 Gilles Bernot

Introduction au système Linux

COURS 1

1. Les 3 concepts fondamentaux d'un système
2. Les utilisateurs
3. Les fichiers

COURS 2

1. Les processus
2. Le shell
3. Les environnements graphiques
4. Les processus démons
5. Le réseau

COURS 3

1. Les expressions régulières sous Unix
2. Usage de `find`

COURS 4

1. La programmation en shell
2. Créer un shell script
3. La programmation en shell
4. Exemple

ANNEXE

Quelques commandes utiles

Nota : le poly contient aussi une fiche en annexe qui récapitule les principales commandes vues durant le cours.

1 Les 3 concepts fondamentaux d'un système

Un système informatique bien structuré met en œuvre trois concepts clefs :
- les *utilisateurs*, qui identifient de manière non ambigüe les acteurs qui peuvent utiliser le système, leurs droits et ce que le système peut faire en leur nom,
- les *fichiers*, qui sauvegardent de manière fiable à la fois les données, les programmes qui les manipulent et les paramètres qui déterminent les comportements du système et de ses services,
- enfin les *processus*, qui sont les programmes en train de « tourner » sur la machine à un moment donné.

Un utilisateur virtuel particulier, appelé `root`, a tous les droits sur le système. Certains fichiers *appartiennent* à `root` ; ce sont typiquement ceux qui gèrent la liste des utilisateurs autorisés à utiliser le système et leurs droits, ceux qui contrôlent le comportement global du système, *etc.*

Un fichier appartient toujours à un utilisateur du système. De même un processus appartient et agit toujours au nom d'un utilisateur du système et l'utilisateur est donc responsable des actions de ses processus. Hormis `root`, seul l'utilisateur propriétaire d'un fichier ou d'un processus peut lui appliquer toutes les modifications qu'il souhaite. Les processus appartenant à un utilisateur disposent eux aussi de tous les droits de l'utilisateur ; ils sont en fait ses « bras agissant » au sein du système.

Le système repose donc sur des « boucles de rétroactions » systématiques entre fichiers et processus : les fichiers dits *exécutables* peuvent être chargés en mémoire pour lancer les processus... et les processus agissent entre autres en créant ou modifiant les fichiers du système, ou en lançant d'autres fichiers exécutables...

> Au démarrage du système, il est donc nécessaire de lancer un premier processus, qui en lancera ensuite un certain nombre d'autres (dont l'écran d'accueil pour les « login », la surveillance du réseau, *etc.*) ; eux-mêmes lanceront d'autres processus et ainsi de suite. Le premier processus lancé au boot s'appelle généralement `init` et il tourne ensuite jusqu'à l'arrêt complet du système (par exemple il relance un écran d'accueil chaque fois qu'un utilisateur de « délogue », il gère les demandes d'arrêt du système, *etc.*). Le processus `init` appartient à `root`, naturellement.

2 Les utilisateurs

Le système identifie ses utilisateurs autorisés non seulement par leur « nom de login » mais aussi et surtout pas leur *identifiant*, qui est simplement un nombre entier positif : l'UID (User IDentification). C'est évidemment la combinaison du nom de login et du mot de passe associé qui permet au système de « reconnaître » ses utilisateurs autorisés et de leur autoriser l'accès à cet UID.

> Lorsque vous êtes connectés au système, vous pouvez connaître votre UID en lançant une fenêtre dite de terminal, et en y tapant la commande « `echo $UID` ».

Les utilisateurs sont structurés en *groupes* : chaque utilisateur appartient à un groupe par défaut (souvent le nom de l'équipe ou du service où il travaille).

Un utilisateur possède également un répertoire où le système le place par défaut au moment du « login », sauf exception l'utilisateur a tous les droits sur ce répertoire (créer des sous-répertoires, y placer des fichiers de son choix, *etc.*). Il s'agit de la « *home directory* » de l'utilisateur.

> Lorsque vous êtes connectés au système, vous pouvez connaître votre home directory avec la commande « `echo $HOME` ». C'est généralement quelquechose du genre `/home/monEquipe/monLoginName`.

Enfin un utilisateur possède un processus d'accueil : c'est le processus qui est lancé par le système au moment où il se connecte avec succès (nom de login et mot de passe reconnus). C'est aussi celui qui est lancé lorsque l'utilisateur ouvre une fenêtre de « terminal ». Ce genre de processus est appelé *un shell*. Il s'agit généralement de `/bin/bash` et ce processus attend tout simplement que l'utilisateur tape une commande (donc un nom de fichier) pour l'exécuter.

Lorsque l'utilisateur bénéficie d'un environnement graphique (c'est le cas en général), c'est en réalité un simple shell qui est exécuté au moment du login et ce dernier lance immédiatement les processus de l'environnement graphique au lieu d'attendre que l'utilisateur tape une commande.

Le fichier `/etc/passwd` contient la liste des utilisateurs reconnus par le système (dont `root` lui-même), à raison de un par ligne, et il contient les informations suivantes : nom de login, mot de passe (encrypté !), UID, GID (entier identifiant du groupe par défaut), vrai nom, home directory et login shell.

3 Les fichiers

Les fichiers sont les lieux de mémorisation durable des données de tout le système. Ils sont généralement placés physiquement sur un *disque dur* ou sur tout autre dispositif de grande capacité qui ne perd pas ses données lorsque le courant est éteint (disque SSD, mémoire flash, clef USB, CDROM, ...).

3.1 Les types de fichier

Les fichiers ne sont pas seulement ceux qui mémorisent des textes, des images, de la musique ou des vidéos :

- il y a des fichiers dont le rôle est de pointer sur une liste d'autres fichiers, on les appelle des *répertoires* ou *directories* (ou encore « dossiers » pour les utilisateurs naïfs ; oui, les dossiers sont des fichiers comme les autres)
- il y a des fichiers dont le rôle est de pointer vers un seul autre fichier, on les appelle des *liens symboliques* (ou encore « raccourcis » pour ces mêmes utilisateurs naïfs)
- il y a des fichiers dont le rôle est de transmettre vers un matériel périphérique ce qu'on écrit dedans, et de transmettre, en lecture cette fois, les informations transmises par le périphérique ; ces fichiers sont appelés des « *devices* » ou encore des « fichiers de caractères spéciaux »
- *etc.*

Les fichers auxquels les gens pensent habituellement, c'est-à-dire ceux qui ont un contenu avec des données, sont souvent appelés des fichiers *plats* par opposition aux fichiers répertoires, liens symboliques et autres.

