

Petit guide gnu/Linux

Ce petit guide à été rédigé dans le but de simplifier l'utilisation de la ligne de commande, et de permettre aux utilisateurs d'un système unix de tirer pleinement partie de celui-ci. On considère ici particulièrement gnu/Linux, mais cela s'applique très bien aux autres systèmes unix.

gnu/Linux est un système d'exploitation, où que l'on utilise souvent en ligne de commandes. Celle-ci offre la possibilité de créer une automatisation des tâches : en effet les commandes qui sont tapés à la ligne de commandes peuvent très bien être insérées dans un fichier, on obtient alors un script. Une fois que l'on exécute ce script, les commandes qui sont contenues dans le fichier s'exécuteront l'une après l'autre de façon séquentielle, ce qui permet d'éviter de devoir retaper toutes les commandes à chaque fois que l'on souhaite faire une manipulation. Dans ce texte nous verrons que ces commandes permettent un nombre relativement important de manipulations. Elles vont de la simple administration de fichiers, comme par exemple pour effectuer des sauvegardes automatiques de sortie de programme, ou plus simplement encore pour lancer ces programmes de façon séquentielle, jusqu'à des tâches plus complexes tels que l'extraction de données de ces fichiers, le lancement à des heures précises, ou l'écriture de petites applications interrogeable à distance. Toutes ces possibilités font de l'interpréteur de commande un véritable langage de programmation, qui permet d'effectuer des opérations complexes.

Unix est un système d'exploitation très ancien, c'est pour cela qu'il ne dispose pas d'interface graphique à la base. Celle-ci est remplacé par une interface dans laquelle on entre des commandes. Celles-ci s'exécuteront par un retour à la ligne (une pression sur la touche entrer). Cette interface au fil des années a été fortement perfectionnée, offrant des fonctionnalité d'**alias** de commandes, d'**autocomplétion** de la ligne de commande, ainsi que d'autres fonctionnalités. Il s'est parallèlement à cela également développé un ensemble important de petite commandes permettant un grand nombre de manipulations sur les fichiers, que ce soit sur les fichiers eux mêmes ou bien sur leur contenu.

Ci dessous je donne une liste succincte de commandes, en citant les options qui à mon sens sont les plus importantes. La liste de commandes présente ci-dessous ne se veut pas exhaustive, mais couvre un ensemble de fonctions importantes, qui permettent une bonne utilisation des système unix

Commandes utile sous gnu/Linux

- Listers les fichiers :
 - ls
- Accès à la documentation
 - man
 - info
 - apropos
- Trouver un fichier
 - find : recherche dans le système de fichier
 - locate : recherche dans un base de données de fichiers
 - grep : recherche dans le contenu des fichiers
- Se déplacer dans le système de fichier :
 - cd/pushd/popd
- Manipuler des fichiers :
 - cat : affiche le contenu d'un fichier

- head
- tail
- cut
- join
- sed
- awk
- diff/patch
- grep
- Manipulation des processus :
 - ps/pstree
 - fg/bg
 - kill/killall
 - top
 - trap
 - lock
- Informations sur la machine :
 - free : affiche l'espace mémoire restant
 - du : espace consommé par un fichier ou un répertoire
 - df : affiche l'espace disque disponible
 - uname : affiche la version du système d'exploitation
 - uptime : affiche l'activité du système
 - who
- Archivage :
 - tar
- compressions/décompression de fichiers :
 - bzip2
 - gzip

La documentation en ligne

Le système gnu/Linux (ainsi que les autres système unix) offrent une vaste documentation en ligne qui permet aux utilisateurs de trouver l'information dont ils ont besoin, celle ci est accessible à l'aide de la commande **man**. Pour avoir la documentation de l'interpréteur de commandes (le programme dans lequel on rentre les commandes pour les exécuter), il faut regarder la page du manuel de l'interpréteur de commandes, celui-ci est par défaut **bash**, d'autres interpréteurs de commandes existent.

man bash

Les pages du manuels sont regroupées en différentes sections, chacune d'elles documentant une certaine catégorie d'éléments sous gnu/Linux :

- 1) Commandes utilisateurs
- 2) Appels systèmes
- 3) Librairies de programmations
- 4) documentation sur des périphériques
- 5) Commandes systèmes, fichier de configuration système : rouage du système

- 6) divers
- 7) Conventions, principes de fonctionnement de certaines commandes
- 8) Commandes systèmes

Il arrive que certaines commandes aient des pages de manuels dans deux rubriques, ceci est dû au fait que la commande à le même nom que les appelle à des bibliothèques de programmation, on peut par exemple citer les appels, **printf** et la commande **printf(3)** qui correspond à un appel à la bibliothèque standard du C.

L'accès au premier ne nécessite pas de spécification :

```
man printf
```

C'est pour le second qu'il est nécessaire de spécifier la page du manuel que l'on souhaite consulter.

```
man 3 printf
```

gnu/Linux est généralement livré sous forme de distribution, celles-ci sont des ensembles cohérents de programmes : les versions des bibliothèques sont telles que tous les programmes fonctionnent entre eux (les bibliothèques sont des ensembles de méthodes informatiques regroupées dans une même entité). L'ensemble est compilé, et on dispose en générale d'un système permettant d'installer facilement des programmes. La distribution que l'on considérera ici est celle basée sur debian, elle repose sur un système de paquets très efficace : les archives debian et le système qui permet de les installer : la suite apt (dont apt-get et aptitude font partie). C'est la distribution la plus fournie, les dérivés de cette distribution (tels que ubuntu) disposent aussi d'une grande partie des programmes de la distribution mère.

Les commandes de bases en rapport avec le système de fichier

Se déplacer dans le système de fichiers

Ceci se fait à l'aide de la commande **cd** (**change directory**) :

```
cd /home/fmunch
```

permet d'aller dans le répertoire personnel. Par défaut **cd** se déplace par rapport au répertoire courant (ce comportement peut être modifié en changeant la valeur de la variable CDPATH) à moins qu'on lui spécifie le chemin complet comme ici. **cd repertoire** permettra de se déplacer dans **/home/fmunch/repertoire**. Les répertoires contiennent deux fichiers particuliers :

- "." correspond au répertoire dans lequel on se trouve, c'est par rapport à celui ci que l'on se déplace quand on ne spécifie pas le chemin absolu (ce dernier consistant à spécifier le chemin à partir de la racine).
- ".." est le répertoire parent de celui où l'on se trouve

Par défaut, dans l'interpréteur de commandes, il s'affiche au début de la ligne ce que l'on nomme l'invité de commande qui nous indique avant le \$ dans quel répertoire on se trouve; on a par exemple :

```
~$ cd /  
/$ cd ~/perso_bin/
```

```
~/perso_bin$
```

En omettant de spécifier un répertoire, **cd** se déplace vers le répertoire personnel. La commande "**cd -**" permet de revenir au répertoire précédent. Il existe aussi les commandes **pushd** et **popd** qui permettent de dispoer d'une pile de répertoire. **pushd** a le même effet que **cd**, il change le répertoire courant, mais en plus place l'ancien répertoire dans une file. La commande **popd** aura pour effet de retourner au répertoire précédent. La commande **dirs** permet de connaître l'ensemble des répertoire qui sont présent dans cette pile.

La commande **pwd** permet elle de connaître le répertoire courant. On désigne le répertoire personnelle par **~** ceci permet de racourcir les noms de chemains utilisés. Si par exemple on veut accéder au répertoire personnelle d'autre_utilisateur, on peut écrire :

```
cd ../../autre_utilisateur
```

Ceci suppose que les répertoire de l'autre utilisateur et de l'utilisateur courant ont le même répertoire parent. De façon plus général on peut aussi écrire :

```
cd ~autre_utilisateur
```

où **~autre_utilisateur** représente le répertoire personnel d'un autre utilisateur.

Lister les fichiers : ls

Cette commande permet d'afficher les fichiers contenue dans un répertoire est fort utile et permet par exemple de savoir où l'on en est dans les sorties d'un programme, ou bien ce qui a déjà été réalisé dans un projet. C'est une commande qui s'applique par défaut sur le répertoire courant. On désigne ce dernier généralement par « ./ ». Sans option cette commande affiche simplement la liste des fichier de la façon suivante :

```
fmunch@home:~/mireille$ ls
info list_apt list_apt~
```

par défaut, c'est une simple liste de fichiers, séparés par des espaces. Cependant on peut vouloir plus d'informations sur les fichiers, pour cela on peut utiliser l'option **-l** qui permet l'affichage sous forme longue :

```
$ ls -l
total 12
drwxr-xr-x 2 fmunch fmunch 4096 2008-06-17 14:05 info
-rw-r--r-- 1 fmunch fmunch 2490 2008-06-15 18:52 list_apt
-rw-r--r-- 1 fmunch fmunch 2481 2008-06-15 18:51 list_apt~
```

on constate que cette sortie est organisée sous forme de colonnes, ces colonnes contiennent différentes informations :

- une colonne avec le type et les permissions des fichiers
- une colonne avec le nombre de liens vers ce fichiers
- une colonne avec le nom de l'utilisateur à qui appartient le fichier
- une colonne avec le nom du groupe à qui appartient le fichier
- la taille du fichier

- la date de modification du fichier
- l'heure de modification
- et enfin le nom du fichier

La première colonne correspond aux types et aux permissions du fichier. Le type de fichiers est désigné par le premier caractère, les permissions par les 9 suivants. Les fichiers peuvent être de divers types, ici on a un répertoire (**directory** en anglais, d'où le **d**) et des fichiers, qui sont désigné par un « - ». Il existe également des fichiers **liens**, des fichiers **block**, des fichiers **caractère**, des fichiers **fifo**; dans ces cas on aura respectivement les préfixes **l**, **b**, **c**, et **f**. Les 9 caractères suivants spécifient les droits d'accès au fichier, de l'utilisateur à qui appartient le fichier, du groupe auquel appartient le fichier, ainsi que des autres utilisateurs. Ces permissions sont de 3 natures : droit de lire le fichier (**read** en anglais, il est donc symbolisé par un **r**), droit d'écrire dans le fichier (**write** en anglais d'où **w**), et exécuter qui est symbolisé par un **x**.

Les deux colonnes suivantes correspondent aux noms de l'utilisateur et du groupe auxquels appartiennent les fichiers. La colonne suivante correspondant à la taille du fichier, exprimée en octets. Puis on a deux colonnes pour la date de modification du fichier, et une dernière pour le nom du fichier.

La commande **ls** dispose de différentes options :

- l** : affichage en mode long
- a** : affichage de tous les fichiers (y compris les fichiers qui commencent par un point)
- b** : utilise des caractères d'échappements : les espaces sont précédé par des antislash
- color** : affichage en couleurs des fichiers listés, la couleur utilisé dépend du type de fichier ou de leurs extensions.
- h** : les tailles sont affichées à un format adapté à la lecture : on donne des tailles en kilooctets, et mégaoctets... selon la taille du fichier
- g** : affiche uniquement le groupe
- i** : affiche le numéro d'inode du fichier : c'est un identifiant spécifique au système de fichier
- m** : sépare les fichiers avec des virgules
- r** : inverse l'ordre de trie
- S** : trie les fichiers par tailles
- t** : trie par date de modification

D'autres options existent :

- on peut personnaliser le format avec **--format=**
- spécifier la largeur de l'écran avec **-w**
- trier d'autres façons comme par exemple par extensions
- afficher la taille des fichiers en unité SI (1ko=1000octets et non 1024)
- personnaliser la coloration des fichiers

Caractères spéciaux pour spécifier des ensembles de noms de fichiers

Il est également possible de spécifier une indétermination dans un nom de fichier, par exemple si l'on veut lister tous les fichiers se terminant par l'extension **txt** :

```
ls *.txt
```

L'étoile spécifie ici un ensemble de caractères quelconque de longueur indéterminée. On peut également lister tous les fichiers se terminant par l'extension txt ou pdf, de la façon suivante :

```
ls *. {txt, pdf}
```

ici les deux terme entre les crochets permet de spécifier un choix multiple, on peut spécifier un nombre plus important d'alternatives, par exemple `*. {txt, text, pdf, ps}`. En réalité la spécification d'alternatives, est transformée en la liste d'arguments correspondants :

```
echo a{b, c}
```

produira :

```
ab ac
```

La commande **echo** est une commande qui permet d'afficher la liste des arguments séparés par un espace.

