Petit condensé d'introduction au Système Linux

2026

Ceci est une version β du manuscrit, à prendre avec des pincettes car de nombreuses typos et erreurs peuvent subsister avant une nouvelle passe de relecture, correction et réécriture

Manuscrit rédigé par Victor Lutfalla pour l'UE Fonctionnement des Ordinateurs en Portail Descartes à l'université d'Aix-Marseille au semestre 2 2025/2026.

Table des matières

Disclaimer

Ce manuscrit n'a pas vocation à être exhaustif ni à être un cours complet et intrinsèquement suffisant. De plus, il contient de nombreuses approximations et raccourcis dans la description et l'usage des commandes shell.
Ce manuscrit est pensé comme une ressource qui vient en complément des cours, travaux dirigés et travaux pratiques de système linux. Il est en particulier possible que certains sujets traités en cours mais non inclus dans ce manuscrit soient évaluées lors du contrôle continu et de l'examen.

Introduction

Dans ce manuscrit nous allons introduire le système Linux en se concentrant sur l'interface textuelle (le *shell*, ou console de terminal) et le système de fichier. Nous n'allons que très sommairement traiter la théorie et les grands concepts de système d'exploitation, et nous n'allons pas du tout traiter la conception de systèmes d'exploitation.

Chapitre 1

Système d'exploitation

1.1 Abstraction

Le système d'exploitation est une couche de gestion/abstraction qui permet d'utiliser du matériel physique (*hardware*) en faisant abstraction des détails pratiques, c'est à dire en ne se souciant pas de quel type de support mémoire est utilisé, quel processeur est utilisé, quelle technologie de communication est utilisée entre les différents composants physiques etc.
En particulier le système d'exploitation permet à l'utilisateurice de complètement ignorer le fait que l'ordinateur est par essence un circuit séquentiel constitué de milliards de transistors combinés en portes logiques, registres, etc.
L'utilisateurice interagit rarement directement avec le système d'exploitation. En général l'utilisateurice interagit avec un logiciel applicatif via une interface graphique (qui constitue une couche supplémentaire de gestion/abstraction entre l'utilisateurice et la machine).

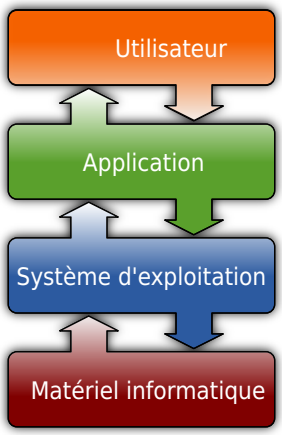


FIGURE 1.1 – Les couches d'abstraction.

1.2 Console de terminal

Les systèmes Unix sont dotés d'une interface pour communiquer directement avec le système d'exploitation (ou du moins le plus directement possible) : la console de terminal.
Cette console de terminal permet d'interagir «directement» avec l'ordinateur à travers des commandes textuelles, c'est pourquoi on parle parfois d'«interface textuelle» (par opposition à «interface graphique») pour le terminal.
L'interface textuelle est «universelle», c'est à dire que tout ce qui peut être fait sur un ordinateur peut être fait depuis l'interface textuelle. En revanche l'interface graphique n'est pas universelle, seules les utilisations/actions pour lesquelles une interface graphique a été prévue/développée peuvent être faites via interface graphique.

Dans l'interface textuelle, une invite (*prompt*) (`[victor@cachytor ~]`, encadré en rouge dans la figure) indique que le terminal est prêt à recevoir

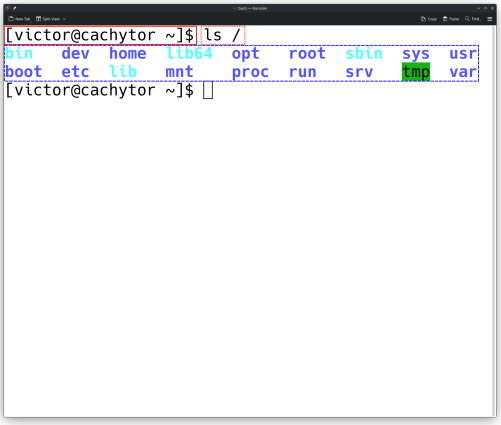


FIGURE 1.2 – Une console de terminal

une commande, l'utilisateurice peut alors taper une commande (`ls /`, encadré en pointillés rouge dans la figure) et la valider avec la touche Entrée, le résultat s'affiche alors en dessous (encadré en lignée brisée bleue dans la figure).

1.3 Raccourcis clavier dans le terminal

Dans le terminal il existe de nombreux raccourcis dont nous allons seulement présenter les plus importants, attention les raccourcis terminal ne sont pas ceux dont vous avez l'habitude (`Ctrl+C` ne fait pas Copier).

- ▷ `Tab` : autocompletion (essaye de compléter une commande partielle)
- ▷ `↑,↓` : navigation dans l'historique
- ▷ `Ctrl+C` : interromps le processus en cours
- ▷ `Ctrl+D` : end-of-file (et alias de `exit`)
- ▷ `Ctrl+L` : ≈`clear` (défile (*scroll*) le terminal afin de nettoyer l'affichage)
- ▷ `Ctrl+R` : recherche dans l'historique
- ▷ `Crtl+Maj+C` : copier
- ▷ `Crtl+Maj+V` : coller

En plus de ces raccourcis, on pourra mentionner `Ctrl+U` (effacement ligne), `Ctrl+W` (effacement mot), `Ctrl+S` (arrêt de l'affichage), `Crtl+Q` (reprise de l'affichage) et `Ctrl+Z` (suspension du processus en cours).

Chapitre 2

Système de fichiers

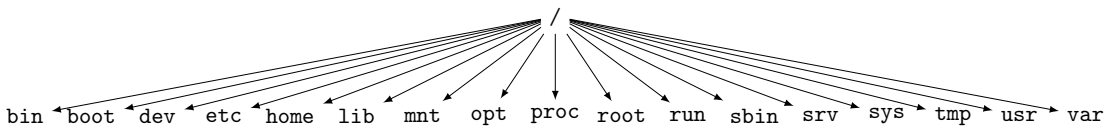
2.1 Tout est fichier

Dans la mémoire d'un ordinateur, tout est fichier : les fichiers sont des fichiers, les programmes sont des fichiers, les exécutables sont des fichiers, les dossiers sont des fichiers, les I/O (input-output) sont des fichiers.
Dans la grande famille des systèmes Unix, il existe 4 types de fichiers :

- ▷ les fichiers ordinaires
- ▷ les répertoires ou dossiers (*directory* ou *folder* en anglais) qui contiennent une liste de description de son contenu
- ▷ les fichiers spéciaux qui sont une abstraction permettant d'interagir avec les entrées/sorties (I/O) telles que le clavier, la souris comme si c'était des fichiers dynamiques textuels (buffers)
- ▷ les liens qui pointent vers un autre fichier

2.2 Filesystem Hierarchy Standard

Afin de pouvoir s'y retrouver, l'organisation des fichiers dans la mémoire logique a été standardisé. Au sein de la grande famille des systèmes Unix c'est le Filesystem Hierarchy Standard (wiki:FHS) qui est le standard dominant [1].
Le système de fichiers peut être vu comme une arborescence où les nœuds sont des dossiers et les feuilles des fichiers (on fait ici l'approximation qu'il n'y a pas de liens).
La racine de cette arbre, est appelé **racine** (*root* en anglais) est désignée par le symbole `/`.
Le symbole `/` est aussi utilisé pour marquer la séparation (imbrication) entre les dossiers. Le dossier `/usr/bin/` sera donc le dossier `bin`, contenu dans le dossier `usr` situé à la racine `/` du système de fichiers.
Voici le premier niveau d'arborescence du FHS :



Chacun des dossiers situés à la racine a un rôle spécifique dont les plus importants sont

- ▷ `/bin` : fichiers binaires (exécutables), dans les distributions modernes c'est souvent un lien vers `/usr/bin`
- ▷ `/home` : contient les répertoires personnels des utilisateurices

1. Certaines distributions linux comme NixOs utilisent un autre système de fichiers

- ▷ `/etc` : `editable text configurations` contient des fichiers de configuration textuels modifiables
- ▷ `/tmp` : contient des fichiers temporaires (qui disparaissent lors du redémarrage)
- ▷ `/lib` : contient des librairies (utilisées par divers exécutables)
- ▷ `/dev` ou `/media` : contient les fichiers spéciaux d' I/O, (`dev=devices`)

On peut aussi représenter cette arborescence de la façon suivante :

```
/
|
|_ /bin
|
|_ /boot
|
|_ /dev
|
|_ /etc
|
|_ /home
|
|_ /lib
|
|_ /mnt
|
|_ /opt
|
|_ /proc
|
|_ /root
|
|_ /run
|
|_ /sbin
|
|_ /srv
|
|_ /sys
|
|_ /tmp
|
|_ /usr
|
|_ /var
```

Vous pourrez trouver la description des autres dossiers dans la page wikipedia wiki:FHS.

