Cours ASR **année 2014-2015** GB4, $2^{nd}$ semestre 2013-2014

Polytech'Nice Notes de cours Gilles Bernot

# *Introduction à l'Administration Systèmes et Réseaux*

**Avertissement au lecteur :**

Ce polycopié n'est pas un document scolaire de référence sur le cours, c'est seulement l'ensemble de mes propres notes de cours. Il ne contient donc pas les explications détaillées qui ont été données en cours.

G. Bernot

*Ce plan n'est que prévisionnel et est soumis à changements*

**COURS 1**

1. Les objectifs du cours
2. Les 3 concepts fondamentaux d'un système
3. Les utilisateurs
4. Les fichiers
5. Les processus
6. Le shell
7. Les environnements graphiques
8. Les processus démons

**COURS 2**

1. Le réseau
2. Configuration d'un ordinateur ou d'un serveur

**COURS 3**

1. Le montage d'un ordinateur, choix des composants
2. Installation d'une distribution de la famille des Unix
3. Quelques packages utiles
4. Protections et mots de passe
5. Mises à jour
6. Annexe : mémo d'installation Linux

**COURS 4**

1. Quelques commandes indispensables
2. Les sauvegardes
3. L'administration du réseau et la sécurité du système
4. Les démons sshd, ftpd et leurs processus clients

**COURS 5**

1. La programmation en shell
2. Syntaxe des shells de la famille sh
3. Exemples
4. Projets

# 1 Les objectifs du cours

Ce cours se veut très pragmatique : il s'agit *en pratique* d'être capable de gérer un petit système informatique en préservant ses utilisateurs des problèmes courants. À l'issue du cours, on saura :

- choisir son matériel en fonction des fonctionnalités attendues du système informatique,
- installer un système *ex nihilo* avec un bon niveau de sécurité (système de type Unix), qui peut éventuellement être un serveur sur le réseau,
- le maintenir (configuration personnalisée, réalisation d'utilitaires dédiés à des tâches routinières, sauvegardes, conseils aux utilisateurs, *etc*)
- et on saura aussi (et surtout) comment chercher et trouver les informations utiles pour réussir une réalisation nouvelle, par conséquent augmenter progressivement ses compétences tout en gérant un système sans mettre en péril la sécurité des données et le confort des utilisateurs.

**Nota :** le poly contient aussi des annexes qui récapitulent les commandes et les fichiers importants vus durant le cours.

# 2 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 (et on peut dire en raccourci que ce cours vous apprend à être un `root` compétent). 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 `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.

# 3 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 avec la commande « `echo $UID` ».
> L'UID de `root` est 0.

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) et s'il est autorisé à rejoindre d'autres groupes, il peut le faire en utilisant la

commande `newgrp` (à condition de connaître le mot de passe du groupe ou d'y être admis dans le fichier `/etc/group`).

L'utilisateur `root` appartient au groupe `root` et certains autres utilisateurs virtuels peuvent appartenir au groupe `root` (ils servent à gérer le système sans prendre le risque de lancer des processus ayant tous les droits de `root`). Il y a également un groupe `wheel` qui joue un rôle similaire.

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 le plus souvent 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 simplement que le shell lance immédiatement les processus de l'environnement graphique au lieu d'attendre que l'utilisateur tape une commande.

> Certains utilisateurs virtuels ont un processus d'accueil plus restreint que `bash`, par exemple l'utilisateur `shutdown`, qui appartient au groupe `root`, a pour « login shell » un processus qui se contente d'éteindre l'ordinateur.

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.

**Les utilisateurs virtuels :** on constate que `/etc/passwd` définit aussi des utilisateurs qui ne sont pas attachés des personnes. C'est le cas de `root` bien sûr mais aussi par exemple `ntp` (net time protocol, pour mettre à l'heure l'ordinateur), `smtp` (send mail transfer protocol, pour distribuer le courrier électronique), `lp` ou `cups` (line printer ou common unix printing system, pour gérer les files d'attende des l'imprimantes), `nfs` (network file system, pour mettre à disposition *via* l'intranet une partition d'un serveur), *etc.*

# 4 Les fichiers

Les fichiers sont les lieux de mémorisation durable des données et des programmes 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 (mémoire flash, clef USB, CDROM, ...).

## 4.1 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 répertoires 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 également 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
```

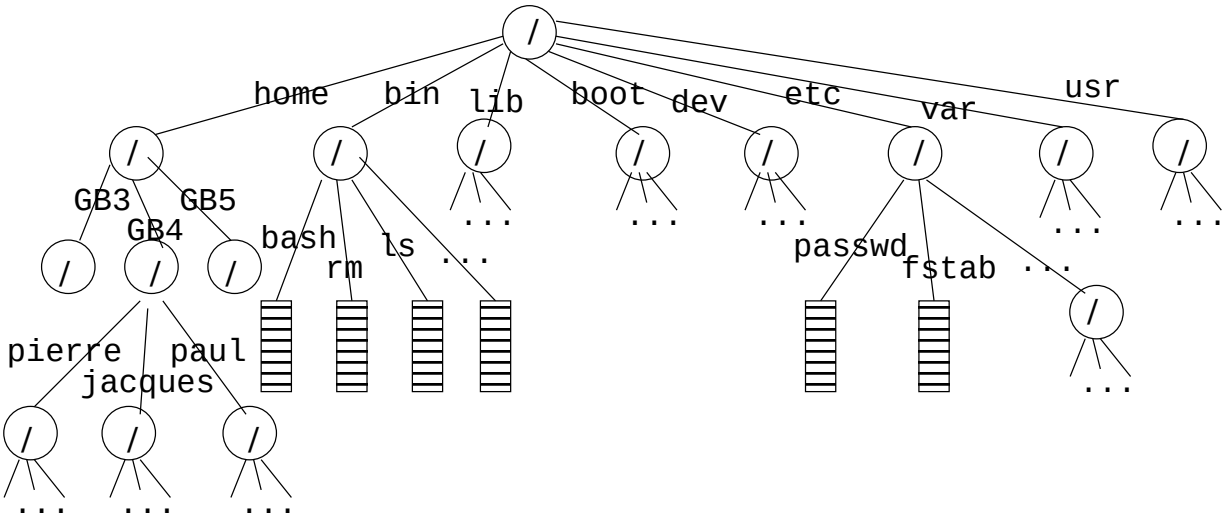
Plusieurs informations sont attachées aux fichiers. On en verra plusieurs dans cette section (propriétaire du fichier, droits d'accès, *etc*) et l'on peut mentionner également deux informations souvent utiles :

- 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.

## 4.2 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 inode. Il peut être, comme on le voit, 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/GB4/paul`). Par conséquent, un nom 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 inode ; ce qui change, c'est l'arborescence, donc seulement les contenus des fichiers de type répertoire qui la constituent.

En particulier, la commande `mv` prend un temps très court quelle que soit la taille du fichier.

Il est possible (mais rare en pratique), pour les fichiers qui ne sont pas des répertoires, qu'un même numéro de inode apparaisse plusieurs fois dans l'arbre. Le fichier correspondant possède alors deux adresses différentes, et donc possiblement deux noms différents.

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.

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.

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 « fils » d'un répertoire (par défaut son répertoire de travail).

