il est souvent suffisant de faire en sorte que le fichier `~/.xsession` soit un lien symbolique sur le fichier `~/.xinitrc`. Attention, ce dernier doit être exécutable. Voici un exemple de fichier de démarrage :

`~/.xinitrc`

```
#!/bin/sh
# .xinitrc
# pas de beep
xset b off
# fond
xsetroot -solid lightseagreen
# un terminal pour commencer
xterm +ls &
# gestionnaire de fenêtres
exec fvwm2
```

On comprend qu'au lancement de X, on va successivement, imposer un bip silencieux, donner une couleur à la fenêtre racine[18], lancer un terminal X, et finalement lancer le gestionnaire de fenêtre.

Dans un tel fichier, le dernier programme — ici fvwm2 — ne doit pas être lancé en tâche de fond, et la terminaison de celui-ci implique la fin de la session X.

18. La fenêtre appelée racine est la fenêtre constituant le « bureau » de l'environnement, elle ne peut être déplacée ou iconifiée, et contient toutes les autres.

Le démarrage de X sur votre système n'est peut être pas tout à fait identique à ce qui est décrit plus haut. Pour savoir comment ça se passe chez vous, jouez les Hercule Poirot, et vous pourrez tomber, comme sur le système de l'auteur, sur un fichier `/etc/X11/xdm/Xsession` dont voici la fin :

/etc/X11/xdm/Xsession

```
 if [ -x "$HOME/.xsession" ]; then
   exec "$HOME/.xsession"
elif [ -x "$HOME/.Xclients" ]; then
   exec "$HOME/.Xclients"
elif [ -x /etc/X11/xinit/Xclients ]; then
   exec /etc/X11/xinit/Xclients
else
   exec xsm
fi
```

extrait que vous êtes à même de comprendre si vous avez lu attentivement le chapitre 5 !

► § 2.4.1 p. 51

Nous donnons ici à nouveau et à titre d'information, l'arborescence des ▲processus◄ correspondant à une session X :

```
PID PPID CMD
324    1 [xdm]
334  324  \_ /etc/X11/X
335  324  \_ [xdm]
346  335      \_ [.xsession]
360  346          \_ fvwm2 -s
448  360              \_xterm -ls
451  448              |  \_-bash
636  451              |    \_ ps --forest -eo pid,ppid,cmd
576  360              \_ /usr/X11R6/lib/X11/fvwm2/FvwmAuto
577  360              \_ /usr/X11R6/lib/X11/fvwm2/FvwmPager
```

6.4.4 X et le réseau

Un des aspects particulièrement souple de X est sa capacité à faire communiquer le client et le serveur à travers un réseau. En d'autres termes, il est possible pour une machine distante exécutant une application graphique (client X), d'envoyer les affichages sur la machine que vous avez devant les yeux — c'est-à-dire utiliser le serveur X local. Cette idée est illustrée à la figure 6.7 page suivante.

La variable d'environnement `DISPLAY`

Lorsqu'une application X démarre, elle cherche à se connecter au serveur spécifié par la variable `DISPLAY` — sauf si l'application a été programmée pour se connecter à un serveur spécifique. La variable `DISPLAY` a la forme suivante :

⟨*machine*⟩:⟨*numéro_display*⟩.⟨*numéro_écran*⟩

⟨*machine*⟩ est le nom de la machine. Selon le contexte, cela peut être :
- le nom de la machine seul : `mekanik` ;
- le nom complet de la machine : `mekanik.zeuhl.org` ;

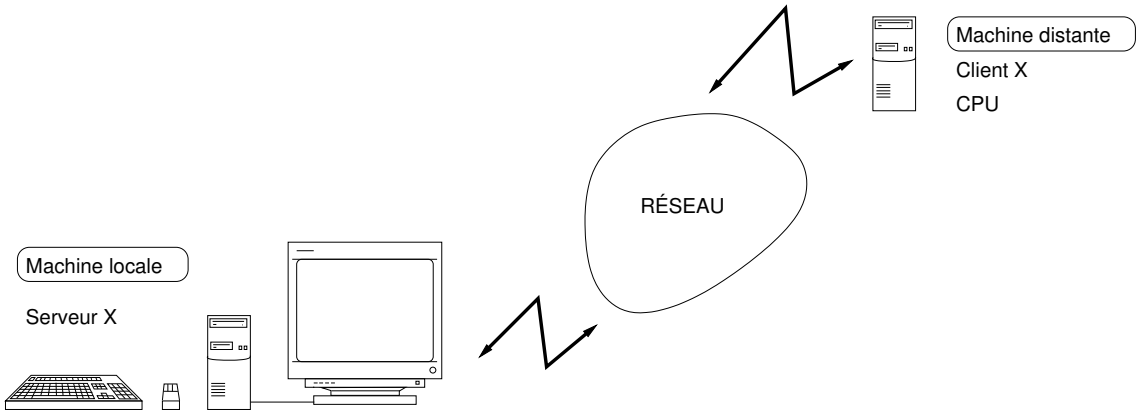


FIGURE 6.7: X en tant que protocole réseau

– rien du tout, l'application cherchera à se connecter au serveur X de la machine sur laquelle elle s'exécute.

Le nombre ⟨*numéro_display*⟩ correspond au numéro du canal initialisé par le serveur, le premier porte le numéro 0. Dans la plupart des cas les machines n'ont qu'un seul display, c'est-à-dire qu'il n'y a qu'un seul ensemble clavier/souris qui communique avec le serveur. Ce nombre ne doit pas être omis. ⟨*numéro_écran*⟩, quant à lui, est le numéro associé aux écrans attachés à un canal donné. Ainsi si une machine possède deux écrans, le premier portera le numéro 0, le suivant 1, etc. La plupart des applications dispose d'une option `-display`, `-d` ou `--display` pour passer outre l'effet de la variable `DISPLAY`, en écrivant par exemple :

```
$ xclock -display machine.domain.org:0 &
[1] 3452
$
```

qui lance une magnifique horloge sur la machine `machine.domain.org` si elle l'a autorisé.

Autoriser des machines à se connecter à un serveur

De manière à éviter que toutes les machines connectées au même réseau que vous (par exemple Internet !) ne puissent utiliser votre serveur X, il est nécessaire d'adopter une politique d'autorisation. La première politique consiste à n'autoriser que certaines machines dont on stockera les noms dans une liste. On verra que la gestion de cette liste est effectuée par l'utilitaire `xhost`.

En supposant qu'un utilisateur assis devant la machine `mekanik` veuille utiliser les ressources de calcul de la machine distante `ayler`, tout en affichant sur l'écran de `mekanik`, voici les étapes à effectuer :

1. autoriser la machine distante `ayler` à utiliser le serveur X de la machine locale `mekanik` ;
2. se connecter sur `ayler` ;
3. spécifier que le serveur X à utiliser est sur la machine `mekanik` ; c'est-à-dire qu'on veut afficher sur l'écran de la machine locale et non sur celui de la machine distante `ayler` ;
4. lancer l'application !

ce qui se traduit dans un shell par [19] :

```
$ xhost +ayler.freejazz.org   <-------------------------- autorisation
ayler.freejazz.org has been added to control list
$ rlogin ayler.freejazz.org   <---------- connexion sur la  machine distante
$ export DISPLAY=mekanik.zeuhl.org:0.0   <---------- affichage sur mekanik
xappli &
[1] 15678
$
```

Autoriser des utilisateurs à se connecter à un serveur

Lorsque la politique basée sur les machines est trop laxiste (autoriser une machine c'est autoriser tous les utilisateurs de cette machine), il faut envisager d'autoriser uniquement certains utilisateurs. Pour ce faire, on s'appuie sur l'utilitaire `xauth` qui gère un ensemble de clefs — appelées *cookies* dans le jargon de `xauth` — associées à chaque display pour un utilisateur donné. Le principe est le suivant : pour autoriser une machine `B` à afficher sur une machine `A`, il faut :

- récupérer un cookie de la machine `A` ;
- le faire connaître à la machine `B`

Pour récupérer un cookie sur la machine locale `A`, on fait :

ə

```
$ xauth nlist A:0
0000 0004 a1031e2f 0001 30 0012 4d49542d4d414749432d434f4f4b4
9452d310010 002e46651c4158584654595d61125506
$
```

qui affiche le cookie correspondant au display `A:0` sur la sortie standard. Dans une autre fenêtre, on se connecte sur la machine distante, et on lance la commande :

```
$ xauth nmerge -
```

qui attend un cookie sur son entrée standard. Par un habile « copier/coller » on peut fournir le cookie de la machine `A`. Une solution élégante, si les deux machines le permettent, est de lancer sur la machine locale `A` :

```
$ xauth nlist A:0 | rsh B /usr/X11R6/bin/xauth nmerge -
$
```

en utilisant la commande `rsh` qui lance le `xauth` sur la machine `A` depuis la machine `B`.

Compter sur ssh pour l'autorisation

► § 4.2.3 p. 94

Aujourd'hui sur certains systèmes, on peut utiliser l'option `-X` de la commande ◂ssh▸ permettant de passer outre la configuration (fastidieuse) des cookies :

```
$ ssh -X machine_distante
Password:
```

Une fois connecté sur la machine distante, les clients X, utiliseront automatiquement le serveur de la machine locale et feront transiter les requêtes X en utilisant le cryptage de `ssh`.

