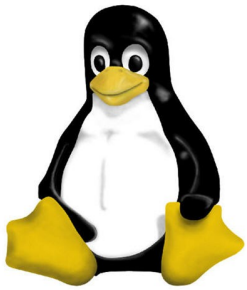

Formation Linux

Initiation au système GNU/Linux

Damien BLANCHARD

2010

v 0.3.2

Table des matières

Licence du document

Introduction

Le but de cette formation est de vous initier à l'utilisation et au fonctionnement d'un système *GNU/Linux*. Pour préparer cette formation j'ai surtout utilisé ce que j'ai pu apprendre par moi même en 8 ans d'utilisation. J'ai aussi utilisé quelques livres. Malheureusement dans la plupart des cas ces livres apprennent à installer et utiliser une distribution en particulier et insiste sur les outils graphiques. Afin de justement vous offrir la formation la plus générale et la plus utile nous allons uniquement voir les éléments qui sont communs à tous les systèmes *GNU/Linux*, qu'ils soient pour l'ordinateur de bureaux comme pour l'embarqué.

Pourquoi *GNU/Linux* et non *Linux* qui est le nom le plus communément utilisé ? Pour comprendre ça il faut voir un peu l'histoire de cette genèse.

1. Histoire de GNU/Linux

1.1. Richard Matthew Stallman

Le premier personnage important dans cette histoire n'est autre que Richard Matthew Stallman ou appelé par ses initiales RMS.

C'est en 1984 que RMS a décidé de créer un système d'exploitation (SE ou OS) entièrement libre nommé système *GNU*. Pour comprendre cette décision il faut revenir un peu en arrière. *Unix* était très répandu dans les universités (dont le MIT où travaillait RMS (plus précisément au MIT AI Lab)) et les entreprises. Cependant les licences des différentes version d'*Unix* rendaient le système propriétaire. Par exemple les universités avaient gratuitement le système et les sources mais ils n'avaient aucun droit de les redistribuer. C'est cette frustration qui a décidé RMS à créer son propre système. Le but du projet GNU était de créer un OS complet et entièrement libre. C'est-à-dire gratuit, que l'on peut modifier et redistribuer. C'est dans cet état d'esprit que RMS créa la licence GPL (General Public Licence).

Il commença par programmer la plupart des commandes *Unix* nécessaires à l'utilisation du système. Puis il chercha à pourvoir GNU d'un compilateur C. Malheureusement, les compilateurs les plus proches de ce qu'il cherchait n'étaient pas libres et ceux qui étaient libres nécessitaient trop de modifications pour que le projet soit viable. Il commença alors l'écriture depuis 0 (*from scratch*) qu'il nomma *GCC* acronyme de *GNU C Compiler*. *GCC* était assez flexible pour permettre de lui ajouter de nouveaux langages comme le C++, objective C, Java etc, et l'acronyme devint ensuite *GNU Compiler Collection*.

Cependant il manquait au projet *GNU* la pièce maîtresse d'un OS : le noyau. RMS ne voulait pas d'un noyau monolithique et s'orienta alors vers un micro noyau. La conception d'un tel noyau est plus compliqué et surtout le débogage est d'une incroyable complexité. Le développement pris du retard, beaucoup de retard. Finalement le retard est tellement important que même aujourd'hui il n'y a pas de version vraiment considérée stable.

1.2. Linus Torvald

Le second personnage clé est Linus Torvald, créateur du noyau Linux. En 1991 Linus était étudiant à l'université d'Helsinki en Finlande. Il utilisait *Minix*, un système non libre, basé sur le modèle d'*Unix*, créé par le professeur Andrew Tanenbaum a des fins didactiques. Il manquait dans *Minix* un émulateur de terminal ce qui aurait permis à Linus de se connecter sur les machines de son université depuis chez lui. Linus entrepris alors de le coder mais il le fit indépendant de *Minix*. Il continua d'ajouter des fonctionnalités à son émulateur de terminal tant et si bien qu'il en avait fait un embryon de noyau. Il porta dessus *bash* et *GCC*, tous deux issus du projet *GNU*. Il posta le fruit de son travail sur des listes de discussions sur internet et rapidement beaucoup de personnes commencèrent à le modifier et à ajouter leurs propres fonctionnalités.

Le choix de Linus de rendre son noyau libre, et le ralliement par internet d'une armée de développeurs permit dès 1994 d'avoir la version 1.0 de *Linux* sur lequel fonctionnait la majorité des programmes *GNU*.

1.3. GNU/Linux l'évolution

Installer un système *Linux* revenait à installer le noyau *Linux*, les outils *GNU* ainsi que d'autres logiciels tiers. C'était réellement une tâche compliquée réservée aux initiés. Des personnes eurent l'idée géniale de faire ce travail et de redistribuer ensuite ce système complet en tant que *Distribution Linux*. C'est dans les années 1993/1994 que deux distributions, qui existent encore aujourd'hui, sont nées : *Slackware Linux* et *Debian GNU/Linux*.

Si aujourd'hui *Slackware* est peu utilisée c'est parce qu'elle est restée très proche de ce qu'elle a été au début : un ensemble cohérent comprenant le noyau, les outils *GNU* et des logiciels tiers. Alors que des distributions comme *Debian* (puis *Red Hat*, *Suse*, *Mandrake* devenue *Mandriva*) apportent une vraie valeur ajoutée par la possibilité d'utiliser des paquets : c'est-à-dire que tous les logiciels disponibles pour la distribution ne sont pas compris sur le support d'installation mais sont disponibles soit sur des supports annexes soit sur des serveurs internet. Ces paquets sont installables via des outils dédiés et permettent la résolution des dépendances. Par exemple si vous voulez installer le logiciel *foo* et qu'il nécessite la bibliothèque *libbar*, le gestionnaire de paquet va vérifier que vous avez *libbar*, dans le cas contraire il va le télécharger et l'installer puis faire de même pour l'application *foo*.

De plus ils ont souvent des outils graphiques permettant une configuration simplifiée de diverses parties du système. Par exemple la distribution qui est actuellement la plus populaire c'est *Ubuntu* (et ses dérivés *Kubuntu* / *Xubuntu*), si vous utilisez du matériel dont les drivers ne sont pas libres ou qu'ils nécessitent un *firmware* non libre, le petit outil graphique va se charger de tout télécharger et tout installer tout seul. Ce genre de choses doivent être faites manuellement lorsqu'on utilise une *Slackware*.

Tout cela concerne les distributions de bureaux (ou de serveurs : il existe des déclinaisons pour serveur d'*Ubuntu*, *Mandriva*, *Red Hat* etc). Mais l'utilisation de *GNU/Linux* ne s'arrête pas là. Outre les *desktops* et les serveurs, on le retrouve de plus en plus dans des systèmes embarqués. Par exemple de plus en plus de téléphones portables utilisent le noyau Linux comme tous les téléphones utilisant *Android* de *Google*. A noter que *Chrome OS* est aussi bâti sur un noyau *Linux*. Les tablettes internet de *Nokia* (770, N800, N810) utilisent aussi un noyau *Linux*. La plupart des routeurs, *Box ainsi que les systèmes GPS de *TomTom* fonctionnent grâce à *Linux*.

Maintenant, et toujours dans cette petite introduction, et avant d'attaquer les choses sérieuses sur la vraie constitution d'un système *GNU/Linux*, essayons de voir dans les grandes lignes ce que c'est, ce qu'il permet et ce qu'il n'est pas afin de démystifier un peu tout cela.

Comme on l'a vu dans sa genèse, *Linux* (le noyau) est inspiré de *Minix* (il n'en contient aucune ligne) qui lui même est un système purement destiné à l'éducation inspiré d'*Unix* (c'est donc un *Unix-like*). Depuis sa constitution *Linux* suit les normes et les standards comme *POSIX* ou *ANSI* ainsi que *BSD*. C'est un énorme avantage puisque lorsqu'on écrit un programme fonctionnant sur *GNU/Linux*, on peut normalement le porter facilement, voire sans modifications, sur d'autres systèmes respectant aussi ces normes comme les *BSD* (*NetBSD*, *OpenBSD*, *FreeBSD*), les *solaris* (*Solaris*, *OpenSolaris*) et dans une moindre mesure *MacOS/Darwin*.

En revanche il sera souvent beaucoup plus difficile de le porter sur un OS ne respectant rien à l'instar de *windows* de *microsoft* dont l'implémentation des normes et standards est très limitée, partielle et parfois ne respecte justement pas les standards.

Un autre point important que nous avons vu, le noyau *Linux* ainsi que la grande majorité des logiciels fournis avec (outils *GNU* inclus) sont libres. Vous pouvez accéder aux sources, les modifier, vous en inspirer, les redistribuer et tout ça gratuitement. C'est une sorte de gigantesque bibliothèque de codes sources entièrement libre.

Donc *GNU/Linux* est un système entier, à part, et c'est aussi un détail important. Il utilise le format *ELF* (*Executable and Linkable Format*) pour ses exécutable et rien que par ce détail on se rend compte qu'il ne peut pas exécuter les applications compilées pour *windows* (utilisant le format *PE*). Cependant il ne permet pas non plus d'exécuter des programmes compilés par d'autres systèmes utilisant *ELF* comme **BSD*, car il n'y a pas de compatibilités entre les bibliothèques C (bibliothèque système qui fait le lien entre le noyau et les applications utilisateurs).

Cependant pour les applications *windows* il existe une implémentation (encore non achevée) des bibliothèques systèmes de *windows*. Cette implémentation appelée *WINE* (*Wine Is Not an Emulator*) permet d'exécuter, plus ou moins bien, un grand nombre d'applications compilées pour les systèmes de *microsoft*.

Puisqu'on parle de *windows*, qu'est ce qui différencie un système *GNU/Linux* d'un *windows* ?

1.4. Les différences entre GNU/Linux et windows

Même pour l'utilisateur final certaines différences sont notables. Contrairement à *windows* qui est une sorte de bloc unique fourni par défaut, un système *GNU/Linux* est un ensemble de "petits" blocs ouverts. Ainsi, alors que sous *windows* il n'y a qu'un seul environnement de bureau sur lequel on peut appliquer des thèmes, sous *GNU/Linux* il y a un serveur graphique sur lequel s'ajoute un environnement de bureau au choix de l'utilisateur. Il en existe beaucoup et pour tout les goûts : de gros environnements tout intégrés comme *Gnome* ou *KDE*, des plus légers comme *XFCE* ou *LXDE*, des très très légers comme *blackbox* ou *openbox* et des rudimentaires. Contrairement à des idées reçues aujourd'hui les distributions sont très simples à utiliser, souvent bien plus que *windows* pourtant jugé intuitif.

La vraie différence vient de la configuration du système, et de ses modifications. En effet sous *windows* on ne peut rien faire sans la souris, tout se fait à coup de clics. La majorité des rouages internes sont cachés. Par exemple, si on peut accéder à la base de registre on ne sait pas vraiment où elle est stockée ni sous quel format. Si avec *Processus Explorer* on peut avoir un arbre graphique des processus exécutés, il est parfois impossible de les tuer.

C'est ici que tout change : les systèmes *GNU/Linux* sont avant tout basés sur des outils non graphiques, sur des commandes dans un terminal. Et même si les distributions récentes intègrent beaucoup d'outils graphiques permettant de quasiment tout faire à coup de clics, c'est cette flexibilité permettant d'utiliser aussi bien la souris que des lignes de commandes qui fait toute la robustesse et la puissance du système. On peut savoir ce que fait le système, et si quelque chose se bloque on peut savoir quoi et on peut y remédier. Les rouages ne sont pas cachés, on a un vrai rôle à jouer.

2. Architecture du système

Un système *GNU/Linux* pour un ordinateur de bureau possède ce type d'architecture :

Le noyau *Linux*
- Les outils *GNU* (bibliothèque C + *shell* + outils de gestion)
- des serveurs (*sshd*, *dhcpcd*, etc)
- Serveur graphique (serveur *X* / *Xorg*)
  - Environnement de bureau
    - applications graphiques (navigateur web, suite bureautique etc)

Au cours de cette formation nous nous arrêterons juste avant le serveur *X*.

2.1. L'arborescence

L'arborescence est l'ensemble des répertoires du système. Par exemple sous *windows* c'est très simple, nous avons quelque chose qui ressemble à cela :

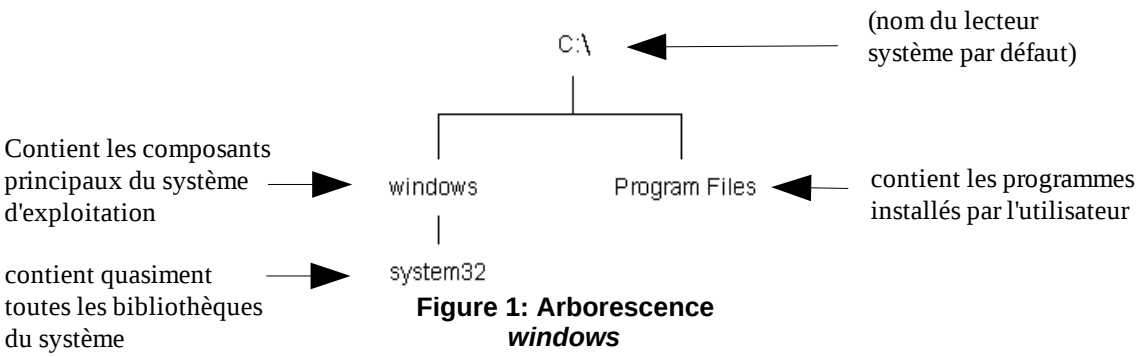

Figure 1: Arborescence *windows*

Sous *GNU/Linux* c'est un peu plus compliqué car les fichiers du système ont chacun une place définie.

Path	**Description**
/	nom de la racine de l'arborescence système
/bin	contenant les exécutables nécessaire au fonctionnement du système
/dev	contenant les fichiers spéciaux correspondant aux (pilotes de) périphériques
/usr/bin	contenant les exécutables des programmes installés
/usr/sbin	contenant des commandes disponibles que pour le super utilisateur (*root*)
/sbin	contenant des commandes disponibles que pour le super utilisateur mais qui sont indispensable au fonctionnement, surtout lors du démarrage, du système
/boot	contenant le(s) noyau(x) ainsi que d'autres fichiers nécessaires à l'amorçage du système
/etc	contenant les fichiers de configuration de la plupart des programmes systèmes, serveurs et parfois les fichiers de configuration par défaut des programmes des utilisateurs.
/var	contenant des fichiers utilisés par différents programmes systèmes, par exemple il contient des *logs*, des *sockets* etc.
/usr/lib	contenant les bibliothèques dynamiques ou statiques disponibles sur le système. C'est le répertoire de recherche par défaut pour l'éditeur de liens.
/usr/lib/X11	contenant les fichiers spécifiques au serveur graphique (*Xorg*)
/usr/include	contenant les fichiers d'en-têtes pour la programmation en C (bibliothèque C standard, API des bibliothèques tierces, en-têtes utilisable pour le noyau Linux comme *V4L* (*video for linux*)).
/usr/local	pouvant contenir une sorte de nouvelle arborescence avec un *bin*, *etc*, *lib* pour les applications ajoutées après l'installation du système.
/mnt	pouvant contenir les points de montage de d'autres périphériques de stockage (clef usb, autre disque dur, *NFS,* etc) aujourd'hui la tendance est plutôt à utiliser */media*
/home	contenant un répertoire par utilisateur dans lequel est stocké tous les fichiers de l'utilisateur.

Tableau 1: Arborescence d'un système *GNU/Linux*

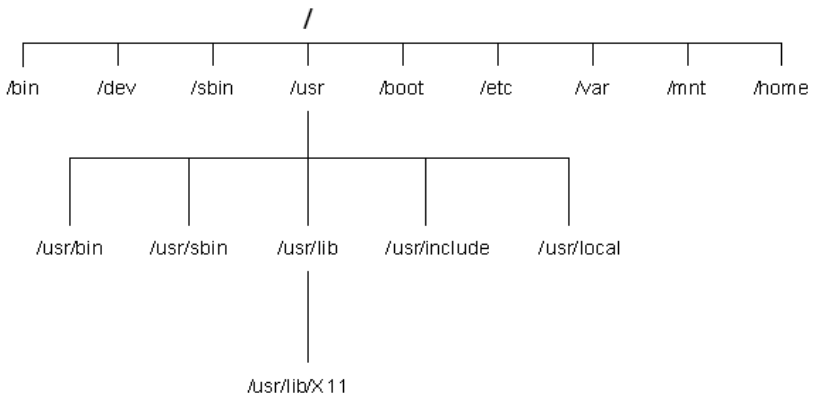

Figure 2: Arborescence sous *GNU/Linux*

Ce survol permet de mettre en évidence des choses très différentes d'un système *windows*. Par exemple, ici il n'y a pas de lettre de lecteur et les chemins (*PATH*) utilisent un *slash* (/) au lieu d'un *backslack* (\).

2.2. Les points de montage

Le point de montage du système s'appelle / (racine = *root*) et chaque répertoire peut être une partition d'un autre disque, ou du même disque. Par exemple nous avons 2 disques dur avec chacun 1 partition nous pouvons faire ceci :

Mettre */*, */boot*, */usr,* etc sur le premier disque et monter le deuxième disque sur */home* ce qui permet de séparer les données propres au système des données des utilisateurs.

2.3. Le nommage des périphériques

Une autre différence très importante est dans le répertoire */dev*. Ce répertoire contient des fichiers "virtuels" qui correspondent aux périphériques et permet de les utiliser via leurs pilotes respectifs de façon transparente.

Par exemple sous *windows* les partitions des disques durs ont des lettres que l'ont peut réattribuer dans le panneau de gestion du poste de travail via une représentation graphique des disques et de leur taille.

Sous Linux c'est totalement différents, chaque périphérique (disque dur et partitions par exemple) ont leurs propres fichiers dans /dev. Le disque dur SATA de la machine sera nommé */dev/sda* et sa première partition */dev/sda1*, la deuxième */dev/sda2*.

Pendant longtemps, et parfois encore aujourd'hui, les disques dur IDE étaient nommés */dev/hdX* (où *X* est le numéro du disque) par exemple le maître (*master*) du port (*slot*) 1 était nommé */dev/hda* et sa première partition */dev/hda1*. Aujourd'hui, sur les noyaux récents, la plupart du temps les disques IDE se nomment comme les disques SATA */dev/sdX*.