La commande `file` permet de savoir quel est le type d'un fichier. Le nom de fichier est souvent choisi pour indiquer son type *via* sont suffixe (par exemple « `.txt` » pour du texte, « `.mp3` » pour du son, ...) mais cette indication n'est pas toujours fiable puisqu'on peut sans problème renommer un fichier à sa guise. La commande `file`, en revanche, analyse le contenu du fichier qu'on lui donne en argument pour déterminer son type.

```
$ file /home
/home: directory
$ file /etc/passwd
/etc/passwd: ASCII text
$ file /dev/audio
/dev/audio: character special
```

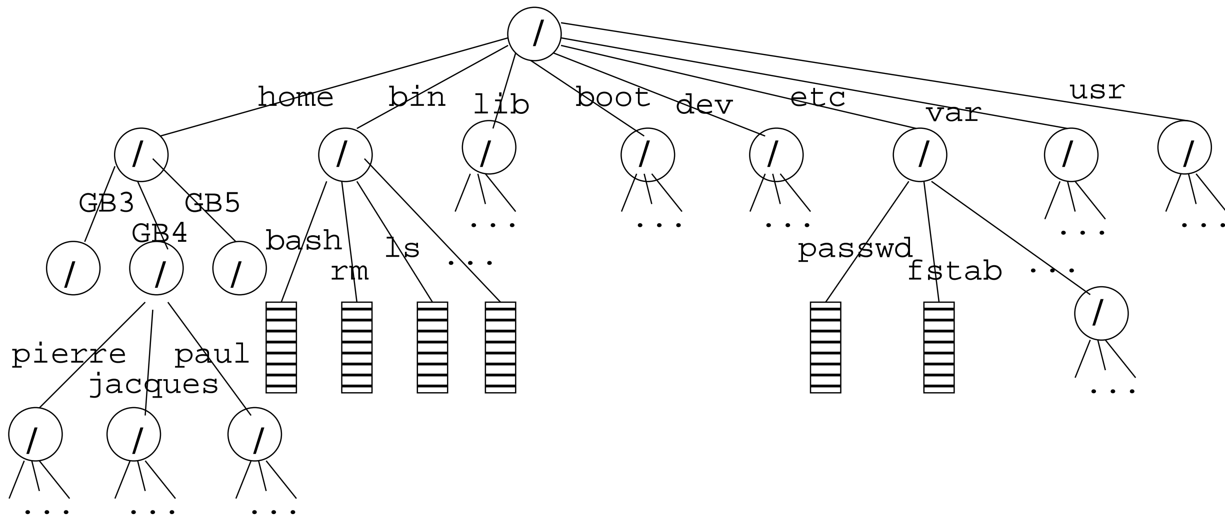
Plusieurs informations sont attachées aux fichiers. On en verra plusieurs dans cette section (propriétaire du fichier, droits d'accès, *etc.*) et l'on peut mentionner également deux informations souvent utiles :

- la date de dernière modification, qui est la date exacte (à quelques fractions de secondes près) où le fichier a vu son contenu être modifié (ou créé) pour la dernière fois,
- la date de dernière utilisation, qui est la date exacte où les données du fichier ont été lues (ou utilisées de quelque façon) pour la dernière fois.

A ce propos, on peut mentionner la commande `touch` qui positionne ces date à l'instant exact où elle est exécutée. Plus précisément, si *`fichier`* n'existe pas alors la commande « `touch` *`fichier`* » crée un fichier plat vide dont les dates de dernière modification et de dernière utilisation sont l'instant présent, et s'il préexistait, elle ne modifie pas le fichier mais positionne néanmoins ces deux dates à l'instant présent.

3.2 L'arborescence du système de fichiers

Les fichiers n'ont pas de nom !
Parler du « nom d'un fichier » est en réalité un abus de langage (que nous faisons tous). *Explication* :
Les fichiers, quel que soit leur type, sont identifiés par un numéro qui indique leur emplacement sur le disque dur (ou autre support). Il s'agit du *numéro de i-node*. Le système de gestion de fichiers gère un *arbre* dont la racine est un fichier de type répertoire qui appartient à `root`. Chaque répertoire est en fait un ensemble de *liens* (des « vrais » liens, pas des liens symboliques) vers les fichiers fils de ce répertoire. Ce sont ces liens qui portent des noms, pas les fichiers eux-mêmes. C'est le nom du lien qui pointe vers un fichier que l'on appelle abusivement le nom de ce fichier (que le fichier soit plat, un répertoire ou de tout autre type).



Un fichier (quel que soit son type) est donc caractérisé par son numéro de i-node mais il peut être par conséquent également désigné sans ambiguité par une *adresse* qui est le chemin suivi depuis la racine de l'arbre jusqu'à lui. Les noms portés par les liens des répertoires sont séparés par des « / » (exemple : `/home/etudiants/bio/paul`). Par conséquent, un nom de lien n'a pas le droit de contenir de « / » !

La racine de l'arbre a simplement `/` pour adresse.

Lorsque l'on « déplace un fichier », avec le gestionnaire de fichier ou avec la commande « `mv` *`ancien nouveau`* », on ne déplace rien en réalité : le fichier reste à sa place sur le disque dur et ne change pas de numéro de i-node ; ce qui change, c'est l'arborescence, donc seulement les contenus des fichiers de type répertoire qui la constituent.

Il en résulte que la commande `mv` prend un temps très court quelle que soit la taille du fichier.

Lorsqu'un processus tourne, en particulier lorsqu'un shell tourne, il possède un *répertoire de travail.* Ceci évite de répéter tout le chemin depuis la racine car l'expérience montre que la plupart des processus utilisent ou modifient des fichiers qui sont presque tous dans le même répertoire. Ainsi, pour un processus (dont le shell) on peut définir une adresse de fichier de deux façons :

- par son *adresse absolue*, c'est-à-dire le chemin depuis la racine, donc une adresse qui commence par un « `/` »
- ou par son *adresse relative*, c'est-à-dire le chemin depuis le répertoire de travail du processus ; une adresse relative ne commence jamais par un « `/` ».