19. Pour l'utilisation de `rlogin`, se référer à § 4.2.2 page 93

6.4.5 Étude de cas : fond d'écran

Dans l'environnement X, on peut imposer une couleur à la fenêtre racine avec la commande `xsetroot`, par exemple :

```
$ xsetroot -solid black
$
```

met le fond d'écran à la couleur noire. Dans cette étude de cas, on se propose de mettre en fond d'écran une image tirée au hasard parmi une série. Pour ce faire, on utilisera le programme xloadimage. Si l'on dispose d'une image `fond.jpg`, on pourra la mettre en fond d'écran avec la commande :

```
$ xloadimage -border black -center -onroot fond.png
fond.jpg is a 1075x716 JPEG image, [...], Huffman coding.
Merging...done
Building XImage...done
$
```

Supposons maintenant que l'on dispose de plusieurs images dans un répertoire :

```
$ ls ~/photos/fonds
fond-00.png  fond-02.png  fond-04.png  fond-06.png
fond-01.png  fond-03.png  fond-05.png  fond-07.png
$
```

Pour choisir une image au hasard, on va d'abord compter le nombre d'images du répertoire et le stocker dans une variable :

```
$ nbf=$(ls -l ~photos/fonds/fond-??.png | wc -l)
$ echo $nbf
8
$
```

On peut ensuite choisir un nombre au hasard en exploitant la variable `RANDOM` et l'opérateur modulo du shell :

```
$ f=$[RANDOM%nbf]
$ echo $f
3
$
```

la variable `RANDOM` contient une valeur tirée aléatoirement entre 0 et 32 767 [20]. L'opérateur `%` renvoie le reste de la division entière. Par conséquent l'expression arithmétique `$[RANDOM%nbf]` renvoie un nombre tiré aléatoirement entre 0 et `nbf`-1. Il ne reste plus qu'à construire le nom du fichier à partir de ce nombre. Pour cela, on peut utiliser la commande `printf` fonctionnant comme celle du langage C :

```
$ printf "~/photos/fond/fond-%02d.png\n" $f
~/photos/fond/fond-03.png
$
```

Outre le fait que `$RANDOM` n'est pas aussi aléatoire qu'on le souhaiterait en particulier dans les bits de poids faible, cette variable ne fait pas partie de la norme Posix mais est disponible dans les shells bash et ksh.

20. Le valeur 32 767 correspond au plus grand nombre entier positif que l'on peut générer à l'aide de 16 bits, soit $2^{16} - 1$. Vous pouvez placer cela lors d'une soirée entre ami, cela fait toujours beaucoup d'effet.

Voici enfin une possibilité de script rassemblant les commandes précédentes :

fond.sh

```
#!/bin/sh
# répertoire stockant les images
dirfonds="$HOME/photos/fonds"

# nombre d'images dans ce répertoires
nbfonds=$(ls -l $dirfonds/fond-??.png | wc -l)

# numéro du fond tiré aléatoirement
nfondchoisi=$(echo $[RANDOM%$nbfonds])
# nom du fond tiré aléatoirement
fondchoisi=$(printf "$dirfonds/fond-%02d.png" $nfondchoisi)

# chargement en fond d'écran
xloadimage -display :0 -border black -center \
           -onroot $fondchoisi > /dev/null
```

► 6.4.3 p. 170
► § 2.5.3 p. 58

Il ne reste plus qu'à lancer ce script dans le fichier de ▲démarrage◀ de la session X (le fichier `.xsession`), et dans la ▲crontab◀ de l'utilisateur séduit par ce merveilleux fond d'écran dynamique. Ainsi par exemple :

```
$ crontab -l
0-59/15 * * * * ~/bin/fond.sh
$
```

changera de fond d'écran tous les quarts d'heure. Notez bien que ce cron n'aura de sens que si le système sur lequel vous vous connectez vous propose un serveur X...

L'option `-display` de la commande `xloadimage` dans le script est nécessaire car au moment où le script est lancé par le daemon `cron`, la variable `DISPLAY` est vide. Vous aurez également noté qu'on redirige dans le fichier « trou noir », les bavardages du programme xloadimage, ceci pour éviter que le daemon `cron` ne nous les envoie par mail tous les quarts d'heure...

6.5 Installer des logiciels

Nous finirons ce chapitre par une présentation succincte de la manière d'installer des logiciels sur son propre compte. Cette activité peut en effet être vue comme la configuration de son environnement de développement, puisqu'on a parfois besoin d'installer des utilitaires pour son usage personnel sans exiger de l'administrateur qu'ils soient installés pour tous les utilisateurs. Les instructions primordiales sont les suivantes :

- **lire la doc !** celle-ci se trouve généralement dans des fichiers portant les noms `README`, `INSTALL`, `LISEZMOI`, etc.
- maîtriser l'utilisation des outils ▲d'archivage◀ et de compression (`gzip` et `tar`) ;
- maîtriser l'utilisation de l'utilitaire ▲`make`◀ dans le cas d'installation sous forme de source.

► § 3.3.7 p. 72
► § 5.3 p. 124

On considérera donc que le « package » binaire ou source à installer se présente sous forme d'une archive compressée et que l'utilisateur est capable de la décompacter dans un répertoire particulier.

6.5.1 Installer des binaires

Une fois l'archive décompressée, il faut suivre les instructions se trouvant dans le fichier décrivant l'installation. Certaines archives contiennent des scripts permettant de copier les fichiers dans des répertoires spécifiques de manière plus ou moins automatique. On peut toujours éditer ces scripts pour les adapter à ses besoins. En tout état de cause, l'installation de programme binaire consiste la plupart du temps à copier :

- un ou plusieurs fichiers exécutables dans un endroit spécifique ;
- des bibliothèques.
- des pages de manuels et autres fichiers `info`.

En supposant que le logiciel s'installe dans le répertoire `~/monsoft`, il faudra s'assurer que le système puisse accéder à ces différents fichiers de la manière suivante :

- mettre à jour la variable `PATH` :
  ```
  export PATH=~/monsoft/bin:$PATH
  ```
- mettre à jour la variable `MANPATH` :
  ```
  export MANPATH=~/monsoft/man:$MANPATH
  ```
- mettre à jour la variable `LD_LIBRARY_PATH` :
  ```
  export LD_LIBRARY_PATH=~/monsoft/lib:$LD_LIBRARY_PATH
  ```

Dans le cas de « petits » utilitaires on peut se contenter d'installer les fichiers avec d'autres utilitaires, par exemple dans les répertoires :

- `~/bin` pour les exécutables ;
- `~/man` pour les pages de manuels ;
- `~/lib` pour les bibliothèques dynamiques.

6.5.2 Installer des sources

L'installation de programmes sous forme de sources peut être plus délicate à cause des problèmes de portabilité entres les différentes versions d'UNIX. Lorsque l'archive est proposée sous forme de sources, elle fournit un `Makefile` contenant diverses informations que l'on peut adapter à son installation :

- le compilateur utilisé, ainsi que les options associées ;
- les répertoires d'installations (pour les binaires, le manuel, etc.)
- ...

Ces informations sont proposées sous la forme de ►variables◄ `make`. Une fois le `Makefile` adapté (le fichier `INSTALL` ou le `Makefile` lui-même contient des instructions qui guident le choix de ces options), il faut créer l'exécutable, généralement en lançant simplement la commande :

§ 5.3.2 p. 127 ◄

```
$ make
...
$
```

Puis il faudra lancer l'installation proprement dite des fichiers ; ce qui est généralement effectué grâce à la commande :

```
$ make install
...
$
```

encore une fois les `Makefile` peuvent être préparés de manière différentes selon le logiciel à installer, il est donc impératif de lire les documentations fournies pour connaître les cibles exactes. À ce propos, la commande :

```
$ make -n install
...
$
```

vous permettera de simuler l'installation et d'avoir un aperçu des répertoires dans lesquels seront installés les fichiers.

Pour simplifier la vie des vaillants utilisateurs, le projet GNU a créé un utilitaire permettant d'adapter automatiquement le `Makefile` à votre système, ceci tout en gardant un certain degré de souplesse. Cet utilitaire se présente dans l'archive des sources sous la forme d'un script nommé `configure` :

```
$ ./configure
...
$
```

permet après un temps de scrutation de votre système de créer un `Makefile` reflétant les caractéristiques de l'UNIX sur lequel vous travaillez. Une fois le `Makefile` créé, les commandes décrites plus haut (`make` et `make install`) ont le même rôle.

Le script `configure` tient généralement compte du fait que le logiciel s'installera dans le répertoire `/usr/local`, ce qui est une pratique courante. Dans la mesure où vous n'avez généralement pas les droits d'accès à ce répertoire, vous pouvez spécifier la racine de l'installation, avec :

```
$ ./configure --prefix=$HOME/monsoft
...
$
```

qui configurera le `Makefile` pour que l'utilitaire soit installé avec comme racine `~/monsoft`. La commande `./configure --help` vous donnera les autres options disponibles pour le package que vous installez.

6.5.3 Installer des paquet Debian

Ce qui suit vous sera utile uniquement si votre UNIX est un système GNU/*LINUX* et si celui-ci est installé via des paquets Debian. C'est la cas bien évidemment de la célèbre Debian mais aussi de la star du moment : la distribution Ubuntu.

Les distributions de GNU/*LINUX* Debian et Ubuntu utilisent le même système de « paquets » pour installer les logiciels. Ce système permet de gérer les différentes versions à mettre à jour et d'inclure dans chaque paquet des scripts de pré-installation et post-installation qui sont lancés, comme leur nom l'indique, avant et après l'installation des binaires [21] du paquet [22].