```
$ pwd
/home/bioinfo/bernot

$ cd bin

$ pwd
/home/bioinfo/bernot/bin

$ ls
RNAplot  hsim  wi  wx  wxd

$ ls -a
. .. RNAplot  hsim  wi  wx  wxd

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

$ ls /etc/X11/ /etc/rc.d
/etc/X11/:
app-defaults  proxymngr    xinit.d      Xmodmap     prefdm
fontpath.d    wmsession.d  xorg.conf.d  Xresources  xorg.conf
gdm           xdm          xsetup.d     Xsession
lbxproxy      xinit        X            lookupdm

/etc/rc.d:
init.d  rc1.d  rc3.d  rc5.d  rc7.d            rc.local
rc0.d   rc2.d  rc4.d  rc6.d  rc.alsa_default  rcS.d
```

La plupart des commandes Unix admettent des *options* qui modifient leur comportement par défaut. Par exemple, le comportement par défaut de `ls` est de donner la liste des fils du répertoire courant en ignorant tous ceux dont le nom commence par un « . » mais l'option « -a » de `ls` modifie ce comportement en montrant tous les fils du répertoire. Il y a également des arguments « standard » qui ne commencent pas par un « - » ; par exemple si l'on indique un ou plusieurs noms de fichiers à `ls`, il fait son travail sur ces fichiers au lieu du répertoire courant.

> *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].

## 4.3 Les droits d'accès

Outre son numéro de inode qui le caractérise, tout fichier quel que soit son type a :

- un *propriétaire*, qui est un utilisateur reconnu du système,

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

— 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` bien sûr, puisqu'il a tous les droits) peuvent effectuer un `chmod` sur un fichier.

L'option « `-l` » (=format long) de `ls` permet de connaître les droit d'accès d'un fichier :

```
$ chmod 640 toto
$ ls -l toto
-rw-r----- 1 bernot bioinfo 5 avril 25 08:13 toto
$ chmod 755 toto
$ ls -l toto
-rwxr-xr-x 1 bernot bioinfo 5 avril 25 08:13 toto
```

Le résultat de `ls -l` fournit les informations suivantes :

— Le premier caractère indique le type du fichier : un tiret pour un fichier plat (comme dans l'exemple), un `d` pour un répertoire [2] (`d`irectory), un `l` pour un lien symbolique, un `c` pour un device (`c`aractère spécial), *etc.*
— Les 3 caractères suivants indiquent les droit du propriétaire du fichier dans l'ordre `r`ead, `w`rite, e`x`ecute : un tiret indique que le droit n'est pas donné, sinon la lettre correspondante apparaît (voir les deux exemples).
— Les 3 caractères suivants indiquent les droit des membres du groupe du fichier selon le même principe.
— Les 3 caractères suivants indiquent les droit des autres utilisateurs, toujours selon le même principe.
— Ensuite vient le nombre de liens qui pointent sur le i-node du fichier. La plupart du temps c'est 1 pour les fichiers qui ne sont pas des répertoires, et de 1 + le nombre de répertoires fils pour un répertoire (il s'agit du lien « standard » sur le répertoire, du lien `.` et du lien `..` de chaque répertoire fils).
— Suivent le nom du propriétaire et le nom du groupe du fichier.
— Puis la taille du fichier (nombre d'octets, 5 dans l'exemple).
— Puis la date et l'heure de dernière modification du fichier (ou la date et l'année si plus d'un an).
— Et enfin le nom du fichier.

Seul `root` peut changer le propriétaire d'un fichier, en utilisant la commande `chown` (change owner) :

```
# whoami
root
# chown untel ceFichier
```

2. « `ls -l nomRepertoire` » fournit la liste des fils du répertoire. Pour avoir les information du répertoire lui-même au lieu de ses fils, on doit ajouter l'option `-d`

