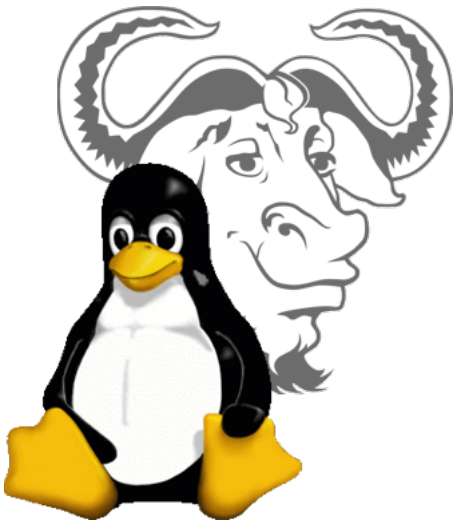
Administration d'un système GNU / Linux

Anthony Labarre



Année académique 2022–2023

Table des matières

Chapitre 1

Introduction

Sommaire

Ce chapitre ne contient aucune information technique et se contente de relater sans trop de détails une partie de l'histoire des systèmes GNU / Linux.

1.1 Qu'est-ce que GNU ? Linux ? GNU / Linux ?

Contrairement à ce que l'expression couramment utilisée pourrait laisser penser, **Linux n'est pas un système d'exploitation : c'est un noyau**. Lorsque l'on parle de Linux dans la vie courante, on veut en fait dire **GNU / Linux**, c'est-à-dire le système qui combine :

1. le **noyau Linux**, un programme qui communique directement avec le matériel ;
2. le **système d'exploitation GNU**, qui gère les ressources en faisant appel au noyau.

Il est donc tout à fait possible d'utiliser Linux avec un autre système (Android, Busybox, ...) ou GNU avec un autre noyau (Hurd, ...). La Figure 1.1 donne une présentation graphique d'un système GNU / Linux, où chaque couche communique directement avec celle qui la contient. Les autres programmes, qualifiés d'utilitaires, dépendent du système d'exploitation pour effectuer leurs tâches.

Sans grande surprise, la mascotte de GNU est un gnou (). Celle de Linux est un pingouin () appelé Tux, pour ***Torvald's Unix***, rendant hommage à son créateur Linus Torvalds et au système **Unix** sur lequel il est basé.

1.2 Logiciels libres

Le système GNU a été créé en 1983 par Richard M. Stallman. L'acronyme GNU est récursif et signifie "**G**NU's **N**ot **U**nix" (acronyme récursif). L'intention de Stallman était de créer un système d'exploitation complètement compatible avec Unix, mais gratuit et "libre" : selon la définition de Stallman, un logiciel est **libre** s'il satisfait les quatre libertés suivantes :

0. le droit d'utiliser le logiciel sans restrictions ;
1. l'accès au code source et le droit d'étudier et modifier le logiciel ;

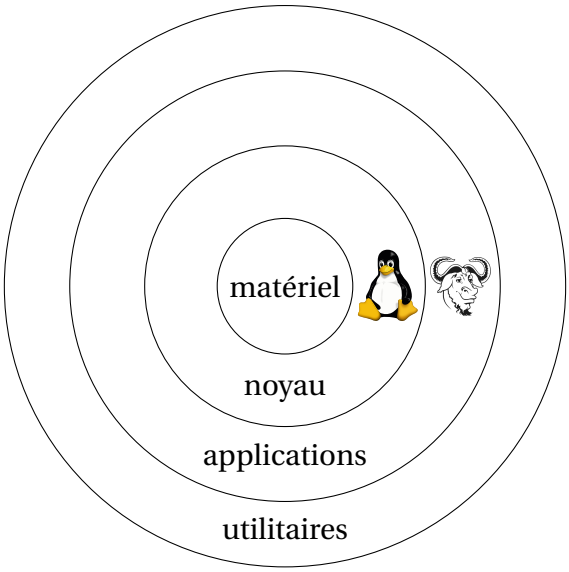


FIGURE 1.1 – La présentation “en couches” d’un système GNU / Linux.

2. la redistribution sans restrictions du logiciel . . .
3. . . . et des modifications qu’on pourrait lui apporter.

Les logiciels libres sont souvent gratuits, ce qui n’aide pas à clarifier leur statut — *free* en anglais pouvant signifier à la fois “gratuit” ou “libre”. Outre l’aspect financier, ces logiciels constituent des ressources précieuses pour les informaticiens : toutes les informations sur leur fonctionnement est disponible, et il est même possible d’y rajouter des fonctionnalités ou de modifier leur comportement puisqu’on a accès à leur code source. De nombreux projets libres invitent d’ailleurs les développeurs à y contribuer (voir `https://sourceforge.net/` ou `https://github.com/`).

Lorsque Stallman a entamé le développement du système GNU, il n’avait pas de noyau pour son système. Plusieurs options ont été étudiées, et le développement d’un noyau nommé Hurd a également été lancé en 1990. C’est finalement le noyau Linux qui a été utilisé pour créer un système exploitable, dont Slackware et Debian sont les plus vieux représentants toujours développés.

Linux a été créé par Linus B. Torvalds en 1992 par Linus B. Torvalds et est disponible indépendamment de GNU sur le site `https://kernel.org/`. Torvalds est également l’auteur du système de gestion de versions `git` (celui de `github`), qu’il a justement développé pour faciliter le développement du noyau Linux.

1.3 Pourquoi GNU / Linux?

Les systèmes GNU / Linux se rencontrent relativement peu chez des utilisateurs grand public, mais leur part du marché augmente considérablement dans les secteurs plus spécialisés. D’après Wikipedia, on retrouve la distribution suivante :

— desktop / laptop : environ 2.87%
— *mainframe* : environ 28%
— embarqué : environ 38.42%

— smartphone / tablette : environ 70.80%
— serveurs : environ 77.4%
— *supercomputers* : 100% (!)

Un informaticien y sera donc inévitablement confronté au cours de sa carrière. De plus, comme on l'a déjà mentionné, ces systèmes sont dérivés du système Unix, qui a donné naissance à la famille de systèmes BSD, Android, OSX/macOS et iOS entre autres. De nombreuses connaissances acquises sous GNU / Linux seront donc transposables à ces systèmes.

1.4 Distributions

GNU / Linux est disponible en plusieurs versions personnalisées, que l'on appelle des **distributions** : il s'agit de systèmes complets directement installables, utilisables, et maintenus par une équipe de développeurs. Il en existe un nombre vertigineux (Distrowatch en recense actuellement 236), et il est même possible de créer sa propre distribution (voir le projet Linux From Scratch).

Ce nombre ne doit pas décourager le débutant, car la plupart des distributions sont très ressemblantes du fait de leurs racines communes. Ainsi, une fois que vous vous serez familiarisé avec une distribution, vous pourrez généralement en changer sans trop de problème si le cœur vous en dit. La distribution que nous utiliserons dans ce cours est Ubuntu (basée sur Debian), qui est plus accessible aux débutants.

1.5 Installation

Installer Ubuntu est assez simple. Pour être plus efficaces et éviter les éventuels problèmes, nous l'installerons sur une machine virtuelle. Comme son nom l'indique, une **machine virtuelle** est un programme qui simule un ordinateur. L'ordinateur simulé est qualifié d'**invité**, tandis que votre machine réelle est qualifiée d'**hôte**. Tous les composants habituels sont simulés sur la machine virtuelle : on y trouvera donc un disque dur virtuel, un lecteur CD virtuel, etc. L'hôte et l'invité peuvent communiquer par divers moyens, et donc partager en particulier des fichiers.

Comme nous utilisons un programme qui simule un ordinateur, les performances seront forcément moins bonnes que si l'on installait Ubuntu directement sur la machine. Pour l'utilisation que nous en ferons ici, les performances resteront cependant acceptables. Nous utiliserons VirtualBox, mais il en existe d'autres.

Chapitre 2

Aperçu d'un système GNU / Linux

Sommaire

Quelles que soient les différences entre les diverses versions, les systèmes GNU / Linux descendent tous des systèmes Unix et en ont hérité les caractéristiques. Ce chapitre présente les particularités de ces systèmes, qu'il est important de connaître pour saisir la suite.

2.1 La ligne de commande

Les systèmes GNU / Linux se distinguent par leur utilisation intensive de la **ligne de commande** : on les écrit dans un programme qualifié de **terminal**. Pour être plus précis, le terminal ne se charge pas directement d'interpréter ces commandes : en général, lorsqu'on lance le terminal, sa première action consiste à exécuter un ***shell***, un autre programme qui permet d'interpréter les commandes que nous allons rentrer. La Figure 2.1 montre un exemple illustrant ces notions.

Exemple 1. La commande suivante nous dit quel *shell* nous sommes en train d'utiliser :

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

La commande `echo` affiche le contenu d'une variable (il existe aussi une commande `printf` qui fait la même chose). Ici, il s'agit de la *variable d'environnement* `SHELL` (voir section 2.5). Dans cet exemple, on détecte que le *shell* utilisé est `bash`, et que le *chemin* vers le programme correspondant est `/bin/bash`.

`echo`
`printf`

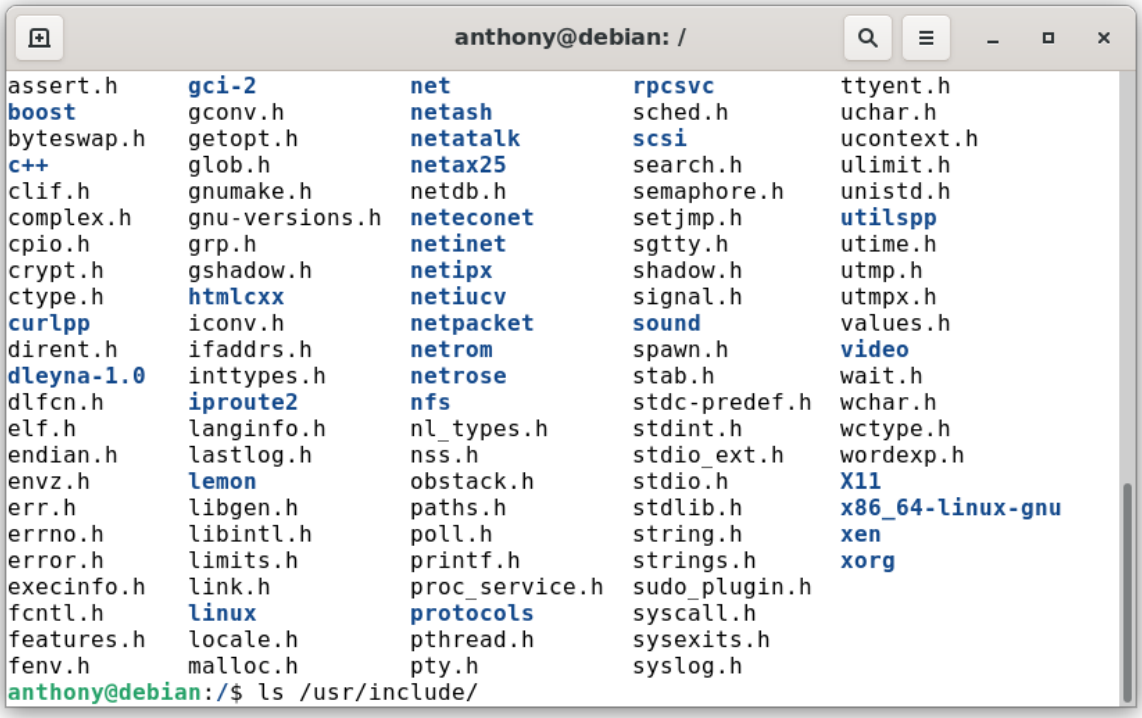


FIGURE 2.1 – Le *shell* `bash` exécuté par le terminal GNOME sous Debian Bullseye (11).

La ligne de commande peut intimider les débutants qui auraient l'habitude d'une interface graphique pour gérer leur système. Pourtant, cette approche présente plusieurs avantages :

— on n'a pas besoin d'une interface graphique, ce qui veut dire que l'on peut par exemple se connecter à un serveur en `ssh` et l'administrer à distance via le terminal ;
— les commandes peuvent se combiner de manière plus ou moins complexe, jusqu'à former de petits programmes que l'on appelle des *scripts* ;
— cette ligne de commande et les programmes de base sont disponibles sur toutes les distributions GNU / Linux, de sorte que l'on ne sera pas désorienté en passant d'un système à un autre.