2.3 Chemin absolu et relatif

L'emplacement d'un dossier ou fichier dans le système de fichier est appelé chemin.
Ce chemin peut être donné en tant que :

- ▷ chemin absolu : c'est à dire le chemin depuis la racine `/`, par exemple `/home/victor/Documents`
- ▷ chemin relatif : c'est à dire à partir du répertoire courant, c'est à dire relativement à l'emplacement actuel dans le système de fichiers par exemple `./Documents`

Pour les chemins relatifs il existes deux méta-noms :

- `.` désigne le répertoire courant (aussi appelé répertoire de travail)
- `..` désigne le répertoire parent du répertoire courant

Par exemple, si je me situe dans mon répertoire de téléchargements `/home/victor/Downloads`, alors :

- `.` désigne le répertoire courant `/home/victor/Downloads`
- `..` désigne `/home/victor` puisque c'est le parent de `/home/victor/Downloads`
- `../Documents/travail` désigne `/home/victor/Documents/travail` puisque `..` désigne `/home/victor`

En plus de ces méta-noms propre au FHS, il existe un métacaractère shell qui désigne le répertoire personnel du user actif : `~` désigne le repertoire `/home/<username>` (dans mon cas `/home/victor`). Pour plus de détails vous pouvez consulter GNU-wiki:~, on peut aussi noter que le fait que `~/Documents` soit un chemin absolu ou relatif est sujet à débats (voir unix.stackexchange:~).

Chapitre 3

Première commandes shell

Le shell est à la fois le nom de l'interface textuelle et du langage (ou plutôt de l'interpréteur et de la structure du langage) utilisé dans l'interface textuelle.
Dans la plupart des distributions linux l'interpréteur shell est `bash` (pour *Bourne Again SHell*, voir wiki:bash), c'est donc avec la syntaxe de cet interpréteur que l'on va travailler. Certains systèmes modernes préfèrent `zsh` à `bash` mais le fonctionnement est très similaire.
Lorsqu'on écrit une commande dans un fichier (comme le condensé ci présent), on la commence toujours par le symbole `$` pour indiquer que c'est une commande shell, et pour symboliser l'invite de commande. Par exemple `$ ls -l | tr -s ' ' | cut -d' ' -f1,9` (nous détaillerons cette commande plus tard). On utilise le symbole `>>>` au début des sorties standard (affichage dans le terminal) reportées dans ce document afin de clarifier qu'il s'agit du résultat d'une commande, et non pas d'une commande. Lorsque vous souhaitez exécuter la commande dans votre shell, il ne faut pas recopier le `$`, si vous oubliez de l'enlever vous aurez le message d'erreur `>>> bash: $: command not found`.
Vous trouverez aussi parfois des commandes qui commencent par `#` au lieu de `$`, cela est souvent l'usage pour les commandes critiques (qui commencent souvent par `sudo` afin d'acquérir les droits «super-utilisateur»).

3.1 Manuel et informations : `man`, `whatis`

L'une des commandes les plus importantes est `man` qui permet d'ouvrir la page de manuel d'une commande, c'est à dire la documentation technique de la commande.

L'expression RTFM (*Read The Fucking Manual*) rappelle que la première chose à faire lorsqu'on essaye (sans succès) d'utiliser ou comprendre une commande bash est de lire la page de manuel de cette commande.
Pour ouvrir la page de manuel du manuel il faut donc exécuter la commande `$ man man` . Lorsqu'on exécute cette commande, la page de manuel s'ouvre (voir capture d'écran à gauche), pour quitter cette page il suffit d'appuyer sur la touche `q` (comme indiqué en bas de l'écran).
La page de manuel est en général assez longue, parfois un peu dure à lire, mais elle est très complète et indique en particulier la liste des options que l'on peut passer à la commande.
Lorsque l'on souhaite avoir une description en une ligne on peut préférer la commande `whatis`, par exemple l'exécution de `$ whatis man` affiche

```
>>> man (1) - an interface to
```

`the system reference manuals`
(qui correspond à la description de `man` dans la page `$ man man`). Dans certains systèmes, d'autres commandes comme `tldr` permettent d'avoir un manuel simplifié (un peu plus détaillé que `whatis` mais moins détaillée que `man`). Je vous encourage à installer sur votre machine personnelle un paquet contenant `tldr` tel que `tealdeer` (disponible avec les package managers `pacman` et `apt`).

3.2 Tableau des commandes abordées

Commande	Squelette : description
`;` (*seq*)	`commande1 ; commande2` : exécute séquentiellement les deux commandes ;
`|` (*pipe*)	`commande1 | commande2` : exécute `commande1` et redirige sa sortie vers l'entrée de `commande2` (et exécute commande 2)
`>` (*redirect*)	`commande >fichier` : exécute `commande` et redirige sa sortie vers le fichier `fichier` (écrase le fichier si il existe)
`>>`	`commande >>fichier` : exécute `commande` et redirige sa sortie sur la fin de `fichier` (ajoute la sortie au contenu du fichier sans l'écraser)
`<`	`commande <fichier` : redirige `fichier` vers l'entrée de `commande` (et exécute la commande sur cette entrée)
`man`	`man commande` : affiche la page de manuel de la commande donnée en argument
`whatis`	`whatis commande` : affiche une description en une ligne de la commande donnée en argument
`ls`	`ls [options] [chemin]` : liste fichiers/dossiers
`cd`	`cd [chemin]` : change de répertoire
`pwd`	`pwd` : affiche répertoire courant
`echo`	`echo chaine` : affiche la chaîne
`cat`	`cat fichier1 [fichier2 ... ]` : affiche les fichiers, souvent utilisé pour afficher un seul fichier
`tac`	`tac fichier1 [fichier2 ... ]` : affiche les fichiers à l'envers (par lignes)
`head/tail`	`head/tail -N fichier` : premières/dernières N lignes
`wc`	`wc [-l|-w|-c] fichier` : compte lignes/mots/caractères
`mkdir`	`mkdir chemin` : crée dossier
`touch`	`touch fichier` : crée fichier vide
`rmdir`	`rmdir chemin` : supprime le dossier vide
`rm`	`rm [-r] chemin` : supprime fichier/dossier (récursif si `-r`)
`mv`	`mv origine destination` : déplace/renomme
`cp`	`cp [-r] origine destination` : copie fichier (dossier si `-r`)
`find`	`find <path> [options] [critères]` : recherche dans le système de fichiers
`grep`	`grep [opts] motif fichier` : recherche motif dans texte
`cut`	`cut -d' ' -fN` : extrait des colonnes (avec séparateur spécifié par `-d`)
`tr`	`tr [otions] classe1 [classe2]` : transforme caractères
`sort`	`sort fichier` : trie lignes
`uniq`	`uniq` : supprime doublons consécutifs
`sort`	`sort fichier` : trie les lignes du fichier
`tee`	`commande1 | tee >fichier` : affiche la sortie de `commande1` et l'écrit dans `fichier`
`sed`	`sed 's/pomme/poire/'` : édite le flux / substitue des regex
`awk`	`awk 'programme'` : extrait, traite et manipule des données dans un flux textuel
`nano`	`nano [fichier]` : ouvre l'éditeur de texte interactif `nano`
`chmod`	`chmod droits chemin` change les droits du fichier ou dossier spécifié
`ps`	`ps -ef` : affiche la liste complète des processus
`top`	`top` : ouvre une console interactive de monitoring système

La plupart de ces options sont décrites dans le manuel GNU-doc:coreutils.