Voici d'autres exemples de noms de périphériques :

```
/dev/ttyS0      port série 1 (COM1 sous windows)
/dev/lp0        port parallèle 1 (LPT1 sous windows)
/dev/ttyUSB0    port série virtuel 1 (via un FT232 par exemple).
```

En revanche, les interfaces réseaux (filaire ou non) ne possèdent aucune entrée dans */dev.*

Il est à noter que les périphériques de stockage de masse (*mass storage device*) comme les clefs USB reprennent le même nommage que les disques SATA.

3. La ligne de commande

Avant de se lancer dans le partitionnement et le formatage des partitions il est temps de se familiariser un peu avec le terminal / la console / ligne de commande.

3.1. Terminal et shell

La ligne de commande est disponible de plusieurs façons. Il y a le terminal disponible quand le serveur *X* est lancé. Via un émulateur de terminal comme *xterm*, *aterm*, *exo-term*, *gnome-terminal*, *konsole,* etc. L'interface ressemble vaguement à celle disponible sous *windows*. Vaguement car elle est bien plus flexible, ergonomique, puissante et surtout utile.

Tout d'abord il faut savoir que l'émulateur de terminal n'est que la partie graphique qui permet de voir et de saisir du texte. Le lancement des applications et l'interprétation des commandes sont pris en charge par le *shell* de l'utilisateur. Le plus connu est le plus utilisé est *GNU bash.* La deuxième façon d'avoir accès à la ligne de commande et d'utiliser la combinaison de touches : *CTRL + ATL + Fx* (avec $1 < x < 6$) (*CTRL + ALT + F7* permet de revenir au serveur *X*). Ainsi on a accès directement à l'invite du *shell*.

3.2. Les commandes de base

Une commande est soit un programme à part entière, soit intégrée au *shell.*

Les commandes utilisent toujours le même formatage à savoir :

```
commande [OPTIONS] [PARAMETRES]
```

Les options permettent de modifier le comportement de la commande, et les paramètres peuvent être la cible (*target*) comme par exemple un fichier ou un répertoire.

Ce qui est entre crochets ([ ]) peut être, ou ne pas être, spécifié. Suivant le comportement que l'on veut de la commande ils ne sont pas nécessaires.

pwd affiche le répertoire courant.

ls permet de lister les fichiers dans le répertoire courant.

Quelques options de *ls* :

-l affiche une liste détaillée (avec heures de création, les droits, taille, etc)
-a affiche TOUS les fichiers
-h convertie la taille (en octet par défaut) en Kio, Mio, Gio pour que ça soit plus lisible.

On peut spécifier les options de 2 façons :

soit une par une exemple ls -l -a -h
soit groupée ls -lah
on peut aussi faire les deux ls -l -ah

cd permet de changer de répertoire. On peut préciser soit un *PATH* (chemin) relatif (= par rapport au chemin courant) soit absolu (= par rapport à la racine /). Donc un chemin absolu commence toujours par '/'.

exemples :

```
cd /home
cd login
```

On peut utiliser une écriture raccourcie pour accéder directement à */home/login* on note alors *~/*

```
cd /home/login
```

est équivalent à

```
cd ~/
```

Si on utilise *cd* sans argument, le répertoire choisi sera le *HOME* de l'utilisateur (*/home/login*).

cp permet de copier des fichiers (ou des répertoires).

```
cp [OPTIONS] FICHIER1 FICHIER2
cp [OPTIONS] FICHIER1 [FICHIER2]... REPERTOIRE/
```

FICHIERn peuvent être des répertoires mais il faut alors utiliser l'option *-r*.

rm permet de supprimer des fichiers (ou des répertoires)

```
rm [OPTIONS] FICHIER1 [FICHIER2]...
```

FICHIERn peuvent être des répertoires mais il faut alors utiliser l'option *-r* comme pour *cp*.

On peut aussi utiliser l'option *-f* pour forcer la suppression et éviter les demandes de confirmations.

mv permet de renommer un fichier ou un répertoire

```
mv FICHIER1 FICHIER2
```

touch permet de mettre à la date courante un fichier existant ou à créer un fichier vide si celui ci n'existe pas.

```
touch nom1
```

Le *shell* fournit une fonctionnalité très utile : l'*auto-completion*. Elle permet de compléter automatiquement ou de fournir une liste des commandes ou des fichiers qui commencent par les lettres que l'on a saisi.

Par exemple si on veut faire *ls* sur un fichier (existant) nommé « ceci est un test.txt », on peut uniquement écrire « *ls ceci* » et appuyer sur la touche tabulation (*TAB*) ce qui complétera automatiquement le nom du fichier.

mkdir permet de créer des répertoire

```
mkdir REPERTOIRE
```

cat est un programme qui lit un ou plusieurs fichiers et les affichent dans le terminal.

La syntaxe est simple :

```
cat FICHIER1 [FICHIER2]...
```

Par convention la plupart des commandes disposent de l'option *--help* (ou parfois *-h*) qui permet d'afficher une aide succincte indiquant les paramètres et les options disponibles.

3.3. Des détails importants

a. Les arguments

Il y a une question importante qu'il faut se poser : comment fait un programme pour parcourir et traiter (*to parse*) les options et les paramètres qu'on lui soumet ?

La réponse est simple : le *shell* va découper les informations en utilisant l'espace comme délimiteur. Le programme va avoir en entrée un tableau (le fameux char *argv[] en C) dont chaque case correspond à un « mot ». C'est un détail très important qui nous permet de comprendre, par exemple, qu'on ne peut pas faire ceci :

```
ls-l
ls -l-a
```

A l'utilisation on s'aperçoit qu'il reste 2 autres points gênants.

b. Les espaces dans les paramètres

Si un paramètres contient un espace, comment faire pour que le *shell* ne l'interprète pas comme 2 paramètres distincts ?

Il existe pour cela 2 méthodes :
Soit on indique au *shell* que c'est une seule chaîne de caractères en utilisant, comme en C, les guillemets (cela fonctionne aussi avec apostrophe) avant et après le paramètre.

Exemple :

```
touch "le fichier"
```

Soit on précède l'espace d'un '\' (*backslash*), ce qu'on appelle un caractère d'échappement, pour indiquer explicitement que l'espace n'est qu'un caractère normal et qu'il ne doit pas avoir d'autres fonctions. Cette méthode fonctionne aussi pour tous (sauf le -) les autres caractères spéciaux comme ' (apostrophe) et " (guillemet) ainsi que d'autres que nous verrons lorsque nous aborderons l'écriture de commandes plus complexes ainsi que les scripts.

c. *Paramètres avec un tiret*

Si un paramètres commence par un tiret, comment faire pour que ça ne soit pas traité comme une option ?

Il est possible d'indiquer explicitement la séparation entre les options et les paramètres en indiquant 2 tirets (--) entre les deux.

Exemple :

```
touch -- -lefichier
```

d. *Le copier / coller*

La question que l'on fini toujours par se poser : comment peut-on faire un copier coller dans un terminal ?

Si certains terminaux graphiques affichent un menu lors d'un clic droit permettant de copier coller ce n'est pas général. On ne peut pas non plus utiliser de raccourcis clavier comme *CTRL+C CTRL+V* car ils ont un autre rôle (envoyer des signaux).

En revanche il existe une autre méthode, très pratique, utilisable dans toutes les applications sous *GNU/Linux*. Il suffit avec la souris de sélectionner le texte, ce qui sélectionne et copie en même temps, ensuite avec le clic central (celui de la molette) on colle le texte. Si la souris ne possède que 2 boutons, le clic central est émulé par l'appui simultané du bouton gauche et du bouton droit.

3.4. Les éditeurs de texte en terminal

Petite parenthèse pour introduire certains programmes qui seront nécessaires tout au long de la formations : les éditeurs. En effet, à de nombreuses reprises nous aurons à manipuler des fichiers textes, soit pour lire leur contenu soit pour le modifier ce contenu. Nous allons donc voir quelques éditeurs souvent présent par défaut sur les distributions.

Nous avons vu qu'avec *cat* nous pouvions lire le contenu d'un fichier, mais lorsqu'il est long cela n'est pas très pratique, nous verrons plus tard que l'on peut compléter le comportement de *cat* avec d'autres outils. En attendant regardons quelques éditeurs.

a. *pico et nano*

Il existe pléthore d'éditeurs utilisables dans un terminal, allant de très simples éditeurs à des outils très complets et assez difficiles à utiliser. Dans les éditeurs très simples nous avons : *nano* et *pico*.

pico est un clone de *nano*, leurs interfaces et leurs commandes sont identiques.

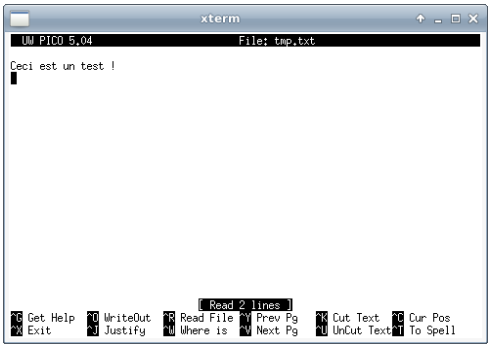

Figure 4: Capture d'écran de *pico*

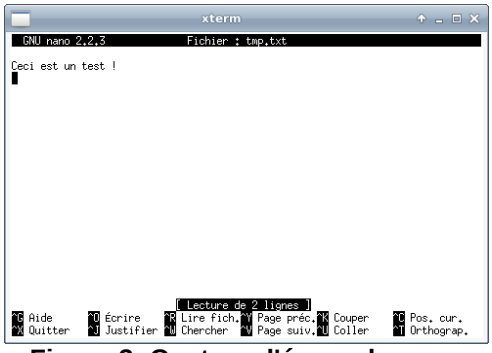

Figure 3: Capture d'écran de *nano*

Les raccourcis sont inscrits en bas, le signe '^' correspond à la touche *CTRL*. Donc, par exemple, pour quitter il faut faire *CTRL + x*.

On édite simplement un fichier par :

```
nano nom_du_fichier
```

ou

```
pico nom_du_fichier
```

Leur utilisation est semblable à n'importe quel éditeur auquel on est habitué : on peut modifier directement le texte, utiliser les touches *backspace* et *suppr*.

Malheureusement, parfois ces 2 éditeurs ne sont pas disponibles par défaut, il faut donc se tourner soit vers *VI* soit vers *emacs* d'utilisation bien moins évidente.

b. VI, le minimum VItal

Chaque changement de mode commence par un appui sur la touche *Echap* (*Esc*), pour pouvoir éditer le contenu il faut passer en mode *Insertion* via *Esc* puis *i*. Attention, car même en mode insertion on ne peut pas utiliser les touches *suppr* ou *backspace*.

Pour revenir en mode commande il suffit d'appuyer de nouveau sur *Esc*, ensuite chaque touche appuyée est considérée comme une commande. Voici un tableau de quelques commandes :

Commande	**Effet**
dd	Coupe la ligne courante
P	Colle au début de la ligne
x	Supprime le caractère sous le curseur
D	Supprime la fin de la ligne
R	Remplace caractère par caractère
u	Défaire les modifications (undo)
ZZ	Sauvegarde les modifications et quitte VI
:w!	Sauvegarde les modifications
:q!	Quitte VI

Tableau 2: Commandes de *VI*

c. *Emacs*

Emacs est un éditeur dans le même style que *VI*, à base de raccourcis clavier. Cependant cette fois on peut effectuer directement les modifications sans devoir passer par un mode insertion et les touches *suppr* et *backspace* sont utilisables.

Commande	**Effet**
CTRL + espace au début de la partie à sélectionner, positionner le curseur à la fin de la zone à sélectionner	Sélectionne
ESC + w	Copie la sélection
CTRL + w	Coupe la sélection
CTRL + y	Colle la sélection (copiée ou coupée)
CTRL + x u	Défaire les modifications (undo)
CTRL + x CTRL + s	Sauvegarde les modifications
CTRL + x CTRL + c	Quitte emacs

Tableau 3: Commandes d'*Emacs*

4. Manipulation des disques

Maintenant que l'on sait comment se nomme un disque et que le terminal n'est plus totalement inconnu, on pourrait voir comment le partitionner.

4.1. Le partitionnement

C'est extrêmement simple, l'outil en console le plus simple pour partitionner se nomme *cfdisk* et s'utilise ainsi :

cfdisk PERIPHERIQUE

par exemple :

```
cfdisk /dev/sda
```

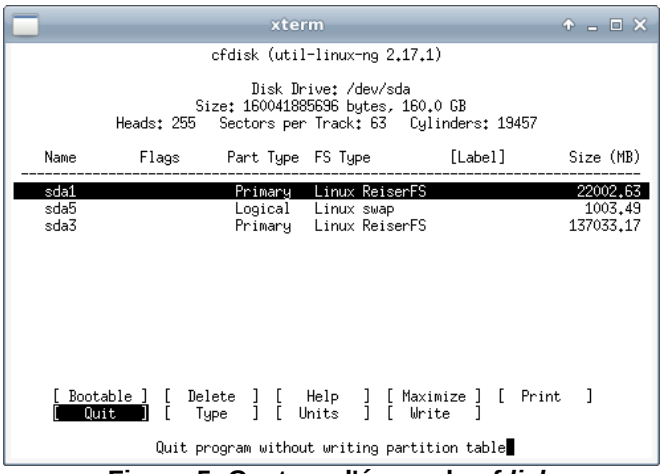

Figure 5: Capture d'écran de *cfdisk*

Si on va dans le menu *type* on peut voir une très longue liste de type de systèmes de fichiers (*FS* = *FileSystem*). C'est l'un des premiers choix à faire lorsqu'on désire utiliser un système *GNU/Linux*.

Le système de fichier est l'organisation à l'intérieur du disque des fichiers ou des morceaux de fichiers. Le choix est important car il influe sur les performances et la fiabilité.

Contrairement à *windows* qui ne gère que la *FAT* et le *NTFS*, on peut utiliser un très grand nombre de *FS* différents sous *GNU/Linux*. Par exemple le *ext2fs* sera utilisé uniquement sur la partition */boot* car il est toujours compilé en dur dans le noyau alors que les autres sont souvent compilés en tant que modules (on reviendra plus tard sur ces notions). Cependant *l'ext2fs* est aussi fiable que la *FAT* : c'est à dire pas du tout. Il faut privilégier un *FS* journalisé, c'est-à-dire qui dispose d'un journal des différentes écritures ce qui permet de corriger les problèmes lors d'un blocage ou d'un arrêt brutal du système.

L'ext3fs est la version *ext2fs* journalisée qui a fait ses preuves, plus récemment *l'ext4fs* est disponible. Mais d'autres sont tout aussi excellents comme le *ReiserFS*, *XFS* ou le *JFS*.

Il y a un autre type de partition, plus particulier : la *SWAP*. La *SWAP* est une partition spéciale qui permet au système, lorsqu'il manque d'espace libre dans la mémoire vive, de déplacer les données peu utilisées par des programmes en cours d'exécution afin de libérer de la mémoire.

Il est important de bien penser que le type indiqué lors du partitionnement avec *cfdisk* n'est qu'indicatif. C'est lors du formatage de la partition qu'elle prendra vraiment son type. *cfdisk* ne fait que créer la table de partition, il ne touche pas aux partitions elles-mêmes.

4.2. Le formatage

On peut maintenant passer au formatage des partitions. Chaque outils de formatage possède ses propres options et paramètres.

Nous n'en verrons donc que quelques uns. Dans la plupart des cas il n'est pas nécessaire d'utiliser des options particulières, les valeurs assignées par défaut sont normalement bien choisies.

Formatage en *ReiserFS* :

```
mkfs.reiserfs PERIPHERIQUE
```

Formatage en *Ext3FS* :

```
mkfs.ext3 PERIPHERIQUE
```

Formatage en *FAT32* :

```
mkdosfs [OPTIONS] -F 32 PERIPHERIQUE
```

L'option -c permet de vérifier l'intégrité physique du disque avant de formater.

Création d'une *SWAP* :

```
mkswap [OPTIONS] PERIPHERIQUE
```

L'option -c permet de vérifier l'intégrité physique du disque avant de formater.

Activation de la *SWAP* :

```
swapon PERIPHERIQUE
```

PERIPHERIQUE est un fichier dans */dev,* par exemple */dev/sda2* pour la deuxième partition du premier disque de la machine.

Il existe aussi différents *FS* spéciaux qui peuvent se révéler extrêmement pratiques. Par exemple *tmpfs* est un système de fichiers dans lequel les fichiers créés restent en mémoire vive et ne sont pas écrits sur le disque dur (mais peuvent passer en *SWAP* si besoin est). *AuFS* et *UnionFS* permettent de fusionner plusieurs points de montage, par exemple dans le cas d'un *LiveCD,* on a un système de fichiers qui est en lecture seule et qui provient du CD, cependant il est quand même pratique de pouvoir modifier ou ajouter des fichiers pendant son utilisation. Ainsi *AuFS* permet de fusionner ce système en lecture seule avec un *TmpFS*. Résultat on peut modifier les fichiers, normalement en lecture seule, et en ajouter sans écriture sur le support.

4.3. Le montage

a. La commande mount

Examinons comment monter un périphérique manuellement.

C'est la commande *mount* qui permet de faire cela et peut s'utiliser assez simplement.

```
mount [OPTIONS] [-t TYPE] PERIPHERIQUE REPERTOIRE
```

mount peut déterminer automatiquement le *FS* d'une partition, la plupart du temps il est donc inutile d'utiliser l'option -t pour indiquer le type.

PERIPHERIQUE est le nom de fichier du périphérique à monter. Comme on l'a vu chaque périphérique possède un fichier dans */dev*. Par exemple */dev/sda1* pour la première partition du premier disque de la machine.

Le système de fichier sera monté dans le répertoire indiqué par le dernier paramètre, c'est qu'on appelle le point de montage (*mountpoint*).

Ainsi la plupart du temps on aura la commande suivant :

```
mount PERIPHERIQUE REPERTOIRE
```

mount possède une option très intéressante permettant de monter des images, comme les ISO pour les CD/DVD, au lieu d'un périphérique physique.

L'option est -o loop

Exemple :

```
mount -o loop -t iso9660 monimage.iso /mnt/tmp
```

La commande *mount* telle qu'on vient de la voir nécessite d'être super utilisateur (*root*).

