Il existe plusieurs environnements console : les shells

Dès le début de ce livre, j'ai fait la distinction entre les deux environnements très différents disponibles sous Linux :

- l'environnement console ;
- l'environnement graphique.

La plupart du temps, sur sa machine, on a tendance à utiliser l'environnement graphique, qui est plus intuitif. Cependant, la console est aussi un allié très puissant qui permet d'effectuer des actions habituellement difficiles à réaliser dans un environnement graphique.

Je vous avais dit qu'il y avait plusieurs environnements graphiques disponibles (Unity, KDE, XFCE…) mais qu'il n'y avait qu'une seule console. J'ai menti.

La différence est moins tape-à-l'œil que dans le mode graphique (où l'on voit tout de suite que les menus ne sont pas à la même place, par exemple).
La console a toujours un fond noir et un texte blanc, je vous rassure (quoique ça se personnalise, ça). En revanche, les fonctionnalités offertes par l'invite de commandes peuvent varier en fonction du **shell** que l'on utilise.

Les différents environnements console sont appelés des shells, c'est ça ?

C'est ça, en effet. Voici les noms de quelques-uns des principaux shells qui existent.

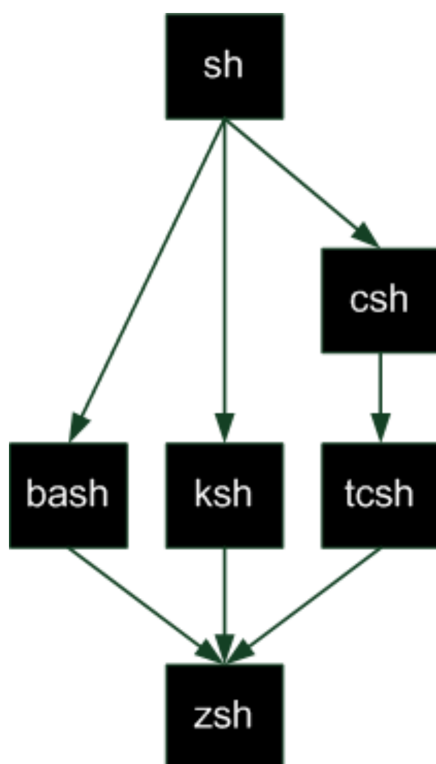
- **sh** : *Bourne Shell*. L'ancêtre de tous les shells.
- **bash** : *Bourne Again Shell*. Une amélioration du *Bourne Shell*, disponible par défaut sous Linux et Mac OS X.
- **ksh** : *Korn Shell*. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- **csh** : *C Shell*. Un shell utilisant une syntaxe proche du langage C.
- **tcsh** : *Tenex C Shell*. Amélioration du *C Shell*.
- **zsh** : *Z Shell*. Shell assez récent reprenant les meilleures idées de bash, ksh et tcsh.

Il y en a quelques autres, mais vous avez là les principaux.

Que faut-il savoir ? Tout d'abord que l'ancêtre de tous les shells est le sh (*Bourne Shell*). C'est le plus vieux et il est installé sur tous les OS basés sur Unix. Il est néanmoins pauvre en fonctionnalités par rapport aux autres shells.

Le bash (*Bourne Again Shell*) est le shell par défaut de la plupart des distributions Linux mais aussi celui du terminal de Mac OS X. Il y a fort à parier que c'est celui que vous utilisez en ce moment sous Linux.
Le bash est une amélioration du sh.

Voici dans les grandes lignes comment ont évolué les shells. Chacun hérite de la plupart des fonctionnalités de son ancêtre (figure suivante).

