Administration Système sous Linux (Ubuntu Server)

Grégory Morel

2022-2023

CPE Lyon - 3IRC / 4ETI / 3ICS

Cours 1

Bash Partie 1 : prise en main

Bash ?

Dans un environnement *serveur*, l'interface avec la machine est (souvent) un *shell* / *interpréteur de commandes*.

Plusieurs variantes ; les principaux :

- `sh` : Bourne Shell, longtemps le shell par défaut sous UNIX,
- `ksh` : Korn Shell, a introduit le contrôle des jobs, les alias, l'historique des commandes...
- `Bash` : Bourne Again Shell, un sh amélioré, et la version par défaut sous Linux
- `zsh` : Un autre shell très populaire

Dans la suite du cours, on utilisera `Bash`.

Bash

Bash est un programme de type REPL, càd une boucle *Read - Eval - Print* [1] :

- *Read* : lit la commande saisie par l'utilisateur,
- *Eval* : exécute cette commande,
- *Print* : éventuellement, affiche un résultat.

Bash est aussi un langage de script, permettant d'automatiser simplement et efficacement des tâches complexes.

1. De la même manière qu'un interpréteur Python

Commandes

Une *commande* est le nom d'un *programme compilé* (ou *binaire*) ou d'un *script* (Bash, Python…).

Il existe aussi quelques commandes *internes* à Bash (ou *builtin*) : ce ne sont pas des « programmes », mais plutôt des commandes d'interaction, avec Bash [1].

La plupart des commandes peuvent prendre des *paramètres* ou *arguments* [2] : par exemple `cat fichier` appelle la commande `cat` avec l'argument `fichier`.

Certains arguments correspondent à des *options*, qu'on peut passer sous forme *courte* (ex. : `cat -n fichier`) ou *longue* (`cat --number fichier`).

1. On retrouve par exemple `cd`, `alias`, `bg` / `fg`…
2. De la même manière que les *fonctions* dans les langages de programmation.

Où trouver de l’aide sur une commande ?

Manuel

Documentation électronique qui décrit le format et le fonctionnement des commandes, des fichiers, d'outils...

Section	Descriptions
1	Commandes utilisateur
2	Appels système (API du noyau)
3	Appels des bibliothèques (fonctions C)
4	Fichiers spéciaux (situés généralement dans `/dev`)
5	Formats des fichiers (ex. : `/etc/passwd`)
6	Jeux, économiseurs d'écran, gadgets...
7	Divers
8	Commandes d'administration

"RTFM !!!"

Manuel

Consulter le manuel : `man` *page_souhaitée*

Informations sur une section : `man 3 intro`

`man -f smail` ou `whatis -r smail` : recherche les pages de manuel nommées *smail* et en affiche les descriptions courtes

`man -k printf` ou `apropos printf` : recherche les pages de manuel comportant le mot-clé *printf* dans leur résumé

`info cat` : doc de la commande *cat* au format *GNU info*

Autres ressources

Beaucoup de programmes admettent aussi une aide "concise" en les appelant avec `--help`, ou `-h` ou encore `-?`

```
adduser --help
```

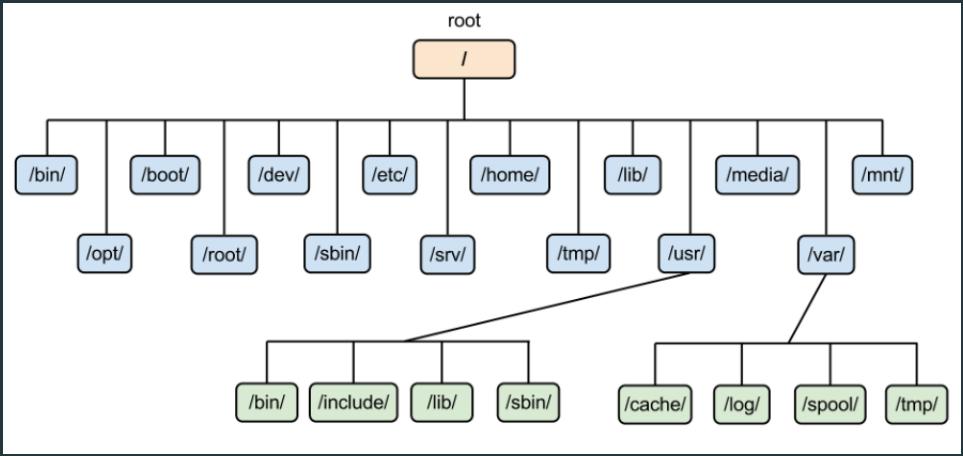
L'interpréteur de commandes (*Shell*) a son propre système d'aide pour les commandes internes : `help commande`

+ livres, web...