On sait que c'est *mount* qui permet de monter des *FS*, on peut se demander comment fait le système au démarrage pour savoir quelles partitions il doit monter et où il doit le faire ?

b. Le fichier fstab

Il existe un fichier très important qui indique tout ça au système c'est *fstab* que l'on peut trouver dans */etc*.

Examinons comment est rempli *fstab* :

```
[thaeron@syndrome ~]$ cat /etc/fstab
/dev/sda5        swap             swap        defaults          0   0
/dev/sda1        /                reiserfs    defaults          1   1
/dev/sda3        /home            reiserfs    defaults          1   2
/dev/sdb2        /mnt/350Go       reiserfs    defaults          1   2
/dev/cdrom       /mnt/cdrom       auto        noauto,user,ro    0   0
/dev/sdc1        /mnt/clef        auto        noauto,user       0   0
/dev/fd0         /mnt/floppy      auto        noauto,owner      0   0
devpts           /dev/pts         devpts      gid=5,mode=620    0   0
proc             /proc            proc        defaults          0   0
tmpfs            /dev/shm         tmpfs       defaults          0   0
```

On reconnaît des éléments qu'on a déjà vu.

Dans la première colonne il y a le nom du périphérique à monter identifié par son fichier dans */dev*, ou le nom du *FS* spécial comme *tmpfs*.

Dans la deuxième colonne il y a le point de montage.

Dans la troisième colonne c'est le type du *FS*. On voit qu'on peut utiliser *auto* pour une détection automatique très utile dans le cas où plusieurs *FS* différents pourraient être utilisés sur le même périphérique. Par exemple on pourrait disposer de 2 clefs USB l'une en *FAT16* et l'autre en *FAT32* ainsi qu'un disque dur USB en *NTFS*.

La quatrième colonne indique les options de montage. *defaults* utilise les paramètres par défaut spécifique du FS. Par exemple pour le *Ext3FS rw*, *suid*, *dev*, *exec*, *auto*, *nouser*, *async*.

Les options communes à tous les types de systèmes de fichiers sont :

Options	**Description**
ro / rw	Montage en lecture seulement/lecture-écriture
suid / nosuid	Autorise ou interdit les opérations sur les bits suid et sgid
dev / nodev	Interprète/n'interprète pas les périphériques caractères ou les périphérique blocs spéciaux sur le système de fichiers
exec / noexec	Autorise ou interdit l'exécution de fichiers binaires sur ce système de fichiers
auto / noauto	Le système de fichiers est (c'est l'option par défaut) / n'est pas monté automatiquement
user / nouser	Permet à tout utilisateur / seulement à root (C'est le paramétrage par défaut) de monter le système de fichiers correspondant
sync / async	Selon cette valeur, toutes les entrées/sorties se feront en mode synchrone ou asynchrone
defaults	Utilise le paramétrage par défaut (c'est équivalent à rw, suid, dev, exec, auto, nouser, async)

Tableau 4: Options de *mount* dans *fstab*

Les colonnes 5 et 6 correspondent à l'archivage et à la vérification des *FS*. On ne détaillera pas car elles ne sont pas très utiles.

Grâce à ce fichier *fstab* on peut spécifier l'option *user* qui va permettre à un utilisateur *non-roo*t de pouvoir monter le périphérique. Pour que la commande *mount* consulte le fichier *fstab* afin d'opérer (ou de refuser) le montage il faut l'utiliser de cette façon :

```
mount PERIPHERIQUE
```

ou

```
mount MOUNTPOINT
```

En omettant un des 2 paramètres *mount* va aller chercher les correspondances dans le fichier *fstab*.

c. Le fichier mtab

Chaque fois qu'une partition est montée par *mount* une ligne est ajoutée dans *mtab* (localisé dans */etc*). C'est ce fichier qui recense tous les périphériques montés et est consulté par *mount* pour éviter de multiples montages.

```
root@geii-dhcp:~# cat /etc/mtab
/dev/sda1 / reiserfs rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
usbfs /proc/bus/usb usbfs rw 0 0
/dev/sdb1 /home reiserfs rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
/dev/sdc1 /mnt/backup_drive reiserfs rw 0 0
```

d. La commande umount

La commande complémentaire de *mount* est *umount* qui permet de démonter les périphériques. Il est très important de penser à démonter (ce qui est fait automatiquement lors de l'arrêt propre du système) car toutes les modifications apportées sur le *FS* sont tamponnées (*bufferisées*). Ainsi lors d'une écriture certaines données, voire toutes selon la taille de l'écriture, restent dans un tampon du noyau et ne seront écrites que lors d'une synchronisation ou d'une demande de démontage du *FS*.

umount s'utilise très simplement :

```
umount PERIPHERIQUE
```

ou

```
umount MOUNTPOINT
```

5. Les fichiers

5.1. Droits des fichiers

A chaque fichier correspond un créateur (*owner*) du fichier, un groupe (pas forcément celui du créateur), et des droits pour le créateur, le groupe et les autres.

Regardons un exemple :

```
thaeron@SquirrelMonkey:~$ ls -lah revmd5.c
-rw-r--r-- 1 thaeron users 7.2K 2009-10-30 09:22 revmd5.c
```

La première colonne correspond aux droits du fichier. Un tiret indique « rien », le premier caractère indique des attributs spéciaux comme : répertoire, lien symbolique (ou physique), *socket*, *suid*.

Ensuite ça se lit par 3 caractères :

rw- droits du owner du fichier (ici « thaeron »)
r-- droits pour le groupe (ici « users »)
r-- droits pour tous les autres.

Autre exemple :

```
thaeron@SquirrelMonkey:~$ ls -lah ns_build_module.sh
-rwx------ 1 thaeron users 5.3K 2009-11-03 22:51 ns_build_module.sh*
```

Ici on a tous les droits pour le *owner* (« thaeron ») et rien pour tous les autres même ceux qui appartiennent au même groupe.

r signifie Read : lecture permise
w signifie Write : écriture permise
x signifie eXecution : exécution permise

Ceci est une manière de représenter les droits, il en existe une autre basée sur les nombres.

Par exemple rw-r—-r-- correspond à 644 et rwx------ à 700

On se rend compte que c'est très simple :

```
x vaut 1 (0b001)
w vaut 2 (0b010)
r vaut 4 (0b100)
```

Il suffit d'additionner la valeur des droits que l'on veut mettre. Le premier chiffre du nombre correspondant au *owner*, le deuxième au groupe et le dernier aux autres comme précédemment.

On peut maintenant se poser la question : comment modifier le *owner*, le groupe et les droits ?

Il a une commande pour chaque :

chmod permet de changer les droits,
chown permet de changer le owner,
chgrp permet de changer le groupe.

```
chmod [OPTIONS] MODE FICHIER1 [FICHIER2]...
```

On peut utiliser pour MODE la valeur numérique que l'on vient de voir. Mais il existe une autre méthode, moins flexible mais bien plus simple : on peut indiquer, pour ajouter des droits, '+' suivi des lettres r,w,x ou '–' pour supprimer des droits. Cependant lorsqu'on utilise cette méthode les droits seront ajoutés pour le *owner*, le groupe et les autres.

Exemple :

```
chmod +rx monfichier
chmod -w monfichier
```

L'option *-R* permet de rendre le comportement de *chmod* récursif : il va aussi modifier les sous-répertoires et leurs contenus.

chown ne peut être utilisé qu'en *root* ou par *sudo* (si l'utilisateur fait parti des *sudoers*) et s'utilise très simplement :

```
chown [OPTIONS] LOGIN FICHIER1 [FICHIER2]...
```

FICHIERn pouvant être un fichier ou un répertoire.

L'option *-R* permet de rendre le comportement de *chown* récursif comme pour *chmod*.

5.2. Les liens symboliques et physiques

Il existe sur les *Unix-like* (dont *GNU/Linux*) un type particulier de fichier qui est extrêmement utile et utilisé : le lien. Il peut être « symbolique » (*symlink*) ou « physique » (*hardlink*).

Le lien permet de créer une sorte de raccourci qui sera vu comme étant le répertoire ou le fichier cible de façon transparente. C'est-à-dire que tout programme agissant sur un fichier ou un répertoire peut utiliser un lien à la place. Alors, à quoi cela pourrait-il servir ?

Imaginons que l'on ait un programme ou une bibliothèque et que l'on en ait besoin dans plusieurs répertoires différents. On pourrait très bien le copier. Cependant en cas de mise-à-jour il faudrait copier de nouveau. C'est là que le lien intervient. Un autre exemple plus concret, nous avons vu les fichiers de */dev* et le contenu de */etc/fstab* et nous avions cette ligne la ligne suivante :

```
/dev/cdrom      /mnt/cdrom      auto        noauto,user,ro   0   0
```

Si nous regardons les propriétés de */dev/cdrom* :

```
[thaeron@syndrome ~]$ ls -l /dev/cdrom
lrwxrwxrwx 1 root root 3 2009-03-10 09:37 /dev/cdrom -> sr0
```

Nous avions vu que la première lettre des attributs du fichier concerne leur type (répertoire / lien / socket / rien = fichier régulier).

Ici nous avons un 'l' ce qui indique que c'est un lien qui pointe sur *sr0* mais le *path* est relatif à celui du lien donc */dev/cdrom* pointe sur */dev/sr0*.

C'est un lien symbolique mis en place par le système au démarrage afin de conserver le même fichier de périphérique pour le CD-ROM quelque soit le vrai périphérique (il pourrait être USB, IDE ou SATA), ou on pourrait en avoir plusieurs.

Les liens symboliques ne sont que des sortes de raccourcis ou de pointeurs pointant sur un fichier (ou répertoire) cible. Ils ont les mêmes propriétés et droits que leur cible. Si on supprime le fichier cible le lien existe toujours mais il est alors cassé (*broken link*). Le lien symbolique n'est en rien une copie du fichier réel.

Pour créer un lien symbolique on utilise la commande *ln* avec l'option *-s*

```
ln -s CIBLE LIEN
```

Les liens physiques (*hardlink*), ou appelés parfois liens matériels, sont des copies « intelligentes ». Ils héritent des mêmes droits et propriétés que leur cible et recopie leur contenu même en cas de modifications. Toutes modifications (contenu ou droits) appliquées sur la cible sont répercutées sur le lien et toutes modifications (contenu ou droits) appliquées sur le lien sont répercutées sur la cible. C'est un véritable miroir de la cible. La suppression du lien n'affecte pas la cible et la suppression de la cible n'affecte pas le lien. Cependant on ne peut pas créer un lien physique d'un répertoire.

Pour créer un lien physique on utilise la commande *ln* sans option :

```
ln cible lien
```

5.3. Rechercher des fichiers

Pour rechercher des fichiers sous *GNU/Linux* nous avons le choix entre 2 programmes : *locate* et *find*.

a. locate

locate permet de faire une recherche très rapide car il ne recherche pas directement les fichiers mais consulte une base indexant tous les fichiers. Il faut donc préalablement créer l'index en utilisant la commande *updatedb* :

```
updatedb
```

updatedb s'utilise sans option et sans paramètre avec les droits *root*. Ensuite on utilise *locate* de la façon suivante :

```
locate [OPTIONS] MOTIF
```

La seule option utile de *locate* est *-i* qui permet de rendre la recherche insensible à la casse. *MOTIF* est une partie du nom des fichiers que l'on recherche.

Exemple :

```
[thaeron@syndrome ~]$ locate malloc
/usr/include/wine/msvcrt/malloc.h
/usr/include/malloc.h
/usr/include/jasper/jas_malloc.h
/usr/include/libguile/mallocs.h
/usr/include/libguile/debug-malloc.h
/usr/include/valgrind/pub_tool_replacemalloc.h
/usr/include/valgrind/pub_tool_mallocfree.h
```

b. find

A l'inverse de *locate*, *find* parcourt le système de fichiers pour trouver les fichiers correspondant au motif recherché. Son utilisation est un peu plus complexe et est détaillée dans la section 9.1.g. Nous ne verrons ici qu'une utilisation basique.

```
find PATH -[i]name "MOTIF"
```

PATH est le chemin de début de la recherche. On utilisera '.' pour rechercher à partir du chemin courant. L'option *-name* est sensible à la casse alors que *-iname* ne l'est pas. *MOTIF* est le motif à rechercher, contrairement à *locate* on ne peut pas juste indiquer un morceau du nom, il faut utiliser les jokers (détaillés dans la section 8.2).

Exemple :

```
[thaeron@syndrome films]$ find /usr/include -name "*malloc.h"
/usr/include/wine/msvcrt/malloc.h
/usr/include/malloc.h
/usr/include/jasper/jas_malloc.h
/usr/include/libguile/debug-malloc.h
/usr/include/valgrind/pub_tool_replacemalloc.h
```

5.4. Identifier des fichiers

Nous avons vu que, presque, toutes les commandes que nous utilisons depuis le début sont des programmes indépendants. Or nous ne tapons jamais d'extensions comme *.exe* sous *windows*. Ce n'est pas parce que le système les ajoute automatiquement, c'est tout simplement parce qu'il n'y en a pas. Sous *GNU/Linux* et les autres *Unix-like* les extensions ne sont pas nécessaires et sont juste des repères pour l'utilisateur.

La question est donc de savoir de quel type est ce fichier sans extension. A l'instar de *l'iphone* où il y a une application pour tout, nous avons une commande pour tout. Dans le cas qui nous intéresse c'est la commande *file*.

file analyse le fichier, et surtout des séquences particulières qui contiennent pour 99% des formats quelques octets spécifiques renseignant sur le type.

file prend en paramètres le ou les noms des fichiers dont on veut savoir le type :

```
file FICHIER1 [FICHIER2]...
```

Prenons le cas d'un fichier *BMP* (*windows bitmap*), son type est indiqué dans les 2 premiers octets du fichier : 0x42 0x4d ce qui donne en ASCII « BM ». Vérifions les premiers octets, ceci grâce au programme *hexdump* qui affiche le contenu du fichier en hexadécimal et en ASCII si on le lui demande.

Comme tous les programmes que nous avons vu depuis le début, *hexdump* s'utilise comme ceci :

```
hexdump [OPTIONS] FICHIER
```

Les options intéressantes d'*hexdump* :

-C : affiche en hexadécimal dans la première colonne et en ASCII dans la deuxième
-n taille_octets : ne traite que les premiers octets spécifié dans taille_octets.

Donc nous faisons :

```
[thaeron@syndrome ~]$ hexdump -C -n 16 imageplateau_2.bmp
00000000  42 4d 36 10 0e 00 00 00  00 00 36 00 00 00 28 00  |BM6.....6...(.|
00000010
```

Nous voyons bien dans les premiers octets « BM ».

Vérifions maintenant si *file* confirme ceci :

```
[thaeron@syndrome ~]$ file imageplateau_2.bmp
imageplateau_2.bmp: PC bitmap, Windows 3.x format, 640 x 480 x 24
```

En plus du type lui-même, *file* peut, suivant les formats et la version du programme, fournir d'autres informations comme ici les dimensions de l'image.

5.5. Connaître l'espace disque

Nous avons vu comment supprimer, ajouter et copier des fichiers. Et nous savons comment voir la taille d'un fichier. Cependant nous ne savons pas encore comment connaître la taille d'un répertoire.

En effet *ls* ne peut pas indiquer la place que prend un répertoire avec tout son contenu. Pour cela il existe un autre programme nommé *du* acronyme de *Disk Usage*.

du s'utilise comme les autres commandes :

```
du [OPTIONS] [REPERTOIRE]
```

Sauf que cette fois on peut omettre *REPERTOIRE*, dans ce cas c'est le *PATH* courant qui sera ciblé. *du* possède beaucoup d'options mais voyons les plus utiles :

Option	Description
-h	« human readable », comme pour *ls* les tailles sont affichées en Ko, Mo ou Go afin que ça soit lisible pour nous humains.
-s	N'affiche que la somme finale. Le comportement par défaut est d'afficher l'espace utilisé pour chaque sous-répertoire (de manière récursive).
-a	Affiche l'espace utilisé pour chaque fichier dans les répertoires et pas seulement des sous-répertoires. Évidemment cette option est incompatible avec -s.

Tableau 5: Options de *du*

On sait, maintenant, comment connaître l'espace utilisé localement. Il serait nécessaire de connaître l'espace restant sur le système. Bien sûr il y a un programme pour ça : *df*.

Son utilisation est la suivante, et est encore une fois conventionnelle :

```
df [OPTIONS]
```

Voyons ses options les plus utiles :

Option	Description
-h	« human readable », comme pour du les tailles sont affichées en Ko, Mo ou Go afin que ça soit lisible pour nous humains.
-a	Affiche tous les FS monté, même ceux non destiné au stockage.
--total	Affiche aussi l'espace utilisé total et l'espace libre total.

Tableau 6: Options de *df*

Exemples :

```
[thaeron@syndrome ~]$ df
Sys. de fich.        1K-blocs       Occupé Disponible Capacité Monté sur
/dev/root             39060816   7939072  31121744  21% /
/dev/sda3            272518332  86742580 185775752  32% /home
/dev/sdb2            341892920 260714356  81178564  77% /mnt/350Go
tmpfs                  1025084         0   1025084   0% /dev/shm
/dev/sdc1              3905516   1038484   2867032  27% /mnt/clef
```

```
[thaeron@syndrome ~]$ df -h --total
Sys. de fich.         Tail. Occ. Disp. %Occ. Monté sur
/dev/root              38G  7,6G   30G  21% /
/dev/sda3             260G   83G  178G  32% /home
/dev/sdb2             327G  249G   78G  77% /mnt/350Go
tmpfs                1002M     0 1002M   0% /dev/shm
/dev/sdc1             3,8G 1015M  2,8G  27% /mnt/clef
total                 628G  340G  288G  55%
```

```
[thaeron@syndrome ~]$ df -ah --total
Sys. de fich.         Tail. Occ. Disp. %Occ. Monté sur
/dev/root              38G  7,6G   30G  21% /
proc                     0     0     0   -  /proc
sysfs                    0     0     0   -  /sys
usbfs                    0     0     0   -  /proc/bus/usb
/dev/sda3             260G   83G  178G  32% /home
/dev/sdb2             327G  249G   78G  77% /mnt/350Go
tmpfs                1002M     0 1002M   0% /dev/shm
none                  0,0K  0,0K  0,0K   -  /proc/fs/vmblock/mountPoint
/dev/sdc1             3,8G 1015M  2,8G  27% /mnt/clef
total                 628G  340G  288G  55%
```

6. Les utilisateurs

6.1. Création de nouveaux utilisateurs et groupes

On a vu que *GNU/Linux* était un vrai système multi-utilisateurs à l'instar d'*Unix*. Il est temps de voir comment on crée des utilisateurs, des groupes et les fichiers qui y sont associés.