Dans son utilisation la plus courante, le principe de la gestion des paquets repose sur l'utilisation de dépôts accessibles depuis le réseau ou depuis des périphériques amovibles (cédérom, dvd, ...). Ces dépôts contiennent les versions officielles des paquets constituant la distribution. Il en existe des « miroirs » dans différents pays accélérant ainsi l'accès aux ressources. Vous devriez trouver sur votre système un fichier `/etc/apt/sources.list` dont voici ce que pourrait être un extrait :

```
$ cat /etc/apt/sources.list
[...]
```

21. Les paquets Debian peuvent également contenir les sources des logiciels.
22. De tels scripts existent également pour l'opération de suppression du paquet.

```
deb http://ftp.fr.debian.org/debian/ lenny main
deb http://security.debian.org/ lenny/updates main contrib non-free
deb http://www.debian-multimedia.org lenny main
[...]
$
```

Dans la mesure où les fichiers stockés sur ces machines distantes évoluent il est nécessaire de mettre à jour la liste des paquets sur votre système. Cette liste est conservée dans le fichier :

`/var/cache/apt/pgkcache.bin`

Même s'il est fort probable que votre distribution mette à jour automatiquement par cron ce fichier, il n'est pas inutile de savoir qu'on peut explicitement rafraîchir ce fichier grâce à la commande :

`apt-get update`

Ensuite un certain nombre d'opérations classiques sont possibles en utilisant entre autres, cette même commande `apt-get` :

- installation d'un paquet :
  `apt-get install` ⟨*nom_du_paquet*⟩
- effacement d'un paquet (y compris ses fichiers de configuration)
  `apt-get -purge remove` ⟨*nom_du_paquet*⟩
- mettre à jour toutes les paquets pour lesquels existe une nouvelle version :
  `apt-get upgrade`
- chercher parmi les paquets (installés ou non) ceux qui correspondent à un motif particulier :
  `apt-cache -names-only search` ⟨*motif*⟩
  Le motif doit être saisi comme une ►expression régulière◄. Par exemple tous les paquets commençant par « `lib` » et finissant par « `pgsql` » peuvent être listés grâce à la commande :
  § 3.5 p. 77 ◄
  ```
  apt-cache -names-only search '^lib.*pgsql$'
  ```
  Pour chercher les paquets dont le nom *ou la description* contiendrait le mot « `penguin` » suivi du mot « `race` » :
  ```
  apt-cache search 'penguin.*race'
  ```

Pour finir cette présentation, sur les paquets Debian, il faut savoir qu'il existe un utilitaire de plus bas niveau permettant d'agir directement sur un paquet, c'est-à-dire sans passer un dépôt : la commande `dpkg`. Cette commande est d'ailleurs finalement appelée par la commande `apt-get` par exemple. Elle permet donc d'installer ou de d'enlever un paquet, mais aussi d'extraire toutes les informations d'un fichier au format des paquets Debian.

Pour conclure

Vous êtes donc armé pour naviguer dans un système UNIX puisque vous avez maintenant les connaissances de base pour manipuler un éditeur de texte, configurer votre shell et votre environnement graphique. Ces connaissances, couplées avec celle du chapitre 5 sur le développement, et du chapitre 4 sur le réseau, vous donnent les bases nécessaires à l'exploration de ce système particulièrement vaste qu'est UNIX. Le chapitre qui suit vous donnera quelques pistes pour trouver les informations qui vous manquent.

7 À l'aide !

Sommaire

7.1 Quoi fait quoi ?
7.2 Les pages de manuel
7.3 Le format info
7.4 Le logiciel
7.5 Les HOWTO
7.6 La documentation en ligne

? h
Sorry, I already gave what help I could...
Maybe you should try asking a human?
TEX Version 3.14159.

L'AUSTÉRITÉ d'un terminal laisse pantois nombre d'utilisateurs affrontant pour la première fois un système Unix. Combien de fois l'Unixien n'a-t-il pas entendu un utilisateur se plaindre du manque de convivialité du système par rapport à ses « concurrents » intégrant aide en ligne, lignes chaudes et autres systèmes d'assistance à l'utilisateur ? Ce chapitre a donc pour principal objectif de tordre le cou à cette idée d'indisponibilité d'information.

7.1 Quoi fait quoi ?

UNIX fournit une pléthore de commandes. En shell bash, par exemple, le fait d'appuyer par deux fois sur la touche tabulation dans un terminal provoque l'apparition du message :

```
There are 2727 possibilities.  Do you really
wish to see them all? (y or n)
```

L'interpréteur de commande signifie par là qu'il nous donne la possibilité de choisir parmi 2727 commandes différentes et ayant *a priori* toutes une action différente. Dans cette section, nous ne donnerons pas la signification et l'action de chacune de ces 2727 commandes mais plutôt le moyen de vous permettre d'accéder à l'information concernant chacune d'entre elle. En effet, l'un des principaux avantages d'UNIX est de permettre l'accès à ce genre d'informations.

7.1.1 À propos

La première question est probablement de savoir ce que vous désirez faire. Supposons que vous cherchiez à visualiser le contenu de votre répertoire de travail.

Malheureusement, vous ne connaissez pas *la* commande qui permet de le faire. Un outil permet de faire une recherche parmi les commandes disponibles : `apropos`.

Son utilisation est des plus simples. Tapez simplement `apropos` suivi d'un mot-clé (en anglais) définissant au mieux votre requête. Pour notre exemple, nous cherchons à visualiser le contenu de notre répertoire de travail (en anglais : *to display working directory content*). Essayons :

```
$ apropos display
$
```

Cette commande devrait faire apparaître un grand nombre (155 sur mon système) de lignes ressemblant probablement à [1] :

```
bell (n)              - Ring a display's bell
bitmap (n)            - Images that display two colors
cal (1)               - displays a calendar
ckalloc, memory, ckfree, Tcl_DisplayMemory,
  Tcl_InitMemory, Tcl_ValidateAllMemory (n)
                      - Validated memory allocation interface.
```

Si vous aussi, vous prétendez être un informaticien plein de qualités [2] vous ne lirez pas (sauf peut-être en dernier recours) ces 75 lignes. Le mot-clé display n'était probablement pas suffisamment significatif. Essayons autre chose :

```
$ apropos working
cd (n)                - Change working directory
chdir, fchdir (2)     - change working directory
getcwd (3)            - Get current working directory
pwd (1)               - print name of current/working directory
pwd (n)               - Return the current working directory
rcsclean (1)          - clean up working files
$
```

Nous obtenons moins d'informations mais malheureusement pas celle qui nous intéresse [3]. Essayons encore :

```
$ apropos content
dir (1)               - list directory contents
gl_copyscreen (3)     - copy the screen contents of contexts
ls (1)                - list directory contents
perltoc (1)           - perl documentation table of contents
pick (1)              - search for messages by content
pair_content (3x)     - curses color manipulation routines
stat (1)              - print inode contents
vdir (1)              - list directory contents
vga_dumpregs (3)      - dump the contents of the SVGA registers
xev (1x)              - print contents of X events
$
```

Il semble que la commande recherchée soit `dir` ou `ls` ou `vdir`. Essayons l'une d'entre elles :

1. Nous vous faisons grâce de 71 lignes.
2. Voir à ce sujet Wall *et al.* (1996) et sa définition de la paresse.
3. Il n'empêche que vous savez maintenant changer de répertoire...

```
$ ls
2Hypergraphes  Impression  Microsoft  Sarah      games
Admin          LaTeX       Mygale     Transfert  local
Archives       Logo        NT         Web        nsmail
Arkeia         Mac         Netscape   archive    test
ConfigNT       Mail        Oueb       bin        tmp
$
```

... et nous avons donc bien obtenu le contenu du répertoire courant. En conclusion, `apropos` peut être considéré comme un « moteur de recherche » sur le rôle des commandes disponibles.

7.1.2 Mais c'est quoi donc ?

Il arrive parfois d'oublier ce que peut faire une commande. Supposons que l'on vous ait parlé de la commande `find` en vous disant qu'elle était fabuleuse. La commande `whatis` devrait vous éclairer.

```
$ whatis find
find (1)        - search for files in a directory hierarchy
$
```

Notez que `whatis` ne sait pas nécessairement de quoi vous lui causez :

```
$ whatis trouver
trouver: nothing appropriate
$
```

`whatis` permet donc d'obtenir une information succincte sur le rôle d'une commande shell.

7.2 Les pages de manuel

Le paragraphe précédent nous a permis de constater que `find` permettait de rechercher des fichiers dans une arborescence. Supposons que vous vouliez rechercher un fichier dénommé `toto` perdu quelque part sur votre *home*. Un `find toto` lancé en ligne de commande produit un navrant `find: toto: No such file or directory`. Vous vous douterez alors que `find` attend peut-être autre chose pour lui permettre de trouver ce fameux `toto`. Une commande va nous aider grandement : `man`. Osons un `man find`. Devant vos yeux ébahis devrait apparaître une page de manuel (raccourci en anglais en *manpage*) concernant `find`.

7.2.1 Une page de manuel

Les pages de manuel ont cette propriété d'avoir une structure assez fixe. Détaillons quelques-unes de ses principales entrées :

Name qui définit en une ligne ce que fait la commande. C'est d'ailleurs cette ligne qui est affichée par `whatis`. Il s'agit de la seule partie requise pour faire d'une page de documentation quelconque une *manpage* ;

Synopsis qui précise la forme d'appel de la commande. Le synopsis précise de plus toutes les options qu'il est possible de passer à la commande et ses arguments éventuels ;

Description qui décrit effectivement l'action de la commande, précise le rôle de chaque argument ;

Options précise le rôle de chaque option (vous vous en doutiez, non ?);

See Also vous donne quelques commandes qui ont une action similaire à la commande dont vous avez la page de manuel sous les yeux et parfois les commandes utilisées par la commande.

Parfois on trouve également en fin de page, des exemples et les bugs connus du programme documenté.

7.2.2 Hiérarchie des pages de manuel

Vous aurez constaté que la page de manuel de `find` commence en fait non pas par NAME, mais en fait par un nébuleux FIND(1). Les pages de manuel sont souvent hiérarchisées en douze sections numérotées de 1 à 9, *n*, *o* et *l*. Décrivons succinctement cette hiérarchie.

1 commandes de l'utilisateur pouvant être lancées par n'importe qui ;

2 appels système (fonctions fournies par le noyau) ;

3 fonctions propres à certaines bibliothèques (C, X windows, etc.) ;

4 ports et périphériques (c'est-à-dire les fichiers de /dev) ;

5 descriptions de formats de fichier de configuration ;

6 jeux ;

7 divers (essentiellement des conventions) ;

8 outils d'administration système ;

9 routines des noyaux (propre à *LINUX*).