```
# chown untel.ungroupe cetAutreFichier
```

(Sous la seconde forme, `chown` permet de modifier aussi le groupe du 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...***

> En fait, le mode d'un fichier contient un quatrième entier rarement utilisé. Il permet au propriétaire du fichier de basculer à vrai un « sticky bit » et dès lors tout utilisateur qui exécute ce fichier lance un processus avec les droit du propriétaire du fichier au lieu de ses propres droits. *C'est extrêmement dangeureux pour le propriétaire du fichier, qui doit dans ce cas être absolument certain que le programme ne peut pas être détourné de ses buts d'origine !*. La commande `su`, qui permet de lancer un shell avec les droits de `root` est naturellement dans ce cas :
> ```
> $ ls -l /bin/su
> -rwsr-xr-x 1 root root 28872 déc.  28 16:51 /bin/su
> ```
> La commande `ls -l` le montre en mettant un `s` à la place du `x` en face des droits du propriétaire.

## 4.4 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 la quasi totalité des 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, 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 spécial : 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.

D'un point de vue technique, les disques qui contiennent les fichiers peuvent être découpés en morceaux appelés des *partitions* afin de faciliter leur sauvegarde et de limiter les dégats en cas de défaillance matérielle. La structure interne d'une partition repose sur une spécification technique qui est généralement appelée `ext4` et il est par ailleurs possible « d'importer » des partitions non Unix, comme FAT, NTFS ou autre.
Une de ces partitions contient le répertoire `/` et les autres sont alors des sous-arbres de l'arborescence globale. Le fichier `/etc/fstab` définit cette arborescence « à gros grains » entre partitions.

# 5 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 `init` est donc identifié par le numéro 1.

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 ! surtout si l'on est `root`...

> 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 son 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 restent, au bilan, lues par de nombreuses personnes dans le monde 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 et utilisez toujours des versions open source lorsqu'elles existent.

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
  `$ 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`

La commande `ps` permet de lister les processus qu'on a lancés depuis le shell et qui tournent au moment du lancement de la commande. Cette commande admet plusieurs options utiles, dont « `-x` » qui permet de voir *tous* les processus qui tournent en votre nom, et « `-aux` » qui liste tous les processus qui tournent sur la machine, quels qu'en soient les propriétaires (il y en a beaucoup, donc c'est une bonne idée de « piper » la sortie standard de `ps` dans l'entrée standard de `less` afin de paginer le résultat).

La commande `top` est similaire à « `ps -aux` » sauf qu'elle ne montre sur une page que les processus les plus gourmands en puissance de calcul. La page est mise à jour en temps réel et on sort de la commande en tapant `q` dans son entrée standard. Très utile lorsqu'on se demande pourquoi la machine a un trou de performance...

Enfin la commande `kill` tente de tuer les processus dont on lui donne les numéros en arguments. Elle « tente » en ce sens qu'elle fait passer un message à ces processus qui leur demande de terminer. Un processus parti dans un bug quelconque peut très bien ignorer cette demande, de même qu'un processus qui s'estime dans une « section critique » et refuse de sortir afin de préserver par exemple la cohérence de ses données. L'option `-9` est plus violente : `kill -9` envoit un signal (très) impératif au processus et l'arrête net.

# 6 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 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 les nœuds ayant le plus de fils différents au cours du temps.

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 généralement un shell qui est lancé pour l'accueillir sur la machine (selon ce qui est défini dans `/etc/passwd`). 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 inaccessibles *via* les menus d'une interface graphique. Pour gérer un système informatique, on utilise donc intensivement les fenêtres de terminal. Lorsque l'on se logue avec un interface graphique, c'est généralement 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 `bash`[3].

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 la commande avait été « `su` », le shell aurait dû trouver le fichier à l'adresse `/bin/su`, donc dans un répertoire différent. Pour la commande « `openarena` » le shell aurait lancé `/usr/games/openarena`, donc un troisième répertoire... 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`.

> Au passage, si vous administrez un système et qu'un utilisateur vous demande une petite réparation qui nécessite de passer `root`, tapez « `/bin/su` » car si vous tapez seulement « `su` » l'utilisateur peut très bien avoir un `$PATH` qui vous fait exécuter une commande `su` qui n'est pas celle que vous croyez, et il peut ainsi récupérer le mot de passe de `root`... En fait, par principe, ne passez `root` qu'à partir de votre propre compte utilisateur.

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=polytech` »), 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 `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 les « *alias* » : lorsqu'on exécute la ligne « `alias ll='ls -l'` » alors il suffira ensuite de taper `ll` au lieu de `ls -l` pour obtenir les informations « longues » sur les fichiers.

3. pour *Bourne-again shell*, jeu de mot avec *born again* parce que `bash` est une extension du shell « historique » de linux (`sh`) programmé par Stephen Bourne

— 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=$HOME/bin:$PATH` » 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.

# 1 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, outre la gestion graphique de la fenêtre, ayant 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 (répertoire souvent appelé « le bureau ») ;
- 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 ou moins) 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 ». 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.*

# 2 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` (ou par des utilisateurs virtuels annexes) et dont le rôle est de faciliter la bonne marche du système complet sont souvent appelés des « services ».

L'un des rôles de `root` est de choisir un bon compromis entre, d'une part, les facilités d'utilisation apportées par ces services, et d'autre part, les performances du système (car une multitude de démons peut finir par occuper le processeur de façon non négligeable) et la sécurité du système : certains services « ouvrent » des accès au monde extérieur (comme ssh, Apache ou Skype) qui sont autant de voies d'attaques pour les pirates, d'autres services renforcent la sécurité en surveillant ces mêmes voies. Il faut toujours éviter d'ouvrir des services qui ne sont pas indispensables aux utilisateurs, et si un service ouvre une voie extérieure, il faut (1) que le firewall (« murs de feu ») la surveille et (2) jeter régulièrement un oeil aux journaux générés par le firewall pour répérer les tentatives de piratages trop fréquentes.

# 3 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 [4], 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 réseau (souvent intégrée à la « carte mère » de l'ordinateur) 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 (ou s'il s'agit d'un broadcast) 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.

# 4 L'administration du réseau et la sécurité du système

Si certaines adresses sont utilisées très souvent, `root` a intérêt à les mémoriser localement sur la machine, dans le fichier `/etc/hosts`.

Il y a en fait plusieurs processus démons (plusieurs *services*) qui surveillent la carte réseau, selon le type de données que les paquets transportent. Cela permet de mettre en œuvre divers *protocoles de communication.* On peut citer : `ftp` (file transfert protocol, pour l'échange de fichiers), `smtp` (send mail transfert protocol, pour l'échange de courriers électroniques), `http` (hypertext transfert protocol, pour l'échange de pages web), `https` (un `http` sécurisé en cryptant les données, de sorte qu'un pirate a du mal à décrypter les informations), `cups` (common unix printing system, communication avec les imprimantes), `ntp` (net time protocol, pour mettre une machine à l'heure), *etc.*)

Le démon `sshd`, s'il tourne, permet à des utilisateurs extérieurs à la machine de se connecter sur la machine locale et d'y effectuer des commandes comme s'ils était physiquement au clavier local. Les utilisateur extérieurs peuvent utiliser `slogin user@nomMachine` pour se connecter, ou `ssh user@nomMachine commande` pour lancer une commande à distance, ou encore `scp fichierLocal user@nomMachine:adresse` pour copier des fichiers entre machines avec la même syntaxe que `cp`.

4. Exception : dans certains cas, un paquet peutr être un « broadcast », ce qui signifie qu'il est envoyé à toutes les machines présentes physiquement sur le même (sous-)réseau.

Les distributions fournissent en général une paramétrisation relativement raisonnable de `sshd` du point de vue sécuritaire. Il est bon cependant que `root` vérifie le fichier de paramétrisation (typiquement `/etc/ssh/sshd_config`) s'il souhaite ouvrir ce service à ses utilisateurs.

Pour des serveurs plus professionnels, on ouvrira aussi typiquement les services `ftpd` permettant aux utilisateurs de transférer des fichiers par exemple avec `lftp`, et Apache (démon `httpd`) pour constituer un serveur web, voire `mysql` pour les bases de données.

Dans tous les cas, l'ouverture de l'un de ces services impose de mettre en place un firewall (pare-feu) comme `shorewall` pour surveiller les communications qui y passent et stopper les tentatives de piratage. La paramétrisation se fait le plus souvent par interface graphique.

Elle impose aussi de mettre en place un démon, comme `msec` par exemple, qui vérifie quotidiennement que les droits des fichiers et surtout les « passe-droits » aux utilisateurs ne sont pas modifiés de manière malveillante. Il faut alors régulièrement lire les journaux du système (là encore, il y a des interfaces graphiques pour le faire) afin de vérifier si `msec` ou le firewall a signalé des anomalies. `msec` relève principalement les fichiers ayant un « sticky bit » qui donne les droits du propriétaire du fichier à tout utilisateur qui exécute ce fichier, les fichiers en écriture non restreinte, le contrôle des `$PATH` abusives, *etc.* Enfin, il est important de restreindre au maximum les exécutables qui peuvent être lancés depuis internet (java ou autres).

# 5 Les sauvegardes

Lorsqu'on gère une machine (même simplement son propre portable), il est ***crucial*** de faire des *sauvegardes régulières.* La fréquence des sauvegardes doit être guidée par la question simple « *quelle durée maximale de travail les utilisateurs acceptent-t-ils de perdre ?* ».

Note : il est inutile de sauvegarder les fichiers du système, une réinstallation est rapide ; ce sont les fichiers impliquant du travail ou difficiles à retrouver ailleurs qui doivent être sauvegardés.

La sûreté des données est toujours assurée par la *redondance* des données. Il faut donc plusieurs copies des données qui changent régulièrement, et il faut que ces copies ne soient pas soumises aux même risques. En particulier, il ne faut pas qu'elles soient détruites simultanément en cas de feu ou autre catastrophe (donc diversifier les bâtiments de stockage !).

Les logiciels de sauvegarde les plus courants sont `dump` et `tar`, et il est aussi possible de faire simplement plusieurs copies sur des disques durs amovibles. Il faut cependant continuer à assurer la confidentialité des données, donc crypter toutes les sauvegardes sensibles et bien gérer les droits d'accès...

Pour des petits volumes de sauvegardes, on peut « griller » des DVD, par exemple avec `k3b` ou tout autre graveur. Pour des sauvegarde de taille moyenne, on préférera des disques externes . Enfin pour de gros volumes on achète des stations de sauvegarde dont le volume et les fréquences de sauvegardes globales sont calibrés en fonction des besoins.

# 6 Les shell scripts et la programmation en shell

Pour gérer un système, on se rend vite compte que l'écriture de shell scripts permet de gagner beaucoup de temps. Un shell script est une simple fichier qui contient des commandes successives comme on les écrirait au clavier. On peut naturellement l'utiliser en le donnant en entrée standard de la commande `bash` mais le plus convivial est de procéder comme suit :

- créer un répertoire pour placer tous ses shell scripts et l'ajouter à la variable `$PATH`,
- faire commencer chaque fichier de shell script par la ligne « `#! /bin/bash -` » afin que le système sache qu'il doit interpréter ce fichier en utilisant le shell `bash`,
- rendre le fichier exécutable avec `chmod`.

Le shell script a alors automatiquement accès à des variables « numérotées » un peu spéciales :

- `$0` est l'adresse absolue du shell script
- `$1` est le premier argument donné au shell script par l'utilisateur qui l'a appelé, `$2` est le second argument, *etc.* Par exemple si l'utilisateur tape « `cmd -t toto tutu` » alors pour le shell script `cmd`, la variable `$1` vaut `"-t"`, `$2` vaut `"toto"` et `$3` vaut `"tutu"` ; les variables suivantes (`$4` *etc*) sont vides.
- `$#` est le nombre d'arguments du shell script (3 dans l'exemple précédent)
- `$*` est égal à `"$1 $2 $3 ..."`

— la commande `shift` décale vers la gauche les variables `$1`, `$2`, `$3`, *etc.* Par conséquent, la valeur de `$1` est perdue et `$#` diminue de 1.

Ces variables là, naturellement, ne doivent pas être exportées...

Le shell offre également les primitives de contrôle habituelles :