On peut également spécifier un intervalle de caractères, à l'aide d'une spécification entre crochet :

```
ls [a-z] ?.c
```

Listera tous les fichiers commençant par une lettre suivie d'un caractère quelconque, et qui ont pour extension `.c`. Le `[a-z]` est l'intervalle des lettres comprises entre `a-z`; la différence majuscule/minuscule (que l'on nomme aussi la case) n'intervient pas ici.

Ces caractères spéciaux ne sont pas les mêmes que ceux utilisés dans les expressions régulières (on les veras plus tards). Dans ces dernières `*` et `?` ont un autre sens, et généralement la case intervient dans les groupes de lettres (spécifier comme un intervalle avant).

Création de répertoires

Une organisation hiérarchisée des données est importante afin de pouvoir retrouver les données qui correspondent à un même projet, à une même analyse de données... La création de répertoire se fait simplement avec la commande **mkdir** :

```
$ ls -l
$ mkdir repertoire
$ ls -l
drwxr-xr-x 2 fmunch fmunch 4096 2008-06-26 13:52 repertoire
$
```

Il est également possible de créer les répertoires intermédiaire, en spécifiant l'option **-p**, la comamnde :

```
mkdir -p ~/répertoire/sous_répertoire
```

créera le répertoire `"~/répertoire/sous_répertoire"`. Il créera égalemnt `"~/répertoire"` si ce dernier n'existe pas encore.

Liens entre fichiers

On peut également faire des liens entre les fichiers, ceci se fait à l'aide de la commande **ln**, deux type de liens existent :

- les liens symboliques, qui sont juste une référence vers un autre nom de fichier
- les liens du même type que le liens entre les noms de fichier et les fichiers eux mêmes

Les plus utilisés sont les liens symboliques, les fichiers source et destination sont placées dans le même ordre que les commandes permettant de déplacer les fichier et de les copier. La création de liens symboliques se fait avec "**-s**" :

\$ ln -s fichier liens

On peut également omettre le nom du lien auquel cas il portera le même nom que le nom du fichier vers lequel le lien pointe et sera créé dans le répertoire courant. Les liens symboliques n'ont pas de permission, ce sont les permissions du fichier vers lequel ils pointent qui s'appliquent. Ces liens peuvent également se faire vers des répertoires.

Les autres types de liens sont les liens "dur" qui ne peuvent que se faire sur un même système de fichiers. Ceux-ci se font quand l'option -s est absente à la commande ln. C'est seulement l'administrateur système qui a le droit de créer des liens dur sur les répertoires.

On a une certaine tendance à privilégier les liens symboliques, la connaissance du système de fichier auquel appartient le fichier, n'étant pas requise.

Informations sur les fichiers dont on est en présence

Les fichiers ont en général des extensions qui permettent leurs identification. Cependant celle-ci peut être absente. Dans ce cas on peut identifier les fichiers à l'aide de la commande **file**. Cette commande utilise le début du fichier pour en identifier le type, une description détaillé se trouve dans la page du manuel de **magic**. On peut spécifier le fichier de reconnaissance à utiliser avec l'option **-m** ou **--magic-file**. L'option **-f** permet de lire une liste de fichier qu'il faut traiter, ceci est plus efficace que d'utiliser la commande **file** pour chaque fichier trouvé. On peut choisir le caractère qui sépare la description du type de fichier, du nom de fichier, avec **-F**. La commande **file** permet également de traiter des fichiers compressés à l'aide de l'option **-z**.

Il peut être intéressant de connaître la taille d'un fichier ou d'un ensemble de fichiers. Cette information peut être obtenue à l'aide de la commande **du**. Selon les implémentations celles-ci donnent la place utilisée par chaque fichiers, ou encore la taille de chaque fichier. Les fichiers sont organisés sous forme de blocs, sur le système de fichier, cependant certains systèmes de fichiers peuvent stocker de petits fichiers dans la table des fichiers plutôt que d'utiliser un bloc entier (qui est en général un multiple de 512 octets). Donc si chaque bloc fait 8 ko, et que l'on a 2'000'000 de fichiers, on occupera au minimum 16 Go car chacun des blocs occupe 8 ko. Les systèmes de fichier récents évitent cela. Comme nous le verrons l'archivage sous forme de fichier **tar** par exemple l'évite également

On peut spécifier différentes options à la commande **du** :

- a** : affiche la taille de tous les fichiers
- b** : affichage de taille en octets

- B n** : définit la taille des blocs à n
- h** : affiche les tailles dans des unités commodes (affichage pour la lecture par des humaines d'où le h)
- si** : comme -**h** mais avec 1ko=1000 octets ...
- k** : affichage en ko
- m** : affichage en mega octets
- max-depth=N** : précise la profondeur maximale à laquelle il faut aller dans le système de fichier
- x** : reste sur un même système de fichiers

Il est aussi important de contrôler l'espace disque restant, ceci se fait à l'aide de la commande **df** (pour connaître l'espace mémoire disponible il y a la commande **free**). En spécifiant un fichier ou un répertoire après, on obtient l'espace restant dans le système de fichier contenant le répertoire ou le fichier. En plus des options précédentes pour les différents types d'unité, on dispose de l'option **-i** qui affiche les informations sur les **inodes**, qui sont l'endroit où les informations sur les fichiers peuvent se mettre.

On peut également obtenir des informations quant à la date de modification, la date d'accès ainsi que la date de création du fichier. Ceci peut se faire à l'aide de la commande **stat**. Celle-ci indique plusieurs informations :

```
~$ stat fichier.sh
```

```
File:  `/home/fmunch2/fichier.sh'
Size: 47      Blocks: 8      IO Block: 4096   regular file
Device: 807h/2055d Inode: 3588799      Links: 1
Access: (0744/-rwxr--r--)  Uid: ( 1000/  fmunch)  Gid: ( 1000/  fmunch)
Access: 2008-06-27 11:24:29.000000000 +0000
Modify: 2008-06-27 11:24:28.000000000 +0000
Change: 2008-06-27 11:24:28.000000000 +0000
```

La première ligne donne le chemin d'accès absolu au fichier. La seconde ligne comporte la taille, le nombre de blocs utilisé, et la taille des blocs minimal (ici 8*512), et le type de fichier : ici on est en présence d'un fichier usuel. La 3^{ème} ligne décrit des informations sur le périphérique de stockage. La 4^{ème} donne des informations sur les permissions d'accès, les 3 suivantes sont respectivement la dernière date d'accès au fichier (si celle-ci est mise à jour (l'option **noatime** dans **mount** désactive cette mise à jour)), la date de modification des données du fichier, et la date de modifications de l'inodes (c'est à dire si les permissions ont été modifiées, elle différera de la date de modification des données). La commande **stat** permet aussi d'obtenir une sortie formaté à l'aide de l'option **-c** ou **--format=** ou encore **--printf** qui interprète également les caractères d'échappement. Cette commande permet aussi de suivre de liens avec l'option **-L** ou **--dereference** .

La date de modification d'un fichier peut être modifiée à l'aide de la commande **touch**, on utilise souvent cette dernière uniquement pour créer un fichier :

```
touch fichier
```

Va mettre à jour toutes les dates du fichier à la date courante, si le fichier n'existe pas il sera créé.

L'option **-c** permet d'éviter la création du fichier comportement. **-a** et permet de changer uniquement la date d'accès au fichier, **-m** uniquement la date de modification. La date par défaut est la date courante. On peut aussi changer la date en utilisant pour date de référence la date d'un fichier : **-r référence**, ou encore spécifier la date avec l'option **-t** au format `[[CC]UU]MMDDhhmm[.ss]`.

L'archivage

Regrouper les fichiers est particulièrement intéressant si l'on souhaite transmettre une hiérarchie complète de fichiers. L'archivage est souvent effectué à l'aide de la commande **tar**. D'autres formats d'archivage existent, tels que CPIO, ar..., on peut trouver plus de détails sur wikipedia : http://en.wikipedia.org/wiki/List_of_archive_formats. Différentes options existent :

- t** : liste l'ensemble de fichiers
- c** : crée un nouveau fichier
- x, --extract** : extrait les fichiers de l'archive
- r, --append** : ajoute un fichier à la fin d'un archive existante
- v** : mode verbeux : précise ce qui est entrain de se dérouler

On peut aussi utiliser la compression lors de la création et de l'extraction des archives.

- Z** : compresse au format Z
- z** : compresse au format gzip
- j** : compressions bzip2

Pour créer une archive, compresser au format bzip2 :

```
tar cjf repertoire.tar.bz2 repertoire/
```

Lister les fichiers contenus dans cette archive

```
tar tjf repertoire.tar.bz2
```

Parfois il est utile d'utiliser les commandes **bzip2**, **gzip** et **compres**, celles-ci permettent la compression d'un fichier unique, on peut aussi y ajuster le taux de compression. Bzip2 est le plus efficace des trois, **gzip** se place en seconde position.

Les niveaux de compressions se spécifient de la façon suivante :

- 1 : compression faible mais rapide
- 4 : compression standard
- 9 : compression élevé mais lente

Il peut être intéressant de spécifier un niveau de compression faible si l'on souhaite faire des testes. Les gains entre le niveau de compression le plus faible et le plus élevé sont souvent assez réduit.

Extraction d'un morceau d'un fichier

La commande **dd** permet d'extraire des régions d'un fichiers :

dd if=fichier of=copie bs=1024 skip=10 count=20

Cette commande prend les blocs 10 à 30 dans le fichier "**fichier**" avec des blocs de taille 1024 octets, et les copie dans le fichier copie. Il est également possible de copier dans un fichier à une certaine position (avec l'option **seek**). Si l'on ne précise pas la sortie, celle-ci est par défaut la sortie standard, si l'entrée n'est pas précisée, celle-ci est l'entrée standard.

echo Bonjour | dd bs=1 skip=2 count=3 2> /dev/null

/dev/null correspond à un périphérique dans lequel on peut mettre n'importe quoi, les données qui rentrent y sont perdues. Ici on redirige les autres informations produite par la commande dd. On verra par la suite ce que signifie 2> . Pour l'instant, il suffit de considérer que cela permet d'afficher la sortie de la commande dd. La commande dd permet également de connaître la vitesse d'un périphérique :

dd if=/dev/hda of=/dev/null bs=1024k count=512

Une autre commande fort utile est la commande **od**, celle-ci affiche les données sous forme octal, ou hexadécimal, ou sous forme de caractères, ceci peut être très pratique pour exprimer des valeurs entières en hexadécimal...

echo A | od -c -> A \n

On n'a gardé ici que la partie intéressante de la commande, on constate qu'il y a une nouvelle ligne après. Les deux lignes suivantes sont les représentations hexadécimale et octale de ces caractères ASCII (ascii étant à droite précédemment, à gauche, cela pourrait être un autre codage).

echo A | od -t x1 -> 410a
echo A | od -o -> 101012

On peut également échanger certains caractères par d'autres ou en supprimer, cela peut être utile pour convertir des fichier texte dos à des fichiers texte unix, la différence entre les deux, est la façon d'exprimer les nouvelles lignes. Sous dos (et d'autres produits Microsoft) les nouvelles lignes sont représenter par deux caractères **\r\n**. Une commande existe pour cela c'est dos2unix. Une autre solution est de supprimer les caractères **\r**. Ceci se fait avec la commande **tr**, qui remplace ou supprime des caractères, le remplacement se fait en spécifiant un ensemble de caractères de départ et un ensemble d'arrivée :

echo Bonjour | tr "[a-z]" "[A-Z]"

Pour supprimer les caractères '**\r**' qui ne nous intéressent pas on pourra simplement utiliser

tr -d '\r' < texte.dos > texte.unix

Découper un fichier

Défois il peut être utile de diviser un fichier en sous blocs, afin de pouvoir échanger ce fichier de façon plus simple, ou le stocker sur des espaces de stockage réduit. Cela peut se faire à l'aide de la commande **split**, elle découpe des fichiers en fichiers ayant le même nombre de lignes

chacun ou encore la même taille chacun :

```
$ split fichier  
$ ls  
xaa xab xac ...
```

On peut spécifier le préfixe des fichiers à créer. L'option **-b** permet de spécifier le nombre d'octets pour chacun des morceaux, ou avec l'option **-l** le nombre de lignes. On peut aussi spécifier la longueur du suffixe à l'aide de l'option **-a**. Enfin l'option **-d** permet d'utiliser des suffixes numériques plutôt que des suffixes alphabétiques.