Pour créer un utilisateur il existe 2 commandes. La première *adduser* est très simple à utiliser puisque l'outil est interactif : il suffit de répondre aux questions posées. La seconde *useradd* est plus compliquée car il faut tout spécifier en paramètres sur la ligne de commande, cependant elle a l'avantage de permettre une utilisation automatisée via des scripts dont nous verrons la création dans la suite de la formation. Évidemment ces deux programmes nécessitent une exécution en tant que root.

Regardons tout d'abord adduser.

```
root@geii-dhcp:~# adduser

Login name for new user []: jack

User ID ('UID') [ defaults to next available ]:

Initial group [ users ]:
Additional UNIX groups:

Users can belong to additional UNIX groups on the system.
For local users using graphical desktop login managers such
as XDM/KDM, users may need to be members of additional groups
```

```
to access the full functionality of removable media devices.

* Security implications *
Please be aware that by adding users to additional groups may
potentially give access to the removable media of other users.

If you are creating a new user for remote shell access only,
users do not need to belong to any additional groups as standard,
so you may press ENTER at the next prompt.

Press ENTER to continue without adding any additional groups
Or press the UP arrow to add/select/edit additional groups
:

Home directory [ /home/jack ]

Shell [ /bin/bash ]

Expiry date (YYYY-MM-DD) []:

New account will be created as follows:

---------------------------------------
Login name.......:  jack
UID..............:  [ Next available ]
Initial group....:  users
Additional groups:  [ None ]
Home directory...:  /home/jack
Shell............:  /bin/bash
Expiry date......:  [ Never ]

This is it... if you want to bail out, hit Control-C.  Otherwise, press
ENTER to go ahead and make the account.


Creating new account...


Changing the user information for jack
Enter the new value, or press ENTER for the default
        Full Name []:
        Room Number []:
        Work Phone []:
        Home Phone []:
        Other []:
Changing password for jack
Enter the new password (minimum of 5, maximum of 127 characters)
Please use a combination of upper and lower case letters and numbers.
New password:
Re-enter new password:
Password changed.


Account setup complete.
```

Regardons les questions en détails.

-L'outil demande d'abord le *login*, c'est le nom de l'utilisateur.

-L'ID (ici UID) est un nombre associé à l'utilisateur, par exemple pour *root* c'est toujours 0. Cet ID est unique, deux utilisateurs ne peuvent avoir le même. La plupart du temps on peut laisser l'outil choisir lui-même.

-Il faut ensuite indiquer le groupe auquel appartiendra l'utilisateur, on verra ensuite comment créer un nouveau groupe. Nous avons vu cette notion de groupe dans les droits des fichiers.

-Il est possible ensuite d'ajouter l'utilisateur dans des groupes supplémentaires pour qu'il ait des droits supplémentaires bien spécifiques. Dans la plupart des cas c'est inutile.

-La question suivante c'est le répertoire *HOME* du user. Par convention c'est toujours */home/login* (sauf pour *root*). C'est dans ce répertoire que l'utilisateur pourra stocker ses données.

-Comme nous l'avons vu au début il existe plusieurs *shell,* le plus populaire et utilisé est *bash*. A moins que vous ayez une préférence particulière pour un autre *shell*, *bash* est le bon choix.

-Les comptes peuvent avoir une date de validité, c'est à ce moment là qu'on l'indique si on veut en fixer une.

-Les autres informations (Name, Room number, Work phone etc) ne sont pas importantes, elles sont même inutiles.

-La dernière question est le mot de passe, on verra ensuite comment l'enlever en cas de besoin.

A l'instar de *adduser*, *useradd* peut définir certaines informations par défaut. Ce sont les mêmes que pour *adduser*. Par exemple le groupe sera *users*, l'UID sera le prochain disponible, et le répertoire *HOME /home/login*.

Attention, car si *adduser* crée le répertoire *HOME* de l'utilisateur *useradd* ne le fait pas par défaut. Les options de *useradd* sont les suivantes et on peut ne pas les spécifier si on accepte les valeurs par défaut :

-u UID
-g GROUPE
-G GROUPE_ADDITIONNEL1[,GROUPE_ADDITIONNEL2[,...]]
-d REPERTOIRE_HOME
-s PATH_DU_SHELL
-m pour créer le répertoire HOME de l'utilisateur

La syntaxe de *useradd* est la suivante :

```
useradd [OPTIONS] LOGIN
```

Exemple :

```
useradd -g users -d /home/jack -s /bin/bash
```

Le compte créé par *useradd* est désactivé par défaut car il ne possède pas de mot de passe. Pour l'activer il faut lui ajouter un mot de passe ce qui activera le compte en même temps.

Le changement de mot de passe se fait par la commande *passwd* :

```
passwd [LOGIN]
```

Si le *login* n'est pas indiqué, le changement de mot de passe sera fait sur l'utilisateur courant. En étant *root*, on peut indiquer le *login* dont on veut mettre (ou changer) le mot de passe.

Pour supprimer un utilisateur il suffit d'utiliser la commande *userdel* :

```
userdel LOGIN
```

Examinons la création de groupe. C'est une tâche bien plus simple que la création d'un utilisateur. Le nom de la commande est explicite : *groupadd.*

La principale option est *-g GID* permettant d'affecter une valeur entière particulière au groupe.

```
groupadd [OPTIONS] GROUPE
```

La suppression est encore plus simple :

```
groupdel GROUPE
```

6.2. Les fichiers associés

Examinons ce qui est sous la surface : les fichiers liés aux groupes et aux utilisateurs.

Fichier	**Contenu**
/etc/passwd	Informations sur les comptes utilisateurs
/etc/shadow	Informations sécurisées sur les comptes utilisateurs
/etc/group	Informations sur les groupes

Tableau 7: Fichiers contenant les informations sur les utilisateurs et groupes

/etc/passwd est formaté de la manière suivante :

```
login:mdp_actif:UID:GID:info_utilisateur:path_home:path_shell
```

On retrouve la plupart des informations que l'on a vu lors de la création des utilisateurs.

Champ	Description
login	le nom de l'utilisateur
mdp_actif	x s'il y a un mot de passe, rien sinon
UID	identifiant (nombre entier) de l'utilisateur
GID	identifiant (nombre entier) auquel l'utilisateur appartient
info_utilisateur	les informations comme le vrai nom, numéro de téléphone, le rôle etc.
path_home	chemin du répertoire HOME de l'utilisateur
path_shell	chemin du shell de l'utilisateur

Tableau 8: Syntaxe de *[/etc/passwd](#)*

Exemples :

```
root:x:0:0::/root:/bin/bash
thaeron:x:1000:100:,,,:/home/thaeron:/bin/bash
apache:x:80:80:User for Apache:/srv/httpd:/bin/false
```

Petite explication sur le dernier. Apache est un serveur HTTP (web) libre et comme de nombreux logiciels orientés réseaux il nécessite de fonctionner via un compte créé spécialement. Ainsi en cas de piratage le pirate n'aura que les droits très limité du serveur. De plus, étant donné que ça n'est pas un vrai utilisateur il n'a pas besoin de *shell*, ainsi on spécifie */bin/false* qui est un programme qui ne fait rien et renvoie une erreur. On remarque que l'utilisateur est quand même censé être protégé par un mot de passe.

Le fichier */etc/shadow* contient des données plus sensibles comme les mots de passes (chiffrés).

Exemples :

```
apache:*:9797:0:::::
root:$1$F7K18K1T$JXBh90hE/uG7/4SCvC0hK/:13686:0:::::
```

Le premier champ est le *login* d'utilisateur. Le deuxième champ est le mot de passe chiffré. Le troisième champ est le nombre de jours écoulés depuis le 1er janvier 1970. Le reste des champs correspond aux jours d'expiration, demande changement de mot de passe, obligation de changement de mot de passe etc.

Si par exemple on enlève le mot de passe de *root* et que l'on sauvegarde ainsi le fichier on pourra devenir super-utilisateur sans qu'aucun mot de passe ne soit demandé. On comprend alors pourquoi il y a une étoile en tant que mot de passe pour l'utilisateur apache : si on essayait d'utiliser ce compte le mot de passe serait toujours refusé.

Le formatage de */etc/group* est encore plus simple et est comme ceci :

```
nom_groupe:mdp_actif:GID:autre_groupe1,autre_groupe2,...
```

Champ	Description
nom_groupe	explicite c'est le nom du groupe
mpd_actif	contrairement à */etc/passwd* un x indique qu'il n'y en a pas (et il n'y en a jamais)
GID	l'identifiant (nombre entier) du groupe
autre_groupe1,autre_groupe2,...	nom des autres groupes qui en font parti, par exemple *root* est quasiment présent dans tous les autres.

Tableau 9: Syntaxe de */etc/group*

6.3. Changement d'utilisateur, de privilèges

Pour lire ces fichiers, pour créer des utilisateurs, etc il faut être *root* (super-utilisateur). Il est donc temps de voir comment faire.

a. Commande su

Il y a une seule commande qui permet de devenir *root* et de changer d'utilisateur c'est *su*.

Si on appelle *su* sans paramètre c'est *root* qui sera choisi pour le changement. Dans le cas où on voudrait changer pour un autre il suffit d'indiquer le *login* en paramètre à su :

```
su [LOGIN] [OPTIONS]
```

Lorsqu'on est utilisateur *su* demande le mot de passe. Ce qui n'est pas le cas lorsqu'on est *root*.

b. Commande sudo

Il existe une autre méthode, utilisée par plusieurs distributions dont *ubuntu*, qui ne nécessite plus d'utiliser un compte *root* mais permet de donner des droits de super-utilisateur temporairement à un utilisateur. Ce programme est *sudo*.

Il faut mettre la commande que l'on veut exécuter avec des droits supérieurs en paramètres de *sudo*.

Exemple :

```
sudo cat /etc/shadow
```

En réalité *sudo* est un peu plus puissant que ça puisqu'il permet d'exécuter des commandes, pas seulement en *root* mais, avec n'importe quel utilisateur en utilisant son propre mot de passe, ou aucun suivant la configuration.

Encore une fois la configuration est dans un fichier texte, lisible et modifiable par un humain. Ce fichier est */etc/sudoers.*

Regardons comment le configurer. Tout d'abord il faut savoir que *sudo* dispose d'un délai d'expiration (*timeout*), c'est-à-dire que la première fois qu'on utilise *sudo* il demande le mot de passe mais ensuite pendant un délai défini il ne le demandera plus.

Donc la première chose à faire et de configurer ce timeout :

```
Defaults:ALL timestamp_timeout=N
```

où N est le temps en minute avant l'expiration. Si N=0 le mot de passe sera toujours demandé.

Ensuite chaque ligne est construite selon ce formatage :

```
entité endroit=(utilisateur_cible)option:commande1,option:commande2,...
```

entité peut être un utilisateur ou un groupe, dans le cas du groupe il est précédé par %
endroit est l'adresse d'où est connecté l'utilisateur (car on peut se connecter à distance avec ssh).

Comme avec *su*, *sudo* permet d'utiliser n'importe quel utilisateur, on peut limiter cette possibilité en renseignant *utilisateur_cible*.

Option peut être *NOPASSWD* pour qu'aucun mot de passe ne soit demandé. Sinon il ne faut rien mettre.

commande1,*commande2*,... ce sont les commandes qui pourront être exécutées, les autres seront rejetées.

Pour chaque champ on peut indiquer *ALL* pour tout autoriser.

Exemples :

```
ALL ALL=(ALL) ALL
```

On autorise tout de tout le monde, où qu'ils soient connectés, à tout exécuter en spécifiant son mot de passe en tant que n'importe quel utilisateur.

```
one ALL=(root) ALL,NOPASSWD:/usr/bin/cat
```

On autorise l'utilisateur *one*, où qu'il soit connecté, à faire toutes les commandes en *root* en spécifiant son mot de passe et à faire *cat* sans qu'il doive spécifier son mot de passe. On comprend qu'une mauvaise configuration peut entraîner un gros trou de sécurité.

c. Qui suis-je ? Commandes whoami, id

Avec toutes ces commandes de changement d'utilisateur, comment savoir qui on est ? Il y a 2 commandes permettant d'avoir des informations sur l'utilisateur courant : *whoami* et *id*.

whoami est trivial et affiche le nom de l'utilisateur courant. *id* est plus complet et affiche toutes les informations de l'utilisateur comme son nom, son *UID*, *GID* et les groupes additionnels auxquels il appartient.

Exemples :

```
thaeron@SquirrelMonkey:~$ whoami
thaeron
thaeron@SquirrelMonkey:~$ id
uid=1000(thaeron) gid=100(users) groups=100(users)
```

7. Processus et signaux

7.1. Processus

Nous avons besoin de connaître quelques éléments sur les processus. Un processus ce n'est rien d'autre qu'un programme en train de s'exécuter. Plus exactement c'est une tâche d'un programme. Un programme exécuté peut créer des processus légers (*threads*) qui ne sont pas des processus à part entière, ou se dupliquer par *fork* qui recopie le processus on obtient alors, ce qu'on appelle, un processus père et un processus fils. Il existe plusieurs programmes permettant d'afficher les processus en cours sur le système. La plus connue et la plus utilisée *ps*.

ps s'utilise comme cela :

```
ps [OPTIONS]
```

Cependant, pour une fois, les options de *ps* sont plus compliquées. On a comme d'habitude des options commençant par un tiret et d'autres, ayant le même nom, qui n'ont pas de tiret. Le comportement par défaut de *ps* (sans option) n'est pas très utile et n'affiche pas grand chose. Voyons ses options les plus intéressantes.

Option	**Description**
-A	Affiche tous les processus.
-L	Affiche aussi les threads.
u	Affiche les informations dans un format utile pour l'utilisateur (avec consommation de CPU et de mémoire).
f	Affiche sous forme d'arbre.

Tableau 10: Options de *ps*

Il existe une autre forme donnant un résultat comparable (voire identique) à -A u, et qui est assez répandue dans les tutoriaux c'est : aux.

L'affichage sous forme d'arbre permet de visualiser la notion de processus père fils. Par exemple en utilisant *ps -A f*

```
27063 ?        S      0:00 xterm -fg green -bg black
27065 pts/2    Ss     0:00  \_ bash
27087 pts/2    S+     0:00      \_ /bin/sh /usr/bin/soffice formation_linux.odt
27104 pts/2    Sl+    3:15          \_ /etc/openoffice.org3/program/soffice.bin
formation_linux.odt
```

On voit que le processus père est *xterm* (le terminal), qu'il est le père de *bash* et que *bash* a exécuté */bin/sh* (qui est un lien symbolique vers *bash*) pour exécuter le programme */usr/bin/soffice* (qui est un lien symbolique vers un script *shell*) et donc ce script exécute */etc/openoffice.org3/program/soffice.bin*

Petit détail :

```
[thaeron@syndrome ~]$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 2009-11-12 17:09 /bin/sh -> bash
```

C'est bien un lien symbolique vers */bin/bash*

Ou on aurait pu faire :

```
[thaeron@syndrome ~]$ file /bin/sh
/bin/sh: symbolic link to `bash'
[thaeron@syndrome ~]$ file /usr/bin/soffice
/usr/bin/soffice: symbolic link to `/etc/openoffice.org3/program/soffice'
```

C'est donc aussi un lien symbolique.

```
[thaeron@syndrome ~]$ file /etc/openoffice.org3/program/soffice
/etc/openoffice.org3/program/soffice: POSIX shell script text executable
```

Et lui c'est bien un script *shell*.

Il n'y a donc aucun mystère, tout est cohérent. Penchons nous un peu plus sur les informations que fournit ps.

```
[thaeron@syndrome ~]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
thaeron  28932  0.0  0.1  58488  3524 tty1     S    10:50   0:00 xterm
```

Colonne	**Description**
USER	Utilisateur propriétaire du processus.
PID	Processus ID, c'est l'identifiant du processus.
%CPU	Le taux d'utilisation du CPU du processus au moment de la commande.
%MEM	Le taux de consommation de RAM du processus au moment de la commande.
VSZ	Une taille, assez incompréhensible, utilisée par le processus.
RSS	Comme VSZ
TTY	Peu important.
STAT	État du processus (voir tableau suivant).
START	Temps ou date de lancement du processus, ici 10min et 50s.
TIME	Temps d'utilisation total du CPU.
COMMAND	Nom du programme auquel correspond le processus.

Tableau 11: Informations de *ps*

L'état du processus est décrit par des lettres dont voici la correspondance :

État	**Description**
D	En sommeil qu'on ne peut pas interrompre.
R	En fonctionnement normal.
S	En attente d'un évènement.
T	Stoppé.
Z	Zombie, le processus est terminé en attente du processus parent.
<	Le processus est en haute priorité d'exécution.
N	Le processus est en basse priorité d'exécution.
s	« session leader », ce n'est pas important.
l	Le processus possède des threads (multi-threadés).
+	Le processus est en avant-plan (foreground).

Tableau 12: État des processus

Nous avons vu que *ps* permettait d'afficher l'utilisation au moment de la commande de l'utilisation mémoire et CPU d'un processus. Cependant on ne peut pas la suivre en temps réel à moins de répéter plusieurs fois rapidement la commande. Il existe bien sûr un programme qui permet de *monitorer* l'utilisation de ces ressources en classant les processus les plus gourmands en premier. Ce programme est *top*. On peut l'utiliser directement, sans option ni paramètre.

Une fois *top* lancé nous sommes confronté à un problème : comment arrête-t-on *top* sans devoir fermer le terminal ?

Pour cela on utilise la combinaison de touche *CTRL + C*. Cette combinaison n'est pas spécifique à *top*, elle envoie un signal au processus lui demandant de s'arrêter.

Nous allons voir ça en détail.

7.2. Les signaux

Un signal est un ordre, identifié par une valeur entière, envoyé au processus qui a obligation de le traiter ou de l'ignorer. Pour détailler un peu : le programme doit renseigner que telle ou telle fonction correspond à un signal. Lorsque le signal est émis l'exécution du programme est suspendu et c'est la fonction enregistrée correspondant au signal qui est exécutée afin de traiter le signal.