```
if [ ... ]
  then ...
  else ...
fi
#
# un "#" indique que la suite de la ligne est un commentaire
#
while [ ... ]
do
  ...
done
#
for variable in liste
do
   ...
done
#
case expression in
  pattern1) # expressions régulières au sens du shell, voir plus loin
    ...
    ;;
  pattern2)
    ...
    ;;
  etc)
esac
```

La syntaxe des conditions entre crochets (`if` et `while`) est définie dans `man test`. En réalité, le crochet ouvrant est une abréviation pour `test` et plus généralement ou peut donner en argument du `if` ou du `while` n'importe quelle commande. La condition est considérée comme « vraie » si la commande n'échoue pas, et comme « fausse » si la commande échoue. `test` est donc simplement programmé pour échouer si la condition qu'on lui donne en arguments est fausse.

Un shell est un langage interprété donc les « shell scripts » sont des fichiers lisibles, aisément vérifiables et modifiables. On peut ensuite programmer exactement ce qu'on taperait pour faire la commande en mode interactif. De plus, on peut programmer des « fonctions » (assez frustres) en shell :

Les redirections comme « `|` », « < », « > » *etc.* ou les backquotes (`` `...` ``) fonctionnent aussi dans un shell script.

Dans une commande, les fonctions s'écrivent sous la forme :

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

et les variables `$1`, `$2`, ... deviennent les arguments *de la fonction* et non plus celles de la commande globale.

Ainsi dans une fonction à l'intérieur d'une commande écrite en shell, les $1, $2, etc. sont les arguments *de la fonction* et on n'a donc plus accès directement aux arguments de la commande. Par exemple, si « `prevision` » est le shell-script suivant :

```
#! /bin/bash
truc () {
  ## ici $1 et $2 sont les arguments d'appel de truc, pas de toto :
```

```
  if [ "$1" = "$2" ]
    then echo "$1"
    else echo "$2$1"
  fi
}
## ici par contre le "$#" est le nbre d'arguments de toto :
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=‘truc $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`.

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 « `truc ()` », la fonction `truc` 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` (voir plus loin) ne peut donc pas faire appel à une fonction définie dans le shell script...

# 7 Quelques commandes indispensables

Dans les shell scripts courants, les commandes Unix suivantes sont très souvent utilisées (se reporter à l'annexe) :

`cat, mkdir, rmdir, man, which, whoami, file, ls, ps et top, emacs, xterm, grep, more et less, sed, find, openOffice, urpmf ou similaire, alias, gimp, lpr lpq et cancel, chmod, chown, chgrp, latex, rm, mv, tar, make, evince ou epdfview, thunderbird, firefox, kompozer...`

# 8 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.

# 9 Configuration d'un ordinateur ou d'un serveur

Les diverses *distributions* d'Unix (ou Linux) proposent toujours des options d'installation par défaut qui sont bien adaptées à un usage personnel. Dans le prochain cours, on verra en pratique comment construire et installer un ordinateur adapté à des usages plus spécifiques.