Permissions des fichiers :

Les permissions : principe de fonctionnement

Les permissions permettent de restreindre l'accès aux ressources de la machine. Cela évite que des virus puissent écrire dans les fichiers systèmes, que l'on puisse supprimer les données d'un autre utilisateur. Cependant la seule notion d'utilisateur n'est pas suffisante, c'est pour cela qu'a été ajouté la notion de groupe, celle-ci fournit des droits d'accès à un ensemble d'utilisateurs. Ces ensembles d'utilisateurs sont gérés par l'administrateur.

A chaque fichier correspond une permission, un identifiant d'utilisateur (auquel correspond un nom) un identifiant de groupe (auquel correspond aussi un nom). Par défaut le nom correspond au nom de l'utilisateur qui a créé le fichier, et le groupe, au groupe auquel appartient le fichier. Il est possible de lancer un interpréteur de commande qui créera des fichiers avec un autre groupe par défaut, à l'aide de la commande **newgrp**. Deux commandes en plus de la commande **ls** sont importantes pour gérer les permissions des fichiers :

- **chown** : change le nom d'utilisateur et de groupe (on peut aussi utiliser **chgrp** si on ne veut que changer le groupe)
- **chmod** : change les permissions des fichiers

chown (change owner) s'utilise de la façon suivante :

```
$ chown utilisateur:groupe fichier
```

tout comme la commande **chmod**, elle dispose d'une option permettant de l'appliquer à une hiérarchie de fichiers : le mode récursif : **-R**.

La commande **chmod** est un peu plus compliquée à utiliser : parfois on ne veut pas forcément simplement assigner le mode, mais on peut aussi vouloir le modifier partiellement. Voyons dans un premier temps comment simplement l'assigner à une certaine valeur. Pour cela deux façons de faire existent, soit on utilise une valeur numérique, soit une chaîne de caractères correspondant au mode d'accès que l'on autorise. Dans les deux cas on aura 3 éléments de permissions :

- un pour l'utilisateur : celui le plus à gauche ou spécifié par un **u** pour user
- un pour le groupe : celui au milieu ou spécifié par un **g**
- un pour tous les autres : celui le plus à droite ou spécifié par un **o** pour other

Chaque élément de permission se décompose de la façon suivante :

4 pour la lecture
2 pour l'écriture
1 pour l'exécution

Pour avoir la permission sous forme numérique il faut sommer chacun des éléments. Par exemple, on veut qu'un fichier puisse être lu et exécuté, alors on obtient : $4+1=5$. Pour permettre la lecture et l'écriture et la lecture on a $4+2=6$. Ainsi si seul l'utilisateur auquel appartient le fichier a le droit de le modifier (par exemple pour un script écrit par un utilisateur pour les autres utilisateurs (dans ce cas les utilisateurs utilisant ce script accordent une grande confiance à l'utilisateur qui a créé le script)), et tout le monde a le droit de le lire et de l'exécuter, on assignera les permissions de la façon suivante :

```
$ chmod 755 fichier
```

Le premier **7** correspondant à l'utilisateur, les deux **5** qui suivent correspondent respectivement aux permissions du groupe et des autres utilisateurs.

Dans le mode textuel, cela donne :

```
chmod u=rwx,go=rx toto
```

Ce mode n'est ici pas très facile à utiliser. Cependant dans le cas où l'on doit uniquement enlever des droits, ou en rajouter, cela est très pratique. Supposons par exemple que l'on veuille autoriser les autres utilisateurs à exécuter un script. Dans ce cas, il est nécessaire d'ajouter la permission d'exécutions, les autres permissions pouvant varier d'un fichier à un autre. Dans ce cas le mode textuel est parfaitement adapté; une autre solution est certainement possible, mais beaucoup plus pénible à mettre en oeuvre. La solution avec le mode textuel est :

```
$ chmod o+x fichier
```

Avec le symbole **-** on peut retirer des permissions sur un fichier. Un caractère désignant **u,g**, et **o** est disponible, cela permet une écriture plus aisée par exemple tout le monde a le droit de lire et d'exécuter (les 3 commandes sont équivalentes) :

```
chmod 555 fichier  
chmod ugo=rx fichier  
chmod a=rx fichier
```

3 modes supplémentaires sont disponibles, deux bits qui autorisent les fichiers à être exécutés en tant que l'utilisateur auquel appartient le fichier (SETUID) ou son groupe (SETGID). Ces modes sont spécifiés de la façon suivante :

```
chmod u+s
```

```
chmod g+s
```

Il existe également le sticky bit qui a pour effet de permettre d'avoir des répertoires tel que **/tmp**, où seuls les propriétaires des fichiers peuvent les supprimer.

La commande **umask** permet de définir un mode par défaut. Il est spécifié à l'aide d'un masque qui est utilisé pour déterminer les permissions des fichiers nouvellement créés. Par défaut il est de 022, les fichiers créés ont au maximum par défaut le mode 666, en retranchant 022, on obtient 644, donc par défaut les fichiers créés ont la permission de lire et d'écrire pour le propriétaire du fichier, et uniquement le droit d'être lus pour les membres du groupe du fichier, ainsi que pour les autres utilisateurs.

Rechercher un fichier

La recherche de fichier peut se faire selon deux types d'informations :

- le contenu des fichiers
- les caractéristiques extérieures du fichier :
 - nom (éventuellement en ignorant la case, ou avec des expressions régulières)
 - taille
 - date de modification, ou d'accès
 - permissions
 - type de fichier
 - système de fichier sur lequel le fichier est présent

Une fois le fichier trouvé, on peut vouloir exécuter une action. Il existe principalement 3 types de commandes pour effectuer ce genre de tâches :

- la recherche de fichier à travers tout le système de fichiers en ne regardant pas le contenu des fichiers; ceci se fait avec la commande **find**
- la recherche rapide en se basant sur un listing de tous les fichiers disponibles sur le système de fichiers, ceci se fait à l'aide de la commande **locate**
- la recherche dans le contenu des fichiers : se fait à l'aide de la commande **grep -r** le **-r** faisant que **grep** fonctionne en mode récursif : il va rechercher.

Usage de la commande **find**

La commande **find** dispose de nombreuses options; la plus importante est **-name fichier** et **-iname fichier** qui permet de chercher un fichier ayant pour nom **fichier** en considérant ou non les différences entre majuscules et minuscules (la case).

On peut ensuite spécifier que l'on ne désire que des fichiers normaux. Ceci se fait à l'aide de l'option **-type f**. Les autres types sont **b,c,d,l,p,s** pour **block**, **caractère**, **répertoire**, **liens**, **tuyaux (pipes)**, **sockets**. Le nom de l'utilisateur peut être spécifié par l'option **-user nom**, le groupe avec l'option **-group**. Les dates de modification et de changements, ou d'accès peuvent être testées à l'aide de respectivement **-atime n**, **-ctime n**, **-mtime n**, n étant une durée en journées on le préfixe d'un + pour indiquer que le fichier a été modifié, changé ou lu il y a plus de n jours avant, avec un - comme préfixe on indique que l'opération a été réalisée moins de n jours avant.

La taille peut être spécifiée avec l'option **-size [+ -]n[bckMG]**, ou le +/- indique si la taille du fichier doit être supérieure ou inférieure. Les suffixes "**bckMG**" sont des unités, elles correspondent à 512 octets, 1 octet, 1 ko, 1 Mo et 1 Go.

Tout comme pour la commande **stat -L** l'option **-L** permet de suivre le lien, **-H** permet de faire

l'opposé. L'option **-maxdepth** permet de spécifier la profondeur de la recherche maximale, **-mindepth**, la profondeur minimale. **-xdev** ou **-mount** spécifient de rester dans un même système de fichier.

Les processus

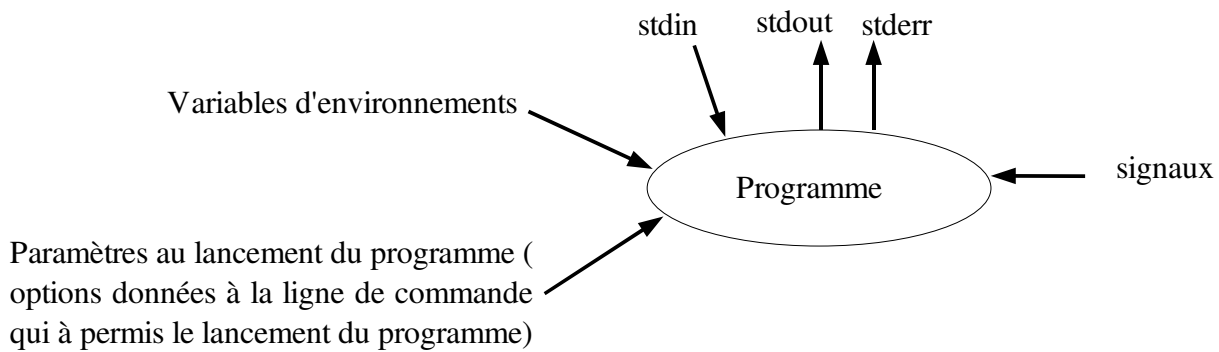
gnu/Linux est un système multiprocessus, c'est à dire qu'il est capable de travailler sur plusieurs tâches à la fois. En réalité ce qui est effectué, c'est que les tâches sont traitées l'une après l'autre. De tels systèmes d'exploitation comprennent de nombreux avantages dont :

- la possibilité d'avoir plusieurs programmes qui tournent au même temps, en effets beaucoup de programmes ne font qu'attendre une entrée de l'utilisateur. A ce titre il est intéressant de rappeler qu'en rien qu'une milliseconde, un ordinateur peut effectuer un grand nombre d'opérations, il est donc parfaitement inutile que l'ordinateur ne fasse rien en attendant les entrées provenant de l'utilisateur.
- partager un ordinateur entre différents utilisateurs, là encore un utilisateur peut effectuer une tâche sur un ordinateur, tout en laissant à disposition cet ordinateur pour d'autres utilisateurs.
- avoir un traitement à la queue-leu-leu des données, ceci peut être réalisé par le mécanisme des tuyaux, qui font que les données sont traités par un programme et la sortie de ce programme est à nouveau traité par un autre programme. Ceci permet en une seule ligne de commande d'effectuer des opérations complexes, comme par exemple, on ne garde que certaines lignes d'un fichier, et on trie ces lignes.
- des applications confinées : une application ne peut pas par accident écrire, ou lire dans la mémoire d'une autre application. Ceci évite des plantages d'une applications à cause d'une autre application. Cette dernière pourrait par exemple écrire dans la mémoire de l'autre application. En mémoire les applications contiennent aussi des localisations des données : cela se présente comme des liens, et ceux-ci peuvent être invalides.

Il est nécessaire de distinguer un processus d'un programme. En effet un programme peut lancer plusieurs processus, c'est ce qui se produit fréquemment avec des programmes comme le serveur internet apache qui lance plusieurs processus pour pouvoir répondre le plus rapidement possible aux requêtes des utilisateurs.