Voyons les principaux signaux :

Signal	**Valeur**	**Action**
SIGINT	2	Demande d'interruption (CTRL + C)
SIGQUIT	3	Arrêt du processus.
SIGKILL	9	Suppression du processus.
SIGSEGV	11	Erreur de segmentation (accès mémoire invalide)
SIGTERM	15	Demande de fermeture du processus.
SIGHUP	1	Terminal déconnecté, souvent utilisé pour demander au processus de se relancer.
SIGFPE	8	Erreur de virgule flottante (souvent division par 0)
SIGILL	4	Instruction illégale.
SIGALRM	14	Temporisation écoulée.
SIGPIPE	13	Utilisation d'un « tube » cassé.
SIGTSTP		Blocage de l'exécution du processus. (CTRL + Z)
SIGCONT		Reprise de l'exécution du processus.

Tableau 13: Principaux signaux

Si certains signaux peuvent être interceptés et traités par le processus, *SIGKILL* ne peut pas l'être et provoque la mort du processus. Pour fermer proprement un processus il vaut mieux utiliser les signaux *SIGTERM* ou *SIGINT*.

Lançons un programme graphique depuis le terminal. Essayons *mousepad* qui est un bloc note comme celui de *windows*. On se rend compte que l'on a plus la main sur le terminal. Pour pouvoir taper d'autres commandes il faudrait que *mousepad* se termine. Ou...

Essayons CTRL+Z, que se passe-t-il ?

Nous voyons : [1]+ Stopped   mousepad

Si nous cherchons mousepad dans les processus actifs nous trouvons :

```
[thaeron@syndrome ~]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
…
thaeron  32323  0.0  0.8 129112 18332 pts/0    T    12:45   0:00 mousepad
```

Nous avons vu que *T* dans la colonne *STAT* signifie que le processus a son exécution bloquée. Si nous cliquons sur la fenêtre de *mousepad* on ne peut plus rien écrire dedans, elle semble « plantée ».

Nous avons 3 possibilité de le relancer : en envoyant le signal *SIGCONT* pour qu'il reprenne son exécution, et 2 commandes *fg* et *bg* qui respectivement relance l'exécution en avant-plant (*fg* = abréviation de *foreground*) ou en arrière-plan (tâche de fond, *bg* = abréviation de *background*).

Essayons *fg*, nous constatons que le processus reprend son exécution mais nous n'avons toujours pas la main sur le terminal.

Si nous refaisons *CTRL+Z* et que cette fois nous tapons *bg* nous constatons que le processus a repris son exécution mais que cette fois le terminal est de nouveau disponible. Le processus tourne maintenant en arrière-plan.

Si nous cherchons de nouveau *mousepad* dans les résultats de ps :

```
[thaeron@syndrome ~]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
…
thaeron  32323  0.0  0.9 129252 18504 pts/0    S    12:45   0:00 mousepad
```

Son état est '*S*' (en attente d'un évènement) il n'est donc plus bloqué et on remarque qu'il n'a pas de '+' indiquant que le processus est en avant-plan.

Nous avons vu qu'on pouvait lui envoyer un signal pour qu'il reprenne son exécution. Pour envoyer un signal directement à un processus il faut utiliser la commande *kill* qui, comme son nom ne l'indique pas, ne sert pas seulement à tuer des processus.

Pour utiliser *kill* :

```
kill [-SIGNAL] PID1 [PID2]…
```

Si on n'indique pas le signal, par défaut c'est le signal *SIGTERM* qui est envoyé.

Donc pour relancer *mousepad* nous pouvons faire :

```
kill -SIGCONT PID_de_mousepad
```

PID_de_mousepad est 32323 dans l'exemple précédent, on le trouve dans la deuxième colonne de ce qu'affiche *ps*.

Mais il existe bien sûr une méthode plus facile pour avoir le *PID* d'un processus. La commande *pidof* récupère le *PID* à partir du nom du programme.

Exemple :

```
[thaeron@syndrome ~]$ pidof mousepad
32323
```

Et il existe un programme regroupant *kill* et *pidof* en une seule commande : *killall*. Il permet d'envoyer un signal à un processus à partir du nom du programme. Attention car si le même programme est lancé plusieurs fois, ils recevront tous le signal.

killall s'utilise comme *kill* sauf qu'il faut spécifier le nom du programme au lieu du *PID*.

Exemple :

```
killall -SIGCONT mousepad
```

Il est possible d'exécuter directement un programme en *background* en terminant la commande par un et commercial (&).

Exemple :

```
[thaeron@syndrome ~]$ mousepad &
[thaeron@syndrome ~]$
```

Nous reprenons directement la main sur le *shell*. Cependant la sortie standard du programme, ainsi que sa sortie d'erreur, se feront toujours dans le terminal.

8. A la conquête du shell

Avant de pouvoir s'attaquer à l'écriture de scripts *shell*, il nous reste à acquérir une plus grande maîtrise du *shell*.

8.1. Les variables d'environnement

Les variables d'environnement sont des variables dynamiques (qui peuvent changer de valeurs) et permettant aux divers processus de communiquer entre eux surtout concernant la configuration. La commande *env* permet d'afficher toutes les variables qui sont enregistrées pour l'utilisateur courant.

Voici quelques exemples :

```
PWD=/home/thaeron
LANG=fr_FR
PATH=/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/lib64/java/bin:/usr/lib
64/kde4/libexec:/opt/kde3/lib64/qt3/bin:/opt/kde3/bin:/usr/lib64/qt/bin:/
usr/share/texmf/bin:.:/sbin:/usr/brlcad/bin
```

Nous avons vu la commande *pwd* qui permettait de connaître le *PATH* courant. La variable *PWD* contient la même chose. La variable *LANG* est utilisée par le système pour changer la langue des messages des programmes. La variable *PATH* est très importante, c'est cette variable qui indique au *shell* où il doit aller chercher les programmes dont on tape le nom. Lorsqu'on appelle le programme *env* par exemple, le *shell* va d'abord chercher dans */usr/local/bin* s'il n'y est pas il va chercher dans */usr/bin* etc. L'ordre est donc important.

Il est possible de voir le contenu d'une variable en particulier. Pour cela on utilise la commande *echo*. Elle permet d'afficher du texte et donc des variables car lorsque le *shell* va interpréter la commande il va remplacer chaque variable par son contenu, si elle n'existe pas elle sera remplacée par une chaîne vide.

Pour différencier le texte PWD de la variable *PWD* on précède le nom de la variable du symbole *$*. Ceci ne s'applique que pour le contenu de la variable, pour une affectation il ne faut pas l'indiquer.

Exemple :

```
echo le PATH courant est : $PWD
```

Pour changer une variable d'environnement ou en ajouter une il faut utiliser la commande export pour *bash*, et souvent set pour les autres *shells*.

Exemple :

```
export MAVAR=contenu
```

Maintenant la variable *MAVAR* est affichée par *env*.

Regardons l'effet de la variable *LANG*.

Si nous appelons le programme *cat* en lui donnant en paramètre un fichier qui n'existe pas :

```
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas
cat: fichier_qui_nexiste_pas: Aucun fichier ou dossier de ce type
[thaeron@syndrome ~]$ export LANG=en_US
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas
cat: fichier_qui_nexiste_pas: No such file or directory
```

8.2. Les jokers

Nous avons plusieurs commandes permettant de traiter les fichiers mais nous n'avons pas encore été dans le cas où nous devons faire un traitement de lot. Par exemple si nous voulons supprimer tous les fichiers ayant l'extension txt. On a recours, pour cela, aux jokers.

Joker	**Description**
*	Remplace toute une chaîne
?	Remplace un seul caractère
{p1,p2,p3,...}	Ensemble de chaînes possibles

Tableau 14: Principaux jokers

Nous voulons supprimer tous les fichiers ayant l'extension .txt :

```
[thaeron@syndrome ~]$ rm *.txt
```

Par exemple, nous avons un répertoire contenant les fichiers suivants :

```
[thaeron@syndrome b]$ ls
b1_k1.png  b1_t1.png  b1_t3.png  b2_t1.png  b2_t3.png  b3_t2.png
b1_k2.png  b1_t2.png  b2_k2.png  b2_t2.png  b3_t1.png  b3_t3.png
```

Nous voulons traiter uniquement les fichiers commençant par b1 et b3 et contenant t1, t2, k1 et k2. Nous pouvons faire :

```
[thaeron@syndrome b]$ ls b{1,3}_?{1,2}.png
b1_k1.png  b1_k2.png  b1_t1.png  b1_t2.png  b3_t1.png  b3_t2.png
```

8.3. Les redirections de flux

Nous voyons, depuis le début, des programmes afficher du texte dans le terminal. En réalité, les programmes envoient les données dans le flux *stdout* (contraction de *standard output*) qui est le flux standard de sortie (affichage), ou encore *stderr* (*standard erro*r) qui est spécifique aux erreurs. Il y a un troisième flux : *stdin* (*standard input*) qui correspond à ce que nous tapons au clavier.

Cette notion de flux est importante car il nous est possible de les rediriger. Il existe pour ça 3 opérateurs.

Op	Description
>	Redirige la sortie vers un fichier
<	Envoi le contenu d'un fichier sur l'entrée standard
|	Redirige la sortie d'un programme vers l'entrée standard d'un autre programme.

Tableau 15: Opérateurs de redirection

Il existe un variante pour la redirection vers un fichier. Si on utilise > le contenu est envoyé dans un fichier qui est créé s'il n'existe pas et est écrasé dans le cas contraire. L'opérateur >> redirige vers un fichier qui est créé s'il n'existe pas et ajoute à la fin du fichier dans le cas contraire.

Regardons quelques exemples :

```
[thaeron@syndrome ~]$ pwd > tmp.txt
[thaeron@syndrome ~]$ cat tmp.txt
/home/thaeron
[thaeron@syndrome ~]$ pwd > tmp.txt
[thaeron@syndrome ~]$ cat tmp.txt
/home/thaeron
[thaeron@syndrome ~]$ pwd >> tmp.txt
[thaeron@syndrome ~]$ cat tmp.txt
/home/thaeron
/home/thaeron
```

La sortie standard de *pwd* est redirigée vers le fichier tmp.txt.

```
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas > tmp.txt
cat: fichier_qui_nexiste_pas: Aucun fichier ou dossier de ce type
```

Le message d'erreur n'est pas redirigé dans le fichier car l'opérateur > redirige *stdout* par défaut et les erreurs sont sur *stderr*.

Il faut savoir que chaque flux a une valeur particulière : 1 pour *stdout*, 2 pour *stderr* et 0 pour *stdin*. En utilisant ces valeurs nous pouvons rediriger *stderr* :

```
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas 2> tmp.txt
[thaeron@syndrome ~]$ cat tmp.txt
cat: fichier_qui_nexiste_pas: Aucun fichier ou dossier de ce type
```

Nous pouvons aussi rediriger *stderr* vers *stdout* :

```
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas 2>&1
cat: fichier_qui_nexiste_pas: Aucun fichier ou dossier de ce type
```

Et rediriger l'ensemble vers un fichier :

```
[thaeron@syndrome ~]$ cat fichier_qui_nexiste_pas >tmp.txt 2>&1
```

L'ordre de redirection est important, si on avait placé *2>&1* avant *>tmp.txt* cela n'aurait pas fonctionné.

Pour comprendre l'utilisation de l'opérateur | (tube en français, *pipe* en anglais) il faut comprendre que beaucoup de commandes peuvent traiter les données que l'on indique en paramètres mais aussi sur leur entrée standard. Prenons l'exemple de *cat*, nous avons vu qu'il pouvait afficher le contenu d'un fichier. Mais il peut aussi répéter ce qu'il reçoit sur son entrée standard.

Exemple :

```
[thaeron@syndrome ~]$ cat
ceci est un
ceci est un
exemple
exemple
```

en gras nous avons ce qui a été tapé au clavier. Pour arrêter *cat* il faut lui envoyer un *EOF* (E*nd Of File*) via la combinaison de touches *CTRL+D*.

On ne voit pas clairement l'utilité de ce comportement. Alors regardons une pratique souvent utilisée pour créer un fichier texte :

```
[thaeron@syndrome ~]$ cat > tmp.txt
ceci est un
exemple
[thaeron@syndrome ~]$ cat tmp.txt
ceci est un
exemple
```

Ce qui nous permet de voir un exemple d'utilisation de l'opérateur '<' :

```
[thaeron@syndrome ~]$ cat < tmp.txt
ceci est un
exemple
```

On peut alors contracter les commandes *cat tmp.txt* et *cat < tmp.txt* en utilisant *pipe* :

```
[thaeron@syndrome ~]$ cat tmp.txt | cat
ceci est un
exemple
```

Évidemment ici ça n'a pas beaucoup d'intérêt mais nous allons voir des programmes très utiles dans le traitement des fichiers et qui s'utilisent dans la majorité des cas avec un *pipe*.

9. Les commandes principales

Commande	Description
awk	Langage d'examen et de traitement de motifs (*patterns*)
basename	Supprime le *PATH* et le suffixe d'un nom de fichier
cat	Affiche la concaténation de fichiers
cut	Supprime des morceaux de chaque ligne d'un fichier
dirname	Ne laisse que le *PATH* d'un nom de fichier
echo	Affiche une ligne de texte
find	Recherche de fichiers dans l'arborescence
grep	Affiche les lignes qui correspondent à un motif
head	Affiche la première partie d'un fichier
less	Affiche interactivement un fichier par morceau
more	Similaire à *less*
sed	Éditeur de flux pour filtrer et transformer le texte
sort	Trie les lignes d'un fichier
tail	Affiche la dernière partie d'un fichier
tr	Remplace ou supprime des caractères
uniq	Affiche ou supprime des répétitions de lignes
wc	Compte les lignes, mots ou caractères d'un fichier
which	Affiche le *PATH* complet d'une commande
xargs	Passe son entrée standard en arguments à une commande

Tableau 16: Principales commandes *Unix*

9.1. Détails de ces principales commandes

Ce qui suit n'est qu'un aperçu des possibilités qu'offrent ces programmes. Seules les options les plus utiles sont référencées. Pour voir en détail tout ce que permet de faire une commande il faut consulter les pages de manuel :

```
man COMMANDE
```

a. awk : Langage d'examen et de traitement de motifs (patterns)

awk est un outils qui dispose de son propre langage. Il est possible de créer un fichier contenant les commandes mais la plupart du temps il est plus intéressant de passer les instructions directement en paramètres.

awk s'utilise ainsi :

```
awk '{instructions}'
```

L'important est qu'*awk* sépare les « mots » qu'il doit traiter. Par exemple le premier mot sera identifié par $1 et le mot 5 par $5. L'instruction la plus utilisée de *awk* est *print*.

Exemple :

```
[thaeron@syndrome ~]$ echo ceci est un exemple | awk '{print $1 $3}'
ceciun
```

Si on veut placer un espace entre :

```
[thaeron@syndrome ~]$ echo ceci est un exemple | awk '{print $1 " " $3}'
ceci un
```

b. basename : Enlève le répertoire et le suffixe d'un nom de fichier.

```
basename NOM [SUFFIXE]
```

Exemples :

```
thaeron@geii-labo5:~$ basename /usr/include/stdio.h
stdio.h
thaeron@geii-labo5:~$ basename /usr/include/stdio.h .h
stdio
```

c. cat : Concaténer des fichiers et les afficher sur la sortie standard

```
cat [OPTION] [FICHIER]...
```

options :
-n : numérote chaque ligne
-b : numérote les lignes non vide
-s : supprime les lignes vides répétées

Exemples :

```
thaeron@geii-labo5:~$ cat test.txt
ceci est un fichier inutile
pour montrer les commandes


Enjoy !
thaeron@geii-labo5:~$ cat -sn test.txt
     1  ceci est un fichier inutile
     2  pour montrer les commandes
     3
     4  Enjoy !
```

Utilisation particulière : si on ne précise pas de fichier, *cat* recopie l'entrée standard. Le programme s'arrête lors de la saisie de *CTRL+D*.

d. cut : Supprime des morceaux de chaque ligne d'un fichier

```
cut [OPTIONS] [FICHIER]...
```

options :

-b LIST : ne coupe que les caractères de cette liste

Exemple :

```
[thaeron@syndrome ~]$ echo exemple | cut -b 1,4,5
emp
```

e. dirname : Ne conserve que la partie répertoire d'un chemin d'accès.

```
dirname NOM
```

Exemples :

```
thaeron@geii-labo5:~$ dirname /usr/include/stdio.h
/usr/include
thaeron@geii-labo5:~$ dirname /usr/include/
/usr
thaeron@geii-labo5:~$ dirname stdio.h
.
```

f. echo : Affiche une ligne de texte

```
echo [OPTIONS] CHAINE
```

options :

-n : n'affiche pas de retour à la ligne à la fin de la ligne.

g. find : Recherche de fichiers dans l'arborescence

```
find [OPTIONS] [PATH] [EXPRESSION]
```

options :

-P : ne pas suivre les liens symboliques
-L : suivre les liens symboliques

PATH : répertoire de départ pour la recherche

EXPRESSION est aussi constitué par des options :
-maxdepth N : ne descend pas dans plus de N sous-répertoires
-name MOTIF : recherche les fichiers correspondant au motif
-iname MOTIF : comme *-name* mais non sensible à la casse (différence minuscule / majuscule)
-executable : ne cherche que les exécutables
-readable : ne cherche que les fichiers dont on a le droit de lecture
-writable : ne cherche que les fichiers dont on a le droit d'écriture

Exemples :

```
[thaeron@synapse ~]$ find ~/tmp -name "te*.sh"
/home/thaeron/tmp/test2/test.sh
/home/thaeron/tmp/test.sh
```

```
[thaeron@synapse ~]$ find ~/tmp -iname "te*.sh" -readable
/home/thaeron/tmp/test2/test.sh
/home/thaeron/tmp/test.sh
/home/thaeron/tmp/TeSt.sh
```

h. grep : Affiche les lignes qui correspondent à un motif

grep est surement le programme le plus utile et le plus utilisé, il permet de filtrer un fichier en n'affichant que les lignes qui correspondent (ou que celles qui ne correspondent pas) à un motif.

```
grep [OPTIONS] MOTIF [FICHIER]...
```

options :

-e MOTIF : utilise MOTIF comme motif, cette option est utile pour spécifier plusieurs motifs
-i : ignore la casse autant pour le fichier que pour le motif
-v : inverse le sens de correspondance, pour n'afficher que les lignes qui ne correspondent pas au motif
-o : n'affiche que la partie qui correspond au motif au lieu de la ligne complète
-n : affiche le numéro de la ligne qui correspond
-c : affiche uniquement le nombre de lignes qui correspondent

Exemples :

```
[thaeron@syndrome ~]$ cat test.txt | grep -n es
1:ceci est un fichier inutile
2:pour montrer les commandes
```

```
[thaeron@syndrome ~]$ cat test.txt | grep -nv es
3:
4:
5:Enjoy !
```

i. head : Affiche la première partie d'un fichier

```
head [OPTIONS] [FICHIER]...
```

options :

-c [-]N : affiche les N premiers octets de chaque fichier, si '-' est spécifié *head* affiche tout sauf les N derniers octets de chaque fichier
-n [-]N : affiche les N premières lignes de chaque fichier, si '-' est spécifié *head* affiche tout sauf les N dernières lignes de chaque fichier

Si on ne spécifie aucune option *head* affiche les 10 premières lignes du fichier.