C'est donc par son premier caractère qu'on distingue une adresse relative d'une adresse absolue.

Lorsque vous êtes sous un shell, la commande `pwd` (print working directory) vous fournit le répertoire de travail dans lequel vous êtes. La commande `cd` (change directory) vous permet de changer de répertoire de travail. Enfin la commande `ls` (list) fournit la liste des liens d'un répertoire (par défaut son répertoire de travail).

```
$ pwd
/home/bioinfo/bernot

$ cd bin

$ pwd
/home/bioinfo/bernot/bin

$ ls
i686  shells  x86_64

$ ls -a
. .. i686  shells  x86_64

$ ls i686 x86_64
i686:
RNAplot  hsim  wi  wx  wxd

x86_64:
wi  wx  wxd
```

```
$ ls /usr
X11R6  etc    include  lib64    local  sbin   src  uclibc
bin    games  lib      libexec  man    share  tmp
```

> *Devinette : c'est généralement une mauvaise idée d'avoir un nom de fichier qui contient un espace ou qui commence par un tiret ; pourquoi ?*

Lorsqu'un processus en lance un autre, il lui transmet son propre répertoire de travail[1].

Un répertoire n'est jamais vide car il contient toujours au moins deux liens fils :
- le lien « `.` » qui pointe sur lui-même
- et le lien « `..` » qui pointe sur le répertoire père dans l'arbre. Exception : dans le répertoire `/` les liens `.` et `..` pointent tous deux sur lui-même.

3.3 Les droits d'accès

Outre son numéro de i-node qui le caractérise, tout fichier quel que soit son type a :
- un *propriétaire*, qui est un utilisateur reconnu du système,
- et un *groupe*, qui n'est pas nécessairement celui par défaut de son propriétaire, ni même obligatoirement un groupe auquel son propriétaire a accès (bien que la plupart des fichiers aient en pratique le groupe par défaut de leur propriétaire) ;
- les « autres » utilisateurs, c'est-à-dire ceux qui ne sont ni propriétaire ni membre du groupe du fichier, peuvent néanmoins avoir le droit d'utiliser le fichier si le propriétaire l'accepte.

Tout fichier peut être *lu* ou *écrit/modifié* ou encore *exécuté*. Pour un fichier de type répertoire, le droit de lecture signifie l'autorisation de faire `ls`, le droit d'écriture signifie l'autorisation de modifier la liste des fils (création, suppression, renommage) et le droit d'exécution signifie l'autorisation d'y faire un `cd`. Pour un fichier plat, le droit d'exécution signifie généralement soit qu'il s'agit de code machine qui peut directement être chargé pour lancer un processus, soit qu'il s'agit des sources d'un programme (en python, en shell ou tout autre langage) qui peut être interprété par un programme *ad hoc* pour lancer un processus.

Le propriétaire peut attribuer les droits qu'il veut aux 3 types d'utilisateurs du fichier (lui-même, le groupe et les autres). Il peut même s'interdire des droits (utile s'il craint de faire une fausse manœuvre !). Il y a donc 9 droits à préciser pour chaque fichier :

	owner	group	others
read	4	4	4
write	2	2	2
exec	1	1	1
SUM	...	...	...

et l'encodage se fait par puissances de 2 de sorte que la somme de chaque colonne détermine les droits de chacun des trois types d'utilisateur. La commande pour modifier les droits d'un fichier est `chmod` ; par exemple :
- `chmod 640` *`adresseDeMonFichier`* donne les droits de lecture et écriture au propriétaire, le droit de lecture aux membres du groupe du fichier, et aucun droit aux autres.
- `chmod 551` *`adresseDeMonFichier`* donne les droits de lecture et exécution au propriétaire et au groupe, et seulement le droit d'exécution aux autres.
- `chmod 466` *`adresseDeMonFichier`* donne le droit de lecture au propriétaire, mais les droits de lecture et écriture aux membres du groupe ainsi qu'aux autres...

Le nombre à trois chiffre est appelé le *mode* du fichier (et le `ch` de `chmod` est pour « change »). Naturellement seuls le propriétaire d'un fichier et `root` peuvent effectuer un `chmod` sur un fichier.

Seul `root` peut changer le propriétaire d'un fichier.

Lorsqu'un utilisateur a le droit d'exécuter un fichier plat, le processus qu'il lance ainsi appartient à l'utilisateur qui l'a lancé, c'est-à-dire qu'***il agit en son nom*** et non pas au nom de l'utilisateur qui possède le fichier. ***Cela suppose donc d'avoir pleinement confiance en l'honnêteté du propriétaire de ce fichier...*** De même si vous utilisez un programme écrit par quelqu'un d'autre ou par une entreprise qui a intérêt à stocker des informations à votre sujet.

1. mais ce processus fils peut tout à fait décider de commencer par changer de répertoire de travail !

3.4 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 suivants sont généralement présents.

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

`/etc` : contient les fichiers qui paramètrent les fonctions principales du système (utilisateurs et mots de passe, structure du réseau local, structure générale de l'arborescence des répertoires, propriétés de l'environnement graphique, *etc*).

`/usr` : contient la majorité des commandes utiles aux utilisateurs ainsi que les bibliothèques et les fichiers de paramètres correspondants.

`/var` : contient la plupart des informations à propos de l'état courant du système (messages d'information réguliers, paramètres changeant avec le temps, *etc*).

`/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.

Rappel de fichiers mentionnés dans ce cours :

— `/` : le répertoire racine du système de fichiers.
— `.` et respectivement `..` : le répertoire lui-même et respectivement son répertoire père dans l'arbre.
— `/home`, `/etc`, `/usr`, `/var`, `/dev` : voir la section 3.4.

1 Les processus

Un processus est un programme en train de tourner sur l'ordinateur considéré. Il est identifié par un numéro. Les numéros partent de 1 au moment où la machine démarre ; le processus de numéro 1 est généralement appelé `init`.