Les sections suivantes ont vraisemblablement été abondonnées mais peuvent être toutefois présentes sur de « vieux » systèmes :

n ce qu'il y a de nouveau. Les documentations résidant dans cette partie se doivent d'être déplacées au bout d'un certain temps dans la section appropriée ;

o vieilles documentations condamnées à la disparition ;

l documentation locale propre à un système, un site.

Notons qu'il n'est pas conseillé d'utiliser les trois dernières sections et que ces dernières ont de plus en plus tendance à disparaître (sur le système où ces lignes sont écrites, seule subsiste la section *n*[4]).

Il peut arriver qu'un mot donné corresponde à des pages de manuel de différentes sections. Par exemple, vous cherchez une information sur la fonction C `printf` :

```
$ apropos printf
format (n)           - Format a string in the style of sprintf
gl_printf (3)        - write formatted output in graphic mode
printf (1)           - format and print data
sprintf, printf,(3)  - formatted output conversion
printftest (6)       - tests the vgagl gl_printf function
snprintf, vsnprintf (3) - formatted output conversion
$
```

4. Et elle ne contient que 210 pages de documentation alors que la section 1 en contient environ 1300.

Si vous vous contentez d'un `man printf`, à votre grand désarroi, vous ne pourrez visualiser que la page de documentation correspondant à la section 1. Comment spécifier à `man` d'aller regarder dans la section 3 ? Facile : essayez un `man 3 printf` et réjouissez-vous :

```
$ man 3 printf
PRINTF(3)        Linux Programmer's Manual         PRINTF(3)
NAME
    printf, fprintf, sprintf, snprintf, vprintf, vfprintf,
    vsprintf, vsnprintf - formatted output conversion
SYNOPSIS
    #include <stdio.h>
...
$
```

Bien entendu, il est nécessaire d'interroger une section dans laquelle existe la page de documentation. Il y a fort à parier qu'un `man 2 printf` vous rétorque narquoisement qu'il n'y a pas d'entrée pour `printf` dans la section 2 du manuel.

7.2.3 La variable d'environnement `MANPATH`

Le plus rageant avec l'utilisation de `man` est d'obtenir un insoutenable *No manual entry for (le nom de la commande)*. Par exemple, un `man xv` produit *No manual entry for xv*. La première idée qui peut vous venir à l'esprit est qu'il n'existe pas de documentation pour cet outil qu'est `xv`. C'est possible mais regardons de plus près une certaine variable d'environnement dénommée `MANPATH`.

```
$ echo $MANPATH
/usr/share/man:/usr/local/man
$
```

Cette variable a le même format que ►`PATH`◄ et indique à la commande `man` les endroits où rechercher les pages de manuel. Dans l'exemple précédent, `man` ira donc dans `/usr/share/man` puis (en cas d'échec) dans `/usr/local/man` pour tenter de trouver l'information qui vous intéresse. Aucune page n'étant associée à `xv`, `man` vous le signifie poliment. Il suffit de rajouter un nouveau répertoire de recherche (`/usr/X11R6/man` sous Linux) à la variable `MANPATH` pour voir apparaître la page de documentation.

§ 6.1.4 p. 147 ◄

```
$ export MANPATH=$MANPATH:/usr/X11R6/man
$ man xv
XV(1)                                                XV(1)
NAME
xv - interactive image display for the X Window System
 ... etc ...
$
```

Cette variable devrait être *a priori* « correctement » configurée pour aller visiter les points de chute classiques des pages de manuel, c'est-à-dire (cette liste ne se veut en aucun cas exhaustive) `/usr/share/man`, `/usr/local/man` et `/usr/X11R6/man`.

Dans le cas où vous souhaiteriez installer certaines pages de manuel à certains endroits précis de votre compte utilisateur, vous pouvez utiliser la même démarche et ajouter dans vos ►fichiers◄ de démarrage :

§ 6.1.6 p. 148 ◄

```
export MANPATH=$HOME/man:$MANPATH
```

de manière à ajouter vos pages de manuels aux pages traitées par la commande `man`.

7.2.4 Recherche exhaustive

Il peut arriver qu'en désespoir de cause, vous désiriez effectuer une recherche exhaustive d'un mot donné dans l'ensemble des pages de manuel. Si vous choisissez de prendre ce risque, sachez que c'est possible mais qu'il y a de grandes chances que cela prenne un certain temps [5].

7.2.5 La commande ultime

Pour toutes les subtilités concernant votre version locale de `man`, vous n'avez finalement qu'une seule commande à connaître : `man man`. La boucle est donc bouclée.

7.3 Le format info

Il existe un système de documentation répandu sur les systèmes UNIX basé sur les outils GNU : le système Texinfo. L'idée de ce système est de pouvoir produire à partir d'un source, à la fois une information de type hypertexte et un document imprimable. D'un point de vue technique, ce même source est passé à la moulinette de TEX pour produire la version papier, et par l'utilitaire `makeinfo` pour produire la version hypertexte. On manipule la version imprimable comme un document au format dvi traditionnel [6], et la version hypertexte avec la commande `info` :

```
$ info whoami
$
```

vous permet de vous esbaudir avec quelque chose devant ressembler à

```
File: sh-utils.info,  Node: whoami invocation,  Next: groups
invocation,  Prev:  logname invocation,  Up: User information

‘whoami’: Print effective user id
=================================

   ‘whoami’ prints the user name associated with the current
effective user id.  It is equivalent to the command ‘id -un’.

   The only options are ‘--help’ and ‘--version’.
   *Note Common options::.
```

Les commandes à connaître pour manipuler `info` dans un terminal sont :
- `SPC`/`-` : pour défiler vers la page suivante/précédente ;
- `u` : pour le lien père (*up*) ;
- `n`/`p` pour le lien suivant/précédent (*next*/*previous*) dans la hiérarchie

5. Par exemple, à la toute fin du siècle dernier, un `man -K yaourt` a pris environ une minute sur notre beau bi-Xeon (500 MHz - 512 Mo de RAM) et plus de cinq minutes sur notre bon vieux 486 (33 MHz - 24 Mo).

6. Voir à ce sujet, le merveilleux ouvrage de Lozano (2008).

– `l` pour le lien précédent dans l'historique (*last*) ;
– `RET` pour pénétrer dans un lien
– `q` pour quitter info.

Si vous disposez d'Emacs, il existe une interface « cliquable » permettant d'éviter de s'encombrer l'esprit avec les commandes précédentes tout en vous donnant la possibilité de manipuler joyeusement le bouton du milieu de votre souris préférée.

Notez enfin que s'il n'y a pas de page info associée à une commande, l'exécution de `info <nom_de_commande>` vous affichera simplement la page de manuel de la commande en question. Si la page de manuel n'existe pas non plus, `info` vous affichera le sommaire de l'ensemble des pages d'info existant sur votre site. Pour boucler la boucle de manière analogue à la commande `man`, essayez :

```
$ info info
$
```

7.4 Le logiciel

Tout bon logiciel devrait être fourni avec une documentation présentant l'ensemble de ses fonctionnalités. Sous *LINUX*, cette documentation se trouve généralement installée dans le répertoire `/usr/doc` ou `/usr/share/doc`. Un `ls` devrait peut-être assouvir votre curiosité. Ces documentations sont généralement rédigées en pur texte ASCII et `more` ou `less` devrait vous permettre de les découvrir. Si vous ne savez vraiment pas à quoi peut servir une commande ou un logiciel donnés, vous pouvez toujours tenter de l'exécuter avec l'option `-h` ou `--help`, c'est parfois suffisant.

7.5 Les HOWTO

Il existe autour de GNU/*LINUX*, un système de documentation dénommé la documentation HOWTO (en engliche, « comment faire »)[7]. Cet ensemble de documents fait lui-même partie d'un vaste projet de documentation nommé le *linux documentation project* sur lequel on peut obtenir des infos sur le site `http://www.linuxdoc.org`. Sur la plupart des distributions de *LINUX*, les Howto's sont installés dans le répertoire recevant la documentation sur votre système[8]. Initialement formatées au format sgml, ces docs sont disponibles aux formats texte, PostScript, html et dvi. On trouve en vrac (plus de 400 thèmes sur une Debian Sarge), des informations destinées à la fois à l'utilisateur, l'administrateur ou au programmeur, sur les modems, l'impression, le son, l'interaction avec le Palm Pilot, la programmation du port série, la configuration des cartes SCSI, le jeu Quake, l'utilisation de *LINUX* en Thaïlande ou en Slovénie, bref tout sujet sur lequel un utilisateur de *LINUX* s'interroge un jour ou l'autre. Toutes ces documentations sont pour la plupart très claires, prenant l'utilisateur novice « par la main » de manière très progressive, avec moult exemples ; elles portent des noms significatifs qui peuvent être utilisés comme argument aux moteurs de recherche : `Printing-Howto`, `Serial-Programming-Howto`, etc. Lisez donc le `Howto-Howto` !

7. Pour les allergiques à Shakespeare (et les autres), certaines documentations ont été traduites dans notre belle langue.
8. Le répertoire `/usr/share/doc/HOWTO` sur la distribution Debian.

7.6 La documentation en ligne

Si malgré les lectures des pages de manuels et de howtos, vous n'arrivez toujours pas à comprendre comment peut fonctionner telle ou telle commande, ou comment faire faire ceci ou cela à un logiciel particulier, rien n'est encore perdu ! Il vous reste le monde extérieur, des personnes qui ont les mêmes problèmes que vous ou qui les ont déjà résolus.

7.6.1 Les newsgroups

La première source d'informations du point de vue de sa richesse mais aussi de son « bruit » est sans doute les groupes ou forums de discussion (*newsgroups*). Apprenez en effet que si vous vous posez une question donnée, la probabilité pour que quelqu'un se la soit déjà posée et surtout ait la réponse, est extrêmement grande.