Ne vous laissez pas décourager par les nombreuses et longues commandes qui peuvent sembler mystérieuses de prime abord. Elles deviendront plus claires au fur et à mesure que vous les utilisez ainsi que le restant du système.

Astuce

Si ce n'est déjà fait, activez la **complétion automatique** sur votre système : cela vous permettra d'être beaucoup plus efficace en complétant automatiquement vos commandes à l'aide de la touche ⇥. Ouvrez votre fichier `.bashrc` :

```
$ nano ~/.bashrc
```

et insérez-y le code :

```
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
```

Sauvegardez le fichier (Ctrl + O), fermez `nano` (Ctrl + X), et validez ensuite les changements à l'aide de la commande :

```
$ source ~/.bashrc
```

Une des lignes directrices des systèmes Unix était de ne faire qu'une seule chose, mais de la faire bien ("*Do One Thing And Do It Well*"). C'est pourquoi on se retrouve de nos jours avec des systèmes comportant énormément de petits programmes basiques, mais qu'on pourra combiner et faire communiquer pour résoudre des problèmes relativement complexes. Il y a donc souvent beaucoup de façons différentes de réaliser une tâche particulière, et il est important de connaître le plus grand nombre possible de ces outils basiques.

Le comportement des programmes est modifiable à l'aide d'options parfois très nombreuses. Ainsi, si la commande que vous utilisez ne fait pas exactement ce que vous voulez, consultez sa documentation pour trouver la ou les options qui vous arrangent. On peut souvent les combiner de manière concise et dans un ordre arbitraire : par exemple, au lieu de lancer la commande `ls` (expliquée plus loin) avec les trois options `-l -S -h`, on peut écrire `ls -lSh`.

Si l'on était obligé d'apprendre toutes les commandes avec toutes leurs options par cœur, personne n'utiliserait ce genre de système. À force de les utiliser, vous finirez par retenir les plus fréquentes, mais personne n'est à l'abri d'un trou de mémoire. Deux moyens existent pour vous aider dans l'utilisation d'un programme :

1. l'option `--help`, qui est disponible pour la grande majorité des programmes sous GNU / Linux, vous donnera des informations concises sur l'utilisation de votre programme ;
2. la **page de manuel** du programme, que l'on obtient par la commande `man programme`, contient beaucoup plus d'informations sur le programme.

Il est donc utile et recommandé de mémoriser *ce que font* les commandes, mais pas leurs options dans les moindres détails.

Astuce

Nous verrons ensemble de nombreuses commandes; pensez à rédiger un résumé pour vous souvenir des noms des commandes et de leurs effets. Ce document contient un index recensant toutes les commandes qui y sont présentées (chapitre 7.3.1).

Remarquons au passage qu'il existe une différence entre les *commandes*, qui sont des instructions spécifiques à un *shell*, et les *programmes* qui sont des logiciels à part entière indépendants du *shell* qu'on utilise. En pratique, cela ne change pas grand-chose du point de vue des utilisateurs que nous sommes, et nous utiliserons ces deux termes de manière interchangeable par abus de langage. Un exemple de cas où l'on verra la différence est que les commandes ne possèdent pas de page de manuel : si `man commande` ne fonctionne pas,
`help` essayez `help commande`.

2.2 Le système de fichiers

Une particularité des systèmes Unix est que "tout est un fichier" : les fichiers traditionnels bien sûr, mais aussi les répertoires, les périphériques, les disques durs, les entrées / sorties, ... Cela surprend de prime abord, mais ce point de vue permet de traiter de manière unifiée beaucoup d'objets a priori très différents.

2.2.1 Structure

Le **système de fichiers** structure les données sur le(s) disque(s). C'est une **arborescence** qui suit les conventions Unix : on retrouvera donc à peu près les mêmes répertoires sur tous les systèmes GNU / Linux. La **racine** de cette arborescence est **/**, qui équivaut au `C:\` sous
`ls` Windows. La commande `ls` nous permet d'obtenir le contenu d'un répertoire.

Exemple 2. Si l'on lance la commande `ls` sur la racine, on obtient tous les répertoires et tous les fichiers qu'elle contient :

```
$ ls /
bin  boot  dev  etc  home  initrd.img  initrd.img.old  lib  lib32
lib64  libx32  lost+found  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var  vmlinuz  vmlinuz.old
```

`tree` On peut aussi visualiser cette arborescence à l'aide de la commande `tree` : l'option `-L` la profondeur d'exploration (ici, un seul niveau).

```
$ tree -L 1 /
/
├── bin -> usr/bin
├── boot
├── dev
├── etc
├── home
```

```
├── initrd.img -> boot/initrd.img-5.10.0-17-amd64
├── initrd.img.old -> boot/initrd.img-5.10.0-16-amd64
├── lib -> usr/lib
├── lib32 -> usr/lib32
├── lib64 -> usr/lib64
├── libx32 -> usr/libx32
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin -> usr/sbin
├── srv
├── sys
├── tmp
├── usr
├── var
├── vmlinuz -> boot/vmlinuz-5.10.0-17-amd64
└── vmlinuz.old -> boot/vmlinuz-5.10.0-16-amd64

22 directories, 4 files
```

Les -> correspondent à des *liens*, un concept qu'on expliquera plus tard. Pour l'instant, on peut retenir qu'il s'agit de raccourcis.

---------- *(fin exemple 2)* --

Le système de fichiers présente plusieurs particularités par rapport à Windows :

- le caractère / sépare les répertoires et équivaut à \ sous Windows ;
- les majuscules et les minuscules importent : par exemple, `/test` et `/Test` sont deux répertoires différents.

Comme sous Windows, **.** désigne le répertoire actuel et **..** désigne le répertoire **parent** (celui qui contient le répertoire actuel). La commande `pwd` donne le chemin complet vers le répertoire actuel. `pwd`

Les répertoires suivants sont particulièrement importants :

- `/dev` (pour **dev**ices) contient le matériel (disques durs, processeurs, ...) ;
- `/etc` contient les fichiers de configuration globaux ;
- `/home` contient les répertoires personnels des utilisateurs (voir section 2.3) ;
- `/mnt` et `/media` contiennent les disques "montés" (voir sous-section 2.2.2) ;
- `/tmp` contient des fichiers temporaires : il est vidé à chaque redémarrage du système.

La commande `cd` (pour ***c**hange **d**irectory*) permet de changer de répertoire. La commande `mkdir` (pour ***make d**irectory*) permet de créer un répertoire ... voire plusieurs répertoire à la fois. Par exemple : `cd` `mkdir`

- `mkdir -p 1/2/3/` crée le répertoire `3` dans le répertoire `2`, lui-même dans le répertoire `1` ; l'option `-p` crée automatiquement les parents qui n'existeraient pas ;

— `mkdir -p test/{premier,second}` crée le répertoire `test/premier` et le répertoire `test/second`.

2.2.2 Partitions et points de montage

Sous les systèmes Unix, "tout est un fichier" … y compris le matériel! Vos périphériques, et en particulier vos disques, sont donc accessibles sous la forme de fichiers dans le répertoire `/dev`; pour avoir accès à leur contenu, il faut les **monter**, c'est-à-dire les raccrocher à l'arborescence via un répertoire appelé **point de montage**.

Pour être plus précis, signalons que tout disque est **partitionné**, c'est-à-dire découpé en morceaux que l'on qualifie de **partitions** et qui peuvent chacune posséder un "type" différent (voir sous-section 2.2.3). On ne monte pas les disques durs, mais bien les partitions qu'ils contiennent. Ceci permet de répartir plusieurs morceaux de l'arborescence sur différentes parties du disque **de manière transparente**, c'est-à-dire sans que l'utilisateur ne s'en aperçoive. Nous expliquerons plus loin les intérêts pratiques de cette approche.

Le fichier `/etc/fstab` contient les informations de montage sur les partitions. Les commandes suivantes sont utiles pour les manipuler :

`mount` — `mount`, pour monter des partitions ou avoir des informations sur ce qui est actuellement monté;

`umount` — `umount`, pour démonter des partitions;

`findmnt` — `findmnt`, pour avoir des informations sur un point de montage.

2.2.3 Types

Indépendamment de la structure du système de fichiers, chaque partition possède un **type** qui détermine le format des données. Les utilisateurs de Windows ont l'habitude des types `fat`, `vfat`, `msdos` et `ntfs`; sous GNU / Linux, on manipule généralement les types `ext3` et `ext4`, mais il en existe bien d'autres.

De nombreux *benchmarks* comparent les performances de nombreux systèmes de fichiers sur des architectures variées. Il n'est pas nécessaire pour un débutant de passer du temps à s'interroger sur ces comparaisons.

Astuce

Si vous ne savez pas quel type de partition choisir, suivez cette règle simple : si vous ne devez manipuler vos données que sous un système GNU / Linux, choisissez `ext4`; sinon, choisissez `vfat`. Dans ce cas, veillez à utiliser des noms de fichiers compatibles, car certains caractères valides pour un type ne le sont pas pour un autre type (par exemple : `mon:fichier` est accepté sous `ext4`, mais pas sous `vfat`).

2.2.4 Fichiers cachés

On remarque que certains noms de fichiers (ou répertoires) commencent par un `.` : il s'agit de **fichiers cachés**, qui n'apparaissent par défaut ni dans un explorateur de fichiers classique, ni lorsqu'on exécute la commande `ls`. Le but n'est pas de protéger ces fichiers en les

rendant invisibles, mais simplement d'alléger le contenu d'un répertoire en cachant des fichiers qui n'ont pas été créés par l'utilisateur, ou qui ne nécessitent pas des manipulations fréquentes. L'option `-a` de la commande `ls` permet de les afficher.

2.3 Utilisateurs, groupes et permissions

GNU est un système **multi-utilisateurs** : plusieurs utilisateurs (en théorie, jusqu'à 2^{32}) peuvent s'y connecter en même temps et partager des ressources. Chaque utilisateur possède un nom et un répertoire personnel dans `/home`. Le raccourci `~` correspond à votre répertoire personnel : si l'utilisateur `pingouin` tape `echo ~`, le terminal affichera `/home/pingouin`.

Le système associe à chaque utilisateur un identifiant unique sous la forme d'un nombre naturel. On appelle ce nombre l'**UID** (***u**ser **id**entifier*). En général, il sera plus simple d'utiliser les noms que les identifiants pour gérer les utilisateurs. Le fichier `/etc/passwd` contient tous les utilisateurs et leurs identifiants.

2.3.1 Le super-utilisateur

Pour des raisons de sécurité, les utilisateurs ordinaires ne peuvent pas tout faire. Les tâches d'administration ((dés)installer des programmes, ou accéder à des fichiers "dangereux") ne peuvent être exécutées que par un **super-utilisateur**, qui s'appelle `root` et possède toujours l'UID 0.

Attention

Ne vous connectez jamais en tant que super-utilisateur ! Les permissions des utilisateurs normaux sont restreintes dans le but de vous éviter des erreurs catastrophiques.

Même si vous êtes seul à utiliser votre système, vous utiliserez deux identités : votre compte d'utilisateur "normal", et le compte de l'utilisateur `root` auquel on peut accéder par deux moyens : la commande `su` (pour ***s**witch **u**ser*), ou la commande `sudo`. `sudo`

Il est recommandé d'utiliser `sudo` pour exécuter des commandes de manière ponctuelle. Ce programme nécessite de préciser dans le fichier de configuration `/etc/sudoers` qui a le droit d'utiliser `sudo` et comment. L'utilisation est ensuite très simple : il suffit de rajouter `sudo` devant la commande que vous voulez exécuter comme super-utilisateur.

Exemple 3. La commande `apt-get upgrade` permet de mettre à jour le système, mais est interdite aux utilisateurs ordinaires :

```
1 $ apt-get upgrade
2 E: Could not open lock file /var/lib/dpkg/lock-frontend - open (13:
  ↪  Permission denied)
3 E: Unable to acquire the dpkg frontend lock
  ↪  (/var/lib/dpkg/lock-frontend), are you root?
```

Pour la lancer, il suffit d'écrire `sudo` devant la commande :

```
1 $ sudo apt-get upgrade
```

```
[sudo] password for user:          # <- entrez le mot de passe de
↪ user
Reading package lists... Done
Building dependency tree... Done
...
```

(fin exemple 3)

On peut aussi obtenir le résultat de l'exemple précédent à l'aide de la commande :

```
$ su -c "apt-get upgrade"
```

Mais dans ce cas, c'est le mot de passe de l'utilisateur `root` qu'il faut rentrer, et pas celui de l'utilisateur qui veut exécuter la commande. Il est donc plus sûr d'avoir recours à `sudo`, puisque ce programme permet de donner des droits plus avancés aux utilisateurs normaux sans jamais diffuser le mot de passe de `root`.

