0.1 L'arborescence typique d'un système Unix

Les fichiers qui sont utiles au système Unix sont souvent placés à des adresses communes à tous les systèmes installés, établies principalement par des habitudes des super-users (`root`) dans le monde entier, donc par le poids de l'histoire. D'un système à l'autre, on constate cependant quelques variations qui sont généralement motivées par les objectifs principaux du système considéré. La spécification de ces différences constitue l'essentiel de ce qu'on appelle une « distribution » d'Unix. Parmi ces distributions, on peut citer BSD, Mageia, Fedora, RedHat, Debian, Ubuntu, *etc.* D'une *distribution* à une autre, la structure de l'arborescence peut donc changer mais les répertoires majeurs pour les utilisateurs restent :

`/home` : contient les home directories des utilisateurs.

`/usr` : contient la majorité des commandes utiles aux utilisateurs ainsi que les bibliothèques et les fichiers de paramètres correspondants. Par exemple `/usr/bin` contient la plupart des commandes que nous utiliserons dans ce cours (`bin` est une abréviation de « binaries ») ; le répertoire `/bin` contient aussi des commandes indispensables au système Linux.

`/dev` : contient tous les fichiers de type « devices » et leur gestion. On peut noter en particulier `/dev/null` qui est un fichier très particulier : tout le monde peut le lire et écrire dedans mais il reste toujours vide, c'est une « poubelle de données » souvent utile pour passer sous silence des messages inutiles comme on le verra plus loin.

0.2 Manipulation de l'arborescence

En bref :

- Pour « se déplacer » dans l'arbre des répertoires, on utilise la commande `cd` en lui donnant pour argument l'adresse (absolue ou relative) du répertoire où l'on veut aller.
- Pour créer un répertoire, on utilise `mkdir` en lui donnant pour argument l'adresse (absolue ou relative) du répertoire que l'on veut créer. Le répertoire père doit préexister.
- Pour renommer et/ou déplacer un fichier ou un répertoire dans l'arborescence, on utilise la commande `mv` en lui donnant pour premier argument l'ancienne adresse et pour second argument la nouvelle adresse.
- Pour copier un fichier, on utilise la commande `cp` avec pour premier argument l'adresse du fichier d'origine et pour second argument l'adresse du nouveau fichier.
- Pour copier un répertoire entier, on peut utiliser la commande `cp` avec l'option `-r` en premier argument, le répertoire d'origine en second argument et l'adresse du nouveau répertoire en troisième argument. (Voir aussi la commande `tar` pour une copie plus avancée de répertoires).
- La commande `man` fournit un manuel en ligne pour toutes les commandes de base du système. Faire `man` pour toutes ces commandes pour en savoir plus. Faire également `man ls` pour extraire des informations sur les répertoires et fichiers d'une arborescence.

1 Les processus

Un processus est un programme en train de tourner sur l'ordinateur à un moment donné. Il a été lancé par un autre processus (par exemple un processus shell) et il est alors considéré comme le *fils* du processus qui l'a lancé. Pour ce faire, le processus père a généralement chargé un fichier exécutable du système de fichier vers la mémoire et a ensuite lancé l'exécution (en parallèle) des instructions mémorisées. Un processus est donc lancé par un autre processus et bien sûr cela pose la question du « premier » processus. C'est l'objectif du « boot » de la machine : un processus appelé `init` est lancé quand on démarre la machine. C'est donc l'ancêtre de tous les processus : il est à l'origine de tous les autres au travers d'un arbre de processus dont il est la racine puisqu'il lance tous les premiers processus, qui en lancent d'autres à leur tour, *etc.*

Un processus agit toujours au nom d'un unique utilisateur du système *et il en a tous les droits.* Cela a pour conséquence immédiate qu'il ne faut lancer un processus que si l'on a toute confiance en ce qu'il fait !

> La possibilité de savoir en détail ce que fait chaque programme qu'on utilise devrait être *un droit inaliénable* pour chaque citoyen car les actions des processus qui travaillent en son nom *engagent sa responsabilité.* Cela implique de pouvoir lire les sources et les spécifications de tout logiciel qu'on utilise. Les logiciels qui respectent ce droit sont dit « open source » ; cela ne signifie pas pour autant que l'usage du logiciel soit gratuit.

Naturellement peu de gens ont le loisir de lire et comprendre toutes les sources de tous les logiciels qu'ils utilisent, cependant les sources d'un logiciel open source sont lues par de nombreuses personnes et c'est ce « contrôle collectif » qui garantit les fonctionnalités exactes du logiciel à tous les utilisateurs (de même que le dépouillement publique des bulletins de vote garantit le respect de la démocratie même si nous n'y participons pas tous).