Premières commandes : gestion et manipulation des fichiers

Comment sont organisés les fichiers sous Linux ?

Une seule arborescence pour tout le système : la FHS [1]



1. Filesystem Hierarchy Standard

Comment sont organisés les fichiers sous Linux ?

Répertoire	Description
/	Racine de la hiérarchie primaire et du système de fichiers
/bin	Commandes *(binaires)* de base nécessaires à l'utilisation d'un système minimal
/boot	Chargeur d'amorçage *(bootloader)*
/dev	Périphériques *(devices)*
/etc	Fichiers de configuration *(Editable Text Configuration)*
/home	Répertoires personnels des utilisateurs
/lib	Bibliothèques logicielles *(libraries)*
/lost+found	Fichiers récupérés après un crash
/media	Point de montage des médias amovibles (clés USB, CD-ROM…)

Comment sont organisés les fichiers sous Linux ?

/mnt	Point de montage temporaire
/opt	Logiciels optionnels (i.e. non inclus de base)
/root	Répertoire du super-administrateur
/run	Informations sur la session en cours
/sbin	Binaires système et des tâches d'administration
/sys	Informations sur les périphériques, les drivers, le noyau...
/tmp	Fichiers temporaires
/usr	Racine de la hiérarchie secondaire : contient essentiellement les applications utilisateurs
/var	Fichiers divers ou dont le contenu est susceptible de changer en permanence *(variable files)* : logs, mails, sites web...

Pensez à consulter le manuel : `man hier`

Navigation dans le système de fichiers

`cd`	revient au dossier $HOME (dossier de l'utilisateur)
`cd chemin`	va dans le dossier spécifié par *chemin*
`cd ..`	remonte dans le dossier parent
`cd -`	revient au dossier dans lequel on était précédemment
`~`	raccourci pour $HOME `cd ~/musique` ⇔ `cd $HOME/musique`
`pwd`	affiche le dossier courant

Opérations de base sur les fichiers

`ls`	liste les fichiers d'un dossier `-a` : liste aussi les fichiers cachés `-l` : affiche les détails
`ll`	alias pour `ls -alF`
`cat fichier`	affiche le contenu d'un fichier
`more fichier`	affiche page par page
`less fichier`	semblable à `more` mais plus élaboré
`head -n`	affiche les *n* premières lignes
`tail -n`	affiche les *n* dernières lignes
`touch fichier`	modifie l'horodatage de *fichier*, ou crée *fichier* s'il n'existe pas
`tar`	crée une archive de plusieurs fichiers
`gzip / gunzip`	compresse / décompresse
`zcat, zless`	id. `cat` et `less`, mais sur des fichiers compressés

Opérations de base sur les fichiers

`cp`	copie un fichier ou un dossier `-r` : copie récursive
`mv`	renomme un fichier ou le déplace dans un autre dossier
`rm`	supprime un fichier [1] `-r` : suppression récursive
`mkdir`	crée un dossier
`rmdir`	supprime un dossier vide
`ln`	crée un lien sur un fichier (≈ copie synchronisée) `-s` : crée un lien symbolique (symlink) (≈ raccourci)
`file`	détermine le type d'un fichier (indépendamment de son extension)

1. Attention ! Un fichier supprimé est définitivement perdu ! (Pas de "corbeille")

Commandes de manipulations

`wc`	compte le nombre de lignes, de mots et de caractères d'un flux de données (fichiers, résultat d'une commande, etc.)
`sort`	trie la sortie (par ordre alphabétique, inverse, aléatoire…)
`uniq`	supprime les doublons de la sortie
`cut`	coupe chaque ligne de la sortie (selon un nombre de caractères, un séparateur…)
`diff`	compare le contenu de deux fichiers
`iconv`	change l'encodage d'un fichier
`grep`	recherche par expressions rationnelles (regex)
`awk`	commande de manipulation très puissante

Chercher des fichiers

`find`	permet de rechercher à partir d'un point de l'arborescence par nom, date de création / modification, taille Exemples : - `find /bin -name 'admin*'` - `find . -mtime 6` - `find ~ -size +1M`
`locate`	recherche dans une base de données des fichiers indexés ⇒ la recherche est beaucoup plus rapide qu'avec `find`
`sudo updatedb`	force la mise à jour de l'index de `locate`

Redirection des flux d’entrées / sorties

Flux d'entrées / sorties standard

Définition

Un flux d'entrées (resp. de sorties) est un canal de communication par lequel transitent des informations vers (resp. depuis) un programme.