Astuce

Dans ce document, les commandes nécessitant les droits du super-utilisateur seront marquées d'un ★. Remarquons que certaines commandes sont exécutables par n'importe qui, mais pas sur n'importe quel fichier, et il faudra donc aussi parfois les exécuter avec les droits du super-utilisateur. Nous en reparlerons en sous-section 2.3.3.

2.3.2 Groupes

En plus des utilisateurs, GNU utilise aussi des *groupes*, qui nous permettent de créer des catégories d'utilisateurs ; par exemple, dans le cadre d'une université, on veut pouvoir distinguer les étudiants des professeurs et du personnel administratif, et pouvoir permettre à un utilisateur d'effectuer certaines actions uniquement s'il appartient au groupe qui en a le droit. Les groupes ne sont pas exclusifs : un utilisateur peut appartenir à plusieurs groupes.

Un **groupe** est un ensemble d'utilisateurs, identifié par un nom et un **GID** (**g**roup **id**entifier). Chaque utilisateur fait au moins partie de son propre groupe, et peut être membre de n'importe quel nombre d'autres groupes. La commande `groups` affiche les groupes dont vous faites partie. Le fichier `/etc/group` contient tous les groupes avec leurs identifiants et leurs membres.

`groups`

Les commandes suivantes permettent de gérer les utilisateurs :

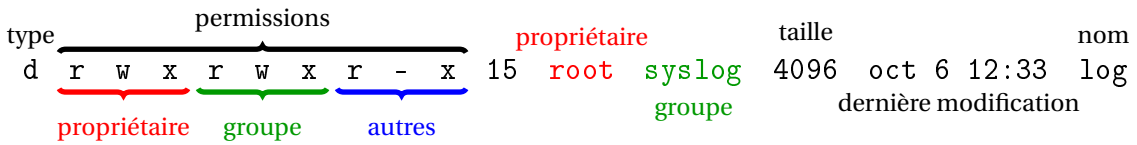
- ★ `adduser nom` : ajoute l'utilisateur avec le `nom` donné ;
- ★ `deluser nom` : supprime l'utilisateur avec le `nom` donné ;
  - son répertoire personnel n'est pas supprimé ; utilisez l'option `--remove-home` ou `--remove-all-files` pour ce faire ;
- ★ `usermod nom` : modifie l'utilisateur avec le `nom` donné ;
- `passwd` : change le mot de passe de l'utilisateur actuel ;

★ `adduser` permet de préciser l'UID qu'on veut recevoir (et `usermod` permet de le modifier). Les commandes ★ `addgroup`, ★ `delgroup` et ★ `groupmod` permettent de gérer les groupes de manière analogue aux commandes sus-mentionnées.

2.3.3 Permissions

Nous avons déjà évoqué le fait que tous les utilisateurs ne sont pas égaux. En plus de cela, le système associe des **permissions** à chaque fichier. Elles déterminent les actions permises sur ce fichier par chaque utilisateur selon son identité : **propriétaire** ou non du fichier (en général, le propriétaire est l'utilisateur qui a créé le fichier, mais cela peut changer), ou membre ou non d'un certain groupe. Les trois permissions les plus courantes sont : `r` (lecture), `w` (écriture), et `x` (exécution). Cette dernière permission est un peu plus particulière : si le fichier est un répertoire, `x` permet de le traverser. Sinon, il s'agit d'un programme, que l'on exécute en rajoutant `./` devant le nom du fichier. L'option `-l` de la commande `ls` affiche les informations détaillées, dont le **type** : `-` pour un fichier ordinaire, `d` pour un répertoire, ou `l` pour un lien.

Exemple 4. La commande `ls -l /var/log` donne le résultat suivant :



Dans cet exemple, on voit que `log` est un répertoire que tout le monde a le droit de lire et de traverser, puisque les permissions `r` et `x` sont activées pour tout le monde. Par contre, seuls le propriétaire du fichier et les membres de son groupe ont le droit de le modifier. Nous omettrons l'explication du 15 pour l'instant.

Il est parfois nécessaire de modifier les propriétés d'un fichier. On peut le faire à l'aide des commandes suivantes :

- `chmod` : change les permissions du fichier ; `chmod`
- `chown` : change le propriétaire et / ou le groupe du fichier ; `chown`
- `chgrp` : change le groupe du fichier. `chgrp`

Il existe également des codes correspondant aux différentes permissions :

code	signification	code	signification	code	signification	code	signification
0	`---`	2	`-w-`	4	`r--`	6	`rw-`
1	`--x`	3	`-wx`	5	`r-x`	7	`rwx`

On utilise alors un chiffre par catégorie. Ainsi, les permissions du fichier de l'exemple 4 auraient pu s'obtenir par la commande `chmod 775 /var/log` (`7 = rwx` pour l'utilisateur, `7 = rwx` pour le groupe, et `5 = r-x` pour les autres).

Enfin, remarquons que même si certains *programmes* peuvent être exécutés par tout le monde, les *fichiers* qu'ils manipulent ne sont pas nécessairement modifiables par tout le monde, selon les permissions associées au fichier. Ainsi, le programme `nano` dont nous parlerons plus loin aura le droit de lire les fichiers de configuration du répertoire `/etc`, mais pas nécessairement de les modifier : cela dépendra de l'utilisateur qui le lancera et de ses droits par rapport au fichier à modifier.

2.4 Processus

Un **processus** est une instance d'un programme en cours d'exécution. Comme les utilisateurs et les groupes, les processus sont identifiés par des numéros : les **PID** (**p**rocess **id**entifier). Et comme les fichiers, ces processus possèdent des propriétaires, qui correspondent à l'utilisateur qui a lancé le processus (et qui peut très bien être différent du propriétaire du programme correspondant).

Tous les programmes, y compris ceux que vous exécutez d'un simple (ou double) clic, peuvent se lancer en ligne de commande à partir du terminal, ce qui est utile pour spécifier d'autres options ou voir les messages d'erreurs quand on veut résoudre des bugs.

Astuce

Si vous lancez `programme` dans le terminal, vous perdrez généralement l'accès à la ligne de commande, c'est-à-dire que vous ne pourrez plus rien taper tant que le programme s'exécutera. Pour éviter cela, lancez le programme avec la commande `programme &`.

`ps` La commande `ps` permet d'obtenir la liste des processus actifs. En général, on l'invoque de la façon suivante : `ps -aux` ; ceci nous donne la liste de tous les processus actifs, avec leur propriétaire et beaucoup d'autres informations.

`top` La commande `top` est souvent plus utile, car elle donne aussi la liste des processus, mais en les "surveillant" : la liste est mise à jour en temps réel selon l'activité des processus. C'est utile entre autres pour savoir "qui" consomme le processeur ou la mémoire.

`pkill` Si un processus est trop consommateur, on peut le "tuer" (l'arrêter) à l'aide des commandes `kill -9 PID` ou `pkill -9 nom_processus`.

Ces commandes envoient un **signal** au processus donné, qui doit alors suivre l'instruction correspondant au signal. `-9` est le signal `SIGKILL`, qui tue les processus, mais il y en a d'autres :

- `SIGSTOP` (`-19`) interrompt le processus ;
- `SIGCONT` (`-18`) reprend l'exécution du processus interrompu ;
- ...

`kill -l` donne la liste de tous les signaux disponibles.

Enfin, quand un processus se termine, il renvoie un **code de retour**. C'est un nombre entier qui permet de vérifier si tout s'est bien passé. Le code de retour du dernier processus qui s'est terminé se trouve dans la variable `?`, dont on affiche le contenu avec `echo $?`.

Astuce

Le code de retour "normal" est 0 et indique que tout s'est bien passé (= "0 problème" !).

2.5 Variables d'environnement

Le système utilise des **variables d'environnement** pour stocker des valeurs facilitant l'exécution de diverses tâches. Elles sont toutes en majuscules ; par exemple :

— `HOME` : le répertoire personnel de l'utilisateur actuel ;
— `USERNAME` : le nom de l'utilisateur actuel ;
— `PATH` : des chemins utiles au système pour exécuter des commandes ;
— ...

Pour afficher la valeur d'une variable, on utilise la commande `echo "$VARIABLE"`. Si l'on veut afficher les noms et les valeurs de *toutes* les variables d'environnement, on utilise la commande `printenv`. Enfin, pour modifier la valeur d'une variable d'environnement, on utilise la commande `export VARIABLE=VALEUR` (pas d'espaces autour de "=" !). `printenv` `export`

Astuce

Si vous possédez des programmes que vous ne pouvez pas installer, vous pouvez les placer dans un répertoire que vous ajouterez à votre variable `PATH` pour pouvoir y accéder facilement dans le terminal. Par exemple :

```
$ mkdir ~/bin  # créer le répertoire qui contiendra vos programmes
$ export PATH=$HOME/bin:$PATH
```

2.6 Fichiers de configuration

De nombreux fichiers de configuration permettent de modifier le comportement des programmes que nous utiliserons. Ils existent parfois en deux exemplaires :

— une version *locale*, située dans votre répertoire personnel : vous pouvez la modifier directement, et ses changements n'affecteront que vous ;
— une version *globale*, située dans `/etc`, dont la modification nécessite généralement les droits du super-utilisateur ; ses changements affectent *tous* les utilisateurs.

On remarquera souvent la présence du caractère `#` dans ces fichiers. Il permet de préciser des commentaires : tout le texte suivant un `#` sur la même ligne est ignoré par le programme qui manipule le fichier (en lecture ou en exécution).

Chapitre 3

Administration basique

Sommaire

Dans ce chapitre, nous allons voir comment sont gérés les programmes sous un système GNU et comment on peut procéder à leur installation, leur suppression, et leur mise à jour. Nous couvrirons aussi quelques tâches fréquentes comme l'édition de fichiers texte.

3.1 Gestion des programmes

Attention

Les informations contenues dans cette section s'appliquent exclusivement aux distributions basées sur Debian, dont Ubuntu. Bien que les principes soient transposables à d'autres distributions, les programmes exacts à utiliser ne seront pas les mêmes.

3.1.1 Principes de fonctionnement

Les distributions basées sur Debian utilisent un système basé sur la notion de *paquets*. Un **paquet** est un fichier à l'extension `.deb`, qui contient tous les fichiers nécessaires au bon fonctionnement d'un programme, ainsi que d'autres informations utiles, notamment les **dépendances**, c'est-à-dire les autres paquets qu'il faut installer pour que le programme fonctionne correctement.

L'installation d'un programme se fait soit à partir d'un paquet que vous avez récupéré vous-même, soit d'un paquet disponible dans la bibliothèque du système. Pour construire sa base de données de paquets disponibles, et détecter les mises à jour, le système utilise des *sources*.

Le système `apt` (pour ***a**dvanced **p**ackaging **t**ool*) est au cœur de la gestion des programmes. Il permet d'installer et de désinstaller des programmes, et également de mettre à jour chaque programme simultanément ou individuellement à l'aide de commandes très simples. `apt`

3.1.2 Les sources

Les **sources** d'`apt` contiennent une liste de **dépôts**, qui sont des adresses où aller chercher les paquets (généralement des URL, mais on peut aussi renseigner des sources locales comme des lecteurs CD). On les trouve dans le fichier `/etc/apt/sources.list`, et ce sont ces adresses que `apt` vérifie lorsqu'on exécute la commande ★ `apt-get update`, qui met à jour la base de données du système `apt`.

Différents types de dépôts sont disponibles. Sous Debian, il est particulièrement important de les connaître, car Debian ne propose par défaut que des logiciels libres, et votre matériel peut nécessiter l'installation de pilotes propriétaires pour fonctionner correctement. L'équipe de Debian a heureusement prévu cela, et propose un large éventail de paquets propriétaires dans les dépôts `contrib` et `non-free`. Il est fortement conseillé de les rajouter à vos sources, pour avoir accès à un éventail plus large (et parfois indispensable) de logiciels. Pour renseigner un dépôt, le format à suivre est le suivant :

`deb URL NOM_DISTRIBUTION TYPES` (pour les programmes)

`deb-src URL NOM_DISTRIBUTION TYPES` (pour les sources des programmes)

où les types possibles sont `main` (les programmes principaux), `contrib` (des paquets ne faisant pas partie officiellement de la distribution) et `non-free` (des logiciels propriétaires).