Le nombre de logiciels open-source que l'on peut installer sur une machine est colossal, à tel point que l'on ne peut pas choisir un par un les logiciels que l'on veut installer. Pour cette raison, les distributions propose des « *packages* » qui sont des ensembles de logiciels utiles pour un type d'activité. Malgré cela, il y a un très grand nombre de packages. Par conséquent, après avoir effectué une installation « standard », on va installer des packages supplémentaires, également offerts par la distribution. L'avantage de choisir un package au sein de la distribution que l'on a installée et que les fichiers de configuration sont par défaut correctement paramétrés et bien placés. Sinon, il est possible qu'il faille modifier quelques fichiers après installation d'un package.

Il n'en reste pas moins que quelques fichiers de paramétrage sont à réviser afin de les adapter à son usage propre. Il s'agit typiquement des partitions, de la paramétrisation du pare-feu et des sécurités, de la personnalisation des fichiers de `/etc` et parfois certains fichiers de `/usr/lib` ou ailleurs...

Ce cours est illustré par la pratique sur une station de travail raisonnablement récente : montage/démontage des composants, partitionnement du disque dur, installation d'une distribution Linux *ex nihilo*, paramétrisation du système.

# 1 Le montage d'un ordinateur, choix des composants

Pour construire une station de travail ou un serveur totalement adapté aux besoins des utilisateurs, il faut d'abord évaluer les capacités utiles de chacun des composants.

On commence toujours par choisir **le processeur** car ses caractéristiques déterminent en partie celles de la carte mère (la plaque bourrée d'électronique qui supporte et connecte la plupart des autres composants de l'orfinateur). Les caractéristiques principales d'un processeur sont

- sa *fréquence*, qui se mesure en Hertz : c'est le nombre d'instructions élémentaires qu'il peut effectuer en une seconde. Plus la fréquence est élevée, plus l'ordinateur fera des calculs rapidement. Actuellement les fréquences standard varient de 2 à 4 Giga-Hertz (par exemple 3,5Ghz)
- le nombre de « *cœurs* » (cores) : face à la difficulté croissante à augmenter la fréquence, les constructeurs de processeurs placent plusieurs processeurs (plusieurs « cœurs ») dans un même composant, qui coopèrent pour simuler un processeur plus puissant. Avoir 4 cœurs dans un processeur ne signifie pas qu'il va 4 fois plus vite car une partie des calculs effectués sert à synchroniser les cœurs entre eux, de sorte que les performances dépendent du niveau d'indépendance les uns par rapport aux autres des processus qui tournent à un moment donné.
- le nombre de bits du processeur (32 bits, 64 bits ou compatible 64 bits) : ce nombre détermine en particulier la taille des adresses en mémoire que peut gérer le processeur, et par conséquent la taille maximale de la mémoire qu'on pourra mettre dans la machine. Un processeur 32 bits ne gère pas plus de 2 Giga-octets de mémoire, ce qui devient obsolète. Les machines raisonnablement récentes ont un processeur 64 bits.

Le processeur et la carte graphique sont généralement les deux composants les plus chers d'un ordinateur. De plus, en l'espace de six mois à un an, un processeur top niveau du marché se voit généralement supplanté par un autre et son prix baisse considérablement. Si le budget est limité, on peut aisément se contenter d'un tel processeur « de la génération précédente ».

Le processeur qu'on aura choisi selon les critères précédents possède une norme pour son « support », appelé *socket* : c'est ce qui détermine l'agencement des nombreuses « pattes » du processeur. Un processeur a en effet sur une de ses faces un grand nombre de connections qui doivent être branchées sur la carte mère, et sur l'autre face une surface métalique plane qui doit être collée avec soin sur un refroidisseur (avec des ailettes et un ventilateur).

**La carte mère** doit ensuite être choisie parmi celles qui offrent la même norme de *socket* que le processeur. La taille de la carte mère est déterminée par un format. Le plus courant est le format ATX. À part cela, en pratique, peu de choses distinguent une carte mère d'une autre, de sorte que l'impact sur les performances de l'ordinateur est faible. Il faut simplement s'assurer qu'il y a assez de connecteurs sur la carte pour le nombre de disques durs internes et de lecteurs/graveurs internes qu'on souhaitera y connecter, et que la norme de la carte graphique qu'on choisira est compatible avec le port graphique de la carte (typiquement « PCI-E »). En général toutes les cartes graphiques standard à une époque donnée partagent la même norme et donc toutes les cartes mères de la même époque l'offrent... Bien souvent, le socket est donc le seul critère de choix objectif pour une carte mère.

La carte mère définit une norme de mémoire, généralement la plus rapide du moment. Il faut donc choisir des **barettes de mémoire** suivant cette norme. Selon l'usage prévu, on peut avoir besoin de 1 Giga-octet à un Tera-octets de mémoire. Il faut toujours être assez généreux en mémoire pour que l'ordinateur ne « swape » jamais faute de mémoire car le swap, qui fait appel à une mémorisation matérielle plutôt qu'électronique des données, fait baisser drastiquement les performances. Un système avec interface graphique peut utiliser quasiment un giga-octet de mémoire ; il faut donc évaluer la mémoire que peuvent utiliser les autres applications envisagées sur cette machine. *Bon à savoir :* les performances sont meilleures si toutes les barettes mémoire ont la même capacité, cependant si vous envisagez de garder l'ordinateur assez longtemps, si vous voulez par exemple 4 Giga-octets de mémoire, autant choisir un seule barette de 4 gigas car cela laisse libre les 3 autres ports mémoire de la carte mère pour des extensions de mémoire ultérieures.

Le choix du ou des **disque(s) dur(s)** se fait en fonction du volume de données à stocker. Les capacités actuelles vont de quelques centaines de Giga-octets à quelques Tera-octets. La vitesse de rotation (typiquement 7200 rotations par minute : 7200 RPM) et la norme (actuellement S-ATA 3) influent sur le temps d'accès aux fichiers.
Pour des serveurs de données, on montera plusieurs disques durs en « RAID » cela signifie que, d'une part, les données

sont dupliquées sur plusieurs disques (par exemple 2 disques de même capacité) et donc si l'un tombe en panne, l'autre contient encore les données et l'on peut donc changer le disque abimé sans interrompre la marche de l'ordinateur, et d'autre part, que l'accès aux disques est plus rapide, puisque le système peut lire la moitié d'un fichier sur un disque pendant qu'on lit l'autre moitié sur l'autre disque. *Bon à savoir :* deux disques durs identiques ont une durée de vie quasiment identique donc mieux vaut monter en RAID des disques de capacité identiques mais de constructeurs différents.