3.3 Listes et navigation du système de fichier : `pwd`, `ls`, `cd`

Afin de naviguer dans le système de fichier les trois commandes fondamentales sont :

- `pwd` (*Print Working Directory*) affiche le chemin absolu du répertoire courant
- `ls` (*LiSt*) affiche le contenu du répertoire courant sous forme de liste
- `cd` (*Change Directory*) change le répertoire courant

Voici un exemple simple d'utilisation de `pw`, `ls` et `cd`.

```
[victor@cachytor ~]$ pwd
/home/victor
[victor@cachytor ~]$ cd /
[victor@cachytor /]$ pwd
/
[victor@cachytor /]$ ls
bin  boot  dev  etc  home  lib  lib64  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
[victor@cachytor /]$ cd home
[victor@cachytor home]$ pwd
/home
[victor@cachytor home]$ ls
victor
[victor@cachytor home]$ cd victor
[victor@cachytor ~]$ pwd
/home/victor
[victor@cachytor ~]$
```

La commande `pwd` s'utilise généralement sans options et ne prends pas d'arguments.

La commande `ls` a de nombreuses options possibles, les plus importantes sont :

- `-a` (*all*) affiche tous les fichiers, y compris les fichiers cachés
- `-l` (*long*) affiche plus d'informations sur les fichiers (droits, propriétaire, date de création ou édition, etc)
- `-R` (*recursive*) affiche récursivement les contenus des sous-dossiers
- `-d` (*directories*) affiche le nom des dossiers mais pas leurs contenus

On peut donner un argument à `ls` afin de ne pas afficher le contenu du répertoire courant, mais plutôt le contenu d'un autre répertoire, ou une sélection du contenu du répertoire.
La syntaxe est `$ ls chemin` où `chemin` désigne un emplacement, sous forme de chemin absolu ou relatif de l'emplacement. Par exemple `$ls /usr/bin` affiche le contenu du répertoire `/usr/bin` (quel que soit le répertoire courant), et `$ls ../` affiche le contenu du répertoire parent du répertoire courant.
Le chemin peut être une regex (expression régulières) simple afin de sélectionner un sous-ensemble de contenu à afficher, par exemple `$ ls *.txt` liste l'ensemble des fichiers qui ont l'extension `.txt` dans le répertoire courant. Pour la syntaxe des regex, voir la section 3.4.
On peut combiner les options et les arguments, par exemple
`$ ls -ld /usr/bin/arp*` affiche la liste (au format long) du contenu de `/usr/bin` (en n'affichant pas le contenu des dossiers mais juste leur nom) dont le nom commence par `arp`.

La commande `cd` s'utilise avec en argument le chemin [1] du répertoire où l'on souhaite se rendre, ce chemin peut être absolu (exemple `$ cd /usr/bin`) ou relatif (exemple `$ cd ../adamant`).

3.4 Regex simples ou expressions globulaires

Comme mentionné pour `ls` on a souvent envie de désigner un ensemble de chaînes de caractères (par exemple des noms de fichiers) avec une formule. L'approche utilisée en bash est la regex

1. Ce chemin peut être une regex à condition que un seul chemin satisfasse cette regex. Si plusieurs chemin satisfont la regex on reçoit l'erreur `cd: too many arguments`.

(*REGular EXpression*) avec l'usage de wildcards. Ces expressions régulières simples sont aussi appellées «expressions globulaires».
Une regex est un suite de symboles dont certains sont standards et certains sont spéciaux, les principaux symboles spéciaux sont :

- `*` : n'importe quelle chaîne de caractère
- `?` : un (unique) caractère quelconque
- `[`*liste de caractères*`]` : un caractère parmi la liste, la liste peut être une plage (ex : `[0-9]` pour les chiffres, ou `[a-z]` pour les lettres minuscules).
- `[!`*liste de caractères*`]` : un caractère **pas** parmi la liste
- `{`*liste de mots*`}` : un mot parmi la liste

Par exemple la regex `a*.txt` contient un symbole wildcard `*` et 5 symboles standard, cette regex désigne toute les chaines de caractères qui commencent par un `a` et finissent par `.txt`.
Quelques premiers exemples :

- `*.pdf` : l'ensemble des chaînes qui finissent par `.pdf`
- `a???` : l'ensemble des chaînes qui font 4 caractères et qui commencent par un `a`
- `*.{png,pdf,jpeg}` : l'ensemble des chaînes qui finissent par `.pdf`, `.png`, ou `.jpeg`
- `[aeiouy]*` : l'ensemble des chaînes qui commencent par une voyelle en minuscule
- `[a-z]*` : l'ensemble des chaînes qui commencent par une lettre minuscule
- `[!a-z]*.txt` : l'ensemble des chaînes qui finissent par `.txt` et ne commencent pas par une voyelle minuscule

Exemple détaillé : on se place dans un répertoire dont le contenu est :

```
aragorn.pdf  aragorn.txt  boromir.pdf  boromir.txt  frodo.pdf  frodo.txt
gandalf.pdf  gandalf.txt  gimli.pdf  gimli.txt  legolas.pdf  legolas.txt
meriadoc.pdf  meriadoc.txt  peregrin.pdf  peregrin.txt  samwise.pdf  samwise.txt
```

Voici quelques commandes et leurs résultats :
`$ ls *.txt` (liste les fichiers qui finissent par `.txt`)

```
>>> aragorn.txt  boromir.txt  frodo.txt  gandalf.txt  gimli.txt
    legolas.txt  meriadoc.txt  peregrin.txt  samwise.txt
```

`$ ls [!aeiouy][!aeiouy]*.txt` (liste les fichiers qui finissent par `.txt` et dont les deux premiers caractères ne sont pas des voyelles minuscules)

```
>>> frodo.txt
```

`$ ls *{or,ri}*` (liste les fichiers qui contiennent `or` ou qui contiennent `ri`)

```
>>> aragorn.pdf  aragorn.txt  boromir.pdf  boromir.txt
    meriadoc.pdf  meriadoc.txt  peregrin.pdf  peregrin.txt
```

3.5 Commandes d'affichage : `echo`, `cat`, `head`, `tail`

La première commande d'affichage est la commande `echo` qui, comme son nom l'indique, fait un écho : affiche à l'écran la chaîne de caractères qu'elle a reçu en argument. Par exemple la commande `$ echo "Hello World"` affichera `>>> Hello World`. Cette commande peut paraître inutile, mais on verra plus tard que avec les redirections (`>`) et les tubes (*pipes*, `|`) on peut utiliser echo dans différents contextes.
Notre seconde commande d'affichage est `cat` (pour *concatenate*) qui affiche à l'écran le contenu du (ou des) fichier(s) texte donnés en argument. Par exemple `$ cat frodo.txt` affichera le contenu du fichier `frodo.txt`.

Si on veut afficher un ou des fichiers mais en ordre des lignes inversés (en commençant donc par la dernière ligne) on peut utiliser `tac` (comme `cat` à l'envers) qui a la même syntaxe que `cat`.
Pour des fichiers très grands, on peut ne vouloir afficher que les premières ou dernières lignes. C'est le rôle des commandes `head` (affiche les premières lignes) et `tail` (affiche les dernières lignes). Par défaut ces commandes affichent les 10 premières (resp. dernières) lignes du fichier donné en argument. On peut ajouter des options pour modifier ce comportement (cf. `$ man head`).
La commande `wc` (pour *word count*) permet de compter le nombre de lignes, de mots et de caractères dans un fichier. Par exemple : `$ wc frodo.txt` . Cette commande affichera le nombre de lignes, de mots et de caractères dans le fichier `frodo.txt`. On peut ajouter l'option `-l` pour n'afficher que le nombre de lignes, `-w` pour n'avoir que le nombre de mots, et `-c` pour n'avoir que le nombre de caractères.