Exemple 5. Voici un exemple de fichier obtenu après l'installation de la version 11 de Debian :

```
1 deb http://deb.debian.org/debian/ bullseye main non-free contrib
2 deb-src http://deb.debian.org/debian/ bullseye main non-free contrib
3
4 deb http://security.debian.org/debian-security bullseye-security main non-free
  ↪ contrib
5 deb-src http://security.debian.org/debian-security bullseye-security main
  ↪ non-free contrib
6
7 deb http://deb.debian.org/debian/ bullseye-updates main non-free contrib
8 deb-src http://deb.debian.org/debian/ bullseye-updates main non-free contrib
```

Attention

La mise à jour des paquets nécessite que les informations de version soient à jour : autrement dit, que le système soit au courant qu'il existe de nouvelles versions des paquets installés. Il est donc nécessaire d'exécuter au préalable la commande ★ `apt-get update`, qui récupère ces informations.

`apt-src` Si l'on veut obtenir le code source d'un paquet, on utilisera la commande `apt-src`, qui fonctionne comme `apt-get`.

3.1.3 Installation et désinstallation de programmes

Les commandes essentielles à connaître pour installer un paquet disponible dans la bibliothèque du système sont :

— pour installer un paquet : ★ `apt-get install nom_du_paquet`

— pour supprimer un paquet :
  — ★ `apt-get remove nom_du_paquet`, ou
  — ★ `apt-get purge nom_du_paquet` pour supprimer en plus ses fichiers de configuration ;
— pour mettre à jour un paquet : ★ `apt-get upgrade nom_du_paquet` ;
— pour mettre à jour tous les paquets : ★ `apt-get upgrade` ;

Si vous avez téléchargé un paquet nommé `paquet.deb`, vous pouvez l'installer à l'aide de la commande ★ `apt install chemin/vers/paquet.deb`. Attention :

1. c'est bien `apt`, pas `apt-get` ;
2. écrivez ★ `apt install ./paquet.deb`, et pas ★ `apt install paquet.deb`, si vous êtes dans le répertoire contenant le paquet.

Que vous utilisiez `apt` ou `apt-get`, la commande installera automatiquement toutes les dépendances nécessaires au bon fonctionnement de votre programme. La seconde méthode fonctionne également pour mettre à jour un paquet déjà installé.

3.1.4 Trouver des programmes à installer

De nombreux programmes que vous utilisez peut-être déjà sur d'autres systèmes existent sous Debian, et il est assez facile de deviner le nom du paquet correspondant qui n'est pas toujours exactement celui du programme (par exemple, le navigateur `firefox` se trouve dans le paquet `firefox-esr`, et le navigateur `chrome` dans le paquet `chromium`). La complétion automatique peut aussi nous aider ; mais comment faire quand on cherche un programme permettant d'accomplir une certaine tâche sans connaître son nom ?

La solution est simple : le programme `apt-cache` nous permet de chercher des informations sur des paquets installables. Il suffit d'exécuter la commande `apt-cache search` suivie des mots-clés qui vous intéressent. `apt-cache`

Dans certaines situations, on est amené à exécuter un programme que l'on s'est procuré autrement que par le système de paquets, et dont on ne connaît pas les dépendances (typiquement : une archive `.tar.gz` dont on doit éventuellement compiler les sources). Il arrivera régulièrement qu'on ait besoin d'un fichier avec un nom particulier, que la commande `apt-cache search` ne nous permettra pas de trouver. On aura alors recours à un autre programme qui s'utilise de la même façon : `apt-file search` suivi du nom du fichier voulu nous donnera la liste des paquets qui le contient. Attention, il faut que la base de données de `apt-file` soit à jour, ce qu'on effectue par la commande ★ `apt-file update`. `apt-file`

3.2 Visualisation et édition de texte

Comme nous serons amenés à chercher et à modifier des informations dans des fichiers de configuration, il est nécessaire de connaître les outils qui permettent de le faire en ligne de commande. Les commandes suivantes seront très utiles :

— `cat fichier` : affiche le contenu d'un fichier en entier dans le terminal ; `cat`
— `less fichier` : affiche le contenu d'un fichier de manière interactive (on peut le faire défiler). La commande `more` fait à peu près la même chose. `less` `more`

vi
emacs
nano

De nombreux éditeurs de texte sont disponibles. Les plus complets et connus sont `vi` et `emacs`, mais ils sont assez complexes à utiliser pour un débutant. Si vous en avez l'occasion, préférez plutôt `nano` à la place, qui suffira. Il existe également des éditeurs non-interactifs, permettant d'automatiser la manipulation du texte, comme `sed` et `awk` qui seront utiles dans des scripts ou des commandes plus ou moins complexes.

3.3 Tâches diverses

3.3.1 Recherche

Trouver des programmes. Grâce à la variable `PATH` et à la complétion automatique, il devient de moins en moins nécessaire de savoir où exactement se situent les programmes sur notre système. Mais dans le cas où ce serait nécessaire, deux commandes permettent d'obtenir cette information :

which
whereis

- `which programme` renvoie le chemin vers le programme passé en paramètre ;
- `whereis programme` renvoie le chemin vers le programme passé en paramètre, ainsi que les sources et la page de manuel du programme.

Trouver des fichiers. Si l'on veut trouver l'emplacement d'un fichier, on a le choix entre deux commandes. L'option la plus simple est d'utiliser `locate motif`, qui renvoie la liste de tous les chemins contenant le motif donné en paramètre.

locate

L'autre option, plus complexe à utiliser mais beaucoup plus puissante, est la commande `find`. Il faut d'abord lui préciser où chercher ses informations à l'aide d'un chemin, et ensuite lui dire quoi chercher. Par exemple, si l'on veut trouver tous les fichiers `.cpp` dans notre répertoire personnel, on écrira :

find

```
1 $ find ~ -name "*.cpp"
```

Chapitre 4

Communication entre processus

Sommaire

Ce chapitre couvre la manipulation des processus, ainsi que les manières dont on peut récupérer leurs résultats ; soit pour les enregistrer dans des fichiers, soit pour les communiquer à d'autres processus pour effectuer d'autres tâches.

4.1 Entrées et sorties

L'**entrée** d'un processus est le moyen utilisé pour lui communiquer des données. La **sortie** d'un processus est le moyen qu'il utilise pour écrire des résultats. On distingue trois types de flux qui sont numérotés :

0. l'**entrée standard** (ou `STDIN`) est le clavier (`/dev/stdin`) ;
1. la **sortie standard** (ou `STDOUT`) est l'écran (`/dev/stdout`) ;
2. l'**erreur standard** (ou `STDERR`) est aussi l'écran (`/dev/stderr`).

La Figure 4.1 illustre les interactions d'un processus avec les flux standards.

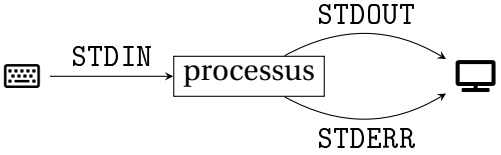


FIGURE 4.1 – Les flux standards gérés par un processus.

4.1.1 Redirections simples

La plupart des commandes affichent leur résultat sur la sortie standard, mais on peut facilement modifier ce comportement, par exemple pour sauvegarder le résultat dans un fichier au lieu de l'afficher. Il suffit pour cela de **rediriger** sa sortie avec l'opérateur >.

Exemple 6. La commande `ls` nous permet d'afficher le contenu d'un répertoire ; si on veut l'enregistrer, il suffit simplement de rediriger sa sortie vers un fichier comme suit :

```
$ ls -l > contenu_répertoire.txt
```

La commande `processus > sortie` redirige ce que `processus` devrait écrire sur `STDOUT` vers le fichier `sortie`. Si `sortie` existe, son contenu est remplacé ; sinon, `sortie` est créée par la commande. Si l'on veut *ajouter* un résultat à une sortie qui existe déjà, on remplace simplement l'opérateur > par >> pour la redirection.

Il est également possible de rediriger les *entrées* : ceci est nécessaire quand le programme que l'on utilise s'attend à recevoir des données sur l'entrée standard, et que l'on préfère l'utiliser en mode non-interactif, en lui fournissant des données provenant d'un fichier. Il nous suffit dans ce cas d'utiliser l'opérateur <.

Exemple 7. Pour installer un nouveau système identique à l'ancien, on peut enregistrer les noms des paquets installés :

```
$ dpkg --get-selections > mes_paquets
```

et puis les réinstaller sur le nouveau système :

```
$ sudo dpkg --set-selections < mes_paquets
$ sudo apt-get dselect-upgrade
```

4.1.2 Redirections multiples

Il est parfois utile de sauvegarder les résultats qui nous intéressent dans un fichier, mais également les erreurs qui ont pu se produire dans un autre fichier (généralement avec l'extension `.log`), qu'on consultera en cas de problème. Cela se fait toujours avec l'opérateur >, mais en précisant cette fois le numéro de la sortie que l'on veut rediriger : > ou `1>` redirigera `STDOUT`, et `2>` redirigera `STDERR`. Bien entendu, le rajout de ces numéros fonctionne aussi avec l'opérateur >>.

`du`

Exemple 8. La commande `du` (pour ***disk usage***) permet d'obtenir des informations sur l'espace utilisé sur le système de fichiers.

```
$ du -sh /
# erreurs
# espace utilisé sur le disque
$ du -sh / > consommation.txt
```

```
# erreurs; espace enregistré dans consommation.txt
$ du -sh / > consommation.txt 2> log.txt
# aucun message; espace -> consommation.txt; erreurs -> log.txt
```

(fin exemple 8)

Enfin, si on ne veut ni voir les erreurs, ni les sauver dans un fichier, il est possible de les rediriger vers le fichier spécial `/dev/null`, que l'on peut voir comme une poubelle.

Exemple 9. La redirection vers `/dev/null` supprime les messages d'erreurs :

```
$ du -sh / > consommation.txt 2> /dev/null
# espace dans consommation.txt; erreurs à la poubelle
```

4.2 Les *pipes*

Les ***pipes***[1] permettent aux processus de communiquer entre eux. Ils constituent un mécanisme très puissant pour accomplir des tâches relativement complexes en combinant des commandes simples.

4.2.1 Premier exemple

En guise d'exemple, utilisons les *pipes* pour résoudre le problème suivant. Nous avons vu en section 4.1 comment rediriger la sortie standard vers un fichier ; mais malheureusement, cela nous empêche de voir ce que le processus a produit comme sortie. Comment faire pour voir la sortie et en même temps l'écrire dans le fichier ?

La solution est fournie en partie par la commande `tee`, qui permet de lire des données de l'entrée standard et de les écrire dans un fichier sans faire de redirection ; autrement dit, `tee` ne supprimera pas l'affichage : elle le lira à l'écran en même temps que nous, mais en profitera pour le recopier en même temps dans un fichier. `tee` en elle-même n'est pas suffisante, car elle ne peut lire les données que sur l'entrée standard ; et l'on ne peut pas rediriger cette entrée vers un fichier, sinon on ne la verra pas à l'écran. Il faut donc que la sortie de notre commande devienne l'entrée de `tee`, et c'est exactement ce que font les *pipes*, représentés par le symbole `|`. Ainsi, pour afficher le contenu détaillé d'un répertoire et en même temps sauvegarder ces informations dans un fichier, on utilisera une des deux commandes suivantes : `tee`

```
$ ls -l | tee sortie.txt      # équivalent de >
$ ls -l | tee -a sortie.txt   # équivalent de >>
```

Remarquons qu'il était possible dans ce cas de s'en sortir autrement : on aurait pu rediriger la sortie vers le fichier (`ls -l > sortie.txt`), et puis afficher le contenu de ce dernier (`cat sortie.txt`). Cette solution est moins efficace, puisqu'on doit traiter la sortie deux fois, alors que le *pipe* nous permet de ne faire qu'une seule passe sur les données.

Il est important de bien saisir la différence entre le fonctionnement des redirections et celui des pipes. Comme l'illustre la Figure 4.2 :

1. En français : "tuyau" ou "tube" … mais tout le monde dit *pipe* (à l'anglaise).

— une redirection *remplace* une entrée ou une sortie par un fichier ; ainsi, `processus1 > fichier && processus2 < fichier` redirige `STDOUT` vers `fichier` pour `processus1` et remplace `STDIN` par `fichier` pour `processus2` ;
— un *pipe établit un canal de communication*, sans stockage intermédiaire, entre les deux processus ; ainsi, `processus1 | processus2` ne remplace pas `STDOUT` pour `processus1`, mais l'utilise comme remplacement de `STDIN` pour `processus2`.