Le choix de **la carte graphique** dépend, d'une part, de la taille de l'écran (s'il est immense, il faut une carte capable d'exploiter sa résolution en nombre de pixels), et d'autre part, de l'usage envisagé (inutile de prendre une carte graphique de haut de gamme si les applications 3D ne sont pas fortement utilisées). Pour un usage bureautique, les cartes les plus simples sont déjà beaucoup plus puissantes que nécessaire. ***Attention :*** plusieurs constructeurs gardent secret les fonctionnalités de leurs cartes graphiques et ne fournissent qu'un « driver » pour Windows. Dans ce cas, Linux peut ne pas offrir de driver, faute d'information sur ce que fait la carte graphique. Actuellement, Nvidia est le constructeur de cartes graphiques le plus connu qui fournisse systématiquement des drivers pour Linux. C'est seulement en partie le cas pour ATI.

**Les lecteurs** et **les graveurs** de CD DVD ou Blu-ray internes partagent tous les mêmes spécifications à une époque donnée.

**L'alimentation électrique** doit être assez puissante pour alimenter tous les composants choisis (de l'ordre de 400W pour un ordinateur standard à 1000W s'il y a beaucoup de lecteurs et une carte graphique puissante). Le niveau de bruit du ventilateur qui refroidit l'alimentation est d'expérience un critère à ne pas oublier...

Enfin le choix du **boîtier** qui portera tous les composants précédents est déterminé d'une part par le format de la carte mère (en général ATX). Sauf si très peu de composants sont à monter et si l'espace manque, mieux vaut choisir une tour assez grande pour brancher, débrancher et dépoussiérer facilement tous les composant. « mini-tour » est donc un format souvent trop petit, « moyen tour » est le format le plus courant, « grand tour » est utile si vous avez beaucoup de disques, lecteurs ou graveurs, enfin le format horizontal « desktop » est similaire à « moyen tour » [l'usage veut qu'on n'accorde pas l'adjectif...]. Il est également utile de prévoir un ventilateur fixé à la tour s'il commence à y avoir un bon nombre de composants dans la machine.

Un dernier point : la poussière est suffisamment conductrice du courant pour créer un court-circuit sur les composants. Il faut donc régulièrement (par exemple tous les 6 mois) ouvrir le boîtier et éliminer la poussière à l'aide d'air comprimé.

# 2 Installation d'une distribution de la famille des Unix

C'est généralement d'une grande simplicité si l'on ne cherche pas à « customizer » l'installation (dans un premier temps...). Il suffit de booter sur le DVD d'installation et de suivre les indications du cliquodrome. NOTE : il se peut qu'il faille d'abord paramétrer le BIOS (Basic Input Output System), qui pilote la carte mère, afin de pouvoir booter sur le lecteur de DVD.

À la fin de l'installation, il est fondamental de choisir un mot de passe pour `root` qui soit difficile à casser et de créer des utilisateurs standard qui n'ont aucun des droits de `root`. On ne passe root que très peu de temps pour la maintenance de la machine (en utilisant `/bin/su`) et l'on se connecte donc en tant qu'utilisateur standard.

Tous les utilisateurs retrouvent facilement leurs habitudes avec les environnements graphiques KDE ou GNOME. Les connaisseurs préfèrent souvent des environnement moins gourmands en temps calcul, comme XFCE ou LXDE par exemple. Ce sont des options qu'on peut choisir au moment de l'installation (mais aussi ajouter plus tard en fonction des besoins).

Les réglages de sécurité sont souvent préconfigurés dans les distributions Linux de sorte qu'on se contente de choisir un « niveau de sécurité ». Une machine qui n'est ni un serveur ni une passerelle aura un niveau standard. Il faut choisir un niveau plus élevé pour les serveurs ou les passerelles. Pour ces derniers, il vaut mieux dédier la machine à cette tâche et prévoir qu'aucun utilisateur ne s'en serve directement (sauf depuis l'extérieur *via* `ssh`).

Les sites de distribution de packages sont à choisir très soigneusement et uniquement parmi ceux bénéficiant d'une totale confiance... Le plus simple est de se limiter aux sites certifiés par la distribution elle même. On choisit aussi des types de packages parmi lesquels on pourra choisir (éviter les packages de type test ou en cours de développement).
Il y a aussi des packages « non libres » dont certains peuvent comporter des parties qui ne sont pas open-source, et les « tainted » qui mettent dans le domaine public des fonctionnalités supplémentaires pour lesquelles il faut s'assurer d'avoir le droit de les utiliser ; certains peuvent être d'un usage interdit dans certains pays qui restreignent certaines libertés individuelles. Il faut bien admettre que ce type de packages améliorent parfois le confort d'utilisation de la

machine pour beaucoup d'utilisateurs.

À la fin d'une installation, lorsque tous les packages sont installés, lancer `makewhatis` afin de créer la table inversée qui permet d'utiliser l'option `-k` de `man` (indispensable). Le refaire régulièrement si de nouveaux packages sont installés.

*Démonstration en temps réel, avec la distribution mageia. Voir le mémo distribué en cours.*

# 3 Quelques packages utiles

Le nombre de packages existants est immense, il est impossible de les connaître tous. Les distributions proposent donc des ensembles de packages par défaut, modulables selon le type d'utilisation (station utilisateur, serveur, passerelles, *etc.*) et il faut ajouter ceux qui correspondent aux usages spécifiques de l'entreprise et du service considéré.

Des packages comme `biopython` (librairies de python spécifiques à la biologie), `Xdialog` (fenêtres de dialogues dans des shell-script), `lamp-php` (pour un serveur web ou un serveur local), `printing` (pour accéder à tous types d'imprimantes), `C-devel` (programmer en C ou C++), `firefox` (navigateur web où l'on peut paramétrer un minimum de confidentialité), `thunderbird` (interface pour le mail), `xterm` (fenêtre « console » pour exploiter un système Unix/Linux avec des lignes de commandes)... sont bien souvent indispensables dans un contexte de bio-informatique.

*Démonstration en temps réel. Voir le mémo distribué en cours.*

# 4 Mots de passe et protections diverses

Recommandations de bon sens :

- Il faut régulièrement (in)former vos utilisateurs à propos des risques qu'ils font courir à l'entreprise ou au laboratoire (et pas seulement à eux-mêmes) s'il choisissent un mot de passe trop simple ou s'il le divulguent.
- Des règles simples (paramétrables) sur les mots de passe peuvent limiter les dégâts :
  - au moins une majuscule et une minuscule, un chiffre, une ponctuation
  - aucun mot d'un dictionnaire de quelque langue que ce soit
  - pas de références à des informations faciles à obtenir (dates de naissances de membres de la famille, nom d'un dispositif présent dans le service ou dans le quartier, *etc.*)
  - imposer aux utilisateurs de changer régulièrement de mot de passe
  - *etc.*
- Il faut régulièrement aller lire les fichiers de « log » des machines (surtout les serveurs ou passerelles) et repérer les messages inhabituels sur les services de communication (`smtp`, `ftp`, `ssh`, `nfs`, *etc.*) et même chose sur les résultats des démons en charge de la sécurité (comme `msec`).
- Il faut installer très régulièrement toutes les mises à jour du système, surtout celles qui touchent à la sécurité.
- Redite... Le choix des sites de mises à jour des packages doit donner une importance majeure de leur niveau de confiance. Sauf exception, ne retenir que ceux certifiés par la distribution.
- Il faut fermer rapidement les comptes utilisateurs obsolètes, *etc.*

# 5 Annexe : mémo d'installation Mageia

Distribué en cours.