Exemples :

```
[thaeron@syndrome ~]$ cat test.txt | head -n 2
ceci est un fichier inutile
pour montrer les commandes
[thaeron@syndrome ~]$ cat test.txt | head -n -4
ceci est un fichier inutile
[thaeron@syndrome ~]$ cat test.txt | head -c 4
ceci
```

j. less : Permet un affichage interactif par morceau d'un long fichier.

```
less [FICHIER]
```

Si aucun fichier n'est spécifié, *less* prendra en compte les données sur son entrée standard.

On peut se déplacer, descendre ou remonter dans le fichier en utilisant les flèches. Ce programme est orienté utilisateur et n'est pas utile dans une commande automatisée comme un script.

k. more : Permet un affichage interactif par morceau d'un long fichier.

Similaire à *less*. Sauf qu'on se déplace en appuyant sur entrée.

l. sed : Éditeur de flux pour filtrer et transformer le texte

```
sed -e EXPRESSION [FICHIER]...
```

sed permet de faire beaucoup de choses, comme *awk*. Cependant on l'utilise principalement pour faire du remplacement d'expression.

Le remplacement se fait ainsi :

's/expression/remplacement/g'

Exemple :

```
[thaeron@syndrome ~]$ echo ceci est un exemple | sed -e 's/un exemple/une
demonstration/g'
ceci est une demonstration
```

m. sort : Trie les lignes d'un fichier

```
sort [OPTIONS] [FICHIER]...
```

options :

-b : ignore les espaces qui précédent le texte
-f : ignore la casse
-R : trie de façon aléatoire
-r : inverse le résultat du trie
-u : n'affiche pas les doublons

Exemples :

```
[thaeron@syndrome ~]$ cat tmp.txt | sort
  souris
chat
chat
dentier
```

```
[thaeron@syndrome ~]$ cat tmp.txt | sort -bu
chat
dentier
  souris
```

n. *tail : Affiche la dernière partie d'un fichier*

Ce programme et le complément de *head*, il a les mêmes fonctionnalités mais en utilisant la fin du fichier comme référentiel.

```
tail [OPTIONS] [FICHIER]...
```

options :

-c[+]N : affiche les N derniers octets, si '+' est spécifié *tail* affiche les tous les octets a partir du Nième octet.
-n[+]N : affiche les N dernières lignes, si '+' est spécifié *tail* affiche les toutes les lignes a partir de la Nième ligne.

Si aucune option n'est spécifiée, *tail* affiche les 10 dernières lignes du fichier.

Exemples :

```
[thaeron@syndrome ~]$ cat test.txt | tail -c+2
eci est un fichier inutile
pour montrer les commandes


Enjoy !
```

```
[thaeron@syndrome ~]$ cat test.txt | tail -c2
!
```

```
[thaeron@syndrome ~]$ cat test.txt | tail -n4
pour montrer les commandes


Enjoy !
```

```
[thaeron@syndrome ~]$ cat test.txt | tail -n+5
Enjoy !
```

o. *tr : Remplace ou supprime des caractères*

```
tr [OPTIONS] SET1 [SET2]
```

options :

-d : supprime les caractères contenu dans *SET1*, pas de remplacements

Exemples :

```
[thaeron@syndrome ~]$ echo test | tr 'et' 'ij'
jisj
```

```
[thaeron@syndrome ~]$ echo test | tr -d 'et'
s
```

Le remplacement se fait caractère par caractère. C'est-à-dire que le premier caractère du *SET1* sera remplacé par le premier caractère du *SET2*, le deuxième caractère du *SET1* sera remplacé par le deuxième caractère du *SET2* etc.

p. uniq : Affiche ou supprime des répétitions de lignes

Attention : *uniq* supprime uniquement les répétitions de lignes consécutives. Si des lignes identiques sont éparpillées dans le fichier, *uniq* ne les traitera pas.

```
uniq [OPTIONS] [FICHIER]...
```

options :

-c : préfixe les lignes par leur nombre d'occurrences
-d : affiche uniquement les lignes répétées
-i : ignore la case
-u : affiche uniquement les lignes uniques

Le comportement par défaut est d'afficher le fichier en supprimant les répétitions (mais en laissant la première occurrence contrairement à l'option **-***u*).

Exemples :

```
[thaeron@syndrome ~]$ cat tmp.txt
chat
chat
dentier
chat
dentier
```

```
[thaeron@syndrome ~]$ cat tmp.txt | uniq -c
      2 chat
      1 dentier
      1 chat
      1 dentier
```

```
[thaeron@syndrome ~]$ cat tmp.txt | uniq -d
chat
```

```
[thaeron@syndrome ~]$ cat tmp.txt | uniq -u
dentier
chat
dentier
```

q. wc : Compte les lignes, mots ou caractères d'un fichier

```
wc [OPTIONS] [FICHIER]...
```

options :

-c : compte le nombre d'octets
-m : compte le nombre de caractères
-l : compte le nombre de lignes
-w : compte le nombre de mots

Exemples :

```
[thaeron@syndrome ~]$ echo ceci est un exemple | wc -c
20
```

```
[thaeron@syndrome ~]$ echo ceci est un exemple | wc -m
20
```

```
[thaeron@syndrome ~]$ echo ceci est un exemple | wc -l
1
```

```
[thaeron@syndrome ~]$ echo ceci est un exemple | wc -w
4
```

r. which : Affiche le PATH complet d'une commande

which utilise le contenu de la variable d'environnement *PATH* pour chercher, dans l'ordre de la liste inscrite dans *PATH*, où est la commande demandée.

```
which COMMANDE
```

Exemple :

```
[thaeron@synapse ~]$ which ls
/usr/bin/ls
```

s. xargs : Passe son entrée standard en argument à une commande

```
xargs COMMANDE
```

Exemple :

Si on désire afficher les détails de tous les fichiers *.sh* qui sont contenus dans le répertoire courant et dans tous ses sous-répertoires. La commande *find* va afficher sur sa sortie standard tous ces fichiers mais *ls* n'accepte pas de données sur l'entrée standard. On ne peut donc pas utiliser directement un pipe.

```
[thaeron@synapse test2]$ find . -name "*.sh" | xargs ls -lah
-rwxr-xr-x 1 thaeron root 4,4K 2010-01-26 20:09 ./guvcview_0.8.2/autogen.sh
-rwxr-xr-x 1 thaeron root  318 2010-01-26 20:09
./guvcview_0.8.2/debian/add_files_deb.sh
-rwxr-xr-x 1 thaeron root  227 2010-01-26 20:09 ./install_example_shell.sh
-rwxr-xr-x 1 thaeron root  190 2010-01-26 20:09 ./load_mod.sh
-rwxr-xr-x 1 thaeron root  214 2010-01-26 20:09 ./mk_nod_pcaxe.sh
-rwxr-xr-x 1 thaeron root   95 2010-01-26 20:09 ./rm_mod.sh
```

9.2. Chainer les commandes

Il est possible de chaîner les commandes afin d'en exécuter plusieurs à partir d'une seule ligne. Dans le cas où la commande2 est indépendante de la commande1 on place un point virgule (;) entre les 2 commandes. La commande 2 sera exécutée juste après la commande1.

Le point virgule indique juste au *shell* que c'est la fin de la commande. Plus intéressant encore, on peut exécuter la commande2 que si la commande1 s'est terminée sans erreurs, pour cela on place 2 et commerciaux (&&) entre les deux commandes.

Dans certains cas on peut désirer exécuter la commande2 si la commande1 s'est terminée par des erreurs, pour cela on utilise 2 pipes (||) entre les 2 commandes.

10. L'écriture de scripts

Nous sommes enfin prêts à aborder le plus intéressant : les scripts *shell*. Un script est un programme interprété (donc sans compilation) par un interpréteur (ici *bash*) permettant d'effectuer automatiquement des tâches rébarbatives. Un script *shell* est un fichier texte contenant un ensemble de commandes, principalement celles que nous avons vu dans les parties précédentes, et d'instructions propres au langage qui est très simple (car très dépouillé).

Nous allons voir la syntaxe de ce langage. Tout d'abord dans la première ligne du fichier il faut indiquer l'interpréteur qui va être utilisé. Ceci n'est pas spécifique et est valable pour tous les langages de scripts que l'on désire rendre exécutable (*chmod +x*) et exécuter directement (sans le passer en paramètre à l'interpréteur).

Voici cette ligne :

```
#!/bin/sh
```

Il est intéressant de noter que dièse (*sharp*) '#' est le caractère permettant de mettre en commentaire. Ce que nous avons vu pour les commandes et les variables est, bien évidemment, toujours valable.

10.1. Conditionnel et boucles

a. if : Teste une valeur booléenne

La syntaxe du *if* est la suivante :

```
if valeur
then
  instructions
fi
```

Le retour à la ligne est nécessaire avant et après le *then*. Il est possible de mettre *if* et *then* sur la même ligne en utilisant le ; (point virgule) comme séparateur.

Exemple :

```
if $PWD == $PATH
then
  echo le path courant est le HOME
fi
```

Comme dans tous les langages il est possible d'exécuter d'autres instructions dans le cas où la condition serait fausse, on utilise alors le mot clé *else*. Dans le cas où on voudrait refaire un *if* on utilise alors *elif* contraction de *else* et de *if*. La structure est la suivante :

```
if valeur
then
  instructions
elif valeur
then
  instructions
else
  instructions
fi
```

b. *test : Évaluation d'une expression*

Dans ce langage, *if* n'évalue pas directement l'expression conditionnelle, il se contente de vérifier la valeur booléenne *TRUE* ou *FALSE*. Il faut pour cela utiliser *test* ou *[*.

Exemple :

```
if [ $PWD = $HOME ]
then
  echo le path courant est le HOME
else
  echo le path courant n\'est pas le HOME
fi
```

ou

```
if test $PWD = $HOME
then
  echo le path courant est le HOME
else
  echo le path courant n\'est pas le HOME
fi
```

Il est important de noter que les espaces avant et après *[* et l'espace avant *]* sont obligatoires !

L'utilisation du point d'exclamation (!) inverse la valeur booléenne.

Exemple :

```
if ! test $PWD = $HOME
then
  echo le path courant n\'est pas le HOME
fi
```

Pour tester ce script il faut l'écrire dans un fichier texte, en oubliant pas d'écrire sur la première ligne *#!/bin/sh*

Une fois sauvegardé (sous le nom *test.sh* par exemple) on a deux possibilités. Soit on l'indique en tant que paramètre à *bash* :

```
thaeron@SquirrelMonkey:~$ bash test.sh
```

Soit on le rend exécutable et on l'exécute :

```
thaeron@SquirrelMonkey:~$ chmod +x test.sh
thaeron@SquirrelMonkey:~$ ./test.sh
```

Pourquoi avoir mis un *backslash* avant l'apostrophe ? Nous avions déjà vu tout au début que le *backslash* permettait de modifier le comportement d'un caractère. En effet dans ce langage l'apostrophe sert à former une chaîne de caractères tout comme les guillemets. Cependant leurs comportements sont différents. Si les guillemets sont juste une indication, l'apostrophe bloque les actions du *shell* sur le contenu de la chaîne. Par exemple elle bloque le processus d'expansion des variables, c'est-à-dire que les variables ne seront pas remplacées par leur contenu.

Exemple :

```
thaeron@SquirrelMonkey:~$ echo "$HOME"
/home/thaeron
thaeron@SquirrelMonkey:~$ echo '$HOME'
$HOME
```

Nous avons utilisé dans l'exemple *$HOME* et *$PWD* qui sont des variables d'environnement, mais il est possible de créer des variables propres aux scripts. La syntaxe est simple, ce n'est qu'une simple affectation :

```
nom_variable=valeur
```

Il n'y a aucune déclaration à faire. La variable est créée dès qu'une valeur lui est affectée. Les variables dans ce langage ne sont pas typées. Les distinctions et les conversions entre une chaîne, entier et fichier ne sont visibles que dans les tests.

c. Opérateurs de test

Nous avions, dans l'exemple précédent, un '=' pour la comparaison, mais il en existe beaucoup d'autres dont voici la liste des plus utiles.

Op	**Syntaxe**	**Condition exprimée**
=	CHAINE1 = CHAINE2	Les chaînes sont identiques
!=	CHAINE1 != CHAINE2	Les chaînes ne sont pas identiques
-n	-n CHAINE	La chaîne n'est pas vide
-z	-z CHAINE	La chaîne est vide
-eq	ENTIER1 -eq ENTIER2	Les entiers sont égaux
-ne	ENTIER1 -ne ENTIER2	Les entiers ne sont pas égaux
-ge	ENTIER1 -ge ENTIER2	ENTIER1 est supérieur ou égal à ENTIER2
-le	ENTIER1 -le ENTIER2	ENTIER1 est inférieur ou égal à ENTIER2
-gt	ENTIER1 -gt ENTIER2	ENTIER1 est supérieur à ENTIER2
-lt	ENTIER1 -lt ENTIER2	ENTIER1 est inférieur à ENTIER2
-e	-e FICHIER	FICHIER existe
-d	-d FICHIER	FICHIER existe et c'est un répertoire
-f	-f FICHIER	FICHIER existe et il est régulier
-x	-x FICHIER	FICHIER existe et il est exécutable

Tableau 17: Opérateurs de comparaisons de *test*

Pour évaluer plusieurs expressions lors d'un test :

Op	**Syntaxe**	**Condition exprimée**
-a	EXPRESSION1 -a EXPRESSION2	Si EXPRESSION1 et EXPRESSION2 sont vraies
-o	EXPRESSION1 -o EXPRESSION2	Si EXPRESSION1 ou (logique) EXPRESSION2 sont vraies

Tableau 18: Opérateurs binaires de *test*

d. boucle while

La boucle *while* (tant que en français) exécute les instructions tant que la valeur de l'expression de sa condition est vraie. Sa syntaxe est la suivante :

```
while valeur
do
  instructions
done
```

e. boucle until

Il existe aussi la boucle *until* (jusqu'à ce que en français) qui exécute les instructions jusqu'à ce que la valeur de l'expression de sa condition soit vraie. Sa syntaxe est la même que celle de *while* :

```
until valeur
do
  instructions
done
```

f. boucle for

Dans ces 2 cas, l'expression de la condition est régie par les mêmes règles que pour *if*. La boucle *for* est totalement différente. Elle permet de traiter les données d'une liste en en prenant une par itération. Sa syntaxe reste simple :

```
for variable in liste
do
  instructions
done
```

Exemple :

```
for i in 1 2 3 4 5
do
  echo $i
done
```

ou encore

```
for i in "ceci" "est" "un" "test"
do
  echo $i
done
```

g. *case*

La dernière structure conditionnelle à voir est *case*. Elle permet de sélectionner différentes instructions à exécuter selon la valeur d'une variable. Sa syntaxe est la suivante :

```
case $variable in
valeur1)
  instructions
;;
valeur2)
  instructions
;;
valeur3)
  instructions
;;
esac
```

h. *read*

bash dispose d'une commande interne (*built-in*) très utile dans l'écriture des scripts : *read*. La commande *read* permet de capturer ligne par ligne ce qui est envoyé sur l'entrée standard et de placer ce contenu dans une ou des variables passées en arguments.

Le comportement de *read* est lui-même intéressant car si on lui donne 3 variables en paramètres il va découper la ligne en mots (le séparateur étant un espace ou une tabulation) et mettre le premier mot dans la première variable, le deuxième dans la deuxième variable et tout le reste dans la dernière variable.

Exemple :

```
read mot1 mot2 mot3
echo mot1 = $mot1
echo mot2 = $mot2
echo mot3 = $mot3
```

Ce qui donne à l'exécution

```
[thaeron@synapse ~]$ bash test.sh
ceci est un test
mot1 = ceci
mot2 = est
mot3 = un test
```

read s'utilise la majorité du temps avec *while*. En effet, nous avons vu que *if* (et donc cela s'applique aussi à *while* et *until*) n'évalue que si la valeur booléenne est VRAIE ou FAUSSE. Ainsi on peut directement évaluer la valeur de retour d'une commande. Tout programme renvoie une valeur à l'OS à la fin de son exécution, si l'exécution s'est bien déroulée elle renvoie 0 (ce qui pour *bash* signifie TRUE) et une autre valeur souvent négative en cas de problème (ce qui est équivalent à FALSE).

Tant que *read* n'a pas rencontré *EOF* (*End Of File*) elle renvoie 0.

Exemple d'utilisation :

```
while read ligne
do
  echo ligne saisie = $ligne
done
```

i. variables spéciales

Nous venons de voir qu'un programme renvoie une valeur lorsqu'il se termine. Si on désire récupérer cette valeur en vue de la traiter ultérieurement il existe pour cela une variable spéciale. Ce n'est pas la seule, voici les principales :

Var	**Description**
$?	Contient la valeur du dernier programme qui s'est terminé
$@	Contient la liste des paramètres passés au script
$#	Contient le nombre de paramètres passés au script
$N	Contient le Nième paramètre à partir de 1 ($1 $2 $3 etc)

Tableau 19: Variables spéciales

j. Affectation des variables et opérations

Nous avons vu que nous pouvions rediriger les flux, renvoyer le flux de sortie standard d'un programme vers le flux d'entrée standard d'un autre. Cependant, comment rediriger la sortie standard vers une variable ?