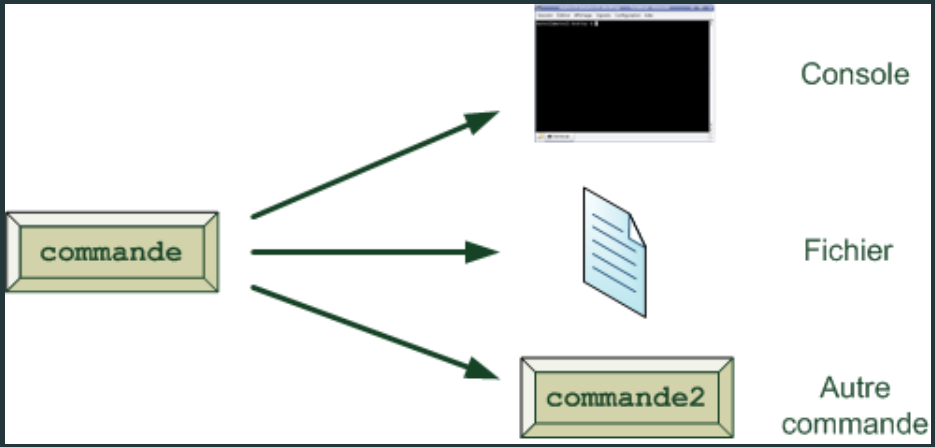
UNIX / Linux fournit trois flux d'entrées / sorties standard :

- l'*entrée standard* (stdin), connectée au clavier,
- la *sortie standard* (stdout), connectée à l'écran,
- la *sortie d'erreur* (stderr), connectée à l'écran.

Sous UNIX / Linux, *tout* est vu comme un fichier, y compris les flux. Chaque fichier ouvert porte un numéro, ou *descripteur de fichier* ; les flux ci-dessus portent les numéros respectifs `0`, `1` et `2`.

Redirection des sorties

Un flux peut être vu comme un tuyau qu'il est possible de déconnecter pour le connecter ailleurs. Par exemple, on peut *rediriger la sortie standard* vers un fichier ou vers une autre commande :



Source : OpenClassRooms

Redirection dans un fichier

L'opérateur > redirige la sortie d'une commande dans un fichier au lieu de l'afficher à l'écran :

```
greg@cpe:~$ ls *.xls
eleves.xls
notes.xls
planning.xls
greg@cpe:~$ ls *.xls > liste_fichiers_Excel.txt
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
```

Si le fichier destination n'existait pas déjà, il est créé automatiquement

Attention ! Si le fichier destination existait déjà, son contenu est effacé et remplacé, sans avertissement (à moins d'activer l'option Bash *noclobber*) !

Redirection dans un fichier

L'opérateur >> redirige la sortie à la fin d'un fichier :

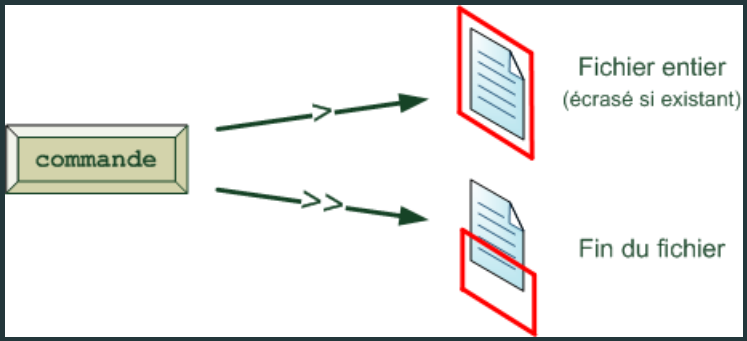
```
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
greg@cpe:~$ ls 2019/*.xls >> liste_fichiers_Excel.txt
greg@cpe:~$ cat liste_fichiers_Excel.txt
eleves.xls
notes.xls
planning.xls
notes2019.xls
```

Si le fichier destination n'existait pas déjà, il est créé automatiquement

Redirection dans un fichier

En résumé :

- `>` : redirige dans un fichier et l'écrase s'il existe déjà ;
- `>>` : redirige à la fin d'un fichier et le crée s'il n'existe pas.