Les *news* constituent un vaste panneau d'affichage que tout le monde peut consulter, sur lequel on peut poser des messages à certaines conditions ; ce vaste panneau est séparé en thèmes (*group*) hiérarchisés. Parmi les milliers de thématiques abordées, on trouve quelques groupes autour d'UNIX. Ces groupes font partie de la hiérarchie `comp.os` pour les groupes traitant de l'informatique (`comp`*uter*) et plus précisément des systèmes d'exploitation (`o`*perating* `s`*ystem*), ou de son pendant francophone `fr.comp.os`. On trouvera parmi ces groupes :

- `fr.comp.os.unix` : discussion autour d'UNIX;
- `fr.comp.os.linux.annonces` : les annonces concernant les nouvelles versions ou les rencontres entre utilisateurs ;
- `fr.comp.os.linux.moderated` : groupe modéré sur l'utilisation de *LINUX* ;
- ...

On pourra également lire les groupes `comp.os.linux.*` de la hiérarchie anglophone. Il est important de comprendre que ces groupes suivent quelques règles d'usage dont l'ignorance peut vous faire envoyer plus ou moins poliment sur les roses :

- on veillera à respecter la nétiquette[9] dans le cadre de la communication « de un à plusieurs » ;
- on testera son logiciel de news sur un groupe comme `fr.test` et non sur le groupe sur lequel on a l'intention de poster ;
- on veillera à lire attentivement « toutes » les documentations disponibles (notamment les ◭faqs◮) avant de poser une question dont la réponse se trouve dans une dizaine de manuels, au risque d'obtenir une réponse du genre : `man` ⟨*softquejarrivepasàfairemarcher*⟩ ou RTFM (qui signifie dans sa version polie : *read the fantastic manual*) ;
- il faudra lire les archives des mois précédents pour éviter de poser une question posée la veille ; une bonne source d'information pour rechercher dans les archives est le site `groups.google.com` qui fonctionne comme un moteur de recherche ;
- on tentera de trouver la charte du groupe si elle existe et la lire ;
- certains groupes sont modérés, c'est-à-dire qu'un groupe de personnes filtre votre message avant de le poster sur le groupe. Le filtrage a pour but de vérifier l'adéquation du contenu avec le thème du groupe. Par conséquent votre

► § 7.6.4 p. 189

9. voir par exemple une traduction en français à `http://www.sri.ucl.ac.be/SRI/rfc1855.fr.html`

message n'apparaîtra pas immédiatement sur le groupe, inutile donc de le reposter !

Le mieux est sans doute de se renseigner précisément sur l'usage des news en consultant des documents destinés aux nouveaux utilisateurs, comme par exemple `http://UsenetFR.free.fr/BU.htm`.

7.6.2 Les mailing lists

Les listes de diffusion (ou *mailing lists*) ont un caractère plus confidentiel puisqu'il faut explicitement s'abonner auprès d'un serveur pour recevoir les messages traitant d'un thème donné. Lorsqu'on envoie un message à la liste, tous les adhérents le reçoivent également ; le principe est donc différent du fonctionnement des news. On pourra trouver des listes de discussions traitant de points particuliers de *LINUX* comme par exemple :

- `linux-8086` - for the 8086 systems
- `linux-admin`
- `linux-kernel` - General kernel discussion
- `linux-kernel`-announce
- `linux-sound`
- ....

plus de soixante listes référencées sur `http://www.linux.org/docs/lists.html`. Pour s'abonner, il faut envoyer un mail poli au gentil robot qui manage tout ça à : `majordomo@vger.kernel.org`. Le *corps* du message doit contenir le mot `subscribe` (s'abonner) suivi du nom de la liste. Sans s'abonner on pourra consulter les archives des messages, par exemple sur :

- `http://www.tux.org/hypermail`
- `http://news.gmane.org`

7.6.3 Le ouèbe

La majorité des logiciels suivant le modèle open source dispose d'une page web dite *home page*. Sur cette page se trouve une description du logiciel, des documentations pour l'utiliser et/ou l'installer, et la dernière version à télécharger. Outre les moteurs de recherche généraux, on pourra consulter :

- `freshmeat.net` : donnant des infos quotidiennes sur les nouveaux projets et disposant d'un moteur de recherche ;
- `sourceforge.net` : hébergeant plusieurs projets de logiciels libres, dispose également d'un outil de recherche.

7.6.4 Les foires aux questions

Le terme faq pour *frequently asked question* est parfois traduit par « foire[10] aux questions ». Certaines personnes collectent donc les questions fréquemment posées par les « p'tits nouveaux » et les « Grands Anciens » et, de ce fait, les réponses fréquemment données pour écrire des compendiums extrêmement riches en informations, astuces et autres indices pour le débutant (et le confirmé aussi parfois). Une recherche sur le Ouèbe avec votre moteur de recherche préféré avec votre question

10. Traduction qui a le léger inconvénient de faire perdre la notion de questions *fréquemment* posées.

et le mot-clé FAQ pourrait vous permettre d'atteindre le nirvana assez rapidement. Vous trouverez sur les sites :

- `http://www.linuxdoc.org/FAQ/Linux-FAQ/index.html`
- `http://www.linuxdoc.org/FAQ/`

la FAQ officielle de *LINUX* et celles de quelques-uns des logiciels connexes.

7.6.5 Les Request for comments

Même s'il s'agit de documents très techniques, il peut être utile de connaître l'existence des RFCs (*request for comments*). Ces documents maintenus par l'IETF (*Internet Engineering Task Force*[11]) décrivent après les standards utilisés sur le réseau Internet. Chacun de ces documents est identifié par un numéro et passe de l'état de brouillon (*draft*) à l'état de standard après un processus de correction. On retrouvera parmi les RFCs la description des protocoles réseau que nous utilisons quotidiennement (http, smtp pour le mail, etc.).

*
* *

Bon courage !

11. `http://www.ietf.org`

Bibliographie

P. Albitz et C. Liu : *DNS and BIND*, chapitre 1. O'Reilly, 3e édition, 1998.

R. Card, E. Dumas et F. Mével : *programmation Linux 2.0*. Eyrolles, 1997.

C. DiBona, S. Ockman et M. Stone : *Open Sources — Voices from the Open Source Revolution*. O'Reilly & Associate, 1999. Traduit en français par un collectif et paru chez le même éditeur, sous le nom « Tribune libre, ténors de l'informatique libre ».

S. Garfinkel et G. Spafford : *Practical UNIX & Internet Security*. O'Reilly, 2e édition, 1996.

H Grégoire : *Mémoires de Grégoire, ancien évêque de Blois*. Ambroise Dupont, Paris, 1837.

Brian Kernighan et Dennis Ritchie : *Le langage C*. Dunod, 2e édition, 2004.

N. Leclerq : Logiciel libre : une volonté de transparence. *http://www.linux-france.org/article/these*, 1999.

Vincent Lozano : *Tout ce que vous avez toujours voulu savoir sur LaTeX sans jamais oser le demander*. Framabook/In Libro Veritas, 2008. `http://www.enise.fr/cours/info/latex` et `http://www.framabook.org/latex.html`.

D. Newham et B. Rosenblatt : *Learning the bash shell*. O'Reilly, 1998. L'ouvrage indispensable pour apprendre le shell de chez gnu.

Eric Raymond : How to become a hacker. *http://www.catb.org/~esr/faqs/hacker-howto.html*, 2000.

J.-M. Rifflet : *La programmation sous UNIX*. Ediscience international, 3e édition, 1995.

Didier Roche : *Simple comme Ubuntu*. Framabook, 2010.

P. H. Salus : *A Quarter-Century Of Unix*. Addison Wesley, 1994. LE livre pour découvrir la genèse d'unix.

R. L. Schwartz : *Introduction à Perl*. O'Reilly International Thomson, 1995. Traduction de Josianne Vinh.

R Stallman : *Free Software Free Society: selected essays of Richard M. Stallman*. CreateSpace, 2009.

R. Stallman, Williams S. et Masutti Ch. : *« Richard Stallman et la révolution du logiciel libre » une biographie autorisée*. Eyrolles, 2010. `http://www.framabook.org/stallman.html`.

A Tanebaum : *Architecture de l'ordinateur*. Sciences Sup. Dunod, 4e édition, 2001.

L. Wall, T. Christiansen et R. L. Schwartz : *Programming Perl*. O'Reilly, 2e édition, 1996.

Glossaire

C

Nom donné au langage avec lequel le système UNIX initial a été conçu. On peut même dire que le langage C a été créé pour bâtir ce système d'exploitation. Deux des fondateurs d'UNIX y ont consacré un livre Kernighan et Ritchie (2004).

Commande

Une commande est un ordre qu'on passe au système d'exploitation via un interpréteur de commande (appelé également *shell*). Chaque commande fait appel soit à un exécutable stocké dans le système de fichiers, soit à une commande interne. La puissance d'UNIX provient essentiellement du fait que ces commandes peuvent être combinées entre elles via des tubes. De plus elles s'insèrent naturellement dans des scripts qu'on peut exécuter comme n'importe quel programme.

Distribution

Dans le monde d'UNIX et de *LINUX* en particulier, une distribution est constituée du noyau *LINUX*, d'un ensemble d'utilitaires du projet GNU et d'un ensemble de logiciels divers et variés. Ce qui caractérise la distribution est également le système permettant d'installer et de mettre à jour ces logiciels. À titre indicatif, en 2010 la distribution Debian contient plus de 25000 paquets. Les dépendances (logiciels nécessaires à l'installation) et la diffusion d'un paquet sont gérés par une personne qui n'est la plupart du temps pas celle qui a conçu le logiciel empaqueté.

Emacs

Éditeur de texte et également religion ou église. Tous les dogmes de cette religion sont présentés à la section 6.3 page 154.

Gnu

Acronyme de GNU is not UNIX. C'est le nom que Richard STALLMAN a donné au projet qu'il a lancé en 1983 pour concevoir un UNIX entièrement libre. Dans le monde UNIX, le projet GNU est connu pour son éditeur Emacs, son compilateur C, son interpréteur de commande Bash et pour tous les utilitaires qui gravitent autour de la ligne de commande (manipulation de fichiers, processus, etc.). Un petit aperçu de la genèse du projet GNU est présenté au chapitre 1.