# 1 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.

# 2 Usage de `tar`

`tar` (comme « tape archive ») est une commande qui permet d'archiver dans un seul fichier toute une arborescence. Les explications suivantes vous faciliteront la lecture du `man` de `tar`.

`tar` possède trois options principales :
- `-c` pour créer un fichier d'archive à partir des répertoires à archiver
- `-x` pour extraire les répertoires archivés à partir d'un fichier d'archive
- `-t` (table of contents) pour voir le contenu d'un fichier d'archive

Pour **créer** un fichier d'archives, la ligne de commande est de la forme

`tar -cαf` *`fichierArchive repertoire_1 repertoire_2 ...`*

où $\alpha$ est en fait un ensemble d'options annexes « modificatrices » du comportement de `tar` et les *`repertoire_i`* sont les répertoires à archiver dans le fichier *`fichierArchive`*. Les options modificatrices les plus courantes sont :

- `v` (verbose) pour que `tar` indique chaque fichier qu'il archive
- `j` pour que le `fichierArchive` soit automatiquement compressé par `bzip2` (sans compression on a coutume de donner le suffixe `.tar` au fichier, avec compression on lui done le suffixe `.tar.bz2`)
- `z` pour une compression plus faible mais plus rapide avec `gzip` (suffixe habituel `.tar.gz` ou `.tgz`)
- de plus l'option `--atime-preserve` évite de mettre à jour la date de dernière utilisation des fichiers archivés (bien que `tar` les lise, naturellement)

Ainsi par exemple pour sauvegarder `/home`, `root` peut taper les commandes :

```
cd /home
tar --atime-preserve -cvjf /sauvegardes/home.tar.bz2 .
```

Le point en dernier argument du `tar` (après avoir fait `cd /home`) permet d'avoir une archive où les adresses de fichier ne commencent pas par `home` : cela permet le cas échéant de restaurer les homedirs dans n'importe quel autre répertoire.

Ensuite il faut évidemment transférer/copier le répertoire `/sauvegarde` en lieu sûr...

Pour **restaurer** un fichier d'archives, la ligne de commande est de la forme

`tar -xαf fichierArchive`

avec les mêmes options modificatrices les plus courantes que pour `-c`.

Par exemple pour restaurer `/home` à la suite d'un crash, `root` peut taper les commandes suivantes :

```
cd /home
for name in *
do
  mv $name $name.damaged
done
tar -xvjf /sauvegardes/home.tar.bz2
```

Et s'il s'agit simplement de vérifier le contenu de l'archive :

```
tar -tvjf /sauvegardes/home.tar.bz2
```

donnera la liste des fichiers de l'archive sur la sortie standard.

# 3 Présentation des projets

Voir feuilles distribuées.

# 4 Un exemple de shell script

Une commande de gestion de la luminosité d'un écran :

```
#! /bin/bash -
#
```

```
minimum=15 #pourcent de brightness
pas=5      #incrément ou décrément
maj=4      #dixièmes de secondes entre deux mises à jour
###
cmd="‘basename $0‘"
tmp=/tmp/$cmd$$
#
aide () {
  echo "Usage:  $cmd [pourcentage|+|-]"
  echo
  echo "Le pourcentage doit etre compris entre $minimum et 100."
  echo "Il est demandé par interface graphique (et mis à jour en temps réel) si aucun"
  echo "argument n’est donné."
  echo "Les arguments + ou - aygmentent ou diminuent le pourcentage de $pas %."
  exit 1
}
#
ajuste () {
  if [ -n "$p" ]
    then # Calcul de la brightness:
      p="‘expr "$1" + 0‘"
      if [ "$p" -gt 100 -o "$p" -lt $minimum ]
        then aide
      elif [ "$p" -eq 100 ]
        then b="1.0"
      elif [ "$p" -ge 10 ]
        then b="0.$p"
        else b="0.0$p"
      fi
      # On y va:
      for e in $ecrans
      do
        xrandr --output $e --brightness $b
      done
  fi
}
# paramètres actuels:
ecrans="‘xrandr -q | sed -e ’/ connected/ ! d’ | sed -e ’s/\([^ ]*\) .*/\1/1’‘"
actuel="‘xrandr --verbose -q | sed -e ’/Brightness: / ! d’ | \
  sed -e ’1 ! d’ -e ’s/ *Brightness: *\([^ ]*\).*/\1/1’ -e s/1\..*/100/1 -e ’s/0\.\(..\).*/\1/1’‘"
actuel=‘expr $actuel + 0‘
# Analyse des arguments:
case "$1" in
  +|++) p=‘expr "$actuel" ’+’ "$pas"‘
    if [ "$p" -gt 100 ]
      then p=100
    fi
    ajuste $p
    ;;
  -|--) p=‘expr "$actuel" ’-’ "$pas"‘
    if [ "$minimum" -gt "$p" ]
      then p=$minimum
    fi
    ajuste $p
    ;;
  [0-9]|[0-9][0-9]|[0-9][0-9][0-9]) p="$1"
    ajuste $p
    ;;
  ’’) msg="Choisir un pourcentage de luminosite :"
    echo $actuel > "$tmp"
    # "stdbuf -oL" réduit le buffering de sortie à une ligne maximum pour alimenter $tmp en temps réel
```

```
    ( stdbuf -oL Xdialog --interval ${maj}00 --stdout --title "$cmd" --no-cancel --rangebox "$msg" \
                                                    0 0 $minimum 100 $actuel 2> /dev/null >> "$tmp"
      rm -f "$tmp" )&
    while [ -r "$tmp" ]
    do
      p="`tail -1 "$tmp"`"
      ajuste $p
      sleep 0.$maj
    done
    ;;
  *) aide
esac
```

et au passage on mentionne l'intérêt de `/dev/null`, des commandes `read`, `Xdialog` (et de son manuel en pages HTML), *etc.*

# 5 Les chaines de caracteres et l'evaluation des expressions en shell

TO DO... (difference entre `"..."` et `'...'`, les commandes `eval` et `source`, le `&`)