Pour ce qui nous concerne un programme ne va plus être ce qui est lié à une console. On lance un programme dans une console, ce dernier s'y place en avant plan (foreground). Si l'on effectue des frappes au clavier, celles-ci seront envoyées au programme qui tourne en avant plan. Il est possible de lancer ce programme en arrière plan, dans ce cas il ne captera pas la tape au clavier dans la console, cependant les sorties de ce programme s'affichent encore sur la console et peuvent gêner l'utilisateur. Souvent un programme lancé en avant plan répond à une combinaison de touches Contrôle-Z qui arrête le programme, et permet à l'utilisateur d'interagir à nouveau avec l'interpréteur de commandes. En réalité cette combinaison de touche envoie un signal au programme. Voyons maintenant comment un programme peut interagir avec son environnement :

- la machine
- l'utilisateur
- les autres programmes



On représente ici 3 types d'interactions :

- celles qui sont présentes au lancement du programme
 - Les paramètres de la ligne de commande sont un tableau de chaîne de caractères
 - Les variables d'environnement associent un nom et une valeur
- Les interactions avec le terminal
 - les entrées/sorties
 - stdin correspond à l'entrée standard
 - stdout et stderr correspondent à la sortie standard et à la sortie des erreurs, les deux peuvent être distingués par des redirections par exemple.
 - Ces deux entités sont quasiment comme des fichiers pour le programme qui est en train de s'exécuter : on utilise le même type d'instructions pour lire ou écrire dans un fichier. Lire ce que l'utilisateur écrit revient à lire dans un flux particulier : stdin. Afficher un message sur la console revient à écrire pour le programme, dans un flux de sortie particulier : soit la sortie standard, soit la sortie des erreurs.
 - les signaux générés par des combinaisons de touches
 - De nombreuses combinaisons de touches existent, une des plus connues est control-C qui envoie le signal 15 qui a pour effet d'arrêter le programme en cours d'exécution.
- Et les signaux ayant potentiellement une autre provenance que le terminal
 - ceux-ci sont par exemple générés à l'aide de la commande **kill**, qui peut envoyer toute un ensemble de signaux. En plus du signal d'arrêt, un signal particulier (-9) permet arrêter le processus, est n'est pas sujet à un comportement particulier du processus en lui même, le processus est juste arrêté, il ne peut esquiver cette arrêt.

D'autres interactions existent : comme par exemple l'appel à des bibliothèques, qui permettent par exemple au cours de l'exécution du programme d'allouer de la mémoire. Ou encore charger dynamiquement de nouvelles fonctions en mémoire à la demande de l'utilisateur.

Ce qui nous intéresse ici principalement ce sont deux types de ressources utilisées par un programme, le processeur et la mémoire. Celles-ci ne sont pas les seules, il existe d'autres ressources tels que l'accès exclusif à des fichiers, ou d'autres ressources telles que les ressources réseau qu'il peut être intéressant d'observer.

Lorsqu'un programme tourne, soit il est en train d'attendre, c'est ce qu'un interpréteur de commande fait la plupart du temps : l'utilisateur ne lance pas des commandes toutes les millisecondes; du moins quand ce dernier utilise la ligne de commande en exécutant les commandes manuellement. Soit le programme est en train de réaliser une opération de calcul (ou simplement déplacer de la mémoire pour effectuer un tri par exemple) . Au fur et à mesure que l'utilisateur

utilise le programme, il rajoute des données dans ce programme, ouvre des fichiers... ceci demande de la mémoire; celle-ci n'étant pas infinie, le programme la demandera au système quand cela est nécessaire. Il est donc indispensable que chaque utilisateur sur une machine n'utilise pas excessivement la ressource mémoire. Une fois que toute la mémoire électronique est utilisée, un mécanisme d'écriture des programmes et des données chargés en mémoire est mis en place, et celles-ci sont écrits sur un espace particulier qui est soit une partition ou un fichier d'échange, communément appelé swap. L'accès au disque dur est beaucoup plus lent que l'accès à la mémoire, les programmes deviennent très lents une fois qu'ils sont mis sur le disque. Pour s'exécuter ils doivent à nouveau se mettre en mémoire, s'exécutent un petit temps, se remettent sur le disque, passent la main à un autre programme qui a été mis sur le disque pour laisser de la mémoire au précédent... La machine est alors très fortement ralentie, et devient quasiment inutilisable.

Le processus se lance lorsque l'on exécute un programme. Un processus est en général rattaché à une console, ce qui permet la communication avec l'utilisateur. L'utilisateur peut alors décider d'interagir avec ce programme à l'aide du clavier, d'arrêter ce programme à l'aide de la combinaison de touche Contrôle-C, ou encore de bloquer ce programme à l'aide de la combinaison de touche Contrôle-Z. Une fois le programme bloqué, il est nécessaire de le débloquer pour qu'il puisse continuer son exécution; ceci se fait soit avec la commande **bg** qui remet le dernier processus bloqué en arrière plan, ou la commande **fg** qui remet ce même processus en avant plan. La commande **jobs** permet de lister tous les processus qui sont contenus dans la console, à l'aide de cette sortie on peut spécifier aux commandes **bg** et **fg** quels processus on souhaite relancer, ceci se fait de la façon suivante :

```
$ man kill
Contrôle-Z
[4]+  Stopped                  man kill
$ fg
affichage du manuel de la commande kill
Contrôle-Z
$ jobs
[1]  Running                  iceweasel &
[3]-  Stopped                  man ps
[4]+  Stopped                  man kill
$ fg %3
affichage de la page du manuel de la commande ps
```

On constate que durant toutes ces manipulations une autre commande, ici un navigateur internet est en train de tourner, sans que cela ne cause la moindre gêne, ce dernier avait été lancé avec un **&** après la commande, ce qui a pour conséquence de lancer la commande en arrière plan : les entrées sont détachées, ce que l'on tape au clavier est envoyé à l'interpréteur de commande, et non plus au programme tournant en arrière plan.

Gestion des processus

Pour gérer les processus, il est indispensable de pouvoir lister tous les processus qui sont en cours d'exécution sur la machine, ou au moins ceux qui nous appartiennent, c'est à dire ceux que l'on a décidé de lancer. La commande **ps** seule affiche tous les processus qui ont été lancés dans la console où l'on se trouve. Ceci est intéressant seulement dans le cas où l'on a lancé qu'une seule commande. On observe 4 colonnes :

- le numéro du processus
- le numéro de la console à laquelle le processus est rattaché
- le temps processeur qui a été consommé par ce programme.
- La commande qui a permis de lancer ce programme.

Différentes options existent :

-a : affiche les processus qui sont attachés à un terminal et qui sont pas les parents des processus, typiquement on ne verra pas les consoles tels que xterm, ou à l'intérieur du terminal gnome

a : affiche tous les processus qui sont rattachés à un terminal

x : affiche tous les processus même s'ils ne sont pas rattaché à un terminal

ax : la combinaison des deux affiche l'ensemble des processus

-A : affiche tous les processus

Les options sans tiret correspondent à des options style BSD, elles affichent une colonne supplémentaire après le numéro du terminal. Celle-ci correspond à l'état du processus, **Z** signifie que le processus est en attente d'une ressource, ce qui engendre son blocage. On nomme de tels processus des processus zombies. **T** correspond à un processus arrêté, **R** est un processus en train de tourner, **D** un processus bloqué à cause d'un accès aux entrées-sorties. **S** signifie qu'un processus est en attente. D'autres options sont spécifiques au mode d'affichage BSD, telles que :

- **s** : un processus parent d'une session. Ce sont par exemple des processus comme l'interpréteur de commande ou encore **screen** qui est un programme qui émule une console. Ce programme permet de disposer de consoles auxquelles on n'a pas besoin de lien permanent. Ceci est très pratique lorsque l'on a un accès non constant à une machine : on peut alors retrouver son environnement comme on l'avait laissé, avec les programmes qui continuent à tourner, malgré que l'on ait détaché ce qui nous lie à la machine sur lesquelles tournent ces programmes.
- **+** signifie que le processus est en avant plan
- **<** le processus à une priorité élevé
- **N** le processus à une priorité faible.

On peut également disposer d'un affichage long, à l'aide de l'option **-l**. Celle-ci permet de visualiser des informations supplémentaires, telles que :

- l'utilisateur (**UID**), et le groupe (**GID**) auxquels appartient le programme.
- le processus parent (**PPID**)
- priorité du processus (**PRI**)
- le niveau d'utilisation du processeur de la machine, niveau nice (**NI**)
- la quantité de mémoire virtuelle utilisé en ko (**VSZ**)
- la quantité de mémoire qui ne peut pas être mise sur la partition d'échange (**RSS**)
- état du processus pour le noyau (**WCHAN**)

La commande **ps** est surtout utile car elle permet d'avoir le numéro du processus (**PID**) associé à un programme. Cependant elle dispose tout comme la commande **ls** d'options de tri, de sélections, et restrictions d'affichage.

Le même type d'informations peut être obtenu avec la commande **top**, et est mis à jour automatiquement à intervalles réguliers, et trié... . La commande **pstree** permet un affichage sous forme d'arbre des processus.

La priorité d'un processus permet de définir si le processus va tourner plutôt qu'un autre; plus la priorité est haute, et plus un processus aura à disposition du temps processeur au dépend d'un autre d'une priorité plus faible. Les priorités les plus hautes correspondent au numéro de priorité les plus faibles. La priorité peut être assignée au lancement du programme, ou quand celui ci tourne. Les niveaux de priorité sont calculés à partir de niveau de gentillesse (nice) fourni par l'utilisateur, et par l'usage que le processus fait de la machine. Par défaut elle correspond au niveau de gentillesse du processus parent, celui qui lance le processus en question. Les utilisateurs peuvent donner des niveau de gentillesse compris entre celui auquel ils se trouvent et 20. Seul l'administrateur peut diminuer le niveau de priorité. Le fait que les utilisateurs puissent assigner au minimum le niveau de gentillesse du programme ayant permis le lancement du programme qui tourne, assure que les processus parents ont toujours la priorité par rapport aux processus fils. Les deux commandes permettant de changer les niveaux de priorité sont :

- **nice -n [--]n programme**
- **renice [--]n pid**

où pid correspond à un numéro de processus.

Lancement particulier de programmes

- **env**

Parfois on peut souhaiter lancer un programme qui est contenu dans le \$PATH, et il est nécessaire de spécifier le chemin complet de ce programme à cet endroit, dans ce cas on peut faire usage de cette commande. Elle s'utilise souvent dans les scripts shell, on l'utilise de la façon suivante :

```
/usr/bin/env prog
```

cela lancera le programme nommé prog en le cherchant dans le \$PATH. Au début des scripts on précise le programme qui sert à interpréter les commandes contenues dans le script, typiquement de la façon suivante :

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

est la première ligne d'un script bash. Si l'on ne connaît pas l'endroit de cet interpréteur de commande, ou que l'on désire que le script soit portable entre différentes machines, on peut spécifier :

```
#!/usr/bin/env bash
```

Le programme **env** cherchera bash dans la variable d'environnement \$PATH et exécutera le script à l'aide cette commande.

- **exec**

Lorsque l'on lance un processus dans le shell ou un scripte, celui-ci existera en parallèle à l'interpréteur de commande. Ceci peut être gênant, étant donné que l'on utilise de la mémoire, on peut l'éviter en remplaçant le shell par la nouvelle commande, c'est ce que la commande **exec** permet de faire. On peut spécifier le nom du programme que l'on exécute avec l'option **-a** et lancer

le nouveau programme en utilisant un environnement vide à l'aide de l'option **-c**.

```
bash
exec env
bash
exec -c env
bash
exec -a "test" sleep 1d
```

A chaque fois avant de lancer la commande **exec**, on lance un interpréteur de commande **bash** à l'intérieur de celui dans laquelle on se trouve. Au premier lancement on remplace l'interpréteur de commande par **env**. Celui-ci s'exécute : il affiche les variables d'environnement, et se termine : le processus **bash** que l'on avait lancé a aussi disparu. A la seconde exécution de **env**, on constate effectivement que l'environnement est vide. La troisième exécution lance la commande **sleep**, celle-ci se contente d'attendre. On aura renommé l'exécutable **toto**, c'est ainsi que la commande **sleep** connaîtra son propre nom, de cette façon la commande **sleep** sera affichée dans le listing des processus. La durée d'attente est ici d'une journée : **one day**.