Howto

« Comment faire pour ... » : c'est l'objectif de cet ensemble de documents qui ont été rédigés au fur et à mesure de l'évolution des logiciels concernés. Ces précieux documents didactiques ont pour but d'assister l'utilisateur dans des domaines très variés autour de l'installation et la configuration de GNU/*LINUX*.

Lien

Dans le système de fichiers d'UNIX un lien physique correspond à l'association entre un nom et les données effectives du fichier. Ce qui peut surprendre au premier abord, c'est qu'on peut créer plusieurs liens vers les mêmes données physiques. En d'autres termes un fichier peut avoir plusieurs noms. Les liens symboliques quant eux, font correspondre un nom à un autre nom. Tout ceci est expliqué à la section 2.3.9 page 42

Linux

C'est un noyau UNIX né d'un projet initié par Linus TORVALDS. Il s'agit donc d'un ensemble de modules logiciels offrant aux programmes les abstractions communes aux systèmes d'exploitation modernes : la notion de fichiers, la notion de processus et la notion de mémoire virtuelle. Il est majoritairement écrit en langage C, les sources qui le constituent occupent environ 40 Mo compressés.

Makefile

L'utilitaire make est un outil dont le propos est d'automatiser la création de fichiers à partir d'autres fichiers selon des règles que définit l'utilisateur. Ces règles sont écrites selon une syntaxe particulière dans un fichier qu'on nomme généralement le `makefile`. Vous pouvez découvrir quelques fonctionnalités de make à la page 124.

Posix

Portable Operating System Interface (le x de ce nom proposé par Richard STALLMAN est là pour rappeler l'héritage d'UNIX) est constitué d'un ensemble de documents définissant ce que devrait être les composants (interfaces utilisateurs et interfaces logicielles) d'un système d'exploitation « compatible »avec un système UNIX.

Processus

Il s'agit de l'objet que manipule le système d'exploitation lorsqu'un programme est en cours d'éxécution. Chaque processus se voit attribuer un numéro unique. Le système (et en particulier le noyau) est chargé de leurs réserver le (ou les) processeur(s) à tour de rôle et de manière optimale. Tout ce qu'il faut savoir sur les processus se trouve à la section 2.4 page 47.

RFC

Les *Request for comments* sont des documents maintenus par l'IETF (*Internet Engineering Task Force*) qui décrivent les standards utilisés sur le réseau Internet. On retrouvera parmi les RFCs la description des protocoles réseau que nous utilisons quotidiennement (http, smtp pour le mail, etc.).

Shell

Le shell est la couche logicielle qui enferme (comme dans une coquille — *shell*) le système d'exploitation et constitue une interface pour l'utilisateur final. Il s'agit donc du moyen qu'à ce dernier de communiquer avec le système. On

appelle également « shell »les programmes implémentant cette interface sous la forme d'un interpréteur de commande. Bash, Zsh, Tcsh en font partie parmi quelques autres. Il faut également noter que le shell, en tant qu'interpréteur de commande, fournit aux utilisateurs un véritable langage de programmation. Il est question du shell dans les chapitres 2 et 3 ainsi qu'à la section 5.2 page 109 pour l'aspect programmation.

Tubes

Un tube est un objet du système permettant de faire communiquer deux processus via deux interfaces (une entrée et une sortie). Un tube permet donc de mettre en relation la sortie d'une commande avec l'entrée d'une autre. On peut ainsi combiner plusieurs commandes élémentaires pour en fabriquer une plus complexe. C'est d'ailleurs en les créant que Douglas MacIlroy a énoncé la désormais célèbre règle de construction des utilitaires du système Unix (voir à ce sujet § 1.2.1 page 5). Les tubes sont présentés à la section 3.2 page 62.

Unix

C'est le nom donné initialement (1969) au système d'exploitation créé par ken Thompson. Ce système faisant suite au projet Multics, Brian Kernighan propose d'abord « Unics » qui deviendra ensuite « unix ». Aujourd'hui unix désigne essentiellement une famille de système d'exploitation.

X

X est à la fois l'ensemble des logiciels permettant de gérer les applications en mode graphique sur un système unix, mais aussi un protocole réseau permettant à ces applications de diriger l'affichage vers une machine autre que celle où elles s'exécutent. X ou X window a été créé au début des années 80 par le MIT. Il en est question dans cet ouvrage à la section 6.4 page 168.

man

C'est la commande permettant aux valeureux utilisateurs du shell d'obtenir le manuel décrivant exhaustivement une commande. Les « pages de manuel » (*man pages*) si elles sont un peu ardues car peu didactique, constituent souvent l'information faisant référence pour un utilitaire donné.

root

C'est le nom donné à l'utilisateur ayant tous les privilèges sur un système unix. Le nom vient du fait que sa « maison »(*home*), c'est-à-dire son répertoire privé, se trouve souvent à la racine du système de fichiers. Étant donné qu'unix est un système où « tout le monde il est égaux » on y trouve, comme souvent, un tyran. Dans le monde unix c'est `root`, qui peut « tuer » vos processus et détruire vos fichiers...

vi

Célèbre éditeur, généralement installé par défaut sur les systèmes unix. Dans sa toute première version, il a été conçu par Billy Joy, l'instigateur de la branche BSD. Il est question de vi à la section 6.2 page 152.

Index

Symboles

; 25
" 112
* 27, 61
/ 27
1>&2 121
2> 63, 121
; 27
< 27, 63
>> 64
> 27
? 27
..... 27, 109
$* 112
$? 120
$@ 112
$# 112
$ 27, 62
&& 25
& 27
\ 61
~ 27, 29
|| 25
| 27, 66

A

a2ps 56
acces control list 34
actif (processus) 48
adresse
    web (url) 103
    électronique 101
adresse IP 87
afficher 67
afterstep 170, 171
aide
    apropos 181
    dans emacs 168
    en ligne 188–190
    FAQs 189
    HOWTOS 187
    info 186
    mailing lists 189
    man 183
    newgroups 188
    Rfc 190
    sur le ouèbe 189
    whatis 183
alias de commande 145
and 117
annuler dans emacs 161
apache 19
append 64
apropos 182, 183
apt-get 179
ar 140
arborescence 31
    DNS 88
    racine 31
    serveur ftp 92
archivage 72
arguments
    dans emacs 158
    de fonction en shell 122
    de la ligne de commande 111
arrière plan 53
ascii (transfert ftp) 92
aspirer un site web 104
at 57
atq 57
atrm 57
autorisation avec X 173–174
avant-plan 53
awk 61, 79–82, 84, 142, 150

B

background 55
backquote 76
backslash 61
backward 160
bash VIII, 109, 144, 145, 149, 175, 181
bash 76, 109, 117, 146, 148
bc 64, 65

`bg` 55
bibliothèque 130
  emplacement 139
bibliothèque dynamique 130
bibliothèque et édition de liens ... 136
binaire (transfert ftp) 92
`bind` 19
boucle `for` 76, 114
bounding box 125
buffer d'emacs 160
bureau d'emacs 168
byte 71
`bzip2` 72, 73

C

`cancel` 57
caractères
  de fin de fichier 63
  espace 27
  génériques 28
  spéciaux 27
`case` 119
`cat` 24, 66, 67, 108, 134
`cc` 141
`cd` 35, 36, 44, 92
change directory 36
charger dans emacs 161
chercher 70
`chgrp` 38
`chmod` 38–41, 46
`chown` 38
cible `make` 125
client Xwindow 169
commande 23
  composition 25
  emacs 156
  expansion 61
  externe 23
  historique 145
  interne 23
  substitution 75
  à distance 94, 100
commentaire en shell 109
commenter dans emacs 163
compilation 130
  avec `make` 140
  en ligne de commande .. 135, 136
  générer les dépendances 142
  optimisation 135
  options 135
  préprocesseur 132
  répertoires de recherche 133
  séparée 135
  warnings 135
complétion
  emacs 158
`compress` 72
compression de fichiers 72
compter 69, 83
concatenate 67
concaténer 67
connexion
  ftp 92
  rlogin 93
  ssh 94
  telnet 93
control-c 54
control-z 54
`convert` 56
convivialité 21
copier/coller dans emacs 161
`coreutils` 119
courrier électronique ..... voir mail
`cp` 36, 75
`cpp` 132, 133
create 72
`cron` 58, 176, 179
`cron` 58, 71, 176
`crontab` 58
`csh` 109
`csh` 109, 144, 148
Ctrl-d (fin de fichier) 63
`curl` 105
`cut` 68, 80

D

daemon 56
`date` 75
se déplacer dans emacs 160
`/dev/null` 65
`df` 33, 71
`dig` 89, 91
`dir` 182
discussion en réseau 100
`DISPLAY` 172, 173, 176
display Xwindow 169

disques
  informations 71
  logiques 31
  physiques 31
djgpp v
dns 88
domain name system 88
done 118
dpkg 179
draft 190
driver 15
droits
  bloc 34
  changer 38–39
  fichier 34
  par défaut 38
  utilisateur 29
du 71
dvips 56
début et fin 69
découper 68
démarrage (fichiers de) 148
démarrage de Xwindow 170
dépendance make 126