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. 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 manière surprenante de nombreux logiciels, non seulement n'offrent pas ces garanties, mais contiennent et exécutent de plus des fonctions non documentées. C'est particulièrement fréquent sur les smartphones par exemple, ou encore avec les moteurs de recherche 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
  ```
- 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
  ```
  $ commande1 | commande2
  ```

2 Le shell

Le 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. Lorsqu'un processus lance d'autre processus, on dit qu'ils sont les processus *fils* du processus lanceur. Ainsi, les processus qui tournent sur une machine possèdent eux aussi une structure arborescente, dont la racine est bien sûr `init` et dont les shells sont en général des nœuds ayant pour fils les commandes lancées par l'utilisateur.

> Au vu de la section précédente, vous aurez tous compris qu'il faut lire « qu'une ligne apparaisse sur son entrée standard » à la place de « que l'utilisateur tape une commande au clavier » dans la première phrase du paragraphe précédent. Le shell est un processus comme un autre.

Comme on l'a vu, lorsqu'un utilisateur se logue avec succès, c'est un shell qui est lancé pour l'accueillir sur la machine. Ce n'est pas le seul usage d'un shell, par exemple lorsque vous ouvrez une fenêtre « de terminal » (qui simule les vieux 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. 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. 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 variables dont les valeurs évitent d'aller chercher dans les fichiers de configuration les informations dont on se sert souvent. Par exemple :

- `USER` contient le nom de login du propriétaire du processus shell. Cela évite de parcourir `/etc/passwd` en cherchant la ligne qui correspond à l'identifiant (l'UID, cf. cours précédent) du propriétaire du processus. 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.
- `LANG` contient votre langue préférée sous une forme codifiée compréhensible par le système (souvent `FR` pour le français) et cette variable est exploitée pour formater la sortie de certaines commandes (par exemple `ls` suit ainsi l'ordre alphabétique du langage de l'utilisateur).
- `HOSTNAME` contient le nom de la machine sur laquelle tourne le processus.
- `DISPLAY` contient un identifiant de l'écran graphique qu'il faut utiliser pour afficher les résultats d'une commande *via* l'interface graphique (note : ceci est indépendant de la sortie standard du processus).
- `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` » le shell aurait lancé `/usr/games/openarena`, donc un répertoire différent. 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`.

Les variables mentionnées jusqu'ici sont dites « globales » c'est-à-dire que lorsque le shell lance une commande, il transmet la valeur de ces variables à ses processus fils. Si vous définissez vous même une variable (par exemple « `LIEU=antibes` »), elle est par défaut locale. Pour la transmettre aux processus fils, il faut la rendre globale avec la commande `export` (par exemple « `export LIEU` »). Par convention, les variables globales portent généralement des noms ne contenant que des majuscules.

Un shell ne se contente pas seulement de lancer des processus tapés par l'utilisateur sur une seule ligne, c'est aussi un langage de programmation impératif, qui possède donc à ce titre des primitives telles que `if`, `while`, `for`, `case`, la gestion des variables, les redirections des entrées et sorties des processus, *etc.* En revanche, les shells sont assez pauvres en termes de structures de données.

On peut donc non seulement taper des commandes assez sophistiquées, mais aussi écrire des programmes dans des fichiers et les faire interpréter par le shell comme s'il s'agissait de son entrée standard.

Un cas particulier important de tels fichiers sont les fichiers d'initialisation :

- Lorsqu'on lance un shell `bash`, avant de donner la main à l'utilisateur il exécute le fichier `$HOME/.bashrc` s'il existe. Ceci permet de personnaliser le shell. Par exemple on peut y placer la ligne « `alias ll='ls -l'` » et dès lors il suffira de taper `ll` au lieu de `ls -l` pour obtenir les informations « longues » sur les fichiers.
- Lorsque le shell est lancé directement par un login de l'utilisateur, il exécute d'abord (donc avant le `.bashrc`) le fichier `$HOME/.profile` s'il existe. Ceci permet typiquement d'initialiser de manière personnelles les variables globales. Par exemple étendre `$PATH` avec un répertoire personnel avec la ligne « `PATH=$PATH:$HOME/bin` » suivie de « `export PATH` ».

Pour terminer cette section, mentionnons que par convention le caractère Ctrl D en début de ligne indique la fin de l'entrée standard d'un processus. Ainsi pour sortir d'un shell, il suffit de taper ce caractère au lieu d'une nouvelle commande ; le shell comprend qu'il ne recevra pas d'autres commandes et il s'arrête donc. On obtient le même résultat avec la commande `exit`. Les shells « de login » (ceux lancés par un login de l'utilisateur) font exception pour éviter de se déloguer intempestivement sur une simple faute de frappe. Il faut utiliser la commande `logout` pour sortir de ces shells là, et donc se déloguer.

3 Les environnements graphiques

Lorsqu'on se connecte en étant physiquement présent au clavier d'un ordinateur, on bénéficie généralement d'une *interface graphique* à l'écran de cet ordinateur. La plupart du temps, l'interface par défaut pour le login des utilisateurs est alors elle-même graphique et, après un login en succès, l'utilisateur n'est pas face à un shell mais face à un *environnement graphique*. Un tel environnement est fondé sur la notion de « fenêtre » et la seule fenêtre qui soit toujours présente est le fond d'écran, appélée *root window*. Cette fenêtre de fond d'écran est conçue pour recevoir/contenir le cas échéant :

— des fenêtres « standard » ; à chaque fenêtre est attaché un processus qui en gère le contenu ; par exemple `firefox` lorsque vous lancez un navigateur web, ou `xterm` lorsque vous lancez une fenêtre de terminal. Ce dernier processus (`xterm`), outre la gestion graphique de la fenêtre, a pour fonction principale de lancer un shell. . .

— des « icônes » ; il s'agit simplement d'une fenêtre de petites taille à laquelle est associé un lien symbolique. Un sous-répertoire de votre homedir est le plus souvent dédié à cette paramétrisation et les fils de ce répertoire sont ceux qui donnent lieu à des icônes visibles à l'écran. Ce répertoire est souvent appelé « le bureau » (desktop).