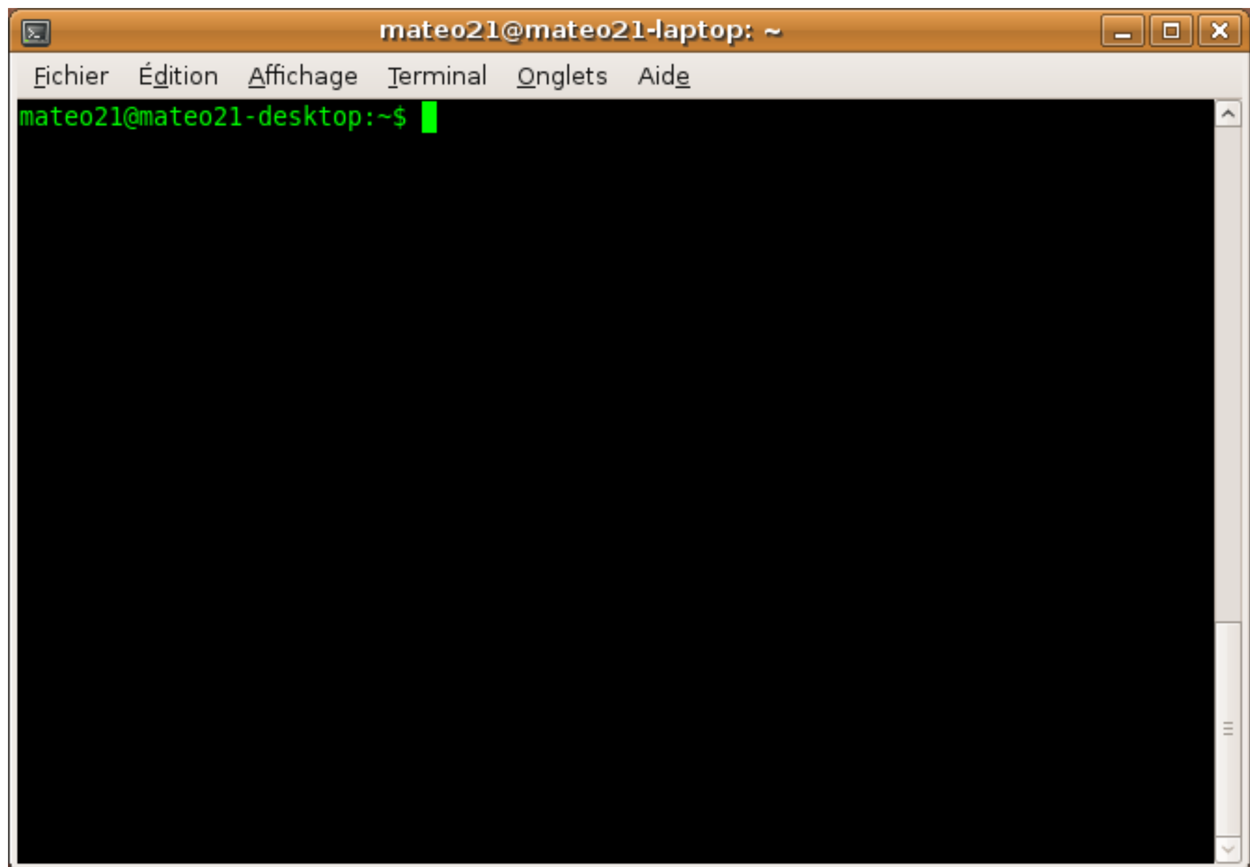


À quoi peut bien servir le sh aujourd'hui alors, si bash est par défaut sous Linux ?

sh reste toujours plus répandu que bash. En fait, vous pouvez être sûrs que tous les OS basés sur Unix possèdent sh, mais ils n'ont pas tous forcément bash. Certains OS basés sur Unix, notamment les OS propriétaires (AIX et Solaris…), utilisent d'autres types de shells ; le ksh y est par exemple très répandu.