3.6 Création et suppressions de dossiers et fichiers : `touch`, `mkdir`, `rm`, `rmdir`, `mv`, `cp`

Les opérations de création et suppressions sont différentes pour les dossiers et les fichiers.
Pour la création, on utilise `touch` pour un fichier et `mkdir` (*make directory*). Pour la suppression, on utilise `rm` (*remove*) pour un fichier, `rmdir` (*remove directory*) pour un dossier vide et `rm -r` (*remove recursively*) pour un dossier non-vide et tout son contenu.
Attention : les suppressions sont définitives (cela ne va pas à la Corbeille) donc `rm -r` est à utiliser avec précaution.
Ces quatre commandes ont la même syntaxe `$ commande chemin`. Par exemple :

- `$ mkdir personnages_secondaires` : crée le sous-dossier `personnages_secondaires` dans le répertoire courant
- `$ touch personnages_secondaires/faramir.txt` : crée le fichier `faramir.txt` dans le sous-dossier `personnages_secondaires`
- `$ touch personnages_secondaires/denetor.txt` : crée le fichier `denetor.txt` dans le sous dossier `personnages_secondaires`
- `$ rm personnages_secondaires/denetor.txt` : supprime le fichier (parce que le nom est erroné donc j'ai décidé de le supprimer
- `$ touch personnages_secondaires/denethor.txt` : crée un fichier avec le bon nom
- `$ mkdir tmp_dir` : crée un dossier `tmp_dir`
- `$ rmdir tmp_dir` : supprime le dossier `tmp_dir`
- `$ rm -r personnages_secondaires` : supprime le dossier personnages secondaires (et son contenu)

On peut aussi déplacer les fichiers et dossiers avec `mv` (*move*), et les copier avec `cp` (*copy*). Dans les deux cas la syntaxe est `commande origine destination`, mais pour copier un dossier il faut spécifier l'option `-r` (*recursively*). Par exemple :

- `$ mv frodo.txt frodo_baggins.txt`
- `$ cp frodo.txt frodo_copy.txt`
- `$ cp -r personnages_secondaires autres_personnages` (en supposant que je n'avais pas supprimé le doissier)

Pour plus de détails vous pouvez vous référer aux pages de manuel de ces commandes.

3.7 Flux standard et redirections : >, >>, <

Les commandes ont 3 flux standard (wiki:flux standard) : un flux d'entrée (usuellement le clavier), un flux de sortie (usuellement l'affichage dans le terminal) et un flux d'erreur (usuellement aussi l'affichage dans le terminal).

La plupart des commandes qui attendent en argument un fichier texte (`cat`, `head`, ...), prennent l'entrée standard comme entrée si l'argument n'est pas fourni. En particulier si un·e utilisateurice exécute la commande `$ cat` (sans argument), la commande prend l'entrée standard c'est à dire que le terminal va afficher à l'écran une copie de ce que l'utilisateur tape au clavier jusqu'à recevoir un EOF (end-of-file, `Ctrl+D`).
Le flux de sortie standard peut être redirigé vers un fichier :

- avec `>`, par exemple `$ cat aragorn.txt boromir.txt >humans.txt` redirige l'affichage de `cat aragorn.txt boromir.txt` vers le fichier `humans.txt`, cela crée (ou écrase) le fichier `humans.txt` qui contiendra donc la concaténation de `aragorn.txt` et `boromir.txt`
- avec `>>` lorsqu'on veut rediriger la sortie sans écraser le fichier destination, par exemple `$ cat aragorn.txt boromir.txt >>humans.txt` qui au lieu d'écraser `humans.txt` (si il existe) ajoute la sortie standard à la fin du fichier (si le fichier destination n'existe pas alors cette commande se comporte comme `>`)

On peut rediriger l'entrée standard avec avec `<` pour remplacer un input attendue en entrée standard par la lecture d'un fichier, mais cela est moins utilisé. Et on peut aussi rediriger la sortie d'erreur et combiner les redirections, mais cela dépasse un peu le sujet de ce manuscrit.

3.8 Combiner des commandes : `;`, `|`

Les commandes peuvent être combinées de différentes manières pour automatiser des tâches complexes. Le point-virgule (`;`) permet d'exécuter plusieurs commandes séquentiellement, indépendamment du succès ou de l'échec des commandes précédentes. Par exemple :

```
$ mkdir test; cd test; touch file.txt
```

Cette commande crée un répertoire nommé `test`, change le répertoire courant vers `test`, puis crée un fichier nommé `file.txt` dans ce répertoire.
Le tube (*pipe* `|`) permet de passer la sortie d'une commande comme entrée à une autre commande. Par exemple :

- `$ ls -l | wc -l` : compte le nombre de fichiers du répertoire courant (liste les fichiers dans le répertoire courant, puis compte le nombre de lignes de la liste),
- `$ ls -l /usr/bin | head` : affiche les 10 premiers fichiers de `/usr/bin` (liste les fichiers dans `/usr/bin`, puis ne conserve que les 10 premières).

3.9 Rechercher dans le système de fichier : `find`

La commande `find` permet de rechercher des fichiers (ou dossiers) dans le système de fichiers. Cette commande est très puissante mais il faut l'utiliser avec prudence, en effet elle recherche les fichiers dans tout le sous-arbre à partir du chemin spécifié en argument (par défaut le répertoire courant) cela peut donc trouver un très grand nombre de fichiers.
Par exemple la commande `$ ls /` affiche le contenu du dossier racine (18 dossiers dans mon cas, cela peut varier un peu selon la distribution), alors que `$ find /` va (tenter de) rechercher et afficher la totalité des fichiers et dossiers du sous-arbre de la racine (il y a des millions de fichiers et dossier au total).
Les options usuelles de `find` sont :

- `-name` : rechercher par nom en regex simple, exemple `$ find ~ -name "*.txt"` affiche la liste de tous les fichiers (et dossiers) finissant par `.txt` dans l'arborescence de mon répertoire personnel (rappel : `~` est un alias de mon `$HOME`.)
- `-iname` : rechercher par nom insensible à la casse, exemple `$ find ~ -iname "*lotr*"` affiche la liste de tous les fichiers (et dossiers) qui contiennent `lotr` (ou `LOTR`, `LotR`, ou toute autre variante min/maj) dans l'arborescence de mon répertoire personnel

- `-maxdepth` : pour limiter la profondeur de recherche dans l'arborescence, exemple `$ find ~ -maxdepth 2 -name "*.txt"` affiche la liste des fichiers `.txt` à profondeur au plus 2 dans mon répertoire personnel (donc soit directement dans `~` soit dans un sous-dossier direct de `~`)
- `-type` pour restreindre par type de fichier/dossier/lien, exemple `$ find ~ -type d -iname "*lotr*"` ne va afficher que les dossiers (`d` pour *directory*) correspondant à la recherche, alors que `-type f` n'aurait afficher que les fichiers (`f` pour *file*)
- `-regextype posix-extended -regex` :pour faire une recherche par regex étendue (cf section 3.10), exemple `$ find /usr/bin -regextype posix-extended -regex ".*[aeiouy].*[0-9]{2,}.*"`
- `-exec` : pour exécuter une commande sur chacun des fichiers trouvés (au lieu d'en afficher la liste), `$ find . -name "*.txt" -exec wc -l {} \;` compte le nombre de ligne de chacun des fichiers `.txt` dans l'arborescence sous mon répertoire courant.
  Dans `-exec`, il faut indiquer `{}` pour le nom du fichier trouvé par find sur lequel la commande est executé, et `\;` à la fin de la commande [2]
  Attention le comportement de `-exec` n'est pas du tout le même qu'un tube, en particulier `$ find . -name ‘‘*.txt’’ -exec wc -l {} \;` compte le nombre de lignes de chaque document trouvé, alors que `$ find . -name "*.txt" | wc -l` compte le nombre de fichier trouvés.