# Quelques commandes utiles mentionnées dans ce cours

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

- `alias` : permet de donner des diminutifs à des commandes souvent utilisées. `alias` *nom*="*vraie commande*"
- `basename` et `dirname` : respectivement de nom du dernier lien dans une adresse et le nom de son répertoire. Par exemple : `basename` appliqué à l'adresse `/toto/tutu/titi.txt` retourne `titi.txt` alors que `dirname` retourne `/toto/tutu`.
- `bash` : le shell le plus courant, souvent appelé simplement `sh`.
- `cat` est plus frustre que `less` : il recopie sur sa sortie standard, le contenu du fichier qu'on lui donne en argument (ou sinon son entrée standard).
- `cd` : change directory (change le répertoire de travail courant).
- `chgrp` : change group, change le groupe d'un fichier (voir aussi `chown`).
- `chmod` : pour attribuer les droits d'accès à un fichier dont on est propriétaire (change mode).
- `chown` : change owner, modifie le propriétaire, et éventuellement le groupe, d'un fichier (seul `root` peut le faire).
- `df` : liste les partitions montées et leur taux d'utilisation. Voir aussi `mount`
- `echo` : écrit sur sa sortie standard les chaînes de caractères qu'on lui donne en argument (utile pour connaître la valeur d'une variable par exemple).
- `emacs` : éditeur de fichiers textes très puissant mais qui requiert d'apprendre quelques suites de clefs pour s'en servir valablement.
- `evince`, `xpdf`, `epdfview`... : outils de visualisation et impression de fichier PDF (versions open-source fiables de « readers » bien connus qui présentent souvent des trous de sécurité tant on ne sait pas ce qu'ils font)
- `file` : détermine le type d'un fichier en explorant son contenu réel.
- `find` : recherche dans l'arborescence de fichiers. Voir la section qui lui est dédiée dans le cours.
- `firefox` : browser internet open-source (qu'il faut néanmoins paramétrer avant usage si l'on veut raisonnablement protéger la vie privée des utilisateurs !)
- freeOffice : version open-source et gratuite des suites bureautiques WYSIWYG du commerce.
- `gimp` : programme très puissant de manipulation d'images.
- `grep` : affiche à l'écran toutes les lignes de son entrée standard, ou d'un fichier donné en argument, contenant une expression régulière (voir section du cours à ce sujet).
- `k3b` ou autres graveurs de CDROM, DVD, BluRay : utiles aussi pour sauvegarder des fichiers `tar`.
- `kompozer` : éditeur convivial de pages HTML
- `latex` ou `pdflatex` : un logiciel professionnel de formatage de texte (qualité d'un livre). L'idée est qu'au lieu de formater à la main visuellement la mise en page, on indique dans un fichier texte ce que l'on veut d'un point de vue logique (ici un nouveau paragraphe, ceci est une figure, ici une section ou une sous-section, un chapitre, *etc.*) et le programme `latex` fait la mise en page en appliquant les règles de mise en page des éditeurs professionnels.
- `less` : montre un fichier texte page par page dans le terminal, la touche *espace* permettant de passer à la page suivante, la touche `u` de remonter, *etc.*
- `lftp` : transférer des fichiers entre machines en utilisant le protocole `ftp`.
- `lpq` : montre la liste d'attente de l'imprimante et/ou indique les problèmes d'impression
- `lpr` : imprime le fichier qu'on lui donne en argument sur l'imprimante désignée par la variable `$PRINTER`.
- `lprm` ou selon les systèmes `cancel` : supprime une demande d'impression de la file d'attente.
- `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*).
- `make` : outil de compilation de gros logiciels.
- `mkdir` : crée un répertoire
- `more` : voir `less... :-)`
- `mount` : permet de rattacher une partition à l'arborescence du système de fichiers.
- `mv` : « move » un fichier (mais en fait, se contente de le renommer).
- `newgrp` : permet à un utilisateur de changer de groupe temporairement (celui de son processus shell en fait)
- `ps` : fournit la liste des processus lancés à partir du terminal par l'utilisateur. Les options « `-x` » et « -aux » montrent plus de processus (beaucoup d'options, voir `man ps`). Voir également `top`.
- `pwd` : print working directory
- `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.
- `rmdir` : supprime un répertoire s'il est vide, indique une erreur sinon.

— `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` ».
— `shift` dans un shell script : décale vers la gauche les variables `$1`, `$2`, `$3`, *etc.* Par conséquent, la valeur de `$1` est perdue et `$#` diminue de 1.
— `shutdown` et `halt` : éteignent l'ordinateur. Voir également `reboot`.
— `/bin/su` : pour « passer `root` » et plus généralement pour lancer un processus au nom d'un autre utilisateur. *Note* : c'est une bonne habitude d'appeler cette commande par son adresse absolue...
— `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.
— `thunderbird` : browser de mails open-source (possède également une extension `lightning` pour gérer des EdT).
— `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` ».
— `touch` : positionne les dates de dernière modification et de dernière utilisation d'un fichier à l'instant présent.
— `urpmf`, `urpmi` : permet à `root` de trouver d'éventuels logiciels pas encore installés et de les installer sous la distribution Mageia. Les distributions Debian et Ubuntu utilisent la commande `apt-get` pour gérer leurs packages, Fedora utilise `yum`, OpenSUSE utilise `zypper`... pour ne citer que les distributions les plus connues.
— `which` : prend en argument un nom de commande et explore la variable `$PATH` pour fournir l'adresse de son fichier exécutable.
— `whoami` : fournit le nom de login de l'utilisateur qui a lancé ce processus.
— `xterm` ou `kterm` ou autre « fenêtre de terminal » : fait apparaître dans une fenêtre graphique un espace textuel qui permet d'utiliser le shell.

## Quelques fichiers utiles 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.
- `/etc/passwd` : fichier contenant les utilisateurs reconnus du système et les informations techniques qui leurs sont associées.
- `/etc/shadow` : fichier contenant les mots de passe cryptés des utilisateurs reconnus du système (seulement lisible par `root`).
- `/etc/group` : fichier contenant les groupes reconnus du système et les informations à leur sujet.
- `/home`, `/etc`, `/usr`, `/var`, `/dev` : voir le cours.
- `/etc/fstab` : paramètres de montage des partitions dans l'arborescence du système de fichiers.
- `/dev/null` : fichier modifiable et lisible par tout le monde mais qui a la particularité d'être toujours vide (taille 0). C'est le « fichier poubelle » du système, dans lequel on redirige typiquement les sorties (standard ou d'erreur) qui n'ont aucun intérêt dans un shell script.