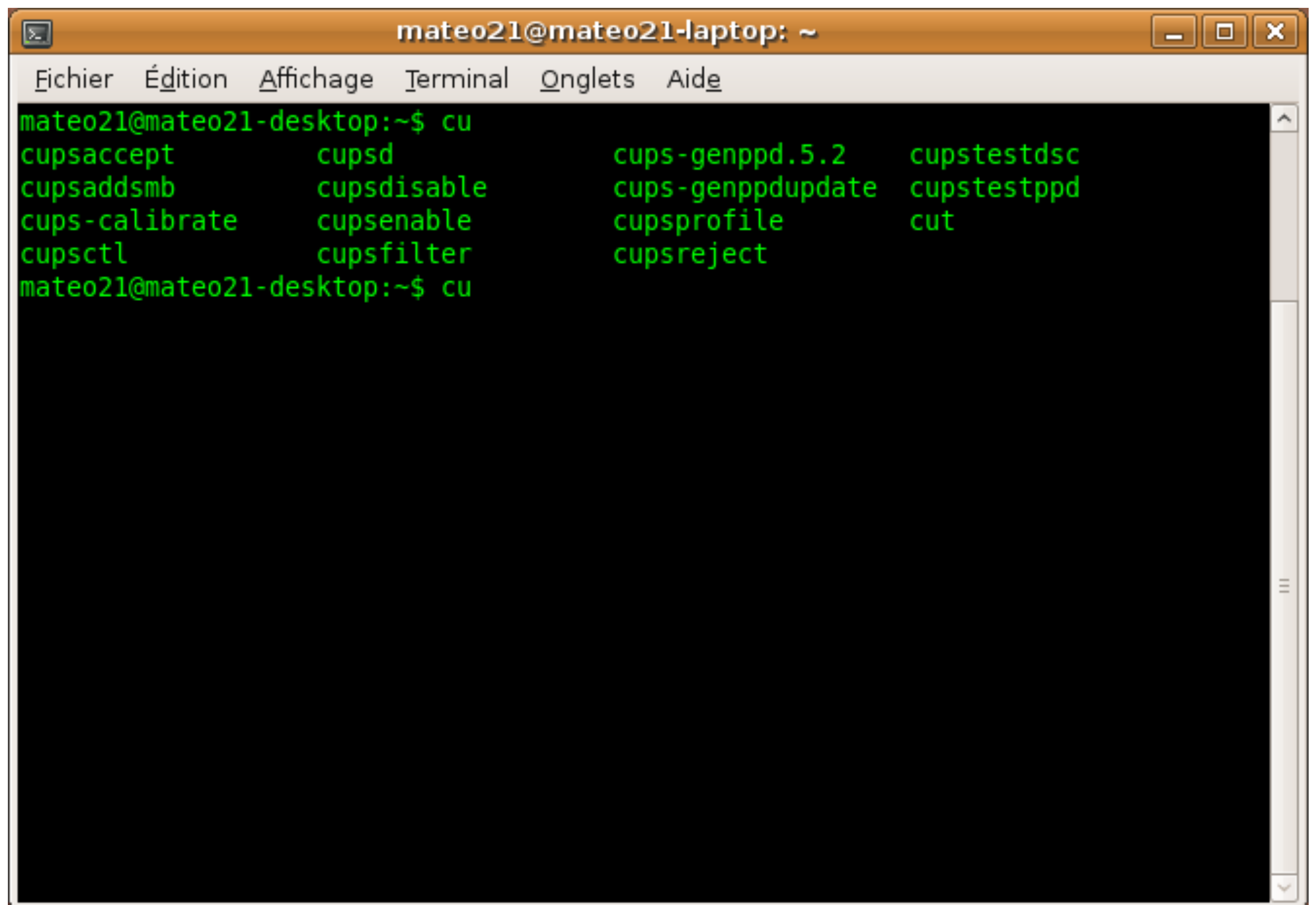
À quoi sert un shell ?

Le shell est le programme qui gère l'invite de commandes. C'est donc le programme qui attend que vous rentriez des commandes (comme l'illustre la figure suivante).



C'est aussi le programme qui est capable par exemple de :

- se souvenir quelles étaient les dernières commandes tapées (vous remontez dans votre historique en appuyant sur la flèche « Haut » ou en faisant une recherche avec un `Ctrl + R`) ;
- autocompléter une commande ou un nom de fichier lorsque vous appuyez sur `Tab` (figure suivante) ;



- gérer les processus (envoi en arrière-plan, mise en pause avec `Ctrl + Z`…) ;
- rediriger et chaîner les commandes (les fameux symboles >, <, |, etc.) ;
- définir des alias (par exemple `ll` signifie chez moi `ls -lArth`).

Bref, le shell fournit toutes les fonctionnalités de base pour pouvoir lancer des commandes.

Souvenez-vous : nous avions modifié un fichier `.bashrc` dans un des premiers chapitres (celui où nous avons appris à utiliser Nano). Le `.bashrc` est le fichier de configuration du bash que Linux vous fait utiliser par défaut. Chaque personne peut avoir son `.bashrc` pour personnaliser son invite de commandes, ses alias, etc.

Installer un nouveau shell

Pour le moment, vous devriez avoir sh et bash installés sur votre système. Si vous voulez essayer un autre shell, comme ksh par exemple, vous pouvez le télécharger comme n'importe quel paquet :

```
# apt-get install ksh
```

Une fois installé, il faut demander à l'utiliser pour votre compte utilisateur. Pour cela, tapez :

```
$ chsh
```

`chsh` signifie *Change Shell*.
On vous demandera où se trouve le programme qui gère le shell. Vous devrez indiquer `/bin/ksh` pour ksh, `/bin/sh` pour sh, `/bin/bash` pour bash, etc.