3.10 Rechercher dans un texte : `grep` et regex étendue

Pour rechercher des motifs dans un ficher (ou flux) texte on utilise la commande `grep` (pour *Global Regular Expression Print*).
La syntaxe générale de cette commande est `$ grep [options] motif fichier`, par exemple pour rechercher pour rechercher le mot `ring` dans le fichier `samwise.txt` on utilise `$ grep "ring" samwise.txt`. Pour rechercher `ring` dans tous les fichiers `txt` du répertoire on peut utiliser `$ grep "ring" *.txt`. Les options usuelles de `grep` sont :

- `-n` (*number*) : affiche le numéro de ligne des occurences trouvées du motif
- `-i` (*insensitive*) : recherche insensible à la casse (min/maj)
- `-c` (*count*) : compte le nombre de ligne où le motif a été trouvé (et non pas le nombre d'occurences)
- `-o` (*only-matching*) : affiche uniquement les occurences trouvées du motif (et non pas les lignes en entier)
- `-E` (*Extended Regex*) : utilise des regex étendues au lieu des regex globulaires
- `-v` (*inv*ert) : inverse la recherche c'est à dire affiche les lignes où le motif n'est pas trouvé

Les regex étendues utilisées par `grep -E` et `find --regextype posix-extended -regex` sont plus expressives et puissantes que les expressions globulaires (regex simples). Comme dans les expressions globulaires il y a des caractères standard (`a`, `1`, etc) qui sont interprétés littéralement, et des caractères spéciaux (ou "métacaractères") qui permettent par exemple de représenter des classes ou des répétitions de caractères. Pour plus de détails vous pourrez consulter wikibooks:POSIX-regex.
Les principaux métacaractères sont :

- `.` : n'importe quel caractère
- `*` : répétitions (entre 0 et $+\infty$) de la classe de caractères précédente, attention n'importe quelle chaîne de caractère est `".*"` et non pas `"*"` car il faut préciser la classe à laquelle appartient le caractère répété

2. on peut indiquer `+` à la place et le comportement sera différent, vous pouvez consulter la documentation pour les différences.

- ▷ `+` : répétitions (entre 1 et $+\infty$) de la classe de caractères précédente
- ▷ `?` : 0 ou 1 occurrences de la classe de caractère précédente
- ▷ `{n,m}` : entre n et m occurrences de la classe de caractère précédente (`{n,}` pour au moins n occurences)
- ▷ `[... ]` : une classe de caractères, cette classe peut être :
  - — par énumération : `[aeiouy]` = une voyelle,
  - — par complémentaire avec `^` au début de la classe : `[^abc123]` = un caractère qui n'est ni `a`, ni `b`, ni `c`, ni `1`, ni `2`, ni `3`
  - — par plage : `[a-t]` = une lettre minuscule entre `a` et `t` ; `[0-5]` = un nombre entre `0` et `5`
  - — standard : `[[:digit:]]` (chiffres), `[[:alpha:]]` (alphabétique), `[[:space:]]` (whitespace char), `[[:alnum:]]` (alphanumérique) (pour une liste complète voir wiki:Posix-regex)
- ▷ `^` : placé en début de motif, force à ce que le motif commence en début de ligne
- ▷ `$` : placé en fin de motif, force à ce que le motif finisse à la fin de la ligne
- ▷ `\` : caractère d'échappement, permet de mettre dans un motif un caractère qui serait sinon interprété comme un métacaractère, par exemple `\.` est un caractère `.` et non pas n'importe quel caractère
- ▷ `(...)` : un groupe (que l'on peut ensuite répéter), par exemple `"(ab)*"` désigne les répétitions du bloc `ab` donc `ab`, `abab`, `ababab`, ...
- ▷ `|` : une disjonction, par exemple `"error|warning"` ou `"(error|warning)"` désigne soit `error` soit `warning`

On utilise souvent `grep` soit sur un ou des fichiers textuels, ou alors via un *pipe* pour filtrer la sortie standard ou d'erreur d'une commande.

3.11 Manipulations simples de texte : `cut`, `tr`, ...

On peut aussi modifier les flux textuels à la volée avec des utilitaires qui vont nous permettre de, par exemple, rendre plus lisible les résultats de certaines commandes.
Le premier de ces utilitaires est `tr` (pour *translate*) qui permet de transformer ou supprimer des caractères. Attention, `tr` s'applique sur l'entrée standard, donc pour l'appliquée sur un fichier texte il faudra faire une redirection d'entrée standard avec `<` (voir exemples). La syntaxe est `$ tr [options] classe1 [classe2]`. Les classes ici peuvent être des listes de caractères (`'abc'` = caractères `a`, `b` et `c`) ou des classes standard `[:lower:]`, `[:upper:]`, `[:digit:]`, etc.
Par exemple pour majusculer un flux de texte on pourra utiliser `$ tr '[:lower:]' '[:upper:]'`.
Les options les plus courantes sont :

- ▷ `-s` (*squeeze*) : supprime les répétitions. Exemple `$ tr -s ' ' < frodo.txt` : renvoie le texte de `frodo.txt` où les répétitions de l'espace ont été supprimées
- ▷ `-d` (*delete*) : supprime les caractères. Exemple `$ tr -d ' ' < frodo.txt` : renvoie le texte de `frodo.txt` où les espaces ont été supprimées
- ▷ `-c` (*complement*) : traduit le complément de la classe1 au lieu de la classe1. Exemple `$ tr -cs '[:alpha:]' <frodo.txt` supprime les répétitions de tous les caractères non-alphabétiques de `frodo.txt`

Notre second utilitaire est `cut` qui permet de découper les lignes du flux entrant en colonnes (avec un délimiteur indiqué par l'option `-d`) et de n'en conserver qu'une sélection (option `-f` pour *fields*).
Par exemple `$ cut -d',' -f1,2 data.csv` va séparer le fichier `data.csv` par virgules (ça tombe bien pour un fichier *CommaSeparatedValue*) et n'afficher que les colonnes 1 et 2. Vous pourrez aussi tester `$ ls -l / | tr -s ' ' | cut -d' ' -f3,9`.
On peut aussi utiliser `uniq` pour supprimer les répétitions de lignes du flux d'entrée, par exemple `$ls -l /usr/bin | tr -s ' ' | cut -d' ' -f3,4 | uniq`.

Et au contraire de `tr -s ' '` on veut parfois ajouter des espaces pour avoir un résultat plus joli (càd un tableau), pour cela on utilise la commande `column` avec l'option `-t`, par exemple `$ls -l / | tr -s ' ' | cut -d' ' -f3,4,9 | column -t`.
On peut aussi vouloir trier un fichier ou une sortie, pour cela on utilise la commande `sort`. Attention le tri par défaut est lexicographique (comme dans le dictionnaire) donc `"1 "` est plus petit que `"14"` qui est lui même plus petit que `"9"`. Pour trier en tant que nombre on utilise l'option `-n` (*number*). On peut aussi trier dans l'ordre décroissant avec `-r` (*reverse*), ou alors randomiser l'ordre avec `-R` (*Random*).
Dans certains cas on veut dupliquer la sortie d'une commande afin de la stocker dans deux fichiers différents, ou bien de l'afficher à l'écran et de la stocker dans un fichier. Pour cela on utilise la commande `tee` (le nom réfère à un "branchement en T" qui, en plomberie, connecte une arrivée d'eau à deux sorties). La syntaxe est alors `$ command | tee >fichier1 >fichier2` pour écrire la sortie de `command` dans deux fichiers, ou alors `$ command | tee >fichier` pour à la fois écrire la sortie de `command` dans un fichier et l'afficher.