De manière surprenante de nombreux logiciels, non seulement n'offrent pas ces garanties, mais contiennent et exécutent de plus des fonctions non documentées ; les spécifications de ces logiciels sont donc pour le moins *incomplètes*. C'est particulièrement fréquent sur les smartphones par exemple, ou encore avec les moteurs de recherche ou les réseaux sociaux les plus connus. Ne soyez pas naïfs ni angéliques : n'utilisez ces logiciels qu'avec la plus grande méfiance.

Un processus possède toujours au moins les 3 canaux standard suivants :
- une entrée standard
- une sortie standard
- une sortie d'erreurs

Les shells savent manipuler ces trois canaux pour tous les processus qu'ils lancent. Par défaut, l'entrée standard est le clavier, la sortie standard est la fenêtre du terminal et la sortie d'erreur est également dirigée vers le terminal. Il est possible de les *rediriger* :
- On peut par exemple utiliser le contenu d'un fichier comme entrée standard, et tout se passe comme si l'on tapait au clavier chacun des caractères contenus dans le fichier
  `$ commande < fichierEnEntree`
- On peut créer un fichier contenant la sortie standard d'un processus (écrase le fichier s'il existait déjà)
  `$ commande > fichierResultat`

  On peut aussi ajouter à la fin d'un fichier existant
  `$ commande >> fichierResultat`
- On peut rediriger la sortie d'erreurs d'une commande (note : aucun espace au milieu de « `2>` »)
  `$ commande 2> fichierDesErreurs`

  Il arrive souvent qu'on veuille totalement ignorer les sorties d'erreur d'une commande et dans ce cas on utilise le fichier spécial `/dev/null`, dans lequel on peut toujours écrire mais qui reste toujours vide...
  `$ commande 2> /dev/null`
- Enfin on peut enchaîner les commandes en fournissant la sortie d'une commande en entrée d'une autre. On dit alors qu'on « pipe » les commandes (il faut prononcer *pipe* à l'anglaise)
  `$ commande1 | commande2`

2 Les processus « shell »

Un shell est un processus dont le rôle est par défaut d'attendre que l'utilisateur tape une commande au clavier, d'interpréter la ligne qu'il a tapée et de lancer le ou les processus qu'implique(nt) cette commande. Le rôle du « login » d'un utilisateur est de demander le lancement du premier processus shell travaillant au nom de la personne qui se logue. Le processus de login (qui agit au nom de `root`) lance alors ce shell comme processus fils, en le faisant appartenir à l'utilisateur qui s'est logué.

Lorsque l'on se logue avec un interface graphique, c'est en fait le shell « de login », dont l'entrée standard n'est plus votre clavier mais un fichier prédéfini, qui lance tous les processus qui vont gérer l'interface graphique pour l'utilisateur.

Ce n'est pas le seul usage d'un shell, par exemple lorsque vous ouvrez une fenêtre « de terminal » (qui simule les terminaux informatiques de l'époque où les interfaces graphiques n'existaient pas), vous obtenez une fenêtre dans laquelle tourne un shell, qui vous permet donc de lancer de nombreuses commandes, avec des finesses de choix des options qui seraient inaccessibles *via* les menus d'une interface graphique. Il existe plusieurs programmes qui sont des shells et nous n'utiliserons ici que le plus courant qui est `/bin/bash`.

Lorsqu'un processus shell est lancé, il dispose de plusieurs variablesen mémoire donnant les informations dont on se sert souvent. Par exemple :
- `USER` contient le nom de login du propriétaire du processus shell. Dans un shell, on obtient la valeur d'une variable en mettant un « $ » devant, ainsi la commande « `echo $USER` » depuis votre shell fournit votre nom de login.
- `HOME` contient l'adresse absolue de votre répertoire personnel.
- `PRINTER` contient le nom de votre imprimante préférée.
- *etc.*

La variable `PATH` mérite à elle seule un paragraphe d'explications :

Lorsqu'un shell doit lancer un processus à la suite d'une commande tapée par l'utilisateur, par exemple « `man ls` » :

- le shell doit d'abord trouver le fichier exécutable qui contient le code du processus à lancer, pour notre exemple le fichier `/usr/bin/man`
- ensuite il le charge en mémoire et indique au système qu'il faut l'exécuter avec les bons arguments, pour notre exemple l'argument « `ls` ».

Si l'exemple avait été « `openarena` » au lieu de « `man` » le shell aurait lancé `/usr/games/openarena`, qui se trouve donc dans un répertoire différent de `/usr/bin/`. Compte tenu du nombre de fichiers présents dans l'arborescence du système, il serait impensable de parcourir tout l'arbre des fichiers pour trouver chaque commande et c'est en fait la variable `PATH` qui contient la liste des répertoires que le shell va explorer pour trouver les commandes de l'utilisateur. Dans cette variable, les répertoires sont séparés par des « `:` ». Par exemple :

```
$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/games:/usr/lib/qt4/bin:/home/bioinfo/bernot/bin
```

La commande `which` permet de savoir où le shell trouve une commande donnée ; par exemple « `which man` » retourne `/usr/bin/man`.