Quelle importance a tout ceci lorsque l'on réalise un script shell ?

Si je vous parle de cela, c'est parce qu'un script shell **dépend** d'un shell précis. En gros, le langage n'est pas tout à fait le même selon que vous utilisez sh, bash, ksh, etc.

Il est possible d'écrire des scripts sh par exemple. Ceux-là, nous sommes sûrs qu'ils fonctionnent partout car tout le monde possède un shell sh.
Il s’agit toutefois du plus vieux shell, or écrire des scripts en sh est certes possible mais n’est franchement ni facile, ni ergonomique.

Avec quel shell va-t-on écrire nos scripts, alors ?

Je propose d'étudier le bash dans ce cours car :

- on le trouve par défaut sous Linux et Mac OS X (cela couvre assez de monde !) ;
- il rend l'écriture de scripts plus simple que sh ;
- il est plus répandu que ksh et zsh sous Linux.

En clair, le bash est un bon compromis entre sh (le plus compatible) et ksh / zsh (plus puissants).

Création du fichier

Nous allons commencer par écrire un premier script bash tout simple. Il ne sera pas révolutionnaire mais va nous permettre de voir les bases de la création d'un script et comment celui-ci s'exécute. Cela sera donc essentiel pour la suite.

Commençons par créer un nouveau fichier pour notre script. Le plus simple est d'ouvrir Vim en lui donnant le nom du nouveau fichier à créer :

```
$ vim essai.sh
```

Si `essai.sh` n'existe pas, il sera créé (ce qui sera le cas ici).

J'ai donné ici l'extension `.sh` à mon fichier. On le fait souvent par convention pour indiquer que c'est un script shell, mais sachez que ce n'est pas une obligation. Certains scripts shell n'ont d'ailleurs pas d'extension du tout.
J'aurais donc pu appeler mon script `essai` tout court.

Indiquer le nom du shell utilisé par le script

Vim est maintenant ouvert et vous avez un fichier vide sous les yeux.

La première chose à faire dans un script shell est d'indiquer… quel shell est utilisé. En effet, comme je vous l'ai dit plus tôt, la syntaxe du langage change un peu selon qu'on utilise sh, bash, ksh, etc.

En ce qui nous concerne, nous souhaitons utiliser la syntaxe de bash, plus répandu sous Linux et plus complet que sh. Nous indiquons où se trouve le programme bash :

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

Le #! est appelé le **sha-bang**.
`/bin/bash` peut être remplacé par `/bin/sh` si vous souhaitez coder pour sh, `/bin/ksh` pour ksh, etc.

Bien que non indispensable, cette ligne permet de s'assurer que le script est bien exécuté avec le bon shell.
En l'absence de cette ligne, c'est le shell de l'utilisateur qui sera chargé. Cela pose un problème : si votre script est écrit pour bash et que la personne qui l'exécute utilise ksh, il y a de fortes chances pour que le script ne fonctionne pas correctement !

La ligne du sha-bang permet donc de « charger » le bon shell avant l'exécution du script. À partir de maintenant, vous devrez la mettre au tout début de chacun de vos scripts.

Exécution de commandes

Après le sha-bang, nous pouvons commencer à coder.
Le principe est très simple : il vous suffit d'écrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

- `ls` : pour lister les fichiers du répertoire.
- `cd` : pour changer de répertoire.
- `mkdir` : pour créer un répertoire.
- `grep` : pour rechercher un mot.
- `sort` : pour trier des mots.
- etc.

Bref, tout ce que vous avez appris, vous pouvez le réutiliser ici ! 😉

Allez, on va commencer par quelque chose de très simple : un `ls`. On va donc créer un script bash qui va juste se contenter d'afficher le contenu du dossier courant :

```
#!/bin/bash

ls
```

C'est tout !

Les commentaires

Notez que vous pouvez aussi ajouter des commentaires dans votre script. Ce sont des lignes qui ne seront pas exécutées mais qui permettent d'expliquer ce que fait votre script.

Tous les commentaires commencent par un #. Par exemple :

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

```
# Affichage de la liste des fichiers
ls
```

Vous avez sûrement remarqué que la ligne du sha-bang commence aussi par un #… Oui, c'est un commentaire aussi, mais considérez que c'est un commentaire « spécial » qui a un sens. Il fait un peu exception.