3.12 Manipulations avancées de texte : `sed`, `awk`

Lorsque l'on veut faire des manipulations plus complexes sur des flux textuels on peut faire appel à des commandes plus avancées comme `sed` (*Stream EDitor*) ou `awk` (pour les initiales des créateurs du langage de script associé à la commande : A. Aho, P. Weinberger et B. Keirnighan).
On ne va pas traiter tous les détails de ces deux commandes (qui sont très puissantes mais avec des syntaxes un peu lourdes parfois), pour plus de détails vous pouvez consulter GNU-doc:sed ($\approx 4\,000$ lignes) et GNU-doc:awk ($\approx 25\,000$ lignes).
L'entrée de `sed` est soit l'entrée standard (par exemple avec un *pipe* `|`) soit un fichier texte.
Le squelette de la commande est donc `$ sed SCRIPT FICHIER` ou `$ sed SCRIPT` (dans le cas où le flux textuel est dans l'entrée standard). Le `SCRIPT` qui décrit la transformation opérée par `sed` est une suite de commandes qui est appliquée successivement (et indépendamment) à chaque ligne du flux textuel. La commande la plus courante est `s` (pour *substitute*, ou *search and replace*) dont la syntaxe est `'s/regex/replacement/flags'` où `regex` est l'expression régulière décrivant le motif à remplacer, `replacement` est la façon dont le motif est transformé, et `flags` (optionnel) sont des options.
Par exemple `$ sed 's/the ring/the one ring/' frodo.txt` remplace dans chaque ligne de `frodo.txt` la première occurrence de `"the ring"` par `"the one ring"`.
Si on rajoute l'option `g` (script `'s/the ring/the one ring/g'`), toutes les occurrences de `"the ring"` sont remplacées (et non pas la première de chaque ligne).
Les options les plus courantes sont :

- `g` la substitution s'applique à toutes les occurrences (et non pas la première de chaque ligne), donc le script script `'s/the ring/the one ring/g'` remplace toutes les occurrences de `"the ring"` (et non pas une par ligne)
- `i` la regex est insensible à la case (*case-insensitive*)

Les regex de `sed` ont presque la même syntaxe que les expressions étendues (de `grep -E`) avec des métacaractères en plus :

- `\w` (*word*) un caractère textuel (lettre, chiffre, underscore)
- `\W` un caractère non-textuel (tout le reste)
- `\b` (*boundary*) les bords d'un mot : la première lettre du mot, et le premier caractère après la fin du mot
- `\B` un caractère textuel qui n'est pas `\b` (donc qui est dans un mot mais pas au début du mot)
- `\s` (*spaces*) une espace ou tabulation
- `\<` (*begin*) le début d'un mot (première lettre)
- `\>` (*end*) la fin d'un mot (premier caractère après le mot)

- ▷ `\‘` le début du flux (à ne pas confondre avec `^` le début d'une ligne)
- ▷ `\’` la fin du flux (à ne pas confondre avec `$` la fin d'une ligne)
- ▷ des blocs délimités par `\( \)` par exemple `’gr\([ae]\)y’`

Le `replacement` est une chaîne de caractères avec aussi des caractères spéciaux :

- ▷ `&` représente l'intégralité du motif trouvé
- ▷ `\n` représente le *n*ième bloc délimité par des `\(` et `\)` dans la regex (numérotés de 1 à 9, si plus de 9 blocs nécessaires il faut passer à un langage de traitement de flux plus puissant, comme `awk`)
- ▷ `\u` majusculise le prochain caractère
- ▷ `\U` majusculise les prochains caractères (jusqu'à une instruction qui y mette fin)
- ▷ `\l` minusculise le prochain caractère
- ▷ `\L` minusculise les prochains caractère
- ▷ `\E` met fin à la conversion min-maj

Par exemple :

- ▷ `$ sed ’s/\<./\u&/g’` majuscule le début de tous les mots
- ▷ `$ sed ’s/\<./\u&/’` majuscule le début du premier mot de la ligne
- ▷ `$ sed ’s/\(\w*\)\(\w\)\(\<\)/\L\1\u\2\3/g’` minisculise le début et majusculise la dernière lettre de chaque mot, ici `\1` fait référence au premier bloc donc `\(\w*\)` qui est une répétition de caractères textuels, `\2` fait référence au second bloc qui est un unique caractère textuel, et `\3` fait référence au troisième bloc qui est une fin de mot (caractère non-textuel). La commande minusculise donc tout sauf la dernière lettre des mots, et majusculise la fin des mots.
- ▷ `$ sed ’s/\(the ring|the one ring\)/\U&/gi’` majusculise toutes les occurrences de `the ring` et `the one ring` (avec regex-matching case-insensitive)

`awk` est une commande encore plus puissante. Les commandes de traitement du flux dans `awk` (qui est à la fois le nom de l'interpréteur et du langage) forment un véritable langage de programmation dans lequel il y a en particulier des boucles `for` et des variables. Nous n'allons pas ici la présenter en détail mais seulement donner une première approche de la syntaxe et quelques exemples.
La syntaxe simplifiée de la commande est `$ awk ’PROGRAM’ FILE` où `’PROGRAM’` est un programme écrit en syntaxe `awk` et `FILE` est un flux ou fichier textuel (ou `$ awk -f program-file text-file` ou `program-file` contient le programme, et `text-file` contient le fichier à traiter).
Voici les fameux exemples que je vous ai promis :

- ▷ `$ awk ’BEGIN { print "Hello World!"}’` affiche `Hello World!`
- ▷ `$ awk ’{ print }’ frodo.txt` affiche le contenu de `frodo.txt` (se comporte comme `cat`)
- ▷ `$ awk ’/hobbit/ {print $0}’ *.txt` affiche toutes les lignes contenant le mot `hobbit` dans les fichier `.txt` : les `/ /` délimitent une regex à chercher, et `$0` désigne la ligne courante
- ▷ `$ awk ’length($0) > 80’ frodo.txt` affiche les lignes les lignes de longueur plus que 80 de `frodo.txt`, vous remarquerez qu'il y a juste un test et que l'action n'est pas spécifiée, l'action par défaut (imprimer la ligne courante) est donc appliquée
- ▷ `$ awk -F’,’ ’NR > 1 {sum += $3} END { if (NR > 0) print sum / (NR - 1)}’ notes_partiel.csv`

  calcule la moyenne de la colonne 3 (représenté par le `$3`, avec séparateur de colonne `’,’` indiqué avec `-F`) de `notes_partiel.csv` en excluant la première ligne (test `NR > 1` avec `NR` le numéro de ligne *Row Number*) qui contient un en-tête.

`awk` est une commande très puissante et rapide, et il est pertinent de l'utiliser pour des programme de traitement de flux textuel qui peut s'écrire en quelques lignes (ou quelques dizaines de lignes). Pour un programme de quelques centaines de lignes voire plus il est recommandé de se tourner vers d'autres langages comme Python (si vous voulez du code facile mais peu efficace) ou C (si vous voulez du code moins facile mais plus efficace).
Pour faire de l'édition interactive de texte, on peut aussi ouvrir un éditeur de texte dans le terminal : `nano` (`Ctrl+X` pour quitter l'éditeur interactif).