- **xargs**

Cette commande permet d'exécuter les arguments qui lui viennent à l'entrée standard : la sortie d'un programme peut être utilisée comme argument d'un autre programme. Ce dernier peut être exécuté éventuellement plusieurs fois quand plusieurs arguments sont à traiter. L'option **-P** indique le nombre de processus à lancer au maximum de façon concomitante, **-n** le nombre d'arguments que chaque commande lancée exécutera. L'option **-p** demandera à l'utilisateur si ce dernier souhaite exécuter la commande. **-I** permet d'effectuer un remplacement. On spécifie exactement où placer le caractère parmi les arguments :

```
echo a | xargs -I{} echo {}b
affichera ab.
```

Envois de signaux aux processus

Envoyer des signaux aux processus permet de leur indiquer un changement d'état, qu'une action a été effectuée et donc qu'il est nécessaire de réagir. Ceci peut par exemple être l'apparition d'une nouvelle tâche à traiter, une relecture d'un fichier de configuration à effectuer, ou simplement se mettre en arrière plan.

Mais cela peut également correspondre à l'arrêt du programme, l'interpréteur de commande dispose d'une commande permettant de générer de tels signaux : **kill**, et **killall**. La première commande nécessite un numéro de processus, la seconde un nom de processus. La seconde va arrêter tous les processus portant ce nom. D'autre part le système d'exploitation peut aussi générer des signaux ou le programme lui-même. Les signaux ne peuvent être envoyés qu'à des processus dont on est le propriétaire.

La commande **kill** s'utilise de la façon suivante :

```
kill [-signal] processus1 [processus2] ...
```

et **killall** de la façon suivante :

killall [-signal] nom_processus

- e permet d'avoir reconnaissance exacte pour des noms longs
- r spécifie une expression régulière
- i interactif
- w attend que tous les processus s'arrêtent
- q mode silencieux
- I ne tient pas compte de la case (différences minuscule/majuscule)
- u ne tue que les processus d'un utilisateur donné

Il existe deux commandes permettant d'avoir des listes de processus : **pgrep** et **pidof**.

pgrep et **pskill** sont également intéressants car ils offrent la possibilité de restreindre la sélection des processus en se basant sur le processus parent, ce qui évite fortement les possibilités d'erreurs.

La commande **killall** n'est pas à confondre avec **killall5**, cette dernière étant utilisée lors de l'arrêt de la machine, elle envoie des signaux à tous les processus dans le but de les arrêter. Cette instruction est utilisée dans les scripts d'arrêt de la machine.

La liste des signaux existants peut être obtenue avec l'option **-l** de la commande **kill**. On remarquera en particulier parmi ces signaux :

15 SIGTERM -> signal d'arrêt par défaut, le processus peut ignorer ce signal
9 SIGKILL -> signal d'arrêt forcé, le processus ne répond pas à ce signal, il ne fait que s'arrêter
2 SIGINT -> signal d'arrêt Contrôle-C envoyé par le clavier

14 SIGALARM -> signal d'alarme : arrête le processus, il s'utilise pour les temporisateurs par exemple.

19 SIGSTOP -> arrêt du processus (ne peut pas être évité)
18 SIGCONT -> relance le processus après qu'il a été arrêté

1 SIGHUP -> engendre la relecture des fichiers de configurations

Les autres signaux correspondent à des exceptions matérielles comme des erreurs de calculs, un excès d'usage de temps machine... Plus d'informations peuvent être obtenues dans la page du manuel 7 sur signal, en tapant : **man 7 signal**.

Détacher un programme de la console

Les programmes qui tournent sur la machine, lorsque l'on ferme la console, les programmes se ferment avec. Ce comportement peut ne pas être souhaité lorsque l'on accède à la machine à l'aide de la commande ssh. Ce programme permet de disposer d'un terminal distant crypté, et qui lorsque la connexion se coupe, la console se ferme également, et donc les programmes s'arrêtent. Il n'y a aucune raison qu'ils ne continuent pas à fonctionner. Deux commandes existent pour faire cela, il s'agit d'un part de la commande **screen** et d'autre part de la commande **nohup**. La première fournit une boîte qui contient plusieurs consoles, et que l'on peut détacher de la console, et rattacher à une autre, et éventuellement même à plusieurs en même temps.

Le programme **screen** s'utilise de la façon suivante :

screen : ouvre un nouveau terminal
control-A C : crée un nouveau terminal
control-A A : passe vers un autre terminal (passe de l'un à l'autre)
control-A n : passe a la console n, n étant un chiffre
control-A D : détache la console
screen -x : attache la console à un autre terminal
screen -ls : liste les sessions **screen** ouvertes

Dans le cas où plusieurs sessions **screen** sont lancées, il est nécessaire de sélectionner celle à laquelle on souhaite se connecter. Bien d'autres commandes existent, elles sont nombreuses, par exemple **screen** dispose d'une ligne de commande. D'autre part une autre commande existe qui est plus légère que **screen** : **nohup**, elle permet de lancer une commande qui est détachée de la console.

Surveillance des ressources

Différentes commandes existent pour surveiller les ressources :

- **free** : permet de connaître la mémoire disponible
- **uptime** : permet de connaître l'usage processeur de la machine
- **ps** permet de trouver les commandes qui consomment beaucoup de ressources

Les ressources peuvent être limitées à l'aide de la commande **ulimit**.

Attendre la fin d'un programme, lancement différé...

On peut vouloir attendre qu'un programme se termine avant d'en lancer un autre, ceci peut se faire à l'aide de la commande de l'interpréteur de commande(s) **wait**. Cette commande ne fonctionnera qu'avec les processus qui ont été lancés dans l'interpréteur de commande. Il est nécessaire de spécifier le numéro du processus après la commande **wait**.

Une commande peut également être lancée quand on est absent, ceci se fait à l'aide de la commande **at**. On peut donner comme exemple :

```
$ at 2030
./program
Control-D
```

Les flux

Les flux correspondent aux entrées-sorties d'un programme. Ils peuvent provenir d'un programme pour rentrer dans un autre programme, ou peuvent être sauvegardés dans un fichier. Ceci permet de sauvegarder les sorties des programmes. On parle de redirection quand on redirige le flux dans un fichier, et de tuyaux (pipe) dans le cas des programmes qui prennent pour entrée la sortie d'un autre programme. On peut aussi diriger un fichier vers l'entrée d'un programme. La redirection est l'opération la plus simple :

```
$ echo "Bonjour tout le monde" > fichier
```

et fichier contiendra "bonjour tout le monde" suivi d'un caractère de nouvelle ligne :

```
$ cat fichier
Bonjour tout le monde
$
```

Supposons que l'on veuille ajouter à ceci un autre message, dans ce cas on pourra simplement faire usage d'une redirection qui ajoute à la fin du fichier :

```
$ echo "Hello World" >> fichier
```

```
$ cat fichier
Bonjour tout le monde
Hello World
$
```

Supposons maintenant que l'on souhaite compter la fréquence des mots de plus de 2 lettres (ou plus exactement commençant par au moins 2 lettres) dans un texte; cela est assez facile :

```
cat 20k_cl.txt | sed 's/\b\s*/\n/g' | egrep "[a-zA-Z]{2,}"
```

le symbole « | » indique de prendre la sortie d'un programme pour la mettre dans l'entrée d'un autre. Ici, la commande **cat** affiche le contenu de 20k_cl.txt, la command **sed** remplace les fins de mots par une nouvelle ligne (une extrémité d'un mot suivie si possible d'autant d'espaces que possible) la commande **grep** est ici pour ne garder que les mots qui commencent par au moins deux lettres; cela évite les lignes vides, ainsi que les symboles de ponctuations. (NB : ce n'est certainement pas la meilleure façon d'utiliser **sed**, mais sûrement la plus simple). Rajoutons quelques étapes :

```
cat 20k_cl.txt | sed 's/\b\s*/\n/g' | egrep "[a-zA-Z]{2,}" | sort |
uniq -c | sort -n > list_mots
```

Voici les mots les plus utilisés sur ce petit extrait de texte :

```
11 to
13 in
14 it
15 that
23 and
35 of
62 the
```

L'interpréteur de commande ne dispose hélas pas de commande permettant d'obtenir les fréquences, pour cela il est nécessaire d'analyser toute la sortie, de faire la somme, et de diviser les décomptes par cette somme. Ceci pourrait par exemple se faire à l'aide d'un langage de script tel que **perl**.

On peut également envoyer un fichier dans un programme :

```
wc -l < /etc/passwd
```

Les données peuvent également être tapées dans la console jusqu'à ce que l'on tape **control-D**
wc -c

toto
Control-D

Il est également possible de spécifier la fin de la chaîne de caractères avec un mot au début d'une nouvelle ligne :

```
cat << FIN > nouveau_fichier
> toto
> FIN
```

Dans ce cas, le nouveau fichier ne comprendra qu'une seule ligne : "toto\n". Le "\n" indique ici qu'il y a une nouvelle ligne. Un mot peut également être redirigé vers un programme

```
cat <<< mot
mot
```

Les programmes peuvent également afficher des erreurs. Dans les redirections que l'on a vues précédemment, ces dernières sont perdues, il est également possible de les rediriger :

créons un script **tcl**, on met dans un fichier texte que l'on nomme **t.tcl** ceci :

```
#!/usr/bin/env tclsh

puts stderr "error"
puts stdout "message"
```

puts est l'instruction qui en **tcl** permet d'écrire, et **stderr**, et **stdout**, correspondent respectivement aux flux de sortie d'erreur, et de sortie standard. On le rend ensuite exécutable par l'utilisateur : **chmod u+x t.tcl** puis on l'exécute :

```
$ ./t.tcl
error
message

$ ./t.tcl | cat -n
error
    1    message
```

On observe clairement que toute la sortie n'a pas été traitée par **cat**, les données ne font que traverser. Ce qu'il est possible de faire c'est de prendre les données du second flux et de les mettre dans le premier :

```
./t.tcl 2&>1 | cat -n
```

Mais on peut encore mieux faire avec les flux : on peut récupérer un flux et le mettre dans un fichier avant de faire des traitements supplémentaires, voici un bout de script :

```
PDB_DB=http://www.rcsb.org/pdb/files/
wget -O ${PDB_DB}2yhx.pdb | tee 2yhx.pdb | grep ^ATOM > 2yhx.pdb.atml
```

on sauvegarde le fichier pdb qui a été chargé, ceci peut bien entendu se faire à d'autres moment lors du traitement. Il est également possible de rediriger dans un fichier spécial que l'on désigne comme tuyau nommé. On y rattache un autre programme, ce dernier est lancé avant, et attend que les données arrivent sur le tuyau nommé. Pour créer un tuyau nommé, on utilise la commande **mkfifo** :

```
mkfifo f
wc -l f
cat -n fichier.txt | tee f
```

On constate ici qu'il est nécessaire de doubler les deux flux si on veut avoir les deux traitements. Un exemple plus utile est :

```
md5sum f
cat fichier.dat | tee f > copie.dat
```

Il existe aussi une commande cat qui fonctionne au travers des réseaux, c'est la commande **netcat**. D'autre part on peut utiliser l'entrée standard comme paramètre d'un programme, ceci se fait à l'aide de la commande **xargs** :

```
ls *txt | xargs wc -l
```

ceci affiche le nombre de lignes des différents fichiers textes. On peut aussi utiliser des pipelines anonymes, ce qui se fait de la façon suivante :

```
$ cat <(echo a)
```

est identique à :

```
$ echo a | cat
```

Cette dernière façon de faire (qui est la première que l'on a vue) a l'inconvénient que l'on ne peut que spécifier un seul fichier, et que de plus elle n'est pas nécessairement compatible avec tous les programmes. L'utilisation de pipeline anonyme permet de passer au delà de ce problème :

```
$ cmp <(echo a) <(echo ab)
/dev/fd/63 /dev/fd/62 differ: byte 2, line 1
```

La commande **cmp**, compare les deux fichiers que l'on a donné en paramètre. Il réalise cette comparaison jusqu'à trouver le premier octet qui diffère entre les deux fichiers. On constate que les deux fichiers sont différents, au second octet (byte en anglais).