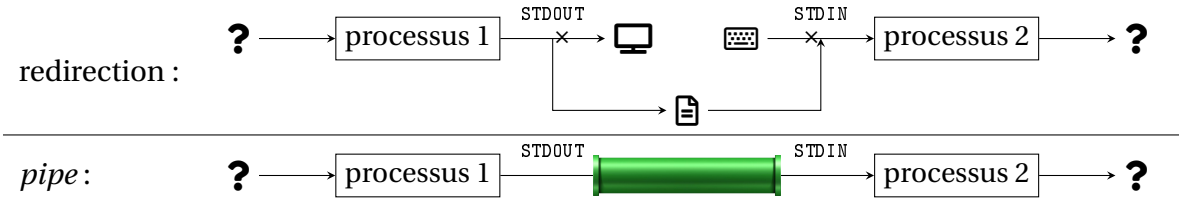


FIGURE 4.2 – Différence entre une redirection et un *pipe*.

Il est possible de combiner autant de pipes que l'on veut dans une seule commande :

```
$ processus_1 | processus_2 | processus_3 | ... | processus_n
```

Dans ce cas, la sortie du processus avant le *pipe* numéro `i` devient l'entrée du processus suivant ce même *pipe*.

4.2.2 Quand utiliser les *pipes* ?

La réponse courte est : presque toujours, car ils possèdent de nombreux avantages, et les situations où ils ne peuvent pas nous aider sont relativement rares. Comme on l'a vu plus haut, il est souvent possible de s'en sortir autrement ; mais on se rend vite compte que les solutions alternatives sont moins efficaces.

`lsof`

Exemple 10. Cherchons tous les fichiers ouverts par `firefox`. On pourrait afficher tous les fichiers ouverts avec `lsof`, rediriger son résultat vers un fichier, puis demander à `grep`, que l'on présentera plus loin, d'afficher toutes les lignes de ce fichier contenant `firefox`, ce qui donnerait ceci :

```
$ lsof > fichiers_ouverts.txt
$ grep firefox fichiers_ouverts.txt
```

Cela fonctionne, mais on produit un fichier gros (58M) et inutile, puisqu'on n'en a besoin que pour la seconde commande. Au lieu de cela, on peut simplement écrire :

```
$ lsof | grep firefox
```

… qui produit le même résultat mais sans fichier intermédiaire et en une seule passe.

Ainsi, si vous n'avez pas besoin des données intermédiaires, utiliser les *pipes* ne présente que des avantages :

— pas de données grosses et inutiles à stocker ;
— c'est plus rapide : on peut traiter chaque ligne directement au lieu d'attendre le résultat complet ;

— c'est plus flexible : les programmes peuvent traiter des données d'un autre processus … et les envoyer à un autre processus … et les envoyer à un autre processus ….

Par contre, si vous avez besoin de réutiliser les données intermédiaires, les *pipes* ne suffiront pas puisque ces données sont perdues et qu'on ne peut les lire qu'une seule fois.

De plus, il existe des cas où l'on ne peut pas les utiliser directement : étant donnée la manière dont les *pipes* agissent sur `STDOUT` et `STDIN` (rappelez-vous de la Figure 4.2 page 24), si l'un des processus n'écrit pas par défaut sur `STDOUT`, ou ne lit pas par défaut sur `STDIN`, il y aura des redirections à effectuer.

4.2.3 `xargs`

Certains programmes acceptent uniquement des arguments, et **pas** des données provenant de `STDIN`. Cela pose problème pour les *pipes*, qui peuvent uniquement rediriger des entrées et sorties. Dans ces cas-là, on utilise le programme `xargs`, qui transforme les données de `STDIN` en arguments pour une autre commande.

Exemple 11. Cherchons tous les fichiers contenant le mot `vmlinuz` et affichons leurs propriétés avec `ls -l` :

```
$ locate vmlinuz | ls -l
# échoue: équivaut à ls -l
$ locate vmlinuz | xargs ls -l
# ok
```

Le fonctionnement de `xargs` est le suivant : supposons que `commande_1` produit le résultat `ligne_1`, `ligne_2`, …. Quand on écrit `commande_1 | xargs commande_2`, cela revient à exécuter `commande_2 ligne_1 ligne_2 ...`. Les lignes ainsi produites deviennent le **dernier** argument de `commande_2` (consultez `man xargs` pour obtenir d'autres comportements).

4.3 Les filtres

On qualifie de **filtre** tout programme qui transforme des données de type texte. Ces filtres ne modifient généralement pas le fichier donné en entrée, mais appliquent un traitement particulier à son contenu pour en produire un autre. Cette définition est très large, mais désigne souvent un des programmes suivants, utilisé en conjonction avec des *pipes* :

— `cut` sélectionne des portions d'un affichage structuré, qu'on pourrait assimiler à des colonnes ; `cut`
— `grep`, qui filtre toutes les lignes d'un fichier contenant une certaine expression ; `grep`
— `head` affiche les premières lignes d'un fichier ; `head`
— `nl`, pour ***n**umber **l**ines*, numérote les lignes d'un fichier ; `nl`
— `shuf` mélange aléatoirement les lignes d'un fichier ; `shuf`
— `sort` trie les lignes d'un fichier ; `sort`
— `tail` affiche les dernières lignes d'un fichier ; `tail`
— `tr`, pour ***tr**anslate*, efface ou remplace des caractères dans un fichier. `sed` et `awk` sont des outils similaires, mais beaucoup plus avancés ; `tr`

`uniq` — `uniq`, très souvent utilisé avec `sort`, élimine les doublons (c'est-à-dire des lignes identiques qui se suivent) dans un fichier ;

`wc` — `wc`, pour ***word count***, compte les caractères, mots ou lignes d'un fichier ;

Remarquons que tous ces programmes ne nécessitent pas forcément l'usage d'un *pipe*. Par exemple, `cat fichier | wc -l` affiche le nombre de lignes d'un fichier, mais on aurait pu obtenir le même résultat simplement avec `wc -l fichier`.

4.3.1 `grep`

`grep` est un des programmes incontournables de GNU et mérite que l'on s'y attarde. `grep` permet de trouver des chaînes ou des expressions dans des fichiers texte. Son format est très simple : `grep texte fichiers`.

Exemple 12. Cherchons toutes les occurrences de "Quasimodo" dans le texte de "Notre-Dame de Paris" :

```
$ grep Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
# affiche toutes les lignes contenant Quasimodo
$ grep -c Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
241 # affiche le nombre de lignes contenant Quasimodo
$ grep -o Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt
# affiche chaque occurrence sur une ligne séparée
Quasimodo
Quasimodo
...
$ grep -o Quasimodo Hugo\ -\ Notre-Dame\ de\ Paris.txt | wc -l
244 # le nombre d'occurrences de Quasimodo dans le livre
```

Par défaut, `grep` est sensible à la casse et ignorerait donc dans les commandes ci-dessus les occurrences de "QUASIMODO" ou de "quasimodo". On peut modifier ce comportement avec l'option `-i` :

```
$ grep -c Quatre Hugo\ -\ Notre-Dame\ de\ Paris.txt
9 # le nombre d'occurrences de "Quatre"
$ grep -c -i Quatre Hugo\ -\ Notre-Dame\ de\ Paris.txt
120 # le nombre d'occurrences de "Quatre" et "quatre" et ...
```

Attention : `grep` ne considère pas "Quatre" comme un mot isolé ! Parmi les occurrences, on retrouvera donc aussi "Quatrelivres", "Quatre-Couronnes", "Quatre-Nations", ...

Il existe quelques variantes utiles de `grep` (qui ne sont pas nécessairement toutes installées par défaut) qu'il est bon de connaître :

`pgrep` — `pgrep` : recherche un motif parmi les noms des processus ;

`agrep` — `agrep` : recherche de motifs approximatifs (avec des "erreurs") ;

`ngrep` — `ngrep` : recherche de motifs dans le trafic réseau ;

`zipgrep` — `zipgrep` : `grep` pour les fichiers zip ... sans devoir les décompresser !

— ...

Chapitre 5

Expressions régulières

Sommaire

Les expressions régulières constituent un outil très puissant pour chercher et modifier du texte. Leur intérêt réside dans le fait qu'elles permettent de spécifier la *structure* de l'objet à chercher, plutôt qu'un mot (ou une phrase) exact(e). On peut par exemple définir une expression qui correspondrait à la structure d'une adresse e-mail, ou d'un numéro de téléphone : ceci permet ensuite de trouver *toutes* les adresses e-mail, ou *tous* les numéros de téléphone d'un document, au lieu d'une adresse ou d'un numéro fixe.

5.1 Les bases

Une **expression régulière** est une chaîne qui encode un *modèle* de texte. Pour les illustrer, examinons un premier exemple simple.

Exemple 13. L'expression régulière suivante représente les dates au format "année-mois-jour", en utilisant quatre chiffres pour les années, deux pour les mois, et deux pour le jour :

`[0-9][0-9][0-9][0-9]-[0-1][0-9]-[0-3][0-9]`
(année) (mois) (jour)

Les crochets permettent de préciser un intervalle de valeurs autorisées : `[0-9]` représente donc n'importe quel chiffre, et est répété quatre fois puisqu'on exige que l'année soit présentée sur quatre chiffres. Les champs "mois" et "jour" contiennent chacun deux chiffres, avec comme restriction supplémentaire que le mois ne peut pas commencer par un chiffre supérieur à 1 et le jour ne peut pas commencer par un chiffre supérieur à 3. Enfin, les deux - autour du mois correspondent au format demandé.

Remarquons que l'expression donnée dans l'exemple 13 ne permet pas d'ignorer les dates incorrectes comme le 30 février. Il est possible d'écrire une autre expression régulière pour régler ce problème, mais elle sera beaucoup plus compliquée : certains mois ont 30 jours, d'autres 31, et le mois de février en a 28 ou 29 selon l'année. Généralement, cela ne se justifie pas : on essaie de construire des expressions régulières correctes, mais leur but est de récupérer les informations respectant une certaine structure, pas de vérifier leur validité. Pour

cette vérification, on peut sans mal intégrer nos expressions à des scripts rédigés dans des langages de programmation qui s'y prêtent mieux.

Symboles et classes de caractères. Les expressions régulières utilisent des symboles et un format qu'il est indispensable de connaître. On peut y insérer des caractères particuliers que l'on a envie de trouver dans le texte, ou utiliser des symboles (voir le Tableau 5.1).

Symboles fréquents		Négations	
`[0-9]` ou `\d`	les chiffres	`\D`	tout sauf un chiffre
`[a-z]`	les lettres minuscules		
`[A-Z]`	les lettres majuscules		
`\w`	les lettres et les chiffres	`\W`	tout sauf une lettre ou un chiffre
`\s`	les espaces	`\S`	tout sauf un espace

TABLEAU 5.1 – Les symboles utilisés par les expressions régulières pour représenter les caractères.

Certaines de ces classes peuvent se restreindre ou se combiner. Par exemple, `[a-k]` n'accepte que les lettres minuscules entre "a" et "k", et `[a-zA-Z]` autorise toutes les lettres, minuscules ou majuscules (mais pas accentuées comme les "é", "è", ...).

Répétitions. En plus de cela, on peut également préciser combien de fois chaque symbole doit être répété. Il suffit de faire suivre l'expression d'un des symboles suivants, sans espace :

- `?` : 0 ou 1 fois ;
- `+` : au moins une fois ;
- `*` : autant de fois qu'on veut ;
- `{n}` : exactement `n` fois ;
- `{n,}` : au moins `n` fois ;
- `{,m}` : au plus `m` fois ;
- `{n,m}` : entre `n` et `m` fois ;

Exemple 14. L'expression régulière pour les dates de l'exemple 13 peut se simplifier comme suit :

```
\d{4}-[0-1]\d-[0-3]\d
```

Emplacements. On peut aussi préciser des conditions sur les emplacements où les résultats doivent apparaître, avec les opérateurs suivants :

- `^expression` : l'expression doit apparaître au début d'une ligne de texte ;
- `expression$` : l'expression doit apparaître à la fin d'une ligne de texte ;
- `\b`, avant ou après l'expression : marque la présence d'un mot. Par exemple, `\b[0-9]` décrit un chiffre situé juste après un mot, et repérera le "2" de "Terminator 2" ;
- `\B`, avant ou après l'expression : marque l'absence d'un mot. Par exemple, `\B[0-9]` décrit un chiffre qui n'est pas situé juste après un mot.