3.13 Encore des combinaisons de commandes : `||` et `&&`

On peut combiner les commandes de façon séquentielle indépendante avec `;` et de façon séquentielle avec redirection de sortie avec `|` , mais on peut aussi avoir des variantes de la combinaison séquentielle où l'exécution de la seconde commande dépend du succès ou non de la première commande. Pour cela on utilise le `||` (*or*) et le `&&` (*and*) :

- ▷ `$ commande1 || commande2` : `commande1` est exécutée, si son exécution est un echec alors `commande2` est exécutée (sinon elle est ignorée)
- ▷ `$ commande1 && commande2` : `commande1` est exécutée, si son exécution est un succès alors `commande2` est exécutée (sinon elle est ignorée)

Par exemple :

- ▷ `$ touch /test.txt || echo "je n'ai pas les droits pour touch dans /"` donne dans la sortie standard :

  ```
  >>> touch: cannot touch '/test.txt': Permission denied
      je n'ai pas les droits pour touch dans /
  ```

- ▷ `$ touch /test.txt && echo "je n'ai pas les droits pour touch dans /"` donne dans la sortie standard :

  ```
  >>> touch: cannot touch '/test.txt': Permission denied
  ```

  car la seconde commande n'est pas exécutée puisque la première est un échec.
- ▷ `$ touch /tmp/test.txt || echo "j'ai les droits pour touch dans /tmp"` n'affiche rien (car la première commande n'affiche rien et la seconde n'est pas exécutée
- ▷ `$ touch /tmp/test.txt || echo "j'ai les droits pour touch dans /tmp"` donne dans la sortie standard :

  ```
  >>> j'ai les droits pour touch dans /tmp
  ```

  puisque la seconde commande est bien exécutée puisque la première est un succès

Chapitre 4

Droits dans le système de fichier

4.1 Les droits dans le système Unix

Les systèmes Unix sont par design multi-utilisateurs. Historiquement, les ordinateurs étaient volumineux et coûteux (équivalent d'un serveur de calcul ou d'un datacenter actuel) et une entreprise ou université (ou administration publique) avait usuellement un unique ordinateur (serveur) auquel se connectaient plusieurs clients légers (terminaux).
De même, les ordinateurs des salles de TP sont multi-utilisateur [1], au même titre qu'un ordinateur familial (avec un compte user [2] par utilisateurice réel) est multi-utilisateur.
En plus des *end-user* (les utilisateurices humain·e·s) il y a dans un système Unix des *users* spéciaux dont le plus connu est `root` (le super-utilisateur).
Vous pouvez consulter la liste complète des users (end-user et users spéciaux) avec la commande `$ cat /etc/passwd`.
Afin de gérer la confidentialité et la sécurité, chaque fichier dans un système unix a un propriétaire et des droits d'accès, d'édition et d'exécution.
Afin de gérer ces droits, chaque fichier (ou dossier, ou fichier spécial) a :

1. un propriétaire désigné par `u` (*user*)
2. un groupe désigné par `g` (*group*)
3. le reste du monde désigné par `o` (*other*)

Pour chaque fichier, le propriétaire (`u`), le groupe (`g`) et les autres (`o`) ont un ensemble de droits séparés.
Pour chaque catégorie, et chaque fichier il y a trois types de droits :

- ▷ lecture `r` (*read*) : permet d'accéder au contenu pour un fichier normal, et d'accéder à la liste du contenu pour un dossier
- ▷ écriture `w` (*write*) : permet d'éditer le contenu pour un fichier normal, et d'éditer la liste du contenu pour un dossier
- ▷ exécution `x` (*execute*) : permet d'exécuter le contenu pour un fichier normal (qui est donc considéré comme un exécutable), et de "traverser" pour un dossier , c'est à dire pour accéder à un fichier ou sous-dossier dont on connais le chemin.

4.2 `ls -l` le retour

Pour consulter les droits d'un fichier, on utilise l'option longue de `ls`.
En particulier l'output de `$ ls -l f*` est

1. En vrai c'est plus compliqué que ça, mais on va faire semblant
2. Dans ce manuscrit j'essaye d'utiliser le terme utilisateurice pour la personne humaine derrière le clavier, et user pour l'entité symbolique dans le système d'exploitation.

```
>>> -rw-r--r-- 1 frodo fellowship    0 26 mars  15:48 frodo.pdf
    -rw-r--r-- 1 frodo fellowship  310 27 mars  12:26 frodo.txt
```

Le premier caractère de la ligne représente le type de fichier (`-` pour un fichier normal, `d` pour un dossier, `l` pour un lien, `c` pour un fichier spécial I/O). Ensuite on a les droits du propriétaire sur trois caractères, les droits du groupe sur trois caractères puis les droits des autres. Dans la plage de droits, un `-` signifie qu'il n'y a pas le droit correspondant. Dans notre exemple `frodo.txt` a les droits `rw-r--r--` donc le propriétaire a les droits de lecture et d'écriture, mais le groupe et les autres n'ont que les droits de lecture ; et personne n'a les droits d'exécution.
Plus en détail on peut décoder l'output de `$ ls -l` de la façon suivante :

```
      +-permissions du propriétaire
      |
      |     +-permissions des autres
      |     |
      |     |  +-nombre de liens physiques
      |     |  |
      |     |  |                  +-taile     +-timestamp (dernière modification)
     _|_   _|_ |                 _|__ ________|_______
    -rw-r--r-- 2 frodo fellowship 1024 2026-03-30 10:44 frodo.txt
    |   ___      _____ __________                      _________
    |    |         |    |                               |
    |    |         |    |                               +- nom du fichier
    |    |         |    |
    |    |         |    +-groupe
    |    |         |
    |    |         +-propriétaire
    |    |
    |    +-permissions du groupe
    |
    +-type
```

4.3 Changer les droits

Lorsque l'on est propriétaire d'un fichier, on peut en modifier les droits avec `chmod` et même le propriétaire avec `chown` (attention le changement de propriétaire peut se révéler irréversible si vous n'avez pas les droits pour revenir en arrière). Lorsqu'on bénéficie des droits d'administration (`sudo`) on peut modifier les droits et propriétaires de tous les fichiers (mais cela peut se révéler dangereux, à faire avec précautions).
Il y a deux syntaxes possibles pour la commande `chmod` :

- ▷ `$ chmod classe(mod)droits fichier` (syntaxe longue ou logique)
- ▷ `$ chmod nombre fichier` (syntaxe courte ou numérique)

En syntaxe longue (ou logique), la `classe` désigne les catégories (propriétaire `u`, groupe `g`, autres `o`, ou combinaisons des catégories [3]) affectées, `(mod)` est soit `+` (ajout de droits), `-` (suppression de droits), `=` (spécification des droits).
Par exemple, sur le fichier `frodo.sh` dont les droits sont `rw-r-----` (les commandes sont exécutées successivement) :

- ▷ `$ chmod u+x frodo.sh` transforme les droits en `rwxr-----`
- ▷ `$ chmod g+w frodo.sh` transforme les droits en `rwxrw----`
- ▷ `$ chmod -w frodo.sh` transforme les droits en `r-xr-----` (pour `+` et `-` on peut ommetre la catégorie auquel cas ça s'applique à toutes les catégories)

3. On peut utiliser `a` pour *all* donc `a` est équivalent à `ugo`.

▷ `$ chmod o=r frodo.sh` transforme les droits en `r-xr--r--`

En syntaxe courte (ou numérique) on représente les droits de chaque catégorie par un chiffre entre 0 et 7. Ainsi on pourra ainsi écrire `$ chmod 754 frodo.sh`.
Pour coder les droits d'une catégorie en chiffre, on convertis les droits en nombre binaire sur 3 bits (le droit est actif →`1`, le droit est désactivé →`0`) dans l'ordre usuel *read-write-execute* puis on convertis ce nombre binaire à 3 chiffres en un chiffre décimal entre $(000)_2 = (0)_{10}$ et $(111)_2 = (7)_{10}$, ainsi `rwx → 111 → 7`, `rw- → 110 → 6`, ..., `--x → 001 → 1` et `--- → 000 → 0`.
Ainsi `754` code les droits `rwxr-xr--`, donc la commande `$ chmod 754 frodo.sh` donne les droits `rwxr-xr--`.
En syntaxe numérique on peut aussi modifier les droits existants avec `+` (ajout de droits) et `-` (retrait de droits). Par exemple `$ chmod +111 frodo.sh` donne les droits d'exécution à toutes les catégories, et `$ chmod -007 frodo.sh` retire tous les droits à `o` (les autres).

4.4 Gestion des `users`

On peut créer, modifier, supprimer, donner des droits etc à des `users` en ligne de commande. En revanche cela nécessite les droits d'administration (les droits `"sudo"`) et on ne va pas traiter cela dans ce cours.