Source : OpenClassRooms

Redirection dans un fichier

Rem. : il n'est pas possible de rediriger un traitement sur un fichier dans le **même** fichier :

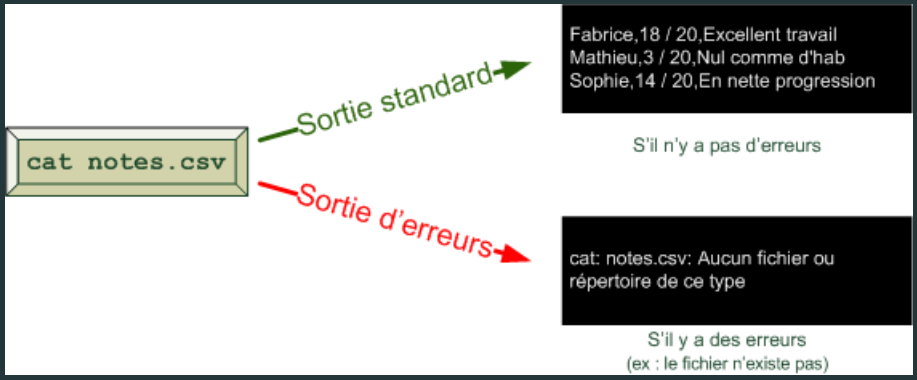
```
greg@cpe:~$ cat f.txt > f.txt
==> Efface le contenu de f.txt !
greg@cpe:~$ cat f.txt >> f.txt
cat: f.txt : le fichier d'entrée est aussi celui de sortie
```

Bash ouvre le fichier de destination avant d'exécuter la commande. Avec le symbole `>`, le fichier est donc supprimé avant d'être traité ; avec le symbole `>>`, `cat` proteste car il devrait écrire à la fin d'un fichier en cours de modification.

Redirection des erreurs

Les messages affichés à l'écran peuvent l'être *via* deux flux différents :

- sortie standard : messages affichés en situation normale
- sortie d'erreur : messages affichés en cas d'erreur



Source : OpenClassRooms

Redirection des erreurs

Pour rediriger les erreurs, on utilise les opérateurs `2>` et `2>>`

A retenir

On retrouve ici le descripteur de fichier associé à la sortie d'erreur.
L'opérateur `>` (resp. `>>`) est d'ailleurs un synonyme de `1>` (resp. `1>>`).

```
greg@cpe:~$ convert *.* -o pdf 2> erreurs.txt
OK : conversion de monsite.html
greg@cpe:~$ cat erreurs.txt
Erreur : impossible de convertir linux.iso
Erreur : impossible de convertir script.sh
```

Souvent, on redirige les erreurs vers le fichier spécial `/dev/null` pour les éliminer

Redirection des erreurs

Si on veut rediriger la sortie standard et les erreurs dans le même fichier, on utilise la syntaxe 2>&1 :

```
greg@cpe:~$ convert *.* -o pdf > sortie.txt 2>&1
greg@cpe:~$ cat sortie.txt
Erreur : impossible de convertir linux.iso
OK : conversion de monsite.html
Erreur : impossible de convertir script.sh
```

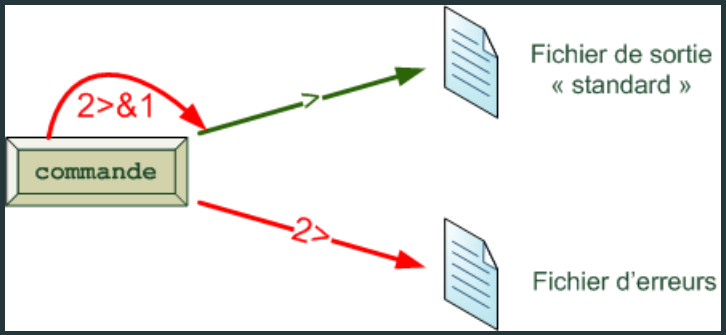
Il n'est pas possible d'écrire 2>>&1. Si on veut ajouter les messages à la fin d'un fichier, on écrira :

```
greg@cpe:~$ convert *.* -o pdf >> sortie.txt 2>&1
```

Redirection des erreurs

En résumé :