Manipuler le texte verticalement

Il peut parfois être intéressant d'avoir les premières lignes d'un fichier texte, pour identifier le fichier par exemple. Ou encore dans le cas de script, connaître l'interpréteur qui est utilisé, pour cela il y a la commande **head**. Par défaut celle-ci affiche les 10 premières lignes, on peut spécifier le nombre de lignes souhaité de la façon suivante :

```
head -n fichier
```

où `n` est un nombre indiquant le nombre de lignes que l'on souhaite afficher. L'option `-c` permet d'afficher que les `n` premiers caractères, par exemple `head -c 1` ne va afficher que le premier caractère.

On peut également vouloir travailler avec la fin d'un texte, dans ce cas il existe la commande **tail** :

```
tail -15 fichier
```

De la même façon qu'avec **head** on peut spécifier un nombre de caractères à prendre à partir de la fin. Une option supplémentaire existe cependant, l'option `-f`, elle permet de scruter l'ajout de caractères au bout du fichier. Ce mode de fonctionnement est particulièrement intéressant car on ajoute souvent des données à la fin d'un fichier (mode d'écriture d'ajout : `append`). La commande **tail** se termine alors soit à la demande de l'utilisateur (Contrôle-C par exemple), soit encore quand le programme fournissant les données sur à l'entrée standard de **tail** se termine. On peut également spécifier l'intervalle de temps entre lequel il doit vérifier que quelque chose à été ajouté, ceci se fait à l'aide de l'option `-s`, le temps est spécifié en secondes.

La combinaison des deux commandes précédentes permet d'extraire une ligne donné d'un fichier. Cela est possible d'une autre façon (pas vraiment plus efficace) :

- numéroter les lignes
- extraire la ligne ayant le bon numéro
- enlever la numérotation des lignes

```
cat -n fichier_text | grep -E "^ +2\b " | cut -f 2-
```

On verra les deux commandes **grep** et **cut** par la suite, ici ce qui nous intéresse, c'est la commande **cat**. C'est une commande très simple : elle se contente d'afficher le contenu d'un fichier, elle dispose de différentes options :

- `-n` : affiche la numérotation des lignes
- `-b` : numérote seulement les lignes non vides
- `-s` : supprime les lignes vides répétées

Ici la commande **grep** permet d'avoir la ligne qui commence par un nombre indéfini d'espaces, suivi d'un 2, après celui-ci il n'y a plus rien. La commande **cut** coupe et garde uniquement les colonnes se trouvant après la première colonne.

Manipuler le texte horizontalement

Ces opérations sont bien plus intéressantes, une ligne pouvant regrouper différentes informations sous forme de colonnes. On pourra par exemple penser au format de sortie de certains tableurs qui est le `cvs` (pour Comma-separated values qui signifie valeurs séparées par des virgules). On peut vouloir garder ou supprimer certaines lignes en fonctions de certains critères, numéroter les lignes, ne garder que certaines colonnes, vérifier des conditions sur certaines valeurs (ou champ), vérifier l'unicité de certaines lignes, trier les lignes en fonction d'un certains champ...

On peut aussi vouloir remplacer un mot par un autre dans des texte, mettre la première lettre en majuscule, remplacer les caractères accentué par une combinaison de caractères (conversion texte en HTML).

On peut regrouper ces opérations de la façon suivante :

- sélections de ligne : **grep**
- garder uniquement certaines colonnes : **cut**
- joindre des fichiers : **paste, join**
- modifier le contenu de lignes : **sed**
- trier un fichier : **sort**
- rendre des éléments uniques : **uniq**

Sélection de lignes

Celle-ci se fait à l'aide de la commande **grep**, on peut par exemple lui indiquer que l'on souhaite conserver toutes les lignes contenant le mot *mot*, tout en ignorant la casse de ce mot à l'aide de l'option **-i**.

```
$ cat fichier | grep -i mot
```

ou toutes les lignes commencent par **ATOM**, dans ce cas on utilisera **grep** de la façon suivante

```
$ cat fichier.pdb | grep ^ATOM
```

Cependant des recherches plus complexes peuvent être entreprises en utilisant des expressions régulières, comme par exemple la recherche d'un mot, c'est à dire une chaîne de caractères qui est contenue entre un début et une fin de mot.

```
$ cat fichier | grep "\bmot\b"
```

Par exemple pour rechercher les doublons :

```
$ echo "Salut Salut" | egrep "\b(\w+)\b\s*\1\b"
```

Ceci utilise une expression régulière. Pour la comprendre, il faut ici savoir que **\b** représente une extrémité d'un mot. L'étoile n'a pas le même sens que pour l'interpréteur de commande, elle signifie répéter entre 0 et n fois l'élément qui précède. Les parenthèses groupent un élément, et le **\1** signifie que cet élément doit apparaître à nouveau. **\w** correspond à un ensemble de caractères possible. **egrep** est une version étendue de **grep**. On peut également l'obtenir en invoquant **grep** avec l'option **-E**. Un autre mode est le mode rapide, celui-ci est parfait pour une recherche d'un simple mot, il est plus rapide car il ne nécessite pas l'interprétation d'expressions régulières.

grep permet également une sélection inverse, avec l'option **-v**. On peut également compter les lignes avec l'option **-c**, et colorer la partie qui a permis de détecter la ligne avec **--color**. On peut arrêter de sélectionner après un certain nombre de comptage, avec l'option **-m**, ceci pourrait également se faire en utilisant **head**, mais de façon moins efficace. L'option **-o** affiche que ce qui a permis la reconnaissance.

Une autre possibilité que **grep** offre c'est de travailler sur plusieurs fichiers, ce mode de fonctionnement est nommé le mode récursif, il s'active à l'aide de l'option **-r**. On fournit alors généralement un répertoire plutôt qu'un fichier. Dans ce mode **grep** affichera le nom de fichier, on

peut aussi lui faire afficher le numéro de la ligne dans le fichier avec l'option **-n**.

??? -A -B options

Manipulations sous forme de colonnes

De nombreux fichiers sont organisés sous la forme de colonnes, on peut par exemple citer les formats de fichier comportant des coordonnées atomiques que sont les fichiers `pdb`, ou encore le fichier contenant les comptes utilisateurs sous `gnu/Linux` : `"/etc/passwd"` . De façon plus générale ceci permet de créer des tables, c'est une forme simplifiée de base de données.

Deux types de commandes existent pour ces manipulations :

- celles qui permettent de récupérer certaines colonnes
- celles qui permettent de joindre des colonnes

La commande **cut** est celle qui permet de récupérer certaines colonnes, ceci peut se faire en spécifiant l'intervalle de caractères, ou encore les champs, à condition de spécifier le séparateur entre les différents champs. Des formats de fichiers comme le format de coordonnées atomiques, se composent de colonnes à position fixe, c'est à dire que l'on trouvera les différentes coordonnées à différentes positions, et ces colonnes se trouvent à la même position pour tous les lignes (à chaque ligne correspond un atome). D'autres formats tels que celui contenant les comptes utilisateurs, sont organisés sous une forme standard, où les colonnes qui sont séparé par des `":"`.

Exemple :

```
$ echo 123456789 | cut -c 2-4,5
235
```

L'option **-c** permet de spécifier que l'on souhaite extraire des caractères à certaines positions. Ces positions sont spécifié sous la forme de groupe et d'intervalles, les groupes sont séparé par des virgules, et les intervalles par un très d'union.

Autre exemple concernant des champs séparé par des espaces

```
$ cat /etc/passwd | grep root
root:x:0:0:root:/root:/bin/bash
$ cat /etc/passwd | grep root | cut -f 6 -d ":"
/root
```

récupère le répertoire perso d'un utilisateur, ici `root`, **-f** permet de spécifier le champ que l'on désire, **-d** permet de spécifier le délimiteur.

une autre solution pour afficher le répertoire personnel d'un utilisateur est :

```
$ echo ~root
```

Une option intéressante est **-s** qui permet d'empêcher les lignes ne contenant pas le délimiteur de s'afficher.

La commande permettant de rattacher des fichiers ensemble est la commande `paste`, celle ci dispose de peu d'options.

```
$ echo a > /tmp/fichier_a
$ echo b > /tmp/fichier_b
$ paste -d : /tmp/fichier_a /tmp/fichier_b
a:b
$ paste <(echo a) <(echo b)
a b
```

Dans le premier exemple, on a utilisé le fichier que l'on vient de créer grâce à des redirections des sorties produite à l'aide des commandes `echo`. Dans le second exemple on a crée des fichiers, ces fichiers sont ensuite utilisés par la commande `paste` qui rassemble les fichiers sous forme de colonnes. On peut utiliser la commande `paste` avec l'option `-d` pour disposer d'un autre séparateur que la tabulation qui est par défaut.

Une autre commande qui permet de rassembler des morceaux est la commande `join`, celle-ci est bien plus perfectionnée que `paste` : elle permet de mettre une condition sur les colonnes que l'on assemble.

L'exemple le plus simple de l'utilisation de cette commande, est :

```
$ join -1 1 -2 1 <(echo 1 a) <(echo 1 z)
1 a z
```

On a rassemblé les deux fichiers (les sorties des commandes entre les parenthèses) en utilisant la première colonne de chacun des fichiers. Comme les colonnes sont les mêmes, on peut spécifier les colonnes sur les deux fichiers, à l'aide de l'option `-j`. L'option `-t` permet de spécifier le séparateur, on peut par exemple utiliser `-t " : "`, pour spécifier que le séparateur entre les colonnes est " : ". L'autre option est `-i` qui permet d'ignorer la case sur les colonnes tests.

Modifier des fichiers à l'aide d'expressions régulières

La commande `sed` permet la manipulation de fichiers en y modifiant du texte. Pour cela elle utilise des expressions régulières qui permettent de sélectionner la partie de texte à modifier; on a ensuite un élément de remplacement :

```
$ echo Bonjour | sed s/ji/j/
```

Dans le cas d'une page web on peut souhaiter mettre en évidence l'erreur, ji n'étant pas une suite de caractères que l'on trouve en français (du moins elle est plutôt très rare), cela correspondra donc très probablement à une erreur.

```
$ echo Bonjour | sed s/ji/\<font color=\"red\">\ji\</font\>/
```

On a mis en place des caractères d'échappements qui permettent d'utiliser des caractères spéciaux dans l'expression régulières, certains d'entre eux sont en réalité pour l'interpréteur de commande, comme par exemple `>` qui est échappé en `\>`.

D'autres expressions régulières plus avancées existent, on rentrera plus dans les détails dans le chapitre suivant :

Expressions régulières

Les expressions régulières sont un outil très puissant pour la manipulation de texte, elle permettent de sélectionner, de remplacer, ou encore d'extraire. Elles sont utilisées dans de nombreux langages de programmation, parmi eux il y a beaucoup de langage des scripts. Dans ce domaine Perl est le langage de référence, intégrant profondément dans le langage les expressions régulières. Elles peuvent aussi être utilisées sur la ligne de commande par l'intermédiaire de **sed**, ou encore de **grep** et directement par l'interpréteur de commande.

Les expressions régulières les plus simples correspondent à une recherche d'un mot dans un texte, une telle recherche n'est cependant pas universelle. On peut souhaiter reconnaître ce mot indépendamment de la présence de majuscule ou de minuscule : on peut vouloir ignorer la case. Pour cela on dispose souvent d'un modificateur tel que **/i** que l'on peut aussi spécifier par une option au programme. On peut aussi vouloir ignorer la case uniquement sur le premier caractère (c'est ce que l'on suppose connu pour l'instant) :

```
cat list_mots | grep "[tT]he"
```

ici cependant des mots contenant "The", tel que simplement "Then" vont aussi être valides, on dit alors qu'ils "matchs". Pour éviter cela il existe deux options :

- on détecte un espace avant et/ou après, cela absorbe un ou deux caractères
- on détecte les extrémité des mots : cela n'absorbe aucun caractères

Par absorber, on entend le faie d'inclure dans ce qui est détecté dans le match, on en vera l'utilité plus tard.