Il existe 2 syntaxes. La première vieille école :

```
variable=`commande`
```

Et la seconde :

```
variable=$(commande)
```

Il est donc possible de passer en argument, à un autre programme, le contenu de variable.

Mais il est possible de contracter tout cela en utilisant *xargs*. *xargs* prend en argument une commande et lui passe en arguments ce qu'il reçoit sur son entrée standard. Par exemple, on désire connaître le type du dernier fichier que contient le répertoire courant.

Sans *xargs* :

```
var=$(ls PWD | tail -n1)
file "$var"
```

Avec *xargs* :

```
ls $PWD | tail -n1 | xargs file
```

Contrairement à quasiment tous les langages, il n'est pas possible de faire directement une opération arithmétique sur les variables. La syntaxe est un peu particulière.

```
var=$((operation))
```

Exemple, si on veut incrémenter la variable :

```
var=$(($var+1))
```

Il est possible de découper un script en plusieurs fonctions. Leur syntaxe est simple :

```
fonction nom_fonction()
{
  instructions
}
```

Le plus grand avantage est qu'elles s'utilisent comme si elles étaient des programmes indépendants : on peut leur passer des paramètres, rediriger leurs flux etc.

k. exemples de synthèse

Exemple de synthèse (*while* + opération arithmétique + *case*) :

```
#!/bin/sh

var=0
while [ $var -ne 4 ]
do
  case $var in
  0)
       echo -n "ceci "
  ;;
  1)
       echo -n "est "
  ;;
  2)
       echo -n "un "
  ;;
  3)
       echo "test"
  ;;
  esac

  var=$(($var+1))
done
```

```
thaeron@SquirrelMonkey:~/tmp$ bash test2.sh
ceci est un test
```

Exemple de synthèse (*while* + *for* + *pipe*) :

```
#!/bin/sh

# ce script lit son entrée standard et sépare chaque mot en utilisant
# le caractère ':' comme délimiteur et les affiche un par un

while read ligne
do
  ligne=$(echo $ligne | sed 's/:/ /g')
  for mot in $ligne
  do
       echo $mot
  done
done
```

```
thaeron@SquirrelMonkey:~/tmp$ echo ceci:est:un:test | test3.sh
ceci
est
un
test
```

11. D'autres commandes utiles

Il existe des centaines de commandes et de programmes sous *GNU/Linux* utilisables dans un terminal. Nous ne pouvons bien sûr pas toutes les voir. Nous allons voir quelques programmes supplémentaires très utiles au quotidien.

Programme	**Utilité**
bzip2	Compresse des données en bzip2
date	Affiche et règle l'heure
dd	Convertie et copie un fichier
gzip	Compresse des données en gzip
ldconfig	Met à jour le cache des bibliothèques dynamiques
tar	Manipulation des archives tar
uname	Permet de connaître des informations sur le système
uptime	Permet de connaître des informations sur le système
wget	Permet de télécharger des fichiers par HTTP / FTP
zip	Compresse des données en zip

Tableau 20: Programmes complémentaires

La plupart des logiciels et plus généralement des archives disponibles pour *GNU/Linux* sont des *.tar.gz* ou des *.tar.bz2*

gz correspond à une compression *gzip* et *bz2* à *bzip2*. Cependant ces algorithmes et les programmes qui leurs sont associés ne se chargent pas de créer une archive pouvant contenir plusieurs fichiers (comme le fait *zip*). Cette tâche est effectuée par *tar*.

a. *bzip2 : Compresse des données en bzip2*

```
bzip2 [FICHIER]
```

Si *FICHIER* est omis, *bzip2* compressera les données provenant de l'entrée standard et écrira sur la sortie standard les données compressées. Bien que la décompression soit quasiment toujours effectuée avec *tar* il arrive que les données compressés ne soient pas une archive *tar*.

On utilise alors *bunzip2* de la façon suivante :

```
bunzip2 [FICHIER]
```

Si *FICHIER* est omis, *bunzip2* décompressera les données provenant de l'entrée standard et écrira sur la sortie standard les données décompressées.

b. *date : Affiche et règle l'heure*

date est très pratique pour la manipulation d'une date et permet à l'utilisateur de spécifier son format ce qui rend son utilisation très souple. Pour le détail complet il faut aller voir la page de manuel de *date*.

```
date [+FORMAT]
```

Exemple :

```
date +'%A %d %m (%B) %Y'
```

```
thaeron@SquirrelMonkey:~/tmp$ date +'%A %d %m (%B) %Y'
Wednesday 24 03 (March) 2010
```

c. *dd : Convertie et copie un fichier*

dd est un merveilleux outils permettant de manipuler des fichiers au sens *Unix* du terme. La majorité du temps on utilise *dd* pour manipuler les données pilotes de périphérique.

```
dd [OPTIONS]
```

options :

if=SOURCE : indique la source des données à copier
of=DESTINATION : indique la destination des données
bs=N : lit et écrit par N octets
ibs=N : lit N octets à la fois (512 par défaut)
obs=N : écrit N octets à la fois (512 par défaut)
count=N : copie uniquement N blocs

Si *if* n'est pas spécifié, les données sont lues sur *stdin*. Si *of* n'est pas spécifié, les données sont écrites sur *stdout*.

Il est possible de suffixer N par des coefficients multiplicateurs :

Suffixe	**Coefficient**
w	2
b	512
kB	1000
K	1024
MB	1000*1000
M	1024*1024
GB	1000*1000*1000
G	1024*1024*1024

Tableau 21: Multiplicateurs de *dd*

Exemple :

Pour créer une image ISO d'un CD sous *GNU/Linux* il suffit d'utiliser *dd*.

```
dd if=/dev/cdrom of=image.iso
```

d. gzip : Compresse des données en gzip

```
gzip [FICHIER]
```

Si *FICHIER* est omis, *gzip* compressera les données provenant de l'entrée standard et écrira sur la sortie standard les données compressées. Bien que la décompression soit quasiment toujours effectuée avec *tar* il arrive que les données compressés ne soient pas une archive *tar*.

On utilise alors *gunzip* de la façon suivante :

```
gunzip [FICHIER]
```

Si *FICHIER* est omis, *gunzip* décompressera les données provenant de l'entrée standard et écrira sur la sortie standard les données décompressées.

e. ldconfig : Met à jour le cache des bibliothèques dynamiques

Les bibliothèques dynamiques sont l'équivalent des DLL sous *windows*. Et à l'instar de *windows* il faut les placer dans des répertoires bien spécifiques comme on l'a vu tout au début. Cependant il existe sous *GNU/Linux* un mécanisme de cache. Ainsi chaque fois qu'une bibliothèque est ajoutée au système il faut relancer *ldconfig* qui va mettre à jour le cache permettant ainsi d'exécuter le programme ayant besoin de la bibliothèque.

```
ldconfig
```

ldconfig possède bien des options mais qui ne sont pas très utiles.

f. tar : Manipulation des archives tar

tar permet de manipuler des archives au format *tar*, mais surtout il sert de base à l'utilisation de *gzip* et *bzip2*.

```
tar [OPTIONS] [FICHIER]
```

options :

-x : extrait les fichiers de l'archive
-j : utilise bzip2 pour décompresser
-z : utilise gzip pour décompresser
-v : passe en mode verbeux et affiche les fichiers extraits
-c : crée une archive
-f NOM : utilise NOM comme archive

Si *-f* n'est pas spécifié *tar* utilisera par défaut l'entrée ou la sortie standard.

Pour décompresser une archive *bzip2* :

```
tar -xjf fichier.tar.bz2
```

Pour décompresser une archive *gzip* :

```
tar -xzf fichier.tar.gz
```

Pour créer une archive :

```
tar -cf archive.tar repertoire
```

ou

```
tar -cf archive.tar fichier1 [fichier2]…
```

g. uname : Permet de connaître des informations sur le système

Il est toujours intéressant de connaître quelques informations sur le système qu'on est en train d'utiliser. *uname* permet de connaître la version du noyau, le type d'architecture et le nom du processeur.

```
uname [OPTIONS]
```

options :

-a : affiche toutes les informations
-s : affiche le nom du noyau (cette commande est aussi disponible sur tous les autres *Unix-like*)
-n : affiche le nom de la machine
-r : affiche la version du noyau
-m : affiche le nom de l'architecture
-p : affiche le nom du processeur
-o : affiche le nom de l'OS

Exemple :

```
thaeron@SquirrelMonkey:~/tmp$ uname -a
Linux SquirrelMonkey 2.6.32.3-smp #2 SMP Thu Jan 7 20:10:41 CST 2010 i686
VIA Nehemiah CentaurHauls GNU/Linux
```

h. uptime : Permet de connaître des informations sur le système

uptime donne des informations supplémentaires sur le système. Nous avons vu comment voir la consommation mémoire et CPU d'un processus mais il est parfois intéressant de connaître la charge globale du système. *uptime* permet de connaître cette information ainsi que temps écoulé depuis le démarrage de la machine.

```
thaeron@SquirrelMonkey:~/tmp$ uptime
10:29:09 up 69 days, 17:45,  1 user,  load average: 0.00, 0.00, 0.00
```

i. wget : Permet de télécharger des fichiers sur un serveur HTTP ou FTP

```
wget [OPTIONS] URL
```

options :

-O FICHIER : enregistre le téléchargement sous le nom FICHIER
-c : si le fichier de destination existe déjà, il tente de reprendre le téléchargement
-q : mode silencieux
--limit=VITESSE : limite la vitesse de téléchargement à *VITESSE*, on peut suffixer *VITESSE* par k pour multiplier par 1000.

wget permet de faire beaucoup de choses comme aspirer entièrement un site en suivant les liens. Pour plus de détails il faut consulter la page de manuel.

j. zip : Compresse des données en zip

Bien qu'on utilise rarement ce format sous *GNU/Linux*, les archives destinées aussi à *windows* sont souvent dans ce format.

Syntaxe :

```
zip -r archive.zip répertoire
```

ou

```
zip -r archive.zip fichier1 fichier2 …
```

Pour décompresser :

```
unzip archive.zip
```

12. Administration et configuration

12.1. Les logs

Les divers messages du système et des applications sous *GNU/Linux* sont répartis dans différents fichiers de *logs*.

Il existe 3 *logs* principaux. Tous sont au même endroit : */var/log*

Fichier	**Description**
dmesg	Contient les messages propres au noyau *Linux*.
syslog	Contient les messages envoyés via le démon *syslog*.
messages	Contient des messages du système.

Tableau 22: Principaux *logs*

Le *log dmesg* est associé à une commande du même nom. Ce *log* est vraiment très important car il contient les messages qu'envoie le noyau dès son démarrage.

Tous les périphériques détectés y figurent.

Exemple lors de la connexion d'une clé USB :

```
[thaeron@synapse ~]$ dmesg
…
usb 2-1: new high speed USB device using ehci_hcd and address 6
usb 2-1: New USB device found, idVendor=0951, idProduct=1624
usb 2-1: New USB device strings: Mfr=1, Product=2, SerialNumber=3
usb 2-1: Product: DataTraveler G2
usb 2-1: Manufacturer: Kingston
usb 2-1: SerialNumber: 0014780D8CF1F97115A808F0
scsi10 : usb-storage 2-1:1.0
scsi 10:0:0:0: Direct-Access     Kingston DataTraveler G2  1.00 PQ: 0
ANSI: 2
sd 10:0:0:0: Attached scsi generic sg4 type 0
sd 10:0:0:0: [sdd] 7827392 512-byte logical blocks: (4.00 GB/3.73 GiB)
sd 10:0:0:0: [sdd] Write Protect is off
sd 10:0:0:0: [sdd] Mode Sense: 16 24 09 51
sd 10:0:0:0: [sdd] Assuming drive cache: write through
sd 10:0:0:0: [sdd] Assuming drive cache: write through
 sdd: sdd1
sd 10:0:0:0: [sdd] Assuming drive cache: write through
sd 10:0:0:0: [sdd] Attached SCSI removable disk
```

Syslog est le *log* associé au démon *syslogd*. *Syslogd* est un programme qui tourne en tâche de fond et permet aux programmes, qui y font appel, de stocker leurs messages avec la date et le niveau d'alerte dans ce seul et même fichier de *log*.

12.2. L'amorçage du système

Nous avons vu comment fonctionne le système une fois démarré. Intéressons nous maintenant au démarrage. Sur un système x86 (nos PC), le *BIOS* recherche un disque amorçable et exécute un petit programme sur ce disque qui prend le relais, c'est le *bootloader*. Pour Linux, il y a *LILO* de moins en moins utilisé et *GRUB*. Le *bootloader* charge ensuite le noyau en mémoire qui va ensuite monter la partition principale (/). Une fois le noyau et le matériel initialisé il exécute le programme *init* qui se charge d'exécuter des scripts d'initialisation du système et démarre *getty* par console virtuelle ce qui permet d'avoir les terminaux.

Ces scripts de démarrage sont situés soit dans */etc/rc.d* soit dans */etc/init.d*

Ces scripts sont des scripts *shell* comme ce que nous avons vu précédemment. Leur démarrage est conditionné par leur droit. Par exemple nous avons *rc.syslog* le fameux démon *syslog*. Si nous examinons ses droits :

```
root[rc.d]# ls -lah rc.syslog
-rwxr-xr-x 1 root root 981 2001-06-09 23:10 rc.syslog
```

Nous voyons qu'il est exécutable et sera donc démarré par *init*.

Il est possible à l'utilisateur *root* d'utiliser manuellement ces scripts pour démarrer, arrêter ou redémarrer le programme correspondant.

Exemple avec *rc.sshd* :

Sur le *LiveCD SSHd* (que nous verrons plus loin) n'est pas exécutable. Alors rendons le exécutable et démarrons le :

```
root[rc.d]# chmod +x rc.sshd
root[rc.d]# ./rc.sshd start
```

Pour démarrer il suffit d'indiquer *start* comme paramètre au script. Pour redémarrer on indique *restart* et *stop* pour arrêter.

Les fichiers *rcN.d* où *N* est compris entre 0 et 6 sont des scripts exécutés selon le niveau d'exécution (*runlevel*). Cela correspond au mode d'exécution du système.

Runlevel	**Description**
0	Arrêt du système
1	Mode mono utilisateur
2	inutilisé
3	Mode multi utilisateurs normal
4	Lancement du serveur Xorg
5	Identique à 4
6	Redémarrage du système

Tableau 23: Les différents *runlevels*

Ces modes et leurs scripts, qui y sont associés, sont visibles dans le fichier */etc/inittab* dont nous ne verrons pas le détail.

Cependant on y note une ligne très intéressante :

```
#System initialization (runs when system boots)
si:S:sysinit:/etc/rc.d/rc.S
```

Ceci signifie que le script utilisé pendant le démarrage du système est *rc.S* dans */etc/rc.d.* C'est donc lui qui initialise le système.

Nous venons de voir qu'il y a des modes pour le démarrage et l'arrêt du système mais nous n'avons pas encore vu les commandes associées : *reboot* et *halt*. Il faut bien sûr les exécuter avec les droits *root* (soit en *root* soit via *sudo*).

12.3. Les modules du noyau

Il est temps de voir un peu plus en détail ce qu'est le noyau *Linux*. *Linux* est un noyau monolithique modulaire. On ne développera pas ce qu'est un noyau monolithique car ce n'est pas très important. En revanche l'aspect modulaire est très important car il permet d'ajouter ou d'enlever à chaud (lorsque le noyau est chargé) des modules. Ces modules sont des fonctionnalités du noyau comme des pilotes par exemple. *Linux* est bien sûr multi utilisateurs et multitâches.

Les sources du noyau *Linux* intègrent la grande majorité des fonctionnalités du noyau ainsi que les drivers. Lors de la compilation on a le choix entre compiler ces fonctionnalités soit en dur, donc directement dans le noyau soit en modules. Les modules sont répartis dans une arborescence contenue dans :

```
/lib/modules/VERSION_DU_NOYAU/kernel
```

Les modules du noyau peuvent utiliser et exporter des fonctions qui sont ainsi disponibles pour les autres modules. Afin d'établir ces dépendances après l'installation d'un module il faut utiliser la commande *depmod* :

```
depmod -a
```

La commande *lsmod* permet d'afficher la liste des modules chargés dans le noyau avec leurs dépendances.

Exemple :

```
Module                  Size  Used by
xt_tcpudp               2015  2
iptable_filter          2026  1
ip_tables               8547  1 iptable_filter
x_tables               10594  2 xt_tcpudp,ip_tables
vt1211                 11396  0
hwmon_vid               1972  1 vt1211
```

On voit par exemple que le module *hwmon_vid* dépend du module *vt1211*, ou encore que le module *iptable_filter* utilise le module *ip_tables*.

Pour charger un module on utilise la commande *modprobe* :

```
modprobe NOM_MODULE
```

Pour décharger un module on utilise la commande *rmmod* :

```
rmmod NOM_MODULE
```

Il se peut que certains modules du noyau soient des implémentations différentes des mêmes fonctionnalités, ou qu'ils soient incompatibles. Par exemple le driver libre *framebuffer nvidia* et le driver propriétaire *nvidia*. Pour éviter que le noyau les charge automatiquement on peut les *blacklister*. Le fichier contenant la liste des modules à ne pas charger est *blacklist.conf* ou *blacklist* dans */etc/modprobe.d*

Sa syntaxe est très simple, pour chaque ligne on a :

```
blacklist NOM_MODULE
```

Lors du démarrage suivant du système, le noyau ne chargera plus ces modules.

12.4. Interfaces réseaux

Nous avions vu, tout au début, que les interfaces réseaux n'étaient pas représentées, comme les autres périphériques, par un fichier dans */dev*. Les interfaces réseaux ethernet sont identifiées par *ethN* où *N* est le numéro de l'interface. Il n'y a qu'une commande pour tout faire : *ifconfig*.

Pour afficher toutes les interfaces et leurs configurations :

```
ifconfig -a
```

Généralement, l'interface se nomme eth0.