— des « tableaux de bord » ou « barres d'outils » ou « panels » qui sont des fenêtres de grande longueur (souvent tout l'écran) et de faible largeur (généralement celle des icônes) qui contiennent elles-mêmes des icônes permettant de lancer d'autres fenêtres ou processus utiles.

— des menus, qui peuvent être lancés depuis un tableau de bord ou par un clic de souris dans la root window, ou encore par un clic dans certaines parties des autres fenêtres. Ils servent eux-mêmes à lancer de nouveaux processus et/ou nouvelles fenêtres.

De plus, le shell de login lance un processus « gestionnaire de fenêtres » (window manager). Le rôle de ce processus est, comme son nom l'indique, de placer, déplacer, iconifier ou changer de taille les fenêtres contenues dans la root window. La plupart du temps le gestionnaire de fenêtres permet d'effectuer ces actions à la souris en dessinant un *cadre* autour de chaque fenêtre (qu'on peut « attraper » avec la souris) ainsi qu'un bandeau au-dessus de la fenêtre pour inscrire le nom de la fenêtre et faciliter ces diverses opérations.

Tous ces éléments peuvent être paramétrés à la guise de l'utilisateur (couleurs, effets des boutons ou des touches, apparence des fenêtres et des icônes, processus préférés, nombre de tableaux de bord, menus et services offerts par les tableaux de bord, contenu des menus, *etc*.

4 Les processus « démons »

Le gestionnaire de fenêtres, l'affichage de la root window, la mise à disposition des panels et certaines autres fenêtres ont ceci de particulier qu'ils sont toujours présent et attendent qu'on les « réveille » et utilise (par exemple en passant/cliquant dessus avec la souris). Il s'agit donc de processus qui ont été lancés par le shell de login et qui surveillent certains paramètres (typiquement la position de la souris, l'heure qu'il est et la mise à jour de l'horloge du tableau de bord, ou autre) pour déclencher diverses actions ou autres processus. De tels processus qui passent leur temps à attendre en surveillant quelques paramètres sont appelés des processus « démons ».

De très nombreux démons tournent sur une machine. Parmi les plus courants, on peut citer ceux qui surveillent l'insertion d'un DVD, d'une clef USB, l'état du réseau, l'arrivée de mails, l'horloge du tableau de bord, *etc*. Certains sont lancés en votre nom (comme ceux qu'on a vus pour l'interface graphique), d'autres par `root` pour le système (remplissage des disques, gestion des imprimantes, du son, recherche de mises à jour, défenses anti-piratage « murs de feu », *etc*). Les processus démons lancés par `root`, dont le rôle est de faciliter la bonne marche du système complet, sont souvent appelés des « services ».

5 Le réseau

Lorsqu'un ordinateur doit envoyer des informations à un autre ordinateur *via* le réseau, il transmet les données par sa carte réseau et elles sont découpées en « paquets ».

— Un paquet commence par une « en-tête » qui contient entre autres un numéro qui identifie la machine qui expédie le paquet, un numéro qui identifie la machine destinataire du paquet, le type des données transmises, *etc*.

Ensuite le « corps » du paquet contient des données et il faut généralement un bon nombre de paquets pour transférer toutes les informations lors d'une communication entre ordinateurs.

— La carte graphique se charge de transformer chaque paquet en modulations d'intensité sur le cable réseau ou

sur la wifi. Dans tous les cas, *le paquet est transmis sans distinction à toutes les machines du réseau.* Toutes les machines lisent l'en-tête du paquet et si elles se reconnaissent en le numéro du destinataire alors le *processus démon* qui surveille la carte réseau va prendre le paquet en considération.
N'oubliez jamais néanmoins qu'une machine gérée de manière malveillante a toujours la possibilité de lire tous les paquets du (sous-)réseau auquel elle est connectée afin de récupérer les données qui transitent, informations confidentielles, mots de passe, *etc.*

Du fait que chaque paquet est transmis par une carte réseau à la totalité des machines connectées au réseau, il y a souvent des *collisions* : plusieurs machines qui émettent en même temps produisent sur le réseau un signal illisible. En cas de collision, les cartes réseau concernées ré-émettent leur paquet après un délai aléatoire. Il est donc matériellement impossible d'avoir un grand nombre de machines connectées à un même réseau. Pour cette raison, le réseau internet mondial est découpé en sous-réseaux (e.g. « `.fr` », « `.com` », « `.net` », « `.uk` », ...), ces sous-réseaux sont eux-mêmes découpés en sous-réseaux (e.g. « `unice.fr` », « `free.fr` », « `cnrs.fr` », « `genopole.fr` », ...), qui peuvent à leur tour être découpés (« `www.unice.fr` », « `bibliotheque.unice.fr` », « `ipmc.unice.fr` », « `i3s.unice.fr` », ...), et l'on peut continuer à découper autant que l'on souhaite. On obtient ainsi une arborescence de réseaux.

Chaque sous-réseau possède une *passerelle* : c'est un ordinateur qui porte 2 cartes réseau, l'une connectée au sous-réseau (par exemple `sophia.bibliotheque.unice.fr`) et l'autre connectée au réseau père (dans notre exemple, `bibliotheque.unice.fr`). La passerelle surveille tous les paquets qui passent dans son sous-réseau et lorsque l'adresse de la machine destinataire n'appartient pas au sous-réseau, le paquet est recopié sur la carte du réseau père. Inversement, la passerelle surveille tous les paquets du réseau père et si la machine destinaraire appartient à son sous-réseau alors elle le recopie sur la carte du réseau local.

Il y a également des annuaires sur chaque sous-réseau (appelés DNS pour domain name server) qui assurent la traduction entre le nom « humain » d'un sous-réseau ou d'une machine (comme *bibliotheque.unice.fr* par exemple) et son numéro « IP » utilisé dans les en-têtes de paquets (par exemple `85.118.46.110`).

Lorsqu'une machine démarre (« boote »), elle commence par envoyer des broadcasts pour connaître le numéro de la passerelle, du DNS, et de divers autres serveurs utiles. Ces serveurs répondent à ces demandes en fournissant leur caractéristiques. C'est comme cela que la machine « sait » dans quel sous-réseau elle se trouve.

Cours n°3

1 Les expressions régulières sous Unix

Pour le shell, concernant les noms de fichiers, « `*` » remplace n'importe quelle suite de caractères dans les noms de fichiers (ou dans les patterns des `case` comme on le verra plus loin). De même « `?` » remplace n'importe quel caractère.

Les expressions régulières « standard » ne suivent pas ces règles simples du shell. Les commandes les plus communes qui utilisent des expressions régulières standard sont par exemple `ed`, `sed`, `expr`, *etc.*