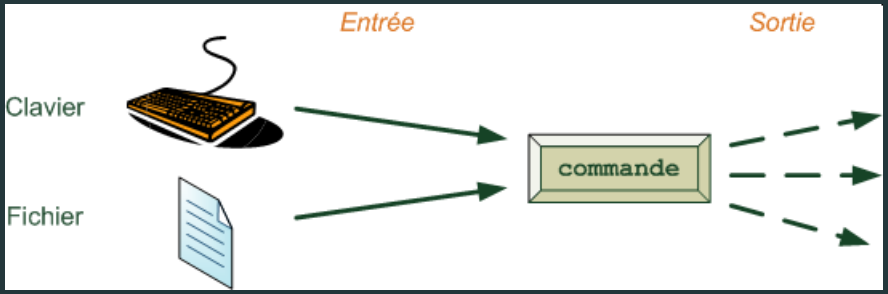
- 2> : redirige les erreurs dans un fichier (écrasé s'il existe déjà)
- 2>> : redirige les erreurs à la fin d'un fichier (créé s'il n'existe pas)
- 2>&1 : redirige les erreurs au même endroit et de la même façon que la sortie standard



Source : OpenClassRooms

Redirection des entrées

L' entrée standard d'une commande est le clavier ; mais il est possible de la rediriger depuis un fichier avec l'opérateur `<` (ou `0<`) :



Source : OpenClassRooms

Redirection des entrées

Par exemple, la commande `tr` lit des chaînes de caractères saisie au clavier et remplace / supprime des caractères :

```
greg@cpe:~$ tr a-z A-Z    # convertit en majuscules
test
TEST
```

Mais on peut rediriger l'entrée vers un fichier, qui alimentera la commande :

```
greg@cpe:~$ cat fichier
hello world!
greg@cpe:~$ tr a-z A-Z < fichier
HELLO WORLD!
```

Bien sûr, on peut rediriger les entrées et les sorties dans la même commande :

```
greg@cpe:~$ cat original
hello world!
greg@cpe:~$ tr a-z A-Z < original > resultat
greg@cpe:~$ cat resultat
HELLO WORLD!
```

Ici, le fichier *original* alimente la commande **tr**, et le résultat est écrit dans le fichier *resultat*.

Redirection des entrées

L'opérateur << permet de créer un *here document* (ou *document en ligne*), c'est-à-dire un document créé directement dans la console (sans passer par un fichier) :

```
greg@cpe:~$ tr a-z A-Z << FIN
> hello
> ceci est un heredoc
> FIN
HELLO
CECI EST UN HEREDOC
```

- La saisie du heredoc s'arrête lorsqu'on tape le délimiteur spécifié après l'opérateur <<.
- Bash dispose aussi de l'opérateur <<< qui lit une seule chaîne (*here string*).

Pipelines

Les redirections permettent d'écrire des commandes complexes comme :

```
ls > fichier && tr [:lower:] [:upper:] < fichier
```

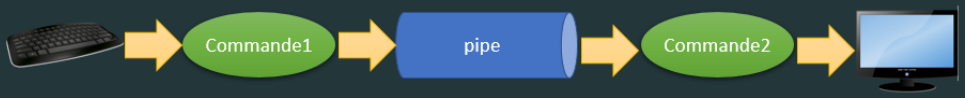
Toutefois, les commandes de ce genre ont plusieurs inconvénients :

- elle crée un fichier intermédiaire qui n'est pas forcément utile ;
- la commande `tr` attend que la commande `ls` se soit terminée avant de commencer son traitement ;
- dans des cas extrêmes, on pourrait manquer d'espace disque pour stocker `fichier`.

Les *pipelines* sont la réponse à ces inconvénients !

Pipelines

Un *pipe* (ou *tube*, ou *tuyau*) est un canal de communication qui relie directement la sortie d'une commande à l'entrée d'une autre commande. Un *pipeline* est un ensemble de commandes reliées par des pipes.



On crée un pipeline en utilisant la syntaxe suivante[1] :

```
commande1 | commande2
```

L'opérateur `|` redirige uniquement la sortie standard ; pour rediriger aussi la sortie d'erreur, Bash propose l'opérateur `|&`, qui est un raccourci pour `2>&1 |`.