Caractères d'échappement. Il arrive parfois que l'on recherche dans un texte des caractères spéciaux qui ont une signification pour les expressions régulières. Pour que l'interpréteur ne se trompe pas, il faut rajouter un \ devant ce caractère (par exemple : `\$`).

Astuce

Le site https://regexr.com/ vous permet de tester en direct vos expressions régulières, avec le texte de votre choix.

5.2 Utilisation

Les expressions régulières sont rarement utilisables dans des programmes classiques de traitement de texte, mais se rencontrent partout dans les programmes GNU, et sont particulièrement utiles avec `grep` et `sed`, entre autres.

Pour expliquer à `grep` que le motif fourni en paramètre est une expression régulière et pas une chaîne "normale", on utilise l'option `-E` ou `-P`.

Exemple 15. Les variables dans les fichiers de configuration se définissent sous la forme "MOT=VALEUR". Pour trouver toutes les lignes d'un fichier contenant cette instruction, on utilise l'expression `\w+=`. Par exemple :

```
$ grep -E "\w*=" /etc/adduser.conf
DSHELL=/bin/bash
DHOME=/home
GROUPHOMES=no
LETTERHOMES=no
SKEL=/etc/skel
FIRST_SYSTEM_UID=100
LAST_SYSTEM_UID=999
FIRST_SYSTEM_GID=100
LAST_SYSTEM_GID=999
FIRST_UID=1000
LAST_UID=59999
FIRST_GID=1000
LAST_GID=59999
USERGROUPS=yes
USERS_GID=100
DIR_MODE=0755
SETGID_HOME=no
QUOTAUSER=""
SKEL_IGNORE_REGEX="dpkg-(old|new|dist|save)"
#EXTRA_GROUPS="dialout cdrom floppy audio video plugdev users"
#ADD_EXTRA_GROUPS=1
#NAME_REGEX="^[a-z][-a-z0-9_]*\$"
```

Si l'on ne veut pas des lignes commentées, c'est-à-dire que l'on veut ignorer les lignes qui commencent par un `#`, on rajoute `^` devant l'expression pour préciser que le mot doit démarrer la ligne :

```
$ grep -E "^\w*=" /etc/adduser.conf
```

```
DSHELL=/bin/bash
DHOME=/home
GROUPHOMES=no
LETTERHOMES=no
SKEL=/etc/skel
FIRST_SYSTEM_UID=100
LAST_SYSTEM_UID=999
FIRST_SYSTEM_GID=100
LAST_SYSTEM_GID=999
FIRST_UID=1000
LAST_UID=59999
FIRST_GID=1000
LAST_GID=59999
USERGROUPS=yes
USERS_GID=100
DIR_MODE=0755
SETGID_HOME=no
QUOTAUSER=""
SKEL_IGNORE_REGEX="dpkg-(old|new|dist|save)"
```

(fin exemple 15)

5.3 Les groupes

Les **groupes**, dans le contexte des expressions régulières, permettent de définir des morceaux d'une expression que l'on a envie de récupérer. Ces groupes sont très simples à préciser : il suffit de placer entre parenthèses les parties de l'expression qui nous intéressent.

Exemple 16. Modifions l'expression régulière de l'exemple 15 pour isoler les noms des variables : il suffit de rajouter des parenthèses autour du nom, comme ceci : `^(\w+)=`. Pour n'afficher que les valeurs de ces groupes, et pas le reste, il est plus commode d'utiliser `pcregrep` avec l'option `-o` suivie du numéro du groupe, et de trier le résultat pour qu'il soit plus lisible :

pcregrep

```
$ pcregrep -o1 "^(\w*)=" /etc/adduser.conf | sort
DHOME
DIR_MODE
DSHELL
FIRST_GID
FIRST_SYSTEM_GID
FIRST_SYSTEM_UID
FIRST_UID
GROUPHOMES
LAST_GID
LAST_SYSTEM_GID
LAST_SYSTEM_UID
LAST_UID
```

```
14 LETTERHOMES
15 QUOTAUSER
16 SETGID_HOME
17 SKEL
18 SKEL_IGNORE_REGEX
19 USERGROUPS
20 USERS_GID
```

(fin exemple 16)

Chapitre 6

Scripts *shell*

Sommaire

Une part non négligeable de l'administration d'un système GNU consiste à développer et à maintenir des scripts automatisant les tâches. Il s'agit de petits programmes écrits dans un langage associé au *shell* utilisé.

Les *shells* étant nombreux et possédant chacun leur propre dialecte, il est vain d'espérer tous les couvrir. Nous ne parlerons ici que des scripts destinés au *shell* `bash`, car c'est le plus répandu ; de plus, les dialectes de chaque *shell* étant relativement proches, il ne sera pas difficile d'apprendre celui d'un autre *shell* sur base des connaissances acquises sur les scripts `bash`.

Astuce

Le site Advanced Bash-Scripting Guide constitue une excellente référence concernant les scripts `bash`. Comme le nom le suggère, il explore en profondeur le sujet, mais reste tout à fait accessible aux débutants (jusqu'à un certain point).

6.1 Intérêt des scripts *shell*

On peut légitimement se poser la question de l'intérêt de ces langages spécialisés à une époque où de nombreuses alternatives modernes sont disponibles et probablement connues du lecteur. Bien que les tâches réalisables dans le *shell* le soient tout autant dans d'autres langages plus avancés, plus efficaces et plus polyvalents, il n'en demeure pas moins que de nombreuses machines tournant sous GNU offriront peu d'alternatives, souvent par choix : tout programme ajouté étant un nouveau risque potentiel de sécurité, il arrivera souvent qu'une machine ne soit équipée que du strict minimum d'un point de vue logiciel, et ce minimum inclura toujours un *shell*.

Par ailleurs, un grand nombre de programmes déjà disponibles utilisent depuis des décennies des fichiers de configuration suivant les normes d'un *shell*, ou nécessitent d'interagir avec lui. Il sera donc nécessaire d'être capable d'écrire ses propres scripts, ou de modifier la multitude de scripts déjà existants servant à la maintenance du système. Vous pouvez d'ailleurs compter le nombre de scripts déjà présents sur votre machine avec la commande :

```
1 $ find / -name "*.sh" 2>/dev/null | wc -l
```

6.2 Les alias

Les alias constituent une première tentative basique de programmation de commandes un peu complexes : au lieu d'écrire une commande difficile à retenir car utilisant beaucoup d'options, ou combinant éventuellement plusieurs programmes avec des *pipes*, on peut attribuer un nom à cette commande, qu'on pourra ensuite utiliser comme un synonyme. Ceci
`alias` se fait avec la commande `alias`.

Exemple 17. Pour mettre à jour notre système en évitant de devoir taper à chaque fois deux commandes, on peut définir l'alias suivant :

```
1 $ alias upgrade_system="sudo apt-get update && sudo apt-get upgrade"
```

On pourra ensuite simplement écrire `upgrade_system` dans notre terminal pour mettre à jour notre système.

Les alias sont pratiques mais limités : on ne peut pas leur communiquer de paramètres. Pour cela, on aura besoin de définir des fonctions. Pour supprimer l'alias `commande`, on utilise
`unalias` simplement `unalias commande`. La commande `unalias -a` supprime tous les alias.

Les alias et les autres commandes que nous définissons sont pratiques ... mais sont perdues dès que la session se termine. Pour les sauvegarder et pouvoir les réutiliser sans devoir les redéfinir à chaque nouvelle session, on les enregistre dans le fichier `~/.bashrc`.

Attention

Comme le fichier `~/.bashrc` n'est chargé qu'en début de session, les changements que vous y apportez ne seront pris en compte que lors de la prochaine connexion. Si vous voulez qu'ils soient pris en compte immédiatement, rechargez le fichier à l'aide de la commande `source ~/.bashrc` (ou sa version moins explicite : `. ~/.bashrc`). `source`

6.3 Format des fichiers

L'extension des scripts *shell* (`bash` ou autres) est `.sh` ; nous donnerons donc à nos programmes cette extension, et il faudra s'assurer que le fichier de script que nous écrivons est exécutable (utilisez `chmod` si nécessaire). De plus, le fichier doit commencer par :

```
1 #!/usr/bin/env bash
```

La séquence **#!** est appelée **shebang**, et permet de renseigner le *shell* à utiliser pour interpréter votre script. Si vous utilisez un autre *shell*, il faudra remplacer `/bin/bash` par le chemin vers cet autre *shell*.

Plus généralement, le # sert à définir des commentaires : ce caractère peut se trouver n'importe où sur une ligne, et tout ce qui le suit sur cette même ligne est ignoré. On ne peut pas commenter plusieurs lignes à la fois, comme on le fait par exemple en C(++) ou en Java à l'aide de `/* ... */`.

Une fois exécutable, le programme se lance simplement avec la commande `./fichier.sh` dans le répertoire courant. On peut aussi utiliser `bash fichier.sh` (remplacez `bash` par le *shell* de votre choix en cas de besoin) : dans ce cas, il n'est pas nécessaire que le fichier soit exécutable.

6.4 Variables

6.4.1 Les bases

Pour déclarer ou modifier une variable, on écrit `nomvariable=valeur`.

Attention

N'écrivez pas d'espaces autour du = ! Sinon, vous aurez l'erreur "command not found".

Pour obtenir la *valeur* d'une variable, on utilise l'opérateur **$** (exemple : `$nomvariable`). On peut supprimer une variable avec la commande `unset nomvariable` ; si la variable possédait une valeur par défaut, elle est recréée avec cette valeur. `unset`

L'opérateur **$** permet aussi d'évaluer le résultat d'une commande ; il faut alors rajouter des parenthèses, comme ceci : `$(ma_commande)`.

Exemple 18. La commande `date -I` fournit la date d'aujourd'hui au format AAAA-MM- `date`

JJ. On peut stocker sa valeur dans la variable `aujourdhui` comme ceci :

```
1 aujourdhui=$(date -I)
```

(fin exemple 18)

Faites bien attention aux parenthèses, car les deux versions sont valides mais ne signifient pas la même chose :

— `$date` est la valeur d'une variable nommée `date` ;
— `$(date)` est le résultat de la commande `date`.

Les variables indéfinies ne provoquent pas nécessairement d'erreur : la commande `echo $alizubd` affichera une ligne vide si la variable `alizubd` n'existe pas.

Astuce

Il est plus sûr d'utiliser des guillemets autour des valeurs des variables, car la présence d'espaces ou de caractères spéciaux peut mener à des comportements inattendus de la part de `bash`. Par exemple, préférez `echo "$var"` à `echo $var`.

6.4.2 Types

Les variables en `bash` n'ont pas de type. Par défaut, `bash` les traite comme des chaînes, sur lesquelles on peut parfois faire des calculs : on doit alors le préciser à l'aide de crochets ou de doubles parenthèses :

```
1 $ echo 1+1
2 1+1
3 $ echo $[1+1]
4 2
5 $ echo $((1+1))
6 2
```

Les opérations habituelles sont disponibles (`+`, `-`, `*`, `/`, `%`, ...), et aussi sous forme condensée (`+=`, `-=`, `*=`, `/=`, ...). Attention ici aussi aux espaces : `x+=1` fonctionne, mais `x += 1` déclenche une erreur.

Attention

`bash` ne gère pas les nombres réels : `x=$((1+1.5))` provoque une erreur ! Il faut utiliser les programmes `bc` ou `dc`.

`declare`

L'absence de types peut rapidement mener à des bugs, puisque l'effet d'un opérateur dépend du type. On peut utiliser la commande `declare` pour donner des attributs aux variables ; par exemple, l'option `-i` force la variable à être entière (***integer***), et l'option `-r` empêche les modifications (***read-only***).

Exemple 19. Quelques instructions avec ou sans `declare` :

```
1 $ x=1
2 $ x+=1
```

```
$ echo $x
11  # x est une chaîne -> 1 + 1 = concaténation
$ declare -i var=42
$ var+=1
$ echo $var
43
$ declare -r var=42
$ var+=1
bash: var: readonly variable # erreur: on ne peut pas modifier var
```

(fin exemple 19)

Certains *shells* proposent une commande `typeset`, qui est un synonyme de `declare`. `typeset`

Attention

Par sécurité, utilisez `let` si vous voulez utiliser les formes condensées : en effet, seule `let` += fonctionne sans problème. Par exemple :

```
$ let x*=2
```

6.5 Instructions conditionnelles

`bash` propose aussi des tests booléens et des instructions conditionnelles. La Figure 6.1 montre la structure d'un `if` complet. Remarquons qu'un `if` doit toujours se terminer par un `fi`. Par ailleurs, chaque bloc d'instructions sous un `if` ou un `elif` doit commencer par un `then`. `if` `fi` `then`

```
if [[ condition1 ]]
then
    # code pour le cas où condition1 est vraie
elif [[ condition2 ]]
then
    # code pour le cas où condition2 est vraie
else
    # code pour le cas où condition1 et condition2 sont fausses
fi
```

FIGURE 6.1 – Syntaxe du `if`.