Pour une définition complète des expressions régulières sous Unix, faire `man ed`. Nous donnons ici seulement les constructions les plus souvent utiles.

Pour ces expressions régulières :
- Un point remplace n'importe quelle lettre.
  Par exemple le pattern « `t.t.` » filtre aussi bien `toto` que `tutu` ou `toti`, mais aussi `twtr`.
- Pour se limiter à certaines lettres, il faut les énumérer entre crochets.
  Par exemple « `t[aiou]t[aiou]` » filtre les 16 possibilités de `tata` à `tutu` en passant par `toti`.
- Avec un tiret, on peut énumérer une séquence de lettres entre crochets.
  Par exemple « [A-Z] » filtre toute lettre majuscule, ou encore « `[A-Z0-9]` » filtre tout caractère qui est une majuscule ou un chiffre.
  Si l'on souhaite un véritable tiret dans l'énumération de caractères, il faut commencer par lui (comme dans « `[-xy]` »).
- On peut également indiquer les caractères qu'on ne veut pas. Dans ce cas on utilise un accent circonflexe pour indiquer une négation.
  Par exemple « `[^0-9]` » signifie « tout caractère qui n'est pas un chiffre ».
- Pour les suites de caractères, une étoile indique une répétition d'un pattern un nombre quelconque de fois (même 0 fois).
  Par exemple « `Gr*oum*f` » filtre `Groumf` ou encore `Grrrroummmf`, mais aussi `Gouf` ou `Grouf`. Si l'on veut au moins un `r` et un `m`, on écrit « `Grr*oumm*f` ».
- Le caractère backslash est un caractère d'échappement. Ainsi « `.*\.jpg` » filtre `toto.jpg` mais pas `totoxjpg`. Si l'on veut un vrai backslash, il faut le doubler (parfois tripler ou quadrupler car backslash est aussi un caractère d'échappement du shell).
- Il y a une exception. Pour beaucoup de commandes de substitution, la forme spéciale « `debut\(milieu\)fin` » permet de ne retenir que le milieu.
  Par exemple le pattern « `toto(.*).jpg` », lorsqu'il est appliqué à `toto123.jpg` retourne la chaîne de caractères `123`.

La commande `expr` sert à calculer toutes sortes de choses. Faire `man expr`. Parmi ses possibilités, il y a la gestion de pattern matching ; dans ce cas, on l'utilise sous la forme :

`expr "`*chaine*`" ':' '`*pattern*`'`

Par exemple la commande « `expr "toto123.jpg" ':' 'toto\([0-9]*\)\.jpg'` » donne la chaîne `123`.

Si le pattern n'est pas reconnu, `expr` retourne une chaîne vide et un code de retour 1 au lieu de 0.

2 Usage de `find`

`find` est une commande très puissante pour appliquer un traitement quelconque dans toute une sous-arborescence du système de fichiers.

Mon premier conseil est bien sûr de faire « `man find` ». Ces explications préliminaires peuvent cependant vous faciliter la lecture du man.

L'usage le plus fréquent de find est de ce type :

`find` ***adresse critères... commandes...***

où :

adresse est simplement l'adresse de la racine de la sous-arborescence à traiter.

critères est une suite de conditions à remplir pour faire l'objet d'un traitement. Par exemple :

- « `-name '*.txt'` » ne retiendra que les adresses dont le nom a pour suffixe `.txt` ; remarquer que `'*.txt'` est entre quotes pour éviter que le « `*` » ne soit interprété par le shell car on veut que ce soit le critère `-name` de `find` qui interprète l'étoile à chaque niveau de l'arborescence, et non pas le shell (qui d'ailleurs ne l'interpréterait que dans le répertoire courant depuis lequel on lance la commande).
- « `-newer toto/truc` » ne retiendra que les fichiers plus récents (en date de dernière modification) que le fichier `toto/truc`.
- « `-mtime +300` » ne retiendra que les fichiers dont la date de dernière modification (ou de création) remonte à plus de 300 jours. Par convention, « `-mtime -300` » ne retiendra que les fichiers modifiés depuis moins de 300 jours et « `-mtime 300` » ceux qui ont été modifiés (ou créés) il y a exactement 300 jours.
- « `-atime +150` » ne retiendra que les fichiers dont la date de dernière utilisation (i.e. de dernière lecture) remonte à plus de 150 jours. Les conventions « `-150` » et « 150 » fonctionnent de même.
- « `-type d` » ne retiendra que les fichiers qui sont des répertoires (autres types bien utiles : `f` pour les « vrais » fichiers et `l` pour les liens symboliques).
- « `-size +2048` » ne retiendra que les fichiers de taille supérieure à 2048 caractères. Les conventions `-2048` pour une taille inférieure à 2048, ou `2048` pour une taille exactement de 2048 caractères fonctionnent de même.
- Si on met plusieurs critères les uns à la suite des autres, c'est par défaut la conjonction (le *et*) des critères.

commandes sont les commandes qui sont exécutées sur les adresses retenues par les critères précédents. Par exemple :

- « `-print` » se contente d'écrire sur la sortie standard les adresses retenues. C'est la commande par défaut si jamais on ne donne aucune commande dans les arguments du `find`.
- « `-exec basename '{}' ';'` » écrira successivement seulement le `basename` de chaque adresse retenue.
- Plus généralement « `-exec` » est très puissant car on peut lui donner n'importe quelle commande à la place de `basename`, même un shell script écrit par ailleurs, et même avec des arguments. Par convention « '{}' » est successivement remplacé par chacune des adresses retenues et on exécute donc autant de fois la commande que le nombre d'adresses retenues par les critères. Enfin, il faut toujours terminer par un point-virgule, que l'on quote lui aussi pour qu'il ne soit pas interprété par le shell.
- « `-ok rm -f '{}' ';'` » proposera de supprimer chacun des fichiers retenus par les critères du `find` ; pour chaque fichier retenu, si l'utilisateur confirme alors la commande est effectuée, sinon elle ne l'est pas. « `-ok` » marche donc comme « `-exec` » sauf que l'utilisateur confirme ou non pour chacune des adresses retenues.

1 Créer un shell script

Un shell comme `bash` est un langage interprété, on écrit donc des « shell scripts » qui sont simplement des fichiers de texte pur, rendus exécutables (mode au moins 5 avec `chmod`) et avec la ligne d'en-tête « `#! /bin/bash` » qui indique que c'est `bash` qui doit interpréter le texte qui suit. On peut ensuite programmer exactement comme on taperait dans un terminal chaque ligne pour faire la commande en mode interactif.

Une fois le fichier rempli et rendu exécutable, il suffit de placer dans un répertoire qu'on a fait reconnaître par la variable `$PATH` (typiquement `$HOME/bin`).
Nota : si le fichier n'est pas dans un répertoire de `$PATH`, on peut toutefois s'en servir en donnant son adresse complète au lieu de donner seulement son nom.

La plupart du temps, on utilise plus souvent les variables et leur manipulation que lorsqu'on tape au clavier :