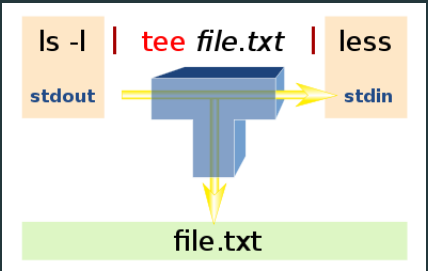
1. On peut bien sûr enchaîner les pipes pour créer des commandes complexes :
`ls | sort -r | tr [:lower:] [:upper:]`

Redirection de flux

`c1 | c2`	relie la sortie de la première commande à l'entrée de la deuxième
`>`	redirige la sortie (mais pas les erreurs) dans un fichier (qui est écrasé s'il existe déjà)
`>>`	redirige la sortie (mais pas les erreurs) à la fin d'un fichier
`<`	prend un fichier en entrée
`<<`	création d'un *here document*
`2>`, `2>>`	redirige uniquement les erreurs dans un fichier
`2>&1`	envoie la sortie d'erreurs sur la sortie standard

Double redirection

La commande `tee` lit depuis l'entrée standard et écrit simultanément sur la sortie standard et un fichier :



Utiliser `tee -a` pour ne pas effacer le contenu du fichier de destination.

xargs

De nombreuses commandes (`head`, `tail`, `sort`, `uniq`, `tr`…) utilisent l'entrée standard. Mais quelques autres (`ls`, `rm`…) ignorent l'entrée standard et n'utilisent que des *arguments* ; il n'est donc pas possible d'utiliser un *pipe* pour rediriger *stdout* vers ces commandes.

Dans ce cas, on utilise la commande `xargs` qui permet de convertir l'entrée standard en *arguments* pour une deuxième commande [1].

Exemple : suppression des fichiers de plus de 14 jours dans le dossier `/tmp` :

```
greg@cpe:~$ find /tmp -mtime +14 | xargs rm
```

1. Par défaut, c'est la commande **echo** qui est utilisée.

Entrée standard vs. arguments

Contrairement à la commande `tr` (et quelques autres), la plupart des commandes qui lisent l'entrée standard acceptent *aussi* leurs entrées *via* des *arguments*.

Exemple :

```
greg@cpe:~$ cat -n < fichier
1 hello world!
greg@cpe:~$ cat -n fichier
1 hello world!
```

Ici, le résultat est le même, mais le principe sous-jacent est légèrement différent :

- dans le 1er cas, le fichier est ouvert par Bash et `cat` lit sont contenu sur son entrée standard ;
- dans le 2nd cas, le *nom* du fichier est transmis au programme `cat`, qui se charge de l'ouvrir et lire son contenu, *parce que le développeur du programme lui a donné cette fonctionnalité*.

Autres commandes à connaître

Utilisation de la console

`help <cmd>`	affiche l'aide de la commande Bash *cmd*
`<TAB>`	autocomplète la commande autant que possible
`<TAB> + <TAB>`	affiche toutes les possibilités de complétion
`CTRL + L`	efface la console
`CTRL + S`	interrompt le défilement d'un résultat trop verbeux
`CTRL + Q`	reprend le défilement
`CTRL + D`	quitte une session
`CTRL + U`	efface la ligne de commande et place le contenu dans le presse-papier
`CTRL + K`	efface ce qui se trouve après le curseur et place le contenu dans le presse-papier
`CTRL + Y`	colle le contenu du presse-papier
`ALT + Fk`	affiche la $k^{ème}$ console virtuelle

Utilisation de la console

`cmd1 ; cmd2`	exécute **cmd1** puis exécute **cmd2** quoi qu'il arrive
`cmd1 && cmd2`	affiche l'historique des commandes tapées (avec un numéro)
`cmd1 !! cmd2`	rappelle la commande **n**

`!!`	rappelle la dernière commande
`history`	affiche l'historique des commandes tapées (avec un numéro)
`!n`	rappelle la commande **n**
`CTRL + R`	recherche dans l'historique
`ECHAP + .`	rappelle le dernier argument de la commande précédente

Multitâches

`commande &`	passe la commande en arrière-plan
`ps aux`	affiche tous les processus en cours d'exécution
`CTRL + Z`	met en pause le processus courant
`bg`	met le processus en pause en arrière-plan
`fg %k`	met le processus n°k au premier plan
`htop`	utilitaire interactif de visualisation des processus
`free`	état de la mémoire
`kill -9 k`	tue (violemment) le processus n°k

Autres commandes utiles

`echo`	affiche ce qui lui est passé en argument
`xargs`	convertit l'entrée standard en arguments pour une commande
`which`	localise une commande, un binaire dans le système de fichiers
`whereis`	localise une commande et sa page de manuel

Arrêter ou redémarrer le système

`halt`	stoppe tous les processus, mais laisse la machine sous tension
`poweroff`	éteint la machine
`reboot`	redémarre la machine
`shutdown`	éteint ou redémarre la machine à une heure donnée