Pour désactiver l'interface :

```
ifconfig interface down
```

Pour l'activer :

```
ifconfig interface up
```

Dans le cas où l'*IP* est fixe et non attribuée par un serveur *DHCP* :

Pour configurer son *IP* :

```
ifconfig interface IP
```

Pour configurer la passerelle :

```
route add default gw IP_PASSERELLE
```

Les *DNS* sont configurés dans le fichier */etc/resolv.conf*

Pour ajouter un *DNS* il faut utiliser la syntaxe suivante :

```
nameserver IP_DNS
```

Les modifications sont immédiates, il n'est pas nécessaire de redémarrer le système. Dans le cas où l'*IP* est attribuée par un serveur *DHCP*, pour faire une demande d'*IP* pour une interface :

```
dhcpcd interface
```

12.5. Le shell distant

On a souvent besoin, lorsqu'on utilise *GNU/Linux*, de se connecter à distance à une machine, serveur etc. Fort heureusement il est possible d'avoir un *shell* distant très simplement grâce à *SSH*.

a. SSH

SSH est l'acronyme de *Secure Shell* car en plus de permettre une connexion il chiffre aussi toutes les données. *SSH* permet une connexion à distance il est donc basé sur le principe client/serveur. Il n'est pas nécessaire que le serveur *SSH* (*SSHd*) soit exécuté pour que le client fonctionne. En revanche il faut que *SSHd* soit exécuté sur la machine cible. Son script de démarrage étant dans */etc/init.d* ou */etc/rc.d* selon le système.

Depuis une machine sous *GNU/Linux* pour se connecter sur une autre il suffit d'utiliser la commande suivante :

```
ssh [-p port] [-l utilisateur] adresse_distante
```

Si l'option *-l* n'est pas spécifié, *SSH* tentera une connexion avec l'utilisateur courant. Une fois l'authentification faite on dispose du *shell* comme si on était connecté localement.

Il existe un client *SSH* sous *windows* qui se nomme *PuTTY*.

b. SCP

SSH permet aussi de faire du transfert de fichier. Pour cela on utilise la commande *SCP*. *scp* s'utilise d'une manière comparable à *cp*. Pour spécifier la machine distante on utilise la notation suivante :

```
utilisateur@adresse_distante:[cible]
```

Pour copier un fichier ou un répertoire local vers la machine distante :

```
scp [-P port] [-r] NOM utilisateur@adresse_distante:
```

Pour copier un fichier ou un répertoire distant en local :

```
scp [-P port] [-r] utilisateur@adresse_distante:NOM .
```

Comme pour *cp* l'option *-r* permet de copier des répertoires et tout ce qu'ils contiennent.

Exemple si ma machine distante a l'adresse 192.168.0.4 et que je désire copier le répertoire « documents » vers la machine :

```
scp -r documents thaeron@192.168.0.4:
```

12.6. Cron le planificateur de tâches

Nous savons écrire des scripts. Mais il se peut qu'on ait besoin d'exécuter un programme, un script ou une simple commande de manière périodique ou à une date fixe. Il existe bien évidemment un tel planificateur de tâches sous *GNU/Linux*. Ce programme est *Cron*.

Comme d'autres logiciels que nous avons vu et qui tournent en tâche de fond *cron*, ou plus précisément *crond*, est un *démon*.

Crond permet de gérer des fichiers, appelés *crontabs*, contenant les règles à suivre pour l'exécution programmée d'une commande. Comme on peut s'y attendre *crond* exécute les commandes programmées avec l'utilisateur créateur de la règle. Pour créer des règles il faut utiliser le programme appelé *crontab* qui va faire le lien entre l'utilisateur et l'écriture du fichier avec des droits supérieurs dans */var/spool/cron/crontabs/LOGIN*

Syntaxe de crontab :

```
crontab [OPTIONS] [FICHIER]
```

options :

-l : affiche les règles de l'utilisateur
-e : édite (par défaut dans *VI*) les règles de l'utilisateur
-d : supprime les règles de l'utilisateur
-u UTILISATEUR : spécifie l'utilisateur

Par défaut c'est l'utilisateur courant qui sera utilisé.

Regardons maintenant la syntaxe des règles. C'est à la fois très simple et extrêmement puissant. Les règles s'écrivent sur une seule ligne formée de 6 colonnes.

Minute	Heure	Jour	Mois	Jour de la semaine	Commande

Comme pour les scripts *shell*, le caractère # sert à mettre en commentaire jusqu'à la fin de la ligne. Pour spécifier « quelque soit » on indique une étoile *.

Crond traite les règles toutes les minutes. On peut donc faire exécuter une commande toutes les minutes en indiquant partout des étoiles.

Pour exécuter toutes les 10 minutes on écrit */10 dans la première colonne.

Il est aussi possible d'exécuter une commande que un intervalle de temps. Pour cela on écrit BORNE_INFERIEURE-BORNE_SUPERIEURE.

Voyons des exemples :

```
#MIN   HEURE     JOUR      MOIS      JDS  COMMANDE
#pour exécuter date toutes les minutes
*      *         *         *         *    date>>date.txt

#pour exécuter date toutes les 2 heures (pile)
0      */2       *         *         *    date>>date.txt

#pour exécuter date toutes les 10 minutes de 23heure à 7heure du matin
*/10   23-7      *         *         *    date>>date.txt

#pour exécuter date à 10heure pile et 14heure pile
0      10,14     *         *         *    date>>date.txt

#pour exécuter date tous les premiers du mois à minuit
0      0         1         *         *    date>>date.txt
```

Pour les colonnes MOIS et JDS il faut utiliser les abréviations anglaises des mois et des jours.

Il est nécessaire de rediriger les flux de sortie, soit vers un fichier soit vers */dev/null*, car par défaut *crond* redirige les sorties sur le *mailer* interne.

12.7. Compilation d'un programme tiers

Nous n'avons pour l'instant utilisé que des programmes qui étaient déjà disponibles par défaut sur le système. Il arrive souvent qu'on ait besoin d'un logiciel supplémentaire. Nous avons évoqué la gestion des paquets propre à chaque distribution. Nous allons voir que nous pouvons compiler nous-même ces programmes.

Pour l'exemple nous prendrons le serveur *tftp-hpa* qui est un serveur *TFTP* protocole souvent utilisé dans un système embarqué. Les sources sont disponibles à cette adresse : http://www.kernel.org/pub/software/network/tftp/

Commençons par récupérer l'archive :

```
[thaeron@synapse tmp]$ wget -q
http://www.kernel.org/pub/software/network/tftp/tftp-hpa-5.0.tar.bz2
Ensuite on décompresse l'archive
[thaeron@synapse tmp]$ tar -xjf tftp-hpa-5.0.tar.bz2
[thaeron@synapse tmp]$ cd tftp-hpa-5.0
```

On a dans le répertoire un fichier *configure* qui est un script *shell* permettant une vérification des dépendances et une configuration automatique des options de compilation. Le script *configure* est généré par la suite d'outils *autotool* (*autoconf*, *automake*, *libtool*). Il contient alors toujours les mêmes options de base, ensuite les développeurs peuvent choisir d'en ajouter.

Pour connaître toutes les options disponibles on appelle *configure* avec l'option *--help*

Les options toujours disponibles sont les suivantes :

--prefix=DIR Répertoire de base pour l'installation
--libdir=DIR Répertoire où seront installées les bibliothèques
--sysconfdir=DIR Répertoire contenant les configurations
--mandir=DIR Répertoire contenant les pages de manuel
--includedir=DIR Répertoire contenant les fichiers d'en-têtes du C
--bindir=DIR Répertoire contenant les exécutables

La plupart du temps le préfixe par défaut est */usr/local* car comme on l'a vu c'est le répertoire censé contenir les programmes non installés par défaut avec le système. Par défaut tous les autres répertoires ont le préfixe comme référence, par exemple pour les bibliothèques *PREFIX/lib*, *PREFIX/bin* pour les exécutables et *PREFIX/etc* pour les fichiers de configuration.

Nous allons plutôt indiquer */usr* comme préfixe et */etc* pour le répertoire des fichiers de configuration.

```
[thaeron@synapse tftp-hpa-5.0]$ ./configure --prefix=/usr
--sysconfdir=/etc
```

On peut ensuite compiler, ceci se fait par l'intermédiaire de la commande *make* car le script *configure* a généré toutes les règles de compilation (*Makefiles*).

```
[thaeron@synapse tftp-hpa-5.0]$ make
```

Il ne reste plus qu'à l'installer, comme l'installation va se faire au sein du système nous devons obtenir les droits *root*.

```
[thaeron@synapse tftp-hpa-5.0]$ su
Password:
[root@synapse tftp-hpa-5.0]# make install
```

Et pour terminer nous re-générons le cache des bibliothèques.

```
[root@synapse tftp-hpa-5.0]# ldconfig
```

Il existe d'autres outils que *autotool* pour la génération de configuration de compilation. Cependant *autotool* est très majoritairement utilisé. Parfois certains logiciels assez simples sont fournis uniquement avec un *Makefile*, il suffit alors de sauter l'étape de *configure* et d'exécuter directement *make* pour procéder à la compilation.

12.8. Le port série

Nous, électroniciens, avons souvent besoin d'utiliser le port série pour dialoguer avec des cartes. Nous allons donc voir comment l'utiliser dans ce but puis nous verrons comment on peut avoir un terminal (avec *shell* et authentification) sur le port série.

a. Lecture et écriture sur le port série

Il existe 2 programmes ayant les mêmes fonctionnalités que le *HyperTerminal* de *windows* : *gtkterm* et *minicom*.

Lors qu'on utilise ces deux programmes en tant qu'utilisateur il faut vérifier qu'on ait les droits de lecture et d'écriture, dans le cas contraire il faut les ajouter :

```
chmod 777 /dev/ttyS0
```

Gtkterm est un terminal graphique (donc il faut avoir le serveur *Xorg* démarré) dédié à l'écriture et à la lecture sur le port série. Il ressemble beaucoup à *HyperTerminal* et tout est configurable avec les souris par les menus.

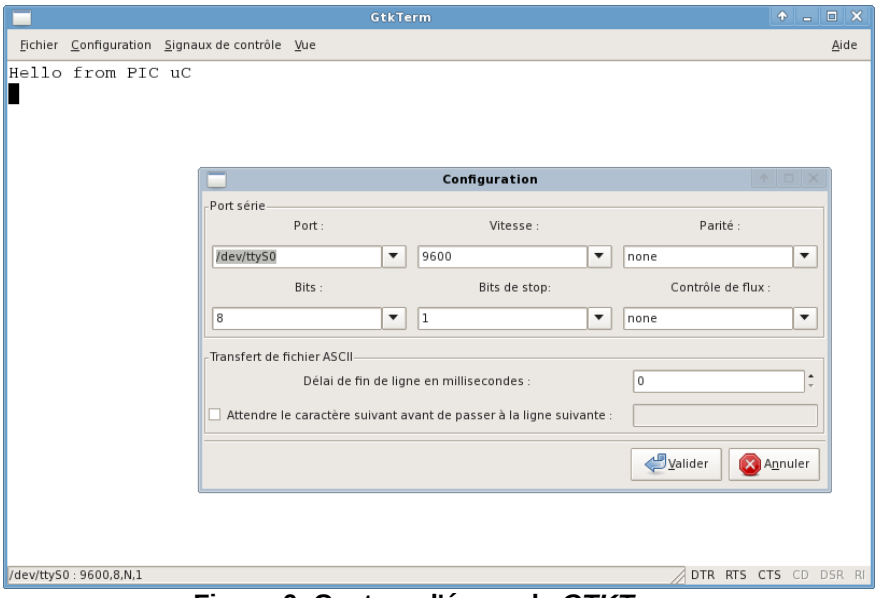

Figure 6: Capture d'écran de *GTKTerm*

minicom est un peu plus compliqué car il est avant tout conçu pour dialoguer avec un modem. Pour le configurer il faut créer un fichier *.minirc.dfl* dans le *HOME* de l'utilisateur ou modifier le fichier générique de configuration : */etc/minirc.dfl*

Examinons sa configuration :

```
pu baudrate 9600
pu bits 8
pu parity N
pu stopbits 1
pu minit
pu mreset
pu mhangup
```

On fixe le transfert 9600 bauds. Les données sont sur 8 bits sans parité avec un seul bit de stop. Ce sont les paramètres les plus couramment utilisé. Ensuite on désactive les options spécifiques à la communication avec un modem (*minit*, *mreset*, *mhangup*).

minicom utilise par défaut le périphérique */dev/modem* et ce n'est pas modifiable par l'utilisateur, si on veut utiliser */dev/ttyS0* il suffit de faire un lien symbolique :

```
ln -s /dev/ttyS0 /dev/modem
```

Il ne reste plus qu'à exécuter *minicom* avec l'option *-o* pour éviter qu'il envoie des commandes d'initialisation.

```
minicom -o
```

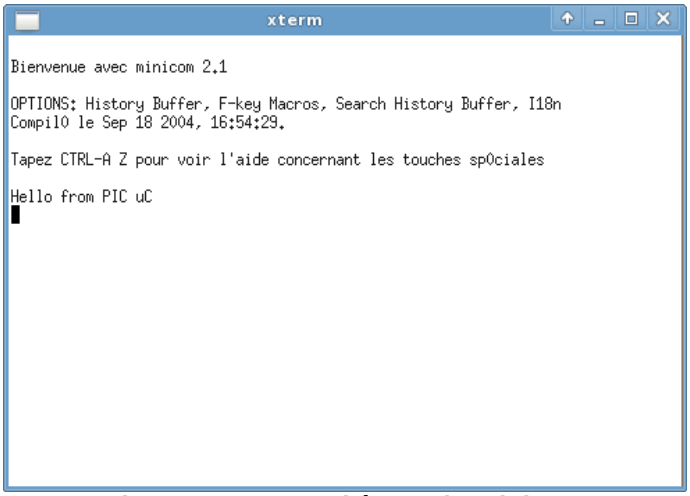

Figure 7: Capture d'écran de *minicom*

b. Le terminal sur le port série

Afin de disposer d'un terminal sur le port série il nous faut un câble *NULL-MODEM*. Ensuite il faut modifier le fichier */etc/inittab* que l'on a déjà entrevu. Nous avions vu que *inittab* était le fichier de configuration de *init*, et qu'*init* se chargeait de démarrer les terminaux virtuels (accessibles par *CTRL+ALT+Fx* avec 0 > x > 7). C'est dans ce même fichier que l'on va pouvoir configurer le terminal sur le port série. Il suffit d'indiquer la ligne :

```
s1:2345:respawn:/sbin/agetty -L 38400 ttyS0 vt100
```

s1 est le nom de la ligne,
2345 sont les *runlevels*,
respawn indique que lorsque la connexion est terminée le processus doit être relancé,
/sbin/agetty est le programme qui va gérer le *shell* sur le port série.

agetty prend les arguments suivant :

-L indique que l'on utilise pas un modem,
38400 c'est la vitesse de transfert,
ttyS0 que l'on utilise */dev/ttyS0* le premier port série
vt100 le type de terminal.

Ensuite on force *init* à relire son fichier de configuration :

```
init q
```

Côté client il faut avoir configuré le programme de lecture du port en 38400bauds, pas de contrôle de flux, 8bits. Le prompt d'authentification devrait apparaître après la commande *init q*. On a donc accès au système comme si on était en local ou par *SSH*.

Mais on peut faire encore plus fort en envoyant les messages d'initialisation du noyau par la liaison série. Il suffit pour cela, juste avant le démarrage du noyau d'indiquer le paramètre suivant :

```
console=ttyS0,38400
```

On obtient alors du côté client :

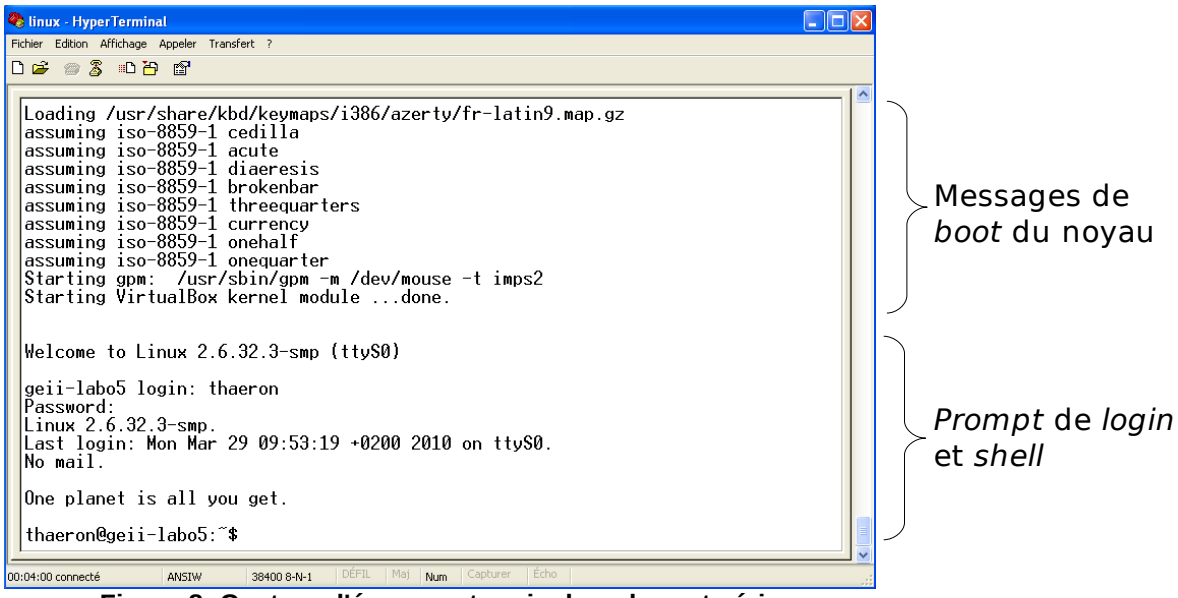

Figure 8: Capture d'écran un terminal sur le port série

Conclusion

Au cours de cette formation, nous aurons appris comment utiliser cette merveilleuse ligne de commande. Utiliser les principales commandes *Unix* et même écrire des scripts *shell*. Nous avons vu les rudiments de l'administration et la configuration. Ces notions vous permettrons de vous débrouiller quelque soit la distribution *Linux* que vous utiliserez, même sur un système embarqué.

Bibliographie

Le système LINUX, O'Reilly France

« Richard Stallman et la révolution du logiciel libre » Une biographie autorisée, Eyrolles

http://www.c-sait.net/cours/scripts.php

http://www.gnu.org/software/bash/manual/bashref.html

http://www.fil.univ-lille1.fr/~sedoglav/OS/main005.html

http://www.vanemery.com/Linux/Serial/serial-console.html

http://linuxdansmonpc.is-a-geek.com/index.php?page=terminal_serie