```
toto="Hello bonjour"
```

affecte pour valeur la chaîne de caractères `"Hello bonjour"` à la variable `toto`. Par la suite, `$toto` fournira cette valeur :

```
echo $toto
```

écrit à l'écran :

```
Hello bonjour
```

Enfin, on peut affecter pour valeur d'une variable le résultat d'une ligne de commande shell. Par exemple :

```
tutu=`echo $toto | sed -e 's/ //'`
```

donne à `tutu` la valeur `"Hellobonjour"` sans espace, puisque le `sed`, qui a pris la valeur de `$toto` en entrée standard, l'a recopié sur sa sortie standard en supprimant les espaces, et ces processus ont été mis entre accents graves, ce qui transforme leur sortie standard en une chaîne de caractère, que l'on a à son tour affectée à `tutu`...

2 La programmation en shell

Outre le fait que l'on peut inclure dans un shell script toute ligne que l'on taperait dans un terminal avec shell, on peut aussi bénéficier des primitives de contrôle de tout langage de programmation impérative :

Les expressions conditionnelles :

```
if [ ici_une_condition ]
  then
    ici_des_commandes
elif [ ici_une_autre_condition ]
  then
    ici_d_autres_commandes
  else
    encore_d_autres_commandes
fi
```

la ligne « `fi` » indique la fin de l'expression conditionnelle « `if` » ; « `elif` » est une contraction de « else if » ; il peut y avoir autant de `elif` que l'on veut ; les `elif` et le `else` ne sont pas obligatoires ; en revanche la partie `then` est obligatoire.

Les boucles while :

```
while [ ici_une_condition ]
do
  ici_des_commandes
done
```

Les commandes sont répétées jusqu'à ce que la condition devienne fausse.

Les boucles for :

```
for nomDeVariable in une_liste_de_noms_séparés_par_des_espaces
do
  des_commandes
  qui_peuvent_utiliser_$nomDeVariable
done
```

Il y a autant de tours de boucles que de noms dans la liste. Par exemple

```
for fichier in *
do
  if [ -d "$fichier" ]
    then
      echo "$fichier est un sous répertoire."
  fi
done
```

donne la liste des sous-répertoires du répertoire courant.

La programmation par cas :

```
case "$uneAdresseDeFichier" in
  /*) echo "c'est une adresse absolue."
    ;;
  */*) echo "c'est une adresse relative dans l'arborescence."
    ;;
  *) echo "c'est une adresse fille du répertoire courant."
esac
```

Comme on le voit, le `case` est effectué en filtrant selon les expressions régulières au sens du shell (cf. cours précédent).

Pour plus de précisions, faire « `man bash` » et pour les conditions utiles pour les `if` et les `while` « `man test` » est parfois plus simple à lire en diagonale que la partie décrivant les conditionnelles du manuel de `bash`.

Dans un shell script, on peut aussi créer des fonctions. Les fonctions s'écrivent sous la forme :

```
nomfonction () {
  ...
  corps de la fonction
  ...
}
```

et les variables `$1`, `$2`, ... deviennent les *arguments de la fonction*.

Par exemple, si « `prevision` » est le shell-script suivant :

```
#! /bin/bash
prefixer () {
  ## ici $1 et $2 sont les arguments d'appel de prefixer, pas de prevision :
  if [ "$1" = "$2" ]
    then echo "$1"
    else echo "$2$1"
  fi
}
## ici par contre le "$#" est le nbre d'arguments de prevision :
case "$#" in
  0|1) echo "Donner plusieurs arguments !!"
       exit 1 ;;
  *) echo "Le prefixe sera \"$1\""
```

```
    prefixe="$1"
    shift
esac
## Par definition : $*="$1 $2 $3 $4 ..."
for suffixe in $*
do
  nom=`prefixer $suffixe $prefixe`
  if [ -f $nom ]
    then echo $nom
  fi
done
```

alors la commande « `prevision old- truc machin chouette` » imprimera à l'écran les noms de fichiers qui existent parmi `old-truc`, `old-machin` et `old-chouette`.

A ce propos, on constate également que les arguments de la commande globale portent les noms $1, $2, *etc.* Ceci permet de passer des informations au shell script lorsqu'on y fait appel.

Pour accéder à l'intérieur d'une fonction aux arguments du shell-script, créer une variable intermédiaire avant d'appeler la fonction. C'est ce que fait la ligne « `prefixe="$1"` » dans l'exemple précédent.

La syntaxe de définition de fonctions est trompeuse : dans l'exemple, bien qu'on écrive « `prefixer ()` », la fonction `prefixer` accepte (et requiert) deux arguments, mais c'est complètement implicite.

Enfin, notons que la fonction est locale à la commande (elle n'est pas « exportée » vers les processus fils) et ne peut donc être utilisée que dans le shell script où elle a été déclarée.
En particulier, l'option `-exec` de `find` ne peut donc pas faire appel à une fonction définie dans le shell script...

3 Exemple

Voici un exemple de shell-script ; il recherche tous les « vrais » fichiers de taille supérieure à un kilo-octet et possédant un suffixe de trois caractères. Ces fichiers sont alors compressés, sauf si ce sont des images.