Chapitre 5

Processus

5.1 Multiprogrammation

Lorsqu'on utilise un ordinateur, de nombreuses tâches sont exécutés «simultanément». Cela est vrai quel que soit le nombre de processeurs et quel que soit le nombre de tâches (gestion input/output, applications, calculs, rendering video, etc).
Dans un système unix (et dans presque tous les systèmes informatiques modernes), chaque tâche indépendante est appelée processus.
La multiprogrammation (ou programmation multitâche) est implémentée en changeant très souvent le processus qui est effectivement en cours d'exécution.
Il existe de nombreux paradigmes de multiprogrammation dont les deux principaux sont le multitâche coopératif (le processus en cours décide à quel moment il donne sa place au suivant) et le multitâche préemptif (l'ordonnanceur peut interrompre un processus à tout moment), de fait les ordinateurs de nos jours ont presque tous un système hybride entre ces deux grands principes.
La discipline informatique qui étudie cela est appelée l'ordonnancement.

5.2 Ordonnancement

5.3 Ordonnancement naïf FIFO

Le système d'ordonnancement le plus simple est le naïf FIFO (*first-in-first-out*), les processus sont exécutés un par un dans leur ordre de création et ce jusqu'à ce qu'ils soient terminés. Ce système n'est absolument pas multitâche car il attend que le processus termine avant de commencer à exécuter un nouveau processus.

5.4 Round-robin

Le système d'ordonnancement multitâche le plus simple est le Round-robin. Dans un système d'ordonnancement Round-robin sur un processeur, chaque processus est exécuté durant un quantum de temps δ (usuellement de l'ordre de 10 à 100 millisecondes) puis rend la place au prochain processus dans la liste d'attente (et se place à la fin de la liste d'attente).
Dans un système Round-Robin chaque processus peut avoir deux états («en attente» ou «en cours d'exécution») jusqu'à être terminé.
Ce système est rarement appliqué tel quel, car il traite de la même façon des processus critiques (comme par exemple `watchdog` dont le rôle est de s'assurer qu'aucun module du kernel n'est en erreur) et les processus de plus faible importance (comme par exemple mon navigateur web).

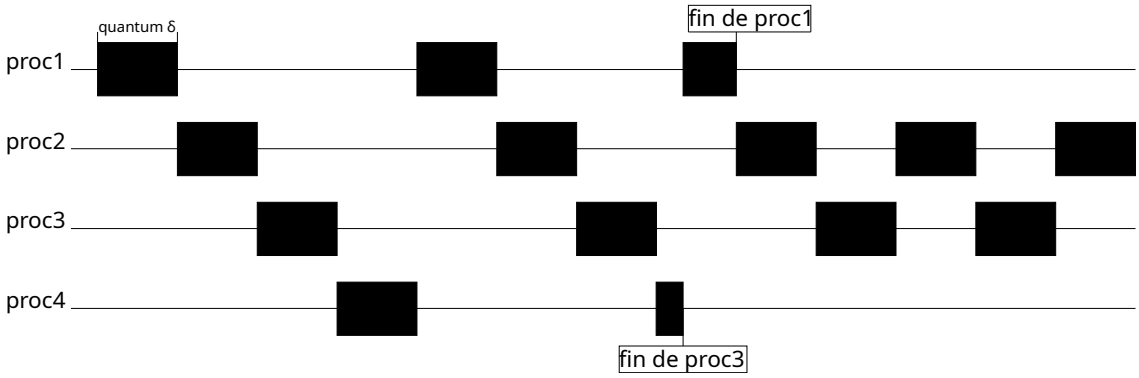


FIGURE 5.1 – L'exécution de 4 processus dans un Round-Robin parfait.

Une solution possible est de gérer la liste d'attente en prenant en compte la priorité, l'objectif étant que tous les processus (même de faible priorité) aient un peu de temps de processeur mais que les processus les plus prioritaires aient une part importante du temps processeur.

5.5 Scheduler linux

Le scheduler linux est un modèle hybride à plusieurs niveaux (voir par exemple ubuntu-explanation:scheduler pour une description synthétique de son fonctionnement).
Les processus les plus critiques sont gérés en mode `SCHED_FIFO` (ils s'exécutent jusqu'à rendre la place aux autres processus), les processus intermédiaires utilisent un Round-Robin légèrement modifié `SCHED_RR`, et les processus standard utilisent un système étendu `SCHED_OTHER` (qui est une variante de Round-Robin où le quantum de temps processeur alloué est déterminé dynamiquement, et où la file d'attente est pondérée par de la priorité).
Dans ce système d'ordonnancement les processus ont plus d'états possibles dont les principaux sont :

- ▷ Élu : en cours d'exécution
- ▷ Prêt / en attente : dans la liste d'attente du scheduler
- ▷ Bloqué / endormi : le processus a été interrompu ou attend un événement (I/O ou fin d'un autre processus)

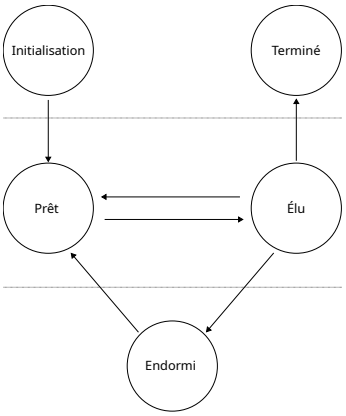


FIGURE 5.2 – Un schéma simplifié des états d'un processus.

5.6 Commandes shell : `ps`, `top`

La commande shell pour afficher les processus existant est `ps` (pour *processes*). Sans options/arguments, cette commande affiche les processus de la session shell ouverte. Afin d'afficher l'intégralité des processus on peut utiliser `$ps aux` (syntaxe BSD) ou `$ps -ef` (syntaxe POSIX). On pourra alors déterminer le nombre total de processus existants avec `$ps -ef | wc -l`.
On peut aussi filtrer les champs affichés avec l'option `-o` , les champs usuels sont :

- `pid` (*process ID*) : le numéro unique du processus
- `cmd` (*command*) : la commande du processus (exemple `firefox`)
- `uid` (*user ID*) : le numéro unique du *user* qui a lancé le processus
- `time` : temps d'utilisation effective du processus
- `stime` (*start time*) : heure à laquelle le processus a été lancé
- `ppid` (*parent process ID*) : le PID du processus parent (celui qui a lancé le processus)
- `rtprio` (*real time priority*) : le niveau de priorité du processus pour le scheduler
- `stat` (*status*) : l'état du processus (`R` = *running*, `S` = *sleeping*,...)
- `policy` (*scheduling policy*) : le type d'ordonnancement du processus (`TS` = `SCHED_OTHER`, `FF` = `SCHED_FIFO`, ...)
- `pcpu` (*percentage of cpu*) : pourcentage d'usage CPU (ratio entre le temps d'exécution processeur et le temps de vie du processus)
- `size` (*memory size*) : quantité de mémoire utilisée

Par exemple si on souhaite seulement afficher les PID, commandes et PPID on peut utiliser la commande `$ ps -e -o pid,cmd,ppid` ou `$ ps -eo pid,cmd,ppid`
On peut de plus trier les résultat avec `--sort=<champ>` (on peut indiquer `+` ou `-` devant le champ pour ordre croissant ou décroissant). Par exemple pour trier par pourcentage d'usage cpu et afficher les dix premiers processus on peut utiliser
`$ ps -eo pid,cmd,pcpu,size --sort=-pcpu | head`
`ps` est un outil de snapshot statique, et non pas un outil de monitoring. Lorsque l'on souhaite pouvoir observer en temps réel les processus et leur utilisation du matériel (mémoire, CPU) on peut utiliser la commande `top` (qui ouvre une interface interactive que l'on quitte avec `q`) ou une variante comme `htop` ou `btop`.
Une fois qu'on a obtenu le PID (process ID) d'un processus, on peut obtenir plus d'information sur lui avec la commande `chrt` ou en consultant sont fichier de status. Par exemple pour consulter le processus de PID `109` je peux consulter son fichier de status avec `$ cat /proc/109/status`, ou avec `$ chrt -p 109`. Il est aussi possible de stopper un processus avec `kill` (à vos risques et périls) avec, pour notre exemple `$ kill 109`.