E

echo 24, 25, 63, 74, 75, 108, 118
éditer un fichier 107
édition de liens 130
  avec bibliothèques 136
  en ligne de commande 136
  répertoires de recherches 139
EDITOR 147
effacer dans emacs 160
elinks 105
Emacs .10, 48, 108, 152, 154, 156, 163
emacs
  aide 168
  arguments 158
  buffer 160
  bureau 168
  charger 161
  commande 156
  commenter 163
  complétion 158
  copier/coller 161
  effacer 160
  .emacs 166
  et les Makefiles 163
  frame 159
  francisation 167
  historique 158
  macros 165
  minibuffer 157
  mode 156
  modes 167
  personnaliser 165–168
  raccourcis 166
  rechercher/remplacer 162
  sauver 161
  se déplacer 160
  sélectionner 160
  undo/redo 161
  window 159
emacs ..... viii, 156
.emacs 166
emacs 143
endormi (processus) 48
entrée standard 62
ENV 148
env 26
environnement
  de développement 147
  des logiciels 147
  variable d' 26
EOF 63
erreur standard 62
espace (nom de fichier) 27
exit 121
expansion 28, 61
export 27
expression régulière 28, 77, 80, 82
extended regexp 78, 82
extract 73
exécutable 130
  lié dynamiquement 136
  lié statiquement 138
  localisation 147
exécution 132

F

FAQs 189
fg 55
fichier
  archivage 72
  attributs 34

avec espace 27
compression 72
copie 36
droits 34
déplacement 36
éditer 107
exécutable 130–132
informations 71
nom 27
objet 130, 136
occupation 85
propriétaire 38
rennommer 37
source 130
système de 30
taille 71
transfert 91
field 68
file 72
`file` 119
`find` 70, 71, 77, 85, 86, 183, 184
`finger` 99
fonction du shell 121
`for` 76
foreground 55
forward 160
`.forward` 103
frame d'emacs 159
francisation d'emacs 167
ftp 91
`ftp` 59, 87
`fuser` 150, 151
fvwm 170, 171
fvwm2 171

G

gcc ..... VI, 134, 142
`gcc` 132–135, 142
gestionnaire de fenêtres 170
`getent` 83
`getfacl` 45–47
Gimp 19
global 81
globbing 28
Gnome 170
gnome-keyring-daemon 97
`grep` 61, 63, 77–79, 86, 150
groff 87
group identifier 29, 83
groupe 29
utilisateur 28
Gtk 19, 170
`gzip` 72, 73, 176

H

`hash` 93
`head` 69
`help` 92
`HISTCONTROL` 145
`HISTFILE` 145
`HISTFILESIZE` 145
historique 145
emacs 158
`history` 145
home 35
home directory 28, 35
`host` 89, 90
HOWTOS 187
human readable 71

I

ICMP 89
`id` 29, 120
identificateur
d'utilisateur 28
de groupe 29
de processus 48
`if` 120
`IFS` 123
ignore case 79
impression 56
improved 154
index node 33
info 187
`info` 186, 187
inode 33, 42
installer des logiciels 176–178
Internet Engineering Task Force[12] 190
interpréteur 109
invite ..... voir prompt
IP 87

12. `http://www.ietf.org`

J

job 55
job control 51, 53
`join` 69
joker 28

K

Kde 170
`kill` 51, 52, 55
ksh 109, 175

L

`LD_LIBRARY_PATH` . 139, 140, 147, 177
`ldd` 137
`less` 67, 187
lftp 105
lien 34, 42–44
  physique 42
  symbolique 43
like this xi
link 42
link ..... voir edition
links2 105
list processing 156
`ln` 42
`locate` 71
logiciels connexes
  a2ps 56
  afterstep 170, 171
  apache 19
  bash viii, 109, 144, 145, 149, 175, 181
  bind 19
  convert 56
  coreutils 119
  cron 58, 176, 179
  csh 109
  curl 105
  djgpp v
  dvips 56
  elinks 105
  Emacs . 10, 48, 108, 152, 154, 156, 163
  emacs ..... viii, 156
  ftp 91
  fvwm 170, 171
  fvwm2 171
  gcc ..... vi, 134, 142
  Gimp 19
  Gnome 170
  gnome-keyring-daemon 97
  groff 87
  Gtk 19, 170
  info 187
  Kde 170
  ksh 109, 175
  lftp 105
  links2 105
  lynx 105
  make 107, 129, 142, 163
  Motif 170
  Mozilla 103
  mutt 102
  mwm 170
  MySql 17
  Netscape 103
  OpenOffice 17
  Opera 103
  Perl 14, 17, 19
  Qt 170
  rlogin 91, 93, 94, 98
  sed ..... xi, 116
  sendmail 19
  sh 109
  ssh 91, 97
  telnet 91, 93, 94, 98
  vi ..... viii, 108, 152–154
  vim 154, 163
  w3m 105
  wget 59, 104
  windowmaker 170, 171
  X 17, 154, 168–176
  X window 168
  Xaw 170
  xdvi 150
  Xlib 168
  xloadimage 175, 176
  Xt 170
`lp` 57
`lpq` 56, 57
`lpr` 56
`lprm` 56, 57
`lpstat` 57
`ls` 23, 28, 33–35, 39, 46, 92, 124, 182, 187

lynx 105

M

machine
adresse 88
nom 88
macros d'emacs 165
Mail 102
mail 101–103
adresses électroniques 101
envoyer 84
faire suivre 103
logiciel de 102
machine relais (MX) 91
mail transport agent 102
mail user agent 102
MTA 102
MUA 102
mail 65, 66, 84, 102
mailing 84
mailing lists 189
mailx 65, 102
make 107, 129, 142, 163
make 124–130, 140–142, 176, 177
cible 125
compiler avec 140
dans emacs 163
dépendance 126
générer les dépendances 142
règle 126
règle implicite 127
variable 128
variables 127
make directory 37
makeinfo 186
man 148, 183, 185–188
MANPATH 177, 185
messages
d'erreur 24
messages console 99
mget 93
minibuffer
emacs 157
mkdir 37
mode
emacs 156, 167
montage de partition 31
more 63, 67, 187
most 67
Motif 170
mount 31, 37
Mozilla 103
MTA 102
MUA 102
multibyte 69
mutt 102
mv 36, 152
mwm 170
MySql 17

N

Netscape 103
newsgroups 188
NFS 33
nice 51
nohup 59
nom
de fichier 27
nom de machine 88
non authoritative 91
nslookup 89–91
number of record 81

O

objet 130
OpenOffice 17
Opera 103
optimisation à la compilation 135
or 117
ordonnanceur 47

P

PAGER 148
partition 31
montage 31
taux d'occupation 33, 71
PATH 24, 26, 31, 132, 147, 177
pathname expansion 28
pax 72
Perl 14, 17, 19
personnaliser emacs 165–168
pid 48
ping 89
pipes 65

PostScript 56
preprocessing 131
print working directory 35
`printf` 74, 75, 108, 175, 184
priorité
    d'un processus 48, 51
privilèges ..... voir droits
processus 47–55
    des utilisateurs 49
    en arrière-plan 53
    en avant-plan 53, 55
    en tâche de fond 53
    et signaux 52
    interrompre 53
    lister 48
    parents 51
    priorité 51
    rependre 53
    état 48
prompt 22, 77, 144
`prompt` 93
propriétaire 38
préprocesseur du C 132
`ps` 48–50, 52, 79, 150
`PS1` 144
`PS2` 77, 144
`pwd` 35, 44, 92

Q

Qt 170

R

raccourci 44
raccourcis d'emacs 166
racine
    de l'arborescence 31
`RANDOM` 175
`ranlib` 140
`rcp` 100
rechercher remplacer dans emacs . 162
recoller 68
redirection
    du flux de sortie 63
redirections 62–67
    du flux d'entrée 64
    du flux d'erreur 63
    tubes 65–67
regular expression 77, voir expression régulière
regular file 117
remove directory 37
`renice` 51
request for comments 190
`return` 122
reverse engineering 15
revert match 79
Rfc 190
`.rhosts` 94
rlogin 91, 93, 94, 98
`rlogin` 87, 94, 100, 101, 174
`rm` 37, 42–44, 124
`rmdir` 37
root
    utilisateur 29
`rsh` 101, 174
runnable 49
`rwho` 98
règle `make` 126
règle implicite `make` 127
répertoire 35
    attributs 39
    création 37
    effacement 37
    et lien 44
    home 29, 35
    taille 71
    utilisateur 28
réseau
    utilitaires 89

S

sauver dans emacs 161
scheduler 47
scheduler 47
`scp` 96, 101
scripts shell 109–122
search permission 35
sed ..... xi, 116
`sed` 61, 81, 82, 116, 142
sendmail 19
`seq` 118, 119
serveur X 169
service
    cron 58
    daemon 56

impression 56
services
at 57
setfacl 45–47
sh 109
sh 109, 143, 148
shell 21
alias 145
arguments (ligne de commande) 111
boucle for 76, 114
commentaire 109
configuration 143–149
démarrage 148
fonction 121
historique 145
interactif 22
langage 73
prompt 22, 144
scripts 109–122
structures de contrôle 117
test 116
utilisateur 28
variable
arithmétique, 114
modification, 112
variables 110
SHLIB_PATH 139
signal 52
CONT 53
KILL 53
STOP 53
Single Unix Specification 48
sleeping 49
sort 66, 67
sortie standard 62
source 130
ssh 91, 97
ssh 87, 94–98, 101, 174
ssh-agent 97
ssh-copy-id 97
ssh-keygen 96
standard
error 62
input 62
output 62
standard error 62
standard input 62
standard output 62
status 49
stream editor 81
structures de contrôle 117
if 120
su 98
substitute 81
substitution de commande 75
summarize 71
suspendu (processus) 48
système de fichier 30
sélectionner dans emacs 160

tabulation avec make 125
tail 69, 70
talk 100
tape archiver 72
tar 72, 73
tcsh 109, 146
telnet 91, 93, 94, 98
telnet 87
TERM 48, 148
terminal 48
et processus 48, 49
test
-d 116
-eq 117
-f 116
-ge 117
-gt 117
-le 117
-lt 117
-ne 117
-x 116
test en shell 116
tiret 124
toto 183
touch 39
tr 76, 77
traceroute 90
transfert
ascii 92
binaire 92
transfert de fichier 91
translate 76
trier 67
trou noir 65
tubes 65–67
type 24, 147

tâche
de fond 53
en arrière-plan 55

U

uid 28
umask 38, 41
`umask` 41
underscore 79
undo/redo dans emacs 161
uniform resource locator 103
url 103
US patent 18
user identifier 28, 29, 83
utilisateur
changer d' 98
identificateur 28
informations 99
liste des 83, 98
logique 28
processus des 49
root 29
répertoire 28
utilitaires disques et fichiers ... 70–73
utilitaires réseau 89