Astuce

Vous rencontrerez parfois des scripts qui utilisent `[ ... ]` plutôt que `[[ ... ]]`, et de longs débats sur la version à préférer. Nous en retiendrons uniquement que les versions utilisant `[[ ... ]]` sont moins portables, mais plus lisibles.

Le Tableau 6.1 reprend les opérateurs de comparaisons disponibles en `bash`. Les comparaisons renvoient 0 pour la valeur "faux" ou 1 pour la valeur "vrai". On peut combiner plusieurs conditions avec les opérateurs logiques "et" (`&&`), "ou" (`||`), et la négation (`!`). Atten-

tion contrairement aux affectations, il est nécessaire d'ajouter des espaces autour des opérateurs.

Comparaison	Opérateur
égalité	`$a == $b`
différence	`$a != $b`
est inférieur	`$a < $b`
est inférieur ou égal	`$a <= $b`
est supérieur	`$a > $b`
est supérieur ou égal	`$a >= $b`

TABLEAU 6.1 – Comparaisons des valeurs de deux variables `a` et `b` en `bash`.

Exemple 20. Souhaitons bon anniversaire à un utilisateur né le 25 octobre :

```
jour=$(date +'%d')
mois=$(date +'%m')
if [[ $jour == 25 && $mois == 10 ]]
then
    echo "Joyeux anniversaire!"
fi
```

6.6 Boucles

Les boucles en `bash` s'utilisent comme dans la plupart des autres langages, à ceci près qu'on doit indiquer le début de la boucle avec un `do` et sa fin avec un `done`.

6.6.1 La boucle `for`

`for` On peut utiliser la boucle `for` pour parcourir des itérables comme des chaînes de caractères, à l'aide de la syntaxe `for element in iterable`.

Exemple 21. Affichons chaque lettre de la chaîne `"bonjour"` sur une ligne séparée :

```
for lettre in "bonjour"
do
    echo $lettre
done
```

`seq` De même, on peut l'utiliser avec `seq` pour simuler un intervalle de nombres à parcourir :

```
# afficher 1 2 ... n (un nombre par ligne)
for i in $(seq n)
do
    echo $i
done
```

Il est également possible d'utiliser un format semblable à ce que propose le langage C.

Exemple 22. On peut initialiser une variable dans la boucle, et décider de la manière dont elle augmente à chaque itération. Le code suivant affiche les nombres de 1 à n (un nombre par ligne) :

```
for ((i=1; i<=n; i++))
do
    echo $i
done
```

6.6.2 La boucle `while`

Les boucles `while` suivent les conventions des autres langages (voir Figure 6.2). `while`

```
while [[ condition ]]
do
    # bloc while
done
```

FIGURE 6.2 – Syntaxe du `while`.

6.7 Fonctions et "modules"

`bash` nous permet de créer nos propres fonctions, avec quelques particularités :

1. les fonctions peuvent avoir des paramètres ..... mais on ne les déclare pas!
2. lorsqu'on appelle la fonction, on écrit son nom *sans parenthèses*, suivi des éventuels paramètres.

Exemple 23. La fonction suivante permet de personnaliser un accueil en fonction du nom de l'utilisateur et de son prénom :

```
saluer() {
    echo "Bonjour $1 $2! Comment allez-vous?"
}

# appel de la fonction
saluer monsieur Pingouin
# affichage: Bonjour monsieur Pingouin! Comment allez-vous?
```

Dans la fonction, `$i` désigne le i-ème paramètre de la fonction (les numéros démarrent à 1). Les fonctions peuvent également renvoyer des valeurs à l'aide du mot-clé `return`. Attention, `return` on ne peut renvoyer que des valeurs numériques!

Un **module** est un fichier fournissant un ensemble de fonctions et de variables que l'on peut réutiliser. En Python, tous les scripts que l'on écrit peuvent servir de modules, et on

récupère ce qu'il nous faut à l'aide du mot-clé `import` ; en C, on enregistre les déclarations des fonctions dans un *header* avec l'extension `.h`, qu'on renseigne dans le programme qui veut les utiliser à l'aide de la directive `#include`. `bash` ne propose pas ce genre de mécanisme, mais on peut le simuler de la façon suivante :

1. on enregistre les définitions des fonctions dans un fichier `mesfonctions.sh` ;
2. pour chaque fonction `mafonction` du fichier `mesfonctions.sh`, on insère dans le fichier `mesfonctions.sh` la commande

```
1 export -f mafonction
```

3. on écrit, au début de notre nouveau script `programme.sh` qui a besoin de ces fonctions, la commande `source mesfonctions.sh` pour charger le fichier.

Il faut bien sûr que `mesfonctions.sh` soit accessible : soit dans le même répertoire que notre nouveau script, soit dans un répertoire renseigné par `PATH`.

Attention

Rappelez-vous que `source` *exécute* le fichier passé en paramètre ; donc, ne placez dans votre "module" que des définitions de fonctions ou de variables !

6.8 Chaînes de caractères

Quelques opérations sur les chaînes permettent d'en extraire des morceaux. Le Tableau 6.2 les récapitule.

Bash	Résultat
`${#chaine}`	la longueur de `$chaine`
`${chaine:i}`	le suffixe de `$chaine` démarrant en `$i`
`${chaine:i:k}`	le suffixe de longueur `k` de `$chaine` démarrant en `$i`
`${chaine#chaine2}`	retire `$chaine2` de `$chaine` si c'est un préfixe
`${chaine%chaine2}`	retire `$chaine2` de `$chaine` si c'est un suffixe

TABLEAU 6.2 – Quelques opérations sur les chaînes.

Les opérations suivantes remplacent dans `chaine` :
— `${chaine/avant/après}` : la première occurrence de `$avant` par `$après` ;
— `${chaine//avant/après}` : **toutes** les occurrences de `$avant` par `$après` ;
— `${chaine/#avant/après}` : `$avant` par `$après` si `$avant` est un préfixe de `chaine` ;
— `${chaine/%avant/après}` : `$avant` par `$après` si `$avant` est un suffixe de `chaine` ;

Attention

L'effet de + diffère selon que l'on manipule des nombres ou des chaînes. Pour ne rien simplifier, les variables de `bash` ne sont pas typées ; c'est donc à nous de préciser de quel + on parle.

Exemple 24. Les trois “additions” suivantes ont un effet différent :

```
$ x=1; y=3; x+=$y; echo $x
13  # concaténation: x et y sont traitées comme des chaînes
$ x=1; y=3; x=$x+$y; echo $x
1+3 # autre concaténation: les valeurs de x et de y sont traitées
 ↪  comme des chaînes
$ x=1; y=3; x=$((x+y)); echo $x
4  # addition: x et y sont traitées comme des entiers
```

Comme dans les autres langages, les caractères spéciaux (`\n`, `\t`, ...) existent; mais si l'on veut que la commande `echo` les affiche correctement, il faut l'invoquer avec l'option `-e`.

6.9 Tableaux

`bash` permet de stocker des tableaux, avec la particularité que ses indices ne doivent pas être consécutifs!

6.9.1 Initialisation

Les tableaux s'initialisent de plusieurs manières très flexibles : en précisant les positions des éléments, “d'une traite”, ou à l'aide d'un fichier.

Exemple 25. On peut déclarer les éléments directement avec leur position :

```
$ tableau[0]="bonjour"
$ tableau[1]="tout"
$ tableau[2]="le"
$ tableau[3]="monde"
```

Ou plus simplement comme ceci (attention aux parenthèses) :

```
$ tableau=(bonjour tout le monde)  # même résultat que ci-dessus
```

On peut aussi initialiser le tableau avec le résultat d'une commande;

```
$ aujourdhui=($(date -Idate | sed s/-/' '/g))
$ echo ${myarray[0]}
2021
$ echo ${myarray[1]}
10
$ echo ${myarray[2]}
22
```

Ceci met l'année, le mois et puis le jour dans le tableau `aujourdhui`.

L'initialisation d'un tableau à partir d'un fichier est très utile, car on peut directement récupérer et isoler les mots ou les lignes d'un fichier.

Exemple 26. L'instruction suivante stocke chaque mot du fichier `fichier.txt` dans un tableau `mots` :

```
$ mots=($(< fichier.txt))
```

Les espaces sont considérés comme les séparateurs des mots et disparaissent. Si l'on préfère stocker les **lignes** du fichier, on utilisera la commande `readarray` :

readarray

```
$ readarray -t lignes < fichier.txt
```

6.9.2 Accès aux éléments

L'accès aux éléments se fait comme d'habitude à l'aide de l'opérateur `[]` et de la position de l'élément désiré (voir les exemples précédents). Si on accède à une position inexistante, il ne se passe rien. Pour afficher les éléments d'un tableau, on aura besoin des accolades.

Exemple 27. Reprenons le tableau initialisé plus haut, et tentons d'afficher sa première valeur :

```
$ tableau=(bonjour tout le monde)
$ echo tableau[1]
tableau[1] # sans $, tableau[1] est interprété par echo comme une
 ↪ chaîne
$ echo $tableau[1]
bonjour[1] # $tableau est interprété comme la première valeur du
 ↪ tableau, et [1] comme une chaîne
$ echo ${tableau[1]}
tout # $ s'applique à la variable tableau[1] et donne enfin le bon
 ↪ résultat
```

Comme on vient de le voir, `$tableau` ne nous donne que le premier élément du tableau (s'il existe). On peut accéder à l'entièreté du tableau avec l'opérateur `@` ; par exemple, `echo ${tableau[@]}` affichera le tableau entier. Enfin, si l'on veut connaître l'ensemble des **positions** valides du tableau, on rajoute l'opérateur `!` devant son nom ; par exemple, `echo ${!tableau[@]}`. Ces opérateurs nous permettent en particulier d'itérer sur les tableaux de deux manières.

Exemple 28. Pour afficher tous les éléments d'un tableau, on peut utiliser la boucle suivante :

```
for elem in ${tableau[@]}
do
    echo $elem
done
```

Si l'on préfère utiliser les positions, on utilisera la boucle suivante :

```
# affiche chaque élément avec sa position
for i in ${!tableau[@]}
do
    echo $i ${tableau[$i]}
done
```

(fin exemple 28)

Enfin, le nombre d'éléments d'un tableau s'obtient à l'aide de `${#tableau[@]}`. Attention, dans le cas d'une chaîne, on utilisait `${#chaine}` ! Ici, le rajout de `[@]` est nécessaire, sinon `tableau` est interprété comme `tableau[0]`, et l'instruction `${#tableau}` nous donnerait alors la longueur *du premier élément* du tableau.

6.10 Fichiers

On devra fréquemment vérifier des propriétés de nos fichiers. Le Tableau 6.3 reprend les tests les plus fréquents.

Expression	Signification
`-e nom`	le fichier (normal, répertoire, ...) nom existe
`-d nom`	le **répertoire** nom existe
`-f nom`	le **fichier** "normal" existe
`-r nom`	le fichier `nom` est lisible
`-w filename`	le fichier `nom` est modifiable
`-x filename`	le fichier `nom` est exécutable

TABLEAU 6.3 – Quelques tests fréquents sur les fichiers.

6.11 Paramètres des scripts

Tout comme les fonctions, les scripts peuvent prendre des paramètres, accessibles par leur position et $:

- `$0` est le nom du script ;
- `$1` est le premier paramètre ;
- `$2` est le deuxième paramètre ;
- ...

Attention : après `$9`, on doit utiliser des accolades (donc `${10}`, `${11}`, etc.) Il y a aussi quelques variables spéciales :

- `$#` est le nombre de paramètres du script ;
- `$*` et `$@` contiennent tous les paramètres du script ;

Attention

Il y a un risque de confusion entre les paramètres d'une fonction et ceux d'un script. Retenons que :

- dans la fonction, `$i` est le i-ème paramètre de la fonction ;
- hors de la fonction, `$i` est le i-ème paramètre du script.