La première solution est la plus simple à mettre en oeuvre :

```
echo -e "The toto\nThen " | grep "The "
```

mais elle ne nous fournit pas toujours ce que l'on souhaite

```
echo -e "The toto\nThen " | grep " The "
```

La vraie solution est de détecter les extrémités des mots, et non pas de sélectionner des espaces inutiles après.

```
echo -e "The toto,\nthen" | grep "\b[Tt]he\b"
```

Deux autres caractères non absorbés (on peut aussi dire mangés) identiques sont disponibles : **^** et **\$** : ils signifient début et fin de ligne, cependant ces deux ne s'appliquent qu'à l'extrémité des chaînes.

D'autre part certains dialectes d'expressions régulières distinguent les débuts et les fins de mots par **"\<"** et **"\>"**.

Dans l'exemple précédent, on a détecté des classes de caractères : celles-ci sont spécifiées entre des crochets : **[]** où l'on insière des caractères. On peut également y spécifier un intervalle de caractères,

un espace, mais on peut aussi exclure certains caractères :

- **[A-Z]** : toutes les lettres majuscules
- **[A-Za-z]** : les majuscules et les minuscules
- **[^)]** tous sauf une parenthèse fermante
- **[^]** pas un espace

Il existe également des classes de caractères qui spécifient un peu tout, ou tous les espaces, certains dialectes d'expressions régulières offrent des classes de caractères supplémentaires, y compris gérant l'UTF (on a par exemple des classes de caractères comprenant tous les "e" avec tous les types d'accents). Celle qui signifie un peu tout est : "." et pour les espaces de façon générale, on a "\s". Une autre classe de caractères correspond aux mots, c'est "\w". Par ailleurs il faut être attentif au fait que les significations des classes de caractères peuvent varier selon le dialecte d'expressions régulières.

Mais souvent on ne souhaite pas détecter un seule caractère, dans ce cas on peut indiquer le nombre de caractères souhaité, ou spécifier au moins un caractères, ou aucun ou un seul.

```
echo aabtt | grep "a*t" -> match
echo aabtt | grep "a+t" -> ne match pas
```

mais on peut être plus quantitatif que cela :

```
echo aatb | grep -E "\ba{2,3}" -> match
```

On peut également capturer des éléments pour vérifier s'ils sont présents par paires; cela permet par exemple de trouver des doublons. Grep ne reconnaît hélas pas \s, on va donc spécifier l'espace dans une classe de caractères ne comprenant pas les caractères composant les mots.

```
echo "a a" | grep -E "(\w+)[^A-Za-z]\1" -> match
```

Cela semble marcher, mais en fait ne fonctionne pas. Voyons dans un premier temps le principe de fonctionnement, \w+ cherche des mots. Ce qui est entre parenthèse capture le mot, le "\1" spécifie que le contenu de ce qui est entre les parenthèses se retrouve à cette position. Entre les deux, aucun élément ne peut faire partie d'un mot. Cette fonctionnalité est appelé le Backreferences. Cependant l'expression régulière ci-dessous ne fait pas ce qu'elle doit faire. Où est le problème ? Le "\w+" ne s'étend pas autant que possible, mais autant que possible tant qu'il y a reconnaissance, et la différence est importante :

```
echo "ha a" | grep -E "(\w+)[^A-Za-z]\1" -> match aussi
```

La bonne solution est de demander à ce que les mots soient des mots entiers :

```
echo "ha a" | grep -E "\b(\w+)\b[^A-Za-z]\b\1\b" -> match pas
echo "ha ha" | grep -E "\b(\w+)\b[^A-Za-z]\b\1\b" -> match
```

NB : cela ne poserait pas de problème de matcher des indicateurs de positions qui ont une taille nulle.

Ces éléments capturés peuvent également servir à réécrire une nouvelle chaîne ou dans des langages comme perl, ils peuvent servir à assigner des variables, de nombreux autres langages permettent de

capturer des éléments de cette façon.

Passons à des modifications :

```
echo "ha a" | sed -r 's/(\w+) (\w+)/\2 \1/'
```

Ceci échange deux mots, et peut également s'appliquer à toutes les paires de mots de chaque ligne d'un texte. Dans certains dialectes d'expressions régulières (Perl notamment) il est possible de détecter des éléments récursifs.

```
sed -r 's/(\w+) (\w+)/\2 \1/g' fichier_text
```

Les parenthèses peuvent avoir un autre usage que d'effectuer des captures, elles permettent également de proposer des alternatives, comme par exemple :

```
domain=`echo $URL | sed -r 's/(http|ftp):\/\/([^\/]*)\/\2/'`
```

Ceci permet d'extraire le nom de domaine du serveur contenu dans une URL, cependant il n'est pas nécessaire de capturer le premier bloc, qui ici ne sert qu'à spécifier une alternative : http ou ftp. Certains langages permettent de spécifier que le groupe est optionnel. Avec (?:, c'est ce que l'on appelle des groupes non capturants. Il existe également des groupes qui font des vérifications avant ou après le texte qui subira la modification, ces derniers ne subiront pas le remplacement. Ces fonctions sont nommées look-ahead et look-behind.

Certains dialectes d'expressions régulières offrent également la possibilité de travailler sur plusieurs lignes, et d'avoir des éléments récursifs; ceci permet de traiter des parenthèses par paires.

Les commandes permettant de comparer des fichiers en entier

La comparaison de fichiers peut être intéressante à faire, soit à des fins analytiques : où l'on cherche à comprendre pourquoi un fichier de configuration est bon et pas l'autre, soit encore pour ne sauvegarder les différences, ce qui permet sur des codes source d'un programme de ne distribuer que les différences, ce qui peut être économique en coûts de distribution de nouvelles versions. En effet, on évite de distribuer un gros programme alors que la modification ne porte que sur une petite partie de celui-ci. Les différentes commandes qui sont expliquées dans cette sections sont :

- **cmp**
- **diff**
- **patch** : permet l'application des différences obtenues avec la commande **diff**.

Les deux méthodes de comparaison ne sont pas forcément les meilleures si l'on souhaite comparer un grand nombre de fichiers entre eux, et si ce que l'on souhaite vérifier, est simplement l'identité ou la différence de fichiers. (on reviendra à cette question plus tard)

La commandes **cmp** est une commande très simple qui se contente de comparer les fichiers entre eux jusqu'à trouver une différence :

```
$ cmp <(echo acf) <(echo abe)
/dev/fd/63 /dev/fd/62 differ: byte 2, line 1
```

De nombreuses options existent à cette commande :

- b** : affiche la première différences en détail
- l** : affiche toutes les différences en détails
- i n** : ignore les n premier octets du fichier
- i n [:N]** : ignore les n premier octets du premier fichier et les N octets du second fichier
- s** : n'affiche rien, modifiera uniquement \$? (la sortie du programme)

La commande **diff** est une commande bien plus perfectionnée, qui permet de créer des différences entre deux fichiers :

soit **a** le fichier suivant à pour contenu:

```
1
2
3 3
b m
zp
```

et **b** le fichier :

```
1
2
b i
3 3
zp
```

La sortie de la commande diff appliquée sur ces deux fichiers sera

```
$ diff a b
2a3
> b i
4d4
< b m
```

On peut utiliser **diff** en spécifiant les fichiers dans l'ordre opposés, dans ce cas la sortie ne sera pas la même, notamment on échange les < par des >, et vice versa. Mais les repères changent eux aussi.

Cette commandes peut ignorer différentes choses lors de la comparaison :

- i** : ignore la case (différences majuscules/minuscules)
- E** : ignore les tabulations
- b** : ignore le nombre d'espaces utilisé
- w** : ignore tous les espaces

Elle peut aussi s'appliquer à toute une hiérarchie de fichiers à l'aide de l'option **-r** : pour activer le mode récursif. L'option **-y** permet d'afficher la sortie sous la forme de deux colonnes ce qui permet de bien mettre en évidence les différences entre les deux fichiers.

Une fois cette sortie générée, on pourra à l'aide de la commande **patch**, à partir de **a** et de cette différences (que l'on nomme généralement un patch) générer b :

```
$ diff a.c b.c > patch.diff
$ patch a.c patch.diff
```

Ceci permet d'appliquer les modifications qui ont été faites pour passer de a.c à b.c, sur le fichier a.c, ce dernier va être modifié pour devenir b.c. De nombreuses autres options existent pour ces deux commandes, on ne rentrera cependant pas dans les détails, on a simplement fait un survol de la commande. Je vous invite à lire la page du manuel, en utilisant :

```
$ man diff
$ man patch
```

Historique des commandes tapées

Connaître les anciennes commandes tapées peut être utile, simplement pour voir si l'on a fait une erreur ou encore pour éviter de devoir retaper toute la commande. L'historique est contenu dans le fichier `~/.bash_history`. On peut remonter aux anciennes commandes à l'aide de la touche flèche vers le haut, et naviguer dans l'autre sens, à l'aide de la touche flèche vers le bas. On peut également réutiliser l'ancienne commande en tapant `!!`, ou encore la commande qui commence par la chaîne de caractères chaîne : `!chaîne`; on peut aussi modifier la dernière commande tapée :

```
$ commande2
$ ^2^1
```

La longueur de l'historique, et ce qui est enregistré, et l'endroit où il est enregistré, peuvent être contrôlés par les variables d'environnement `$HISTCONTROL` `$HISTFILE`

HISTCONTROL peut prendre différentes valeurs, qui peuvent éventuellement être séparées par des virgules :

ignorespace : ne sauve pas les commandes qui commencent par un espace dans l'historique
ignoredups : ignore les commandes qui sont répétées et qui se suivent
ignoreboth : ignore les deux précédents cas
erasedups : supprime les anciennes occurrences de la commande que l'on vient de taper

HISTFILE : contient le lieu où se situe l'historique, si cette variable est supprimée, l'historique ne sera pas sauvegardé.

HISTFILESIZE : taille de l'historique

HISTTIMEFORMAT : format du stockage de l'heure, il est spécifié dans la page du manuel de la commande date.

L'écriture de scripts

L'interpréteur de commande fournit un langage complet. On a vu que les commandes peuvent être exécutées séquentiellement; cependant deux fonctionnalités importantes manquent :

- la possibilité d'exécuter des instructions de façon conditionnelle, mais aussi de pouvoir les répéter (par exemple autant de fois qu'il y a de fichier), c'est ce que l'on nomme le contrôle

du flux d'exécution.

- Un peu de mémoire de façon à conserver une entrée d'un utilisateur, ou pour réaliser des calculs simples, pour manipuler des chaînes de caractères

Donnons un premier exemple :

```
#!/bin/bash
ARCHIVE=~/archive.tar"
DATE=`date "+%Y%m%d%H%M%S"`
bzip2 -9 < $1 > $1.bz2
tar -append -file=$ARCHIVE `pwd`/$1.bz2
```

on constate que pour des raisons de clarté il est intéressant d'utiliser des variables, celles-ci s'utilisent de la façon suivante :

- quand on leur donne une valeur on ne met pas de \$ devant : on dit que l'on assigne la variable
- quand on veut récupérer leur contenu, il est nécessaire de les préfixer d'un \$, ce qui extrait le contenu de la variable.

On utilise ici une variable un peu particulière : \$1, cette variable correspond au premier argument donné au programme, de la même façon \$2 correspond au second argument... D'autres variables particulières existent, nous y reviendront plus tard.

La première ligne est la ligne qui indique quel interpréteur doit lire le script, ici c'est **/bin/bash**. Les deux premiers caractères correspondent au shebang, c'est "#!". Ensuite on assigne les variables ARCHIVE et DATE, puis on compresse le fichier spécifié en premier argument, que l'on ajoute à l'archive **tar**. Pour le shebang c'est le caractère, "#" qui a été retenu car ce caractère correspond aussi au commentaire.

Réaliser des tests

Intérêt des tests

Le programme que l'on a réalisé auparavant est assez intéressant, cependant on n'est pas nécessairement obligé de compresser un nouveau fichier, on peut proposer deux solutions possible :

- Le fichier bzip2 existe déjà, on ne va pas le recréer
- l'archive contient déjà le fichier à ajouter dans la même version, il n'est pas nécessaire de l'ajouter.