V

variable
d'environnement 26
du shell 110
`make` 127, 128
shell
arithmétique, 114
modification, 112
`vdir` 182
verbose 72
vi ..... VIII, 108, 152–154
`vi` 6, 58, 143, 147, 152, 153
vim 154, 163

W

w3m 105
warning à la compilation 135
`wc` 69
wget 59, 104
`whatis` 183
`which` 24
`while` 118
`who` 98
`whoami` 29
wildcard 119
wildcard 28
window d'emacs 159
window manager 170
windowmaker 170, 171
word 78
word count 69
World wide web ..... voir www
`write` 99
www 103–105
adresse (url) 103
aspirer un site 104
naviguer en mode texte 105

X

X 17, 154, 168–176
X window 168
`xargs` 85, 86
`xauth` 174
Xaw 170
xdvi 150
`xdvi` 150
`xfig` 125
`xhost` 173
Xlib 168
xloadimage 175, 176
`xloadimage` 176
`.xinitrc` 171
`.xsession` 170
`xsetroot` 175
Xt 170
`xterm` 148
`xv` 185
Xwindow
autorisation 173–174
client 169
display 169
`DISPLAY` 172
démarrage 170
principe général 169
serveur 169
window manager 170
`.xinitrc` 171
`.xsession` 170

xzgv 76

Table des matières

1 Unix et les logiciels libres **1**
1.1 Avant-propos : la naissance d'un logiciel . . . 1
1.1.1 Du source . . . 1
1.1.2 De la portablilité . . . 2
1.2 Unix . . . 5
1.2.1 Historique . . . 5
1.2.2 Architecture et caractéristiques . . . 8
1.3 Les logiciels libres . . . 9
1.3.1 Les différents types de logiciels . . . 9
1.3.2 Historique du projet Gnu . . . 10
1.3.3 Principe de la gpl . . . 11
1.4 Le cas de Gnu/Linux . . . 11
1.4.1 Qu'est-ce que Linux ? . . . 11
1.4.2 Historique . . . 12
1.5 Quelques réflexions sur les logiciels libres . . . 13
1.6 Actualité et avenir des logiciels libres . . . 15
1.6.1 Le problème des « drivers » . . . 15
1.6.2 Le problème des « virus » . . . 16
1.6.3 De l'utopie à la loi du marché . . . 16
1.6.4 Des brevets sur les logiciels . . . 18
1.6.5 Quelques beaux exemples . . . 18

2 Petit guide de survie **21**
2.1 Le shell . . . 21
2.1.1 Qu'est-ce qu'une commande ? . . . 23
2.1.2 « Convivialité »et ergonomie . . . 25
2.1.3 Rudiments sur les variables d'environnement . . . 26
2.1.4 Caractères spéciaux . . . 27
2.1.5 Espaces dans les noms de fichiers . . . 27
2.1.6 Caractères génériques . . . 28
2.2 Utilisateurs . . . 28
2.3 Le système de fichiers . . . 30
2.3.1 Référencement des fichiers et des répertoires . . . 30
2.3.2 Arborescence . . . 31
2.3.3 Privilèges . . . 34
2.3.4 Parcourir l'arborescence . . . 35
2.3.5 Manipuler les fichiers . . . 36
2.3.6 Et les répertoires dans tout ça ? . . . 37
2.3.7 Gestion des supports amovibles . . . 37
2.3.8 Changer les droits . . . 38
2.3.9 Liens . . . 42
2.3.10 Access control list (ACL) . . . 44

2.4 Processus . . . 47
2.4.1 Examiner les processus . . . 48
2.4.2 Modifier le déroulement d'un processus . . . 51
2.5 Quelques services . . . 55
2.5.1 Impression . . . 56
2.5.2 Le service `at` . . . 57
2.5.3 Le service `cron` . . . 58
2.5.4 L'utilitaire `nohup` . . . 59

3 La boîte à outils 61
3.1 Introduction à l'expansion . . . 61
3.2 Redirections et tubes . . . 62
3.2.1 Redirections . . . 63
3.2.2 Les tubes (*pipes*) . . . 65
3.3 Les outils de base . . . 67
3.3.1 Afficher . . . 67
3.3.2 Trier . . . 67
3.3.3 Découper en colonnes . . . 68
3.3.4 Recoller les colonnes . . . 68
3.3.5 Compter . . . 69
3.3.6 Tête-à-queue . . . 69
3.3.7 Utilitaires disques et fichiers . . . 70
3.4 Le shell en tant que langage . . . 73
3.4.1 Afficher des informations avec `printf` . . . 74
3.4.2 Substitution de commande . . . 75
3.4.3 La structure `for` de `bash` . . . 76
3.4.4 Revenons à nos moutons . . . 76
3.5 `grep` et la notion d'expressions régulières . . . 77
3.6 `awk` . . . 79
3.7 `sed` . . . 81
3.8 Études de cas . . . 83
3.8.1 Manipuler la liste des utilisateurs . . . 83
3.8.2 Envoyer des mails . . . 84
3.8.3 Estimer l'occupation de certains fichiers . . . 85

4 Communiquer ! 87
4.1 Concepts à connaître . . . 87
4.1.1 Notion d'adresse IP . . . 87
4.1.2 Notion de DNS . . . 88
4.1.3 Quelques utilitaires réseau . . . 89
4.2 Quatre grands classiques . . . 91
4.2.1 ftp . . . 91
4.2.2 telnet et rlogin . . . 93
4.2.3 Secure shell (ssh) . . . 94
4.3 Outils de communication d'Unix . . . 98
4.3.1 who . . . 98
4.3.2 Changer d'identité . . . 98
4.3.3 write . . . 99
4.3.4 finger . . . 99

4.3.5 talk . . . 100
4.3.6 « remote »commandes . . . 100
4.4 Courrier électronique . . . 101
4.4.1 Format des adresses . . . 101
4.4.2 Mail user agents . . . 102
4.4.3 Faire suivre son courrier . . . 103
4.5 Le ouèbe . . . 103
4.5.1 Format des adresses . . . 103
4.5.2 Wget l'aspirateur . . . 104
4.5.3 Lynx l'extraterrestre . . . 105

5 Développer ! 107
5.1 Éditer un fichier . . . 107
5.1.1 Sans éditeur . . . 108
5.1.2 Avec un éditeur . . . 108
5.2 Faire des scripts en shell . . . 109
5.2.1 Commentaires . . . 109
5.2.2 Choisir l'interpréteur . . . 109
5.2.3 Variables . . . 110
5.2.4 Structure de contrôle et tests . . . 116
5.2.5 Fonctions . . . 121
5.2.6 Input field separator . . . 122
5.2.7 Fiabilité des scripts . . . 123
5.3 Makefile . . . 124
5.3.1 Principes de base . . . 125
5.3.2 Variables . . . 127
5.3.3 Règles implicites et variables prédéfinies . . . 127
5.3.4 Outils sur les variables . . . 128
5.3.5 Options et fonctionnalités diverses . . . 129
5.4 Faire des projets en langage C . . . 130
5.4.1 Remarques préliminaires . . . 130
5.4.2 Étude du cas simple : un seul fichier source . . . 131
5.4.3 Compilation séparée . . . 135
5.4.4 Bibliothèques . . . 136
5.4.5 Se simplifier la vie avec `make` . . . 140

6 Se mettre à l'aise ! 143
6.1 Avec le shell . . . 143
6.1.1 Le prompt . . . 144
6.1.2 Historique des commandes . . . 145
6.1.3 Alias et fonctions . . . 145
6.1.4 Environnement de développement . . . 147
6.1.5 Interaction avec les logiciels . . . 147
6.1.6 Fichiers de démarrage . . . 148
6.1.7 Étude(s) de cas . . . 149
6.2 Avec vi . . . 152
6.3 Avec Emacs . . . 154
6.3.1 Concepts de base . . . 154
6.3.2 Notations utilisées . . . 156

6.3.3 Appeler une commande . . . 157
6.3.4 Manipuler les objets d'Emacs . . . 158
6.3.5 Les tâches basiques . . . 160
6.3.6 Emacs et les Makefiles . . . 163
6.3.7 Personnaliser . . . 165
6.3.8 À l'aide . . . 168
6.4 Avec Xwindow . . . 168
6.4.1 Principe général . . . 169
6.4.2 Les différentes couches . . . 170
6.4.3 Comprendre le démarrage . . . 170
6.4.4 X et le réseau . . . 172
6.4.5 Étude de cas : fond d'écran . . . 175
6.5 Installer des logiciels . . . 176
6.5.1 Installer des binaires . . . 177
6.5.2 Installer des sources . . . 177
6.5.3 Installer des paquet Debian . . . 178

7 À l'aide ! 181
7.1 Quoi fait quoi ? . . . 181
7.1.1 À propos . . . 181
7.1.2 Mais c'est quoi donc ? . . . 183
7.2 Les pages de manuel . . . 183
7.2.1 Une page de manuel . . . 183
7.2.2 Hiérarchie des pages de manuel . . . 184
7.2.3 La variable d'environnement `MANPATH` . . . 185
7.2.4 Recherche exhaustive . . . 186
7.2.5 La commande ultime . . . 186
7.3 Le format info . . . 186
7.4 Le logiciel . . . 187
7.5 Les HOWTO . . . 187
7.6 La documentation en ligne . . . 188
7.6.1 Les newsgroups . . . 188
7.6.2 Les mailing lists . . . 189
7.6.3 Le ouèbe . . . 189
7.6.4 Les foires aux questions . . . 189
7.6.5 Les Request for comments . . . 190

Bibliographie 191

Glossaire 193

Index 197