```
#! /bin/bash
#
## le find (cf plus bas) fera un appel de cette commande sous la
## forme "$0 -auto-appel-par-find fichier" pour chaque fichier retenu :
if [ "$1" = "-auto-appel-par-find" ]
  then
    case "$2" in
      *.jpg|*.png|*.bmp) ## suffixes d'images, ne rien faire
        ;;
      *) ## sinon on compresse le fichier
        bzip2 "$2"
        ;;
    esac
## appel par utilisateur, on veut un unique répertoire en argument :
elif [ "$#" -neq 1 -o ! -d "$1" ]
  then echo "Arguments invalides. Abandon."
    exit 1
fi
## Bon, tout roule, on lance le find :
find "$1" -size +1024 -type f -exec $0 -auto-appel-par-find '{}' ';'
## nota: cette commande est donc appelée n+1 fois, c.a.d. une fois par
## l'utilisateur et n fois par le find (n = nombre de fichiers retenus).
```

Astuce : pour tester ce genre de commande sans risquer d'endommager l'arborescence, on peut ajouter des « `echo` » en divers endroits. En premier lieu :

```
find "$1" -size +1024 -type f -exec echo $0 -auto-appel-par-find '{}' ';'
```

Puis si c'est bon :

```
case "$2" in
  *.jpg|*.png|*.bmp) ## suffixes d'images, ne rien faire
    echo RIEN POUR "$2"
    ;;
  *) ## sinon on compresse le fichier
    echo bzip2 "$2"
    ;;
esac
```

Quelques commandes utiles mentionnées dans ce cours

Rappel : « `man` *commande* » fournit une notice d'utilisation de la *commande*, à commencer par « `man man` »...

- `echo` : écrit les chaînes de caractères qu'on lui donne en argument (utile pour connaître la valeur d'une variable par exemple).
- `bash` : le shell le plus courant, souvent appelé simplement `sh`.
- `xterm` ou `kterm` ou autre « fenêtre de terminal » : fait apparaître dans une fenêtre graphique un espace textuel qui permet d'utiliser le shell.
- `mv` : « move » un fichier (mais en fait, se contente de le renommer).
- `pwd` : print working directory
- `cd` : change directory (change le répertoire de travail courant).
- `ls` : liste le contenu d'un répertoire ; cette commande admet de nombreuses options, dont `-a` (all) pour afficher aussi les « fichiers cachés » c'est-à-dire ceux qui commencent par un point, et aussi `-l` (format long) pour afficher les informations principales à propos des fichiers (droits, propriétaire, taille, *etc*).
- `touch` : positionne les dates de dernière modification et de dernière utilisation d'un fichier à l'instant présent.
- `chmod` : pour attribuer les droits d'accès à un fichier dont on est propriétaire (change mode).
- `file` : détermine le type d'un fichier en explorant son contenu réel.
- `mkdir` : crée un répertoire
- `rm` : supprime (définitivement, il n'y a pas de « corbeille ») un fichier plat ou un lien symbolique. Avec l'option `-r` (`r` comme récursif), « `rm -r` *répertoire* » supprime le *répertoire* et tout son contenu.
- `less` : montre un fichier texte page par page dans le terminal, la touche *espace* permettant de passer à la page suivante.
- `cat` est plus frustre que `less` : il recopie sans mise en page, sur sa sortie standard, le contenu du fichier qu'on lui donne en argument.
- `alias` : permet de donner des diminutifs à des commandes souvent utilisées. `alias` *nom*`="`*vraie commande*`"`
- `ps` : fournit la liste des processus lancés à partir du terminal par l'utilisateur. Les options « `-x` » et « `-aux` » montrent plus de processus (faire `man ps`).
- `top` est plus sophistiqué que `ps` : il fournit en temps réel la liste des processus les plus gourmands en puissance de calcul (ordre décroissant) de l'ordinateur. On en sort avec la touche « `q` ».
- `which` : prend en argument un nom de commande et explore la variable `$PATH` pour fournir l'adresse de son fichier exécutable.
- `grep` : affiche à l'écran toutes les lignes de son entrée standard ou d'un fichier contenant à un motif (voir section du cours à ce sujet) donné en argument.
- `gedit` : éditeur de fichiers textes d'utilisation assez simple, bien adapté pour écrire des shell-scripts.
- `sed` : stream editor, prend en argument des commandes de remplacement de texte (faire `man sed` en portant surtout attention à l'option `-e` et à la commande de remplacement de la forme « `s/`*vieux*`/`*nouveau*`/g` ».
- `find` : voir la section qui lui est dédiée dans ce cours.
- `evince`, `xpdf`, `epdfview`... : outils de visualisation en impression de fichier PDF (versions open-source de « readers » bien connus qui présentent souvent des trous de sécurité tant on ne sait pas ce qu'ils font)
- openOffice ou freeOffice : versions open-source des suites bureautiques du commerce.
- `gimp` : programme très puissant de manipulation d'images.
- `thunderbird` : browser de mails open-source
- `firefox` : browser internet open-source
- `kompozer` : éditeur convivial de pages HTML
- `lpr` : imprime le fichier qu'on lui donne en argument sur l'imprimante désignée par la variable `$PRINTER`.
- `lpq` : montre la liste d'attente de l'imprimante et/ou indique les problèmes d'impression
- `cancel` ou selon les systèmes `lprm` : supprime une demande d'impression de la file d'attente.
- `tar` : permet de créer un fichier d'archive de toute une arborescence de fichiers. Utile pour les sauvegardes ou pour les échanges par mail de données structurées.
- `k3b` ou autres graveurs de CDROM, DVD, BluRay : utiles aussi pour sauvegarder les fichiers `tar` de la commande précédente...
- `chown` : change owner, permet à `root` de changer le propriétaire d'un fichier.
- `chgrp` : change group, change de même le groupe d'un fichier.