Notons cependant que l'on pourrait vérifier les dates de modification pour savoir si l'archive bzip2 est à jour.

Les différents tests qui existent :

Il existe différents types de tests : on peut vérifier l'état d'une variable, on peut également faire des tests sur des fichiers.

Test sur les fichiers :

- a** ou -**e** : vrais si le fichier existe
- b** vrai si le fichier est un fichier de type périphérique block

- c** vrai si le fichier est de type périphérique caractère
- f** vrai si le fichier est de type standard
- h** vrai si un fichier est un lien symbolique
- p** vrai si le fichier est un pipeline
- r** vrai si le fichier est lisible
- w** vrai si l'on peut écrire dans le fichier
- x** vrai si le fichier est exécutable
- S** vrai si le fichier est une socket
- O** vrai si on est le propriétaire du fichier

on a également des tests comparant deux fichiers :

fichier1 test fichier2

test entre paire de fichiers existants : ???

- nt** : vrai si fichier1 est plus récent que le fichier2
- ot** : vrai si fichier1 est plus ancien que le fichier2
- ef** : vrai si les deux fichiers réfèrent à la même chose

test sur les chaînes de caractères :

- z** : vrai si la longueur de la chaîne est 0
- n** : l'inverse : vrai si la longueur de la chaîne n'est pas nulle

différences entre les chaînes de caractères :

- ==** vrais si les deux chaînes sont égales
- !=** vrais si les deux chaînes ne sont pas égales
- <** la chaîne 1 vient avant la chaîne 2 dans l'ordre lexicographique
- >** la chaîne 1 vient après la chaîne 2 dans l'ordre lexicographique

Mise en pratique

Les tests s'utilisent pour exécuter du code conditionnellement dans différents cas :

- exécuter un bout de code de façon unique
- exécuter un bout de code de façon répétée

Voici un exemple simple d'expressions conditionnelles :

```
if [ -e $1 ]; then
    echo "File already existe"
else
    echo "File don't existe"
fi
```

L'informatique est particulièrement efficace réaliser des tâches répétitives, voilà un exemple de boucle :

```
for a in `ls *.pdf`; do
    a=${a%.pdf}
    pdftotext $a.pdf $a.txt
```

done

On utilise ici la manipulation du contenu d'une variable, ici on enlève la partie terminal de la variable. Avant la première exécution de cette commande, il peut être sage de l'essayer avec la commande **echo** plutôt qu'une commande qui va potentiellement écraser des fichiers qui existent déjà. On constate qu'il y a un problème lors de l'exécution de cette commande : elle ne tient pas compte des espaces, ce qui peut être très gênant : **for** découpe et prend tous les mots, et pas toutes les lignes en sortie du ls

Il existe également la boucle **while**, dont le bloc de code est répété tant que le paramètre de cette instruction est à vrai. Dans les deux types de boucles on peut sortir de la boucle à la demande. On peut aussi générer des séquences de nombre à l'aide de la commande **seq** la commande **for** pourra utiliser ces séquences pour boucler dessus.

```
for a in `seq 0 10`; do echo $a; done
```

Il est nécessaire de mettre des ; à la place des lignes que l'on avait utilisées précédemment. Les séquences peuvent également être deux par deux, en utilisant **seq 0 2 10**, où deux est l'incrément. On peut aussi préciser un format à l'aide de **-f**, de la même façon que l'on utilise **printf**.

On peut également utiliser des entrées données par l'utilisateur à l'aide de la commande **read**, comme par exemple :

```
$ read a
```

qui va demander à l'utilisateur de saisir ce qui va devenir le contenu dans la variable **a**

Les fonctions sont des bouts de code qui se comportent comme des instructions, elles permettent d'écrire une unique fois une tâche.

Petite opérations mathématiques

Le shell permet également des calculs simples, on peut les réaliser de différentes façons :

- avec une syntaxe particulière de l'interpréteur de lignes de commande : **\$ ((opération))** ou **\$(opération)**
- à l'aide de la commande shell **let**
- à l'aide de la commande externe à l'interpréteur de commande : la commande **bc**

Dans les deux premiers cas les opérations portent sur des entiers, on ne pourra donc avoir des nombres fractionnaires. Dans le troisième cas on fait appel à un programme externe :

```
a=$((a+5)) # incrémente a de 5  
a=`echo $a+5|bc`
```

L'option **-l** à la commande **bc** permet de fonctionner en mode nombre à virgule, ce qui signifie qu'il fonctionne en interne avec des nombres flottants, et donnera donc le résultat souhaité par exemple quand on réalise une division.

Chargement d'un bout de code depuis l'interpréteur de commande

On peut charger un bout de code provenant d'un autre fichier à l'aide de l'instruction **source**. Ceci permet un chargement dans le même interpréteur de commande, c'est à dire qu'en "**sourcant**" un fichier on peut modifier les variables d'environnements actuelles. Ou encore créer des alias. Ceci se fait généralement au lancement de l'interpréteur de commandes : le fichier `~/ .bashrc` est alors lu et les instructions qu'il contient sont exécutées. C'est pour cela que dans le cas de certains script on préférera les sourcer plutôt que exécuter le script.

Variable particulière

- **\$1... \$n** : les arguments qui ont été fournis au script que l'on a lancé.
- **\$0** : contient le nom du programme qui a été lancé
- **\$BASH_ARGC** : nombres d'arguments qui ont été reçus par le script.
- **\$BASH_ARGV** : un tableau contenant tous les arguments
- **\$BASH_COMMAND** : la commande qui est en train de s'exécuter
- **\$?** : été retourné par le programme qui vient de tourner :
 - true retourne vrais : **true; echo \$? -> 0**
 - false retourne faux : **false; echo \$? -> 1**
- **\$UID** : user id
- **\$EUID** : UID effectif
- **\$USER** : nom de l'utilisateur
- **\$HOME** : répertoire personnel
- **\$RANDOM** : nombre aléatoire
- **\$CDPATH** : répertoire par rapport au quel cd change de répertoire, par défaut uniquement "."
- **\$PATH** : répertoire où se trouvent tous les exécutables
- **\$PPID** : PID du processus parent
- **\$PWD** : contient le répertoire courant
- **\$SHELL** : localisation de l'interpréteur de commande
- **\$PS1 ... \$PS4** : personnalisation de l'invité de commande
- **\$LANG** : langue du système, est utilisé par les programme que l'on exécute

Réception de signaux par un script, et utilisation de verroux

Petits outils utiles :

echo : permet d'afficher une chaîne de caractères

cal : permet d'avoir n'importe quelle calendrier

tac : inverse les lignes d'un fichier : affiche la dernière en premier

sleep t : marque une pause de pendant une durée **t** exprimé en secondes, à moins qu'un suffixe tel que **m**, **h** ou **d** ne soit utilisé respectivement pour **minutes**, **hours** ou **days**

La commande **date** permet d'afficher ou de modifier la date du système, différents formats de date existent, on peut en afficher un standard :

```
date -R
```

```
$ date -R
```

```
Tue, 26 Aug 2008 22:21:16 +0200
```

L'option **-u** permet d'avoir une date UTC, c'est à dire l'heure universelle; de plus elle est sans décalage horaire (c'est l'heure anglaise, sans l'heure d'été).

Différents éléments de la date peuvent être récupérés, comme par exemple :

date +%a

affiche le nom du jour, %F la date entière, %H les heures au format 24 h, %j le jour de l'année, %m le mois, %n une nouvelle ligne, %Y l'année ... (cf manuel de la commande date).

Elle permet aussi d'afficher les dates de modification d'un fichier à ce format **-r** , ou encore une date que l'on donne à l'aide de l'option **-d**.

La commande **echo** nécessite une description un peu plus complète tout spécialement quand celle-ci est utilisée pour l'écriture de scripts. En effet, l'option **-n** évite l'affichage d'une nouvelle ligne après que la chaîne de caractères a été affichée. D'autre part l'option **-e** permet d'interpréter des caractères d'échappement, tels que des nouvelles lignes **"\n"** ou encore des tabulation **"\t"**, et de nombreux autres.

Personnalisation de l'interpréteur de commande

Plusieurs modifications sont possibles, on en verra quelques unes :

Modification du PATH

Le PATH est une variable d'environnement qui contient l'endroit où se trouvent les exécutables, les différents endroits sont séparés par des ":". C'est l'exécutable du premier répertoire qui est utilisé. Ce que l'on utilise en général pour la modifier est :

```
$ echo $PATH
/home/fmunch/perso_bin/:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
$ PATH=~ /other_bin/:$PATH
```

ainsi les exécutables qui se trouvent dans le répertoires **~/other_bin/** seront également utilisés. Si l'on écrit des scripts shell que l'on met dans ces répertoires, et que l'on utilise des commandes portant leur nom, il peut se produire un appel récursif du shell; ceci est à éviter à tout prix, pour cela il est préférable de spécifier les noms complets des exécutables, et non le nom utilisant le PATH.

Ajouts de commandes supplémentaires

Malgré le grand nombre de commandes que l'interpréteur de commande et le système fournissent, celle-ci ne sont parfois pas toujours suffisantes. Parfois on veut également effectuer des opérations simples, qui nécessitent plusieurs commandes, ou des commandes longues à taper.

Une commande qui peut souvent être utile est :

```
find . -iname "*\~" -exec rm {} \;
```

Pour en créer un script il suffit simplement de mettre cette commande dans un fichier ayant comme entêtes **#!/bin/bash**. On peut également demander à l'utilisateur s'il souhaite bien effectuer l'opération de suppression. Un tel mode peut être obtenu par la lecture du variable à l'aide

de la commande **read**, ou on peut encore demander lors de chaque appels à la commande de suppression de vérifier si l'utilisateur veut bien supprimer ce fichier, ceci se fait à l'aide de **rm -i**.

Un autre script qui peut être utile, est un script permettant de se connecter à un serveur rdesktop distant :

```
#!/bin/bash
RDESKTOP=/usr/bin/rdesktop
SERVER=...
USER=...
exec $RDESKTOP -u $USER -k fr $SERVER -g 800x920 130.79.60.1 $@
```

Dans un autre exemple, on va créer un script qui gère le téléchargement de fichier pdb :

```
#!/usr/bin/env bash

#1aar.pdb
WGET="wget"
PDBBASE="http://www.rcsb.org/pdb/files"
PDBDB="/home/fmunch/pdbdb"
if [ ${#1} != 4 ]; then
    echo "this is not a pdb number"
    exit 1;
fi
if [ -r $PDBDB/$1.pdb ]; then
    cp $PDBDB/$1.pdb $1.pdb
else
    $WGET -o /dev/null $PDBBASE/$1.pdb -O $PDBDB/$1.pdb
fi
```

Dans un premier temps ce scripte vérifie que le numéro donné est bien numéro de pdb valide. Une autre modification à effectuer est de convertir en minuscules le nom du pdb, ainsi que de gérer la sortie du wget, dans le cas où ce dernier ne trouve pas le pdb, et vérifier que le fichier n'a pas déjà été stocké.

Le script suivant permet de sauvegarder quand on est dans un interpréteur le répertoire dans lequel on se situe, et le restaurer ultérieurement. Ce script n'en est pas vraiment un : on le charge dans notre propre interpréteur de commande.

```
s :
#!/bin/sh
pwd > ~/.path

j.sh :
read p<~/.path
cd $p

~/.bashrc :
alias j="source j.sh"
```

la dernière ligne doit se trouver dans le fichier d'initiation de l'interpréteur de commande, qui est

`~/ .bashrc`

On peut faire de même, mais avec des variables d'environnements

Forcer les variables d'environnement sur tous les shells de l'utilisateur (y compris pour l'utilisateur root qui a le droit d'envoyer des signaux à tous).

- ajouts d'alias qui permettent d'invoquer plus simplement des commandes
- personnalisation de l'interpréteur de commande
- ajouts d'auto-complétation
- les possibilité de remplacement
- let