Par souci de lisibilité, on utilise souvent des variables initialisées à l'aide des paramètres du script plutôt que ces paramètres directement. Ceci nous permet aussi de définir des valeurs par défaut, à l'aide de la syntaxe suivante (attention, il y a bien un - devant la valeur) :

```
variable=${numéro_paramètre:-valeur_par_défaut}
```

Exemple 29. Si l'on veut que notre script possède comme unique paramètre un répertoire sur lequel agir, et qu'il s'agisse par défaut du répertoire actuel, on insèrera ceci au début de notre script :

```
rep=${1:-.}
```

et l'on travaillera ensuite sur la variable `rep`.

Chapitre 7

Divers

Sommaire

Nous regroupons dans ce chapitre divers outils qu'il est utile de connaître.

7.1 Outils de maintenance

7.1.1 Programmation de tâches

Bien souvent, il sera utile de pouvoir exécuter périodiquement et automatiquement un script de maintenance, par exemple pour effectuer des sauvegardes. Deux outils permettent d'y arriver : `cron` et `anacron`.

7.1.1.1 `cron`

`cron` est un programme qui permet de planifier l'exécution de tâches à des moments précis : par exemple : “sauvegarder mes fichiers tous les jours à 17h30”. Sa configuration globale, qui affecte donc *tous* les utilisateurs du système, se trouve dans le fichier `/etc/crontab`. On y précise pour chaque entrée :

`cron`

1. l'heure de début ;
2. les jours ou mois d'exécution (par défaut : tous les jours) ;
3. l'utilisateur qui lancera la tâche ;
4. et la commande à exécuter ;

Exemple 30. Voici un exemple de fichier `/etc/crontab` :

```
1 # /etc/crontab: system-wide crontab
2 # Unlike any other crontab you don't have to run the `crontab'
3 # command to install the new version when you edit this file
4 # and files in /etc/cron.d. These files also have username fields,
```

```
# that none of the other crontabs do.

SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 *        * * *        root    cd / && run-parts --report /etc/cron.hourly
25 6        * * *        root        test -x /usr/sbin/anacron || ( cd / && run-parts --report
 ↪ /etc/cron.daily )
47 6        * * 7        root        test -x /usr/sbin/anacron || ( cd / && run-parts --report
 ↪ /etc/cron.weekly )
52 6        1 * *        root        test -x /usr/sbin/anacron || ( cd / && run-parts --report
 ↪ /etc/cron.monthly )
#
```

(fin exemple 30)

On peut programmer des tâches de manière *globale*, donc pour tous les utilisateurs ; ou de manière *locale* : chaque utilisateur peut programmer des tâches qui ne s'exécuteront que pour lui. La configuration globale est dans `/etc/crontab` ; les tâches locales se configurent avec la commande `crontab -e`.

crontab

`cron` possède plusieurs avantages : il exécute les commandes planifiées avec un grand degré de précision (jusqu'à la minute), et est disponible sur tous les systèmes GNU / Linux par défaut. En revanche, si votre machine est éteinte au moment où la tâche doit s'exécuter, elle ne s'exécutera pas ! Heureusement, l'outil suivant nous permettra de remédier à ce problème.

7.1.1.2 `anacron`

anacron

Comme `cron`, `anacron` permet de paramétrer l'exécution de tâches périodiques. Mais alors que `cron` exige de son utilisateur qu'il fixe un moment précis pour l'exécution d'une tâche (par exemple : tous les jours à 17h30), `anacron` se fixe une *période* (par exemple : tous les jours, "quand c'est possible"). `anacron` est donc préférable si : votre machine n'est pas allumée en permanence, ou si l'heure précise d'exécution n'est pas importante.

Comme pour `cron`, des tâches peuvent être programmées de façon globale ou locale. La configuration globale se trouve dans `/etc/anacrontab`. Chaque entrée suit le format :

```
période délai job-ID commande/script
```

avec :

- `période` : nombre de jours ou `@daily`, `@weekly`, `@monthly` ;
- `délai` : nombre de minutes à attendre avant de démarrer ;
- `job-ID` : l'identifiant unique de la tâche prévue ;

Exemple 31. Voici un exemple de fichier de configuration global pour `anacron` :

```
# /etc/anacrontab: configuration file for anacron

# See anacron(8) and anacrontab(5) for details.

```

```
SHELL=/bin/sh
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
HOME=/root
LOGNAME=root

# These replace cron's entries
1        5        cron.daily        run-parts --report /etc/cron.daily
7        10        cron.weekly        run-parts --report /etc/cron.weekly
@monthly        15        cron.monthly        run-parts --report
 ↪ /etc/cron.monthly
```

(fin exemple 31)

Pour les tâches locales, on “imite” la configuration globale :

```
$ mkdir -p ~/.anacron/{etc,spool}
$ echo $SHELL >> ~/.anacron/etc/anacrontab
$ echo $PATH >> ~/.anacron/etc/anacrontab
```

Puis on utilise `crontab -e` pour exécuter `anacron` périodiquement (par exemple toutes les heures) avec l’entrée :

```
@hourly /usr/sbin/anacron -s -t $HOME/.anacron/etc/anacrontab \
    -S $HOME/.anacron/spool
```

7.1.2 Sauvegardes

Il est capital de réaliser des sauvegardes fréquentes des fichiers importants. GNU propose de nombreux outils de sauvegarde, avec différents “niveaux d’intelligence” :

- tout recopier à chaque sauvegarde ;
- ou recopier uniquement les fichiers nouveaux ou modifiés ;
- ou recopier uniquement les changements.

Parmi les outils les plus recommandés, il y a `rsync`, qui possède de nombreux avantages : `rsync`

- la sauvegarde / restauration locale ou à l’aide d’un serveur ;
- l’utilisation des *différences* entre exemplaires de fichiers pour la sauvegarde ;
- une très grande flexibilité.

De nombreux outils lui servent d’interface et sont plus faciles à configurer, notamment `backintime`.

7.2 Outils de développement

7.2.1 *Patches*

Un ***patch*** est une modification que l’on rajoute à un logiciel, par exemple pour corriger un bug ou rajouter une fonctionnalité. Ils sont très fréquents en informatique ; on les retrouve dans :

- les mises à jour de sécurité ou de fonctionnalités (Windows ou autres) ;
- les corrections dans les jeux vidéo ou l’ajout de contenu ;
- …

En général, ils sont distribués sous forme binaire ; mais avec les logiciels libres, on peut aussi distribuer les corrections des sources ! L'intérêt est que les *patches* sont beaucoup plus petits que la nouvelle version entière du logiciel.

Pour créer des *patches*, il nous faut d'abord un outil qui permette d'identifier les différences
diff entre deux fichiers. Le programme `diff` fait exactement cela.

Exemple 32. Utilisons `sed` pour réaliser une copie d'un fichier où l'on remplace toutes les occurrences de "Quasimodo" par "Superman" :

```
$ sed s/Quasimodo/Superman/g notredamedeparis.txt > superman-de-paris.txt
```

`diff` indique les lignes et les caractères dans les deux fichiers où des différences apparaissent ; ici, < désigne les lignes du premier fichier, et > celles du second :

```
$ diff notredamedeparis.txt superman-de-paris.txt
2138,2139c2138,2139
< ``C'est Quasimodo, le sonneur de cloches! c'est Quasimodo, le bossu de
< Notre-Dame! Quasimodo le borgne! Quasimodo le bancal! Noël! Noël!''
---
> ``C'est Superman, le sonneur de cloches! c'est Superman, le bossu de
> Notre-Dame! Superman le borgne! Superman le bancal! Noël! Noël!''
```

Pour créer un *patch* permettant de passer de l'ancienne version d'un fichier à la nouvelle, on enregistre les différences dans un fichier à l'aide de la commande :

```
$ diff -u ancien nouveau > changements.patch
```

On peut maintenant distribuer uniquement `changements.patch`, que les utilisateurs pour-
patch ront utiliser pour mettre à jour leur version du fichier `ancien` à l'aide de la commande `patch` comme suit :

```
$ patch ancien changements.patch
```

Le résultat est un fichier identique à `nouveau`. Si nécessaire, il est aussi possible d'annuler les changements (voir `man patch`).

7.2.2 `comm`

comm Comme `diff`, `comm` est un outil de comparaison de textes, dont l'utilisation se limite toutefois aux fichiers dont les lignes sont triées. Sans option, la commande `comm premier second` affiche trois colonnes côte à côte :

1. la première colonne contient les lignes n'apparaissant que dans `premier` ;
2. la deuxième colonne contient les lignes n'apparaissant que dans `second` ;
3. la troisième colonne contient les lignes apparaissant dans `premier` et `second`.

On peut donc facilement produire les analogues des opérations ensemblistes, à l'aide de l'option - suivie des numéros des colonnes à supprimer :

— `premier` \ `second` s'obtient avec `comm -23 premier second` ;
— `second` \ `premier` s'obtient avec `comm -13 premier second` ;
— `premier` ∩ `second` s'obtient avec `comm -12 premier second`.

7.2.3 `make`

Les commandes que l'on utilise pour compiler des programmes peuvent parfois être compliquées et longues. Pour se faciliter la tâche, on utilise un programme appelé `make` qui exécute les instructions stockées dans un fichier de compilation appelé `Makefile`. L'intérêt de cette approche est de pouvoir permettre à n'importe qui, même des utilisateurs n'ayant aucune connaissance technique, de compiler des projets à l'aide d'une commande simple : en général, il suffira à l'utilisateur de taper `make` dans le répertoire qui contient le `Makefile`. `make`

L'autre avantage de `make` est qu'il analyse les sources et les cibles afin de déterminer ce qui a changé depuis la dernière compilation : ceci permet de ne recompile que les parties nécessaires. `make` utilise un système de dépendances, qui permet aussi de déterminer quels fichiers n'ayant pas changé doivent néanmoins être recompilés parce qu'ils dépendent de fichiers qui ont changé.

Les `Makefile` peuvent être très compliqués, mais le principe est toujours le même : on définit des **cibles** (= *targets*), qui sont les résultats à obtenir ; et pour chaque cible, on définit des règles (= *rules*) de construction, qui expliquent à `make` comment construire les cibles. Les cibles doivent être écrites de la manière suivante :

```
nom_de_la_cible : liste de prérequis
	règle 1
	règle 2
	...
```

Dans ce format :

- `nom_de_la_cible` est un mot sans espaces
- `liste de prérequis` est une liste d'autres cibles qui doivent avoir été construites ;
- `règle 1`, `règle 2`, ... sont les instructions de compilation de la cible ;

Attention

Les tabulations sont importantes : ne les remplacez pas par des espaces !

Exemple 33. Voici un exemple plus complet de `Makefile`, tiré de https://www.gnu.org/software/make/manual/html_node/Simple-Makefile.html :

```
edit : main.o kbd.o command.o display.o \
	insert.o search.o files.o utils.o
	cc -o edit main.o kbd.o command.o display.o \
	        insert.o search.o files.o utils.o

main.o : main.c defs.h
	cc -c main.c
kbd.o : kbd.c defs.h command.h
	cc -c kbd.c
command.o : command.c defs.h command.h
	cc -c command.c
display.o : display.c defs.h buffer.h
	cc -c display.c
insert.o : insert.c defs.h buffer.h
	cc -c insert.c
```

```
search.o : search.c defs.h buffer.h
	cc -c search.c
files.o : files.c defs.h buffer.h command.h
	cc -c files.c
utils.o : utils.c defs.h
	cc -c utils.c
clean :
	rm edit main.o kbd.o command.o display.o \
	   insert.o search.o files.o utils.o
```

(fin exemple 33)

7.3 Autres

7.3.1 Commandes "*offline*"

On a parfois besoin d'accéder à des machines beaucoup plus puissantes que la nôtre pour effectuer des calculs. En général, on se connecte à la machine distante à l'aide de `ssh`, puis on lance la tâche dans le terminal. Malheureusement, si l'on se déconnecte, la tâche se termine ... ce qui nous oblige donc à rester connecté si l'on veut la voir se terminer.

`nohup`

Bien entendu, ce n'est pas pratique si la tâche dure plusieurs jours. Pour éviter qu'elle ne se termine lors de la déconnexion, on peut l'exécuter avec `nohup`. On peut demander l'écriture du résultat dans un fichier situé dans notre répertoire personne. Et l'on peut même demander à être prévenu par e-mail quand la tâche se termine.

Index

L'index suivant recense tous les concepts présentés dans le document.

Commandes

L'index suivant recense toutes les commandes présentées dans le document.