Donner les droits d'exécution au script

Nous avons écrit un petit script sans prétention de deux-trois lignes. Notre mission maintenant est de parvenir à l'exécuter.

Commencez par enregistrer votre fichier et fermez votre éditeur. Sous Vim, il suffit de taper `:wq` ou encore `:x`.
Vous retrouvez alors l'invite de commandes.

Si vous faites un `ls -l` pour voir votre fichier qui vient d'être créé, vous obtenez ceci :

```
$ ls -l
total 4
-rw-r--r-- 1 mateo21 mateo21 17 2009-03-13 14:33 essai.sh
```

Ce qui nous intéresse ici, ce sont les droits sur le fichier : `-rw-r--r--`.
Si vous vous souvenez un petit peu du chapitre sur les droits, vous devriez vous rendre compte que notre script peut être lu par tout le monde (`r`), écrit uniquement par nous (`w`), et n'est pas exécutable (pas de `x`).

Or, pour exécuter un script, il faut que le fichier ait le droit « exécutable ». Le plus simple pour donner ce droit est d'écrire :

```
$ chmod +x essai.sh
```

Vous pouvez vérifier que le droit a bien été donné :

```
$ ls -l
total 4
-rwxr-xr-x 1 mateo21 mateo21 17 2009-03-13 14:33 essai.sh
```

Tout le monde a maintenant le droit d'exécuter le script. Si vous voulez, vous pouvez limiter ce droit à vous-mêmes mais pour cela je vous invite à revoir le cours sur les droits car je ne vais pas me répéter.

Exécution du script

Le script s'exécute maintenant comme n'importe quel programme, en tapant « ./ » devant le nom du script :

```
$ ./essai.sh
essai.sh
```

Que fait le script ? Il fait juste un `ls`, donc il affiche la liste des fichiers présents dans le répertoire (ici, il y avait seulement `essai.sh` dans mon répertoire).

Bien entendu, ce script est inutile ; il était plus simple de taper `ls` directement. Cependant, vous devez vous douter que l'on va pouvoir faire beaucoup mieux que ça dans les prochains chapitres.

Vous pouvez déjà modifier votre script pour qu'avant tout chose il vous donne également le nom du répertoire dans lequel vous vous trouvez :

```
#!/bin/bash

pwd
ls
```

Les commandes seront exécutées une par une :

```
$ ./essai.sh
/home/mateo21/scripts
essai.sh
```

Exécution de débogage

Plus tard, vous ferez probablement de gros scripts et risquerez de rencontrer des bugs. Il faut donc dès à présent que vous sachiez comment déboguer un script.

Il faut l'exécuter comme ceci :

```
$ bash -x essai.sh
```

On appelle en fait directement le programme bash et on lui ajoute en paramètre un `-x` (pour lancer le mode débogage) ainsi que le nom de notre script à déboguer.

Le shell affiche alors le détail de l'exécution de notre script, ce qui peut nous aider à retrouver la cause de nos erreurs :

```
$ bash -x essai.sh
+ pwd
/home/mateo21/scripts
```

```
+ ls
essai.sh
```

Créer sa propre commande

Actuellement, le script doit être lancé via `./essai.sh` et vous devez être dans le bon répertoire.
Ou alors vous devez taper le chemin en entier, comme `/home/mateo21/scripts/essai.sh`.

Comment font les autres programmes pour pouvoir être exécutés depuis n'importe quel répertoire sans « ./ » devant ?

Ils sont placés dans un des répertoires du PATH. Le PATH est une variable système qui indique où sont les programmes exécutables sur votre ordinateur. Si vous tapez `echo $PATH` vous aurez la liste de ces répertoires « spéciaux ».

Il vous suffit donc de déplacer ou copier votre script dans un de ces répertoires, comme `/bin`, `/usr/bin` ou `/usr/local/bin` (ou encore un autre répertoire du PATH). Notez qu'il faut être root pour pouvoir faire cela.

Une fois que c'est fait, vous pourrez alors taper simplement `essai.sh` pour exécuter votre programme et ce quel que soit le répertoire dans lequel vous vous trouverez !

```
$ essai.sh
/home/mateo21/scripts
essai.sh
```

En résumé

- Contrairement aux apparences, il existe plusieurs environnements console différents : ce sont les shells. Ce sont eux qui gèrent l'invite de commandes et ses fonctionnalités comme l'historique des commandes, la recherche `Ctrl + R`, l'autocomplétion des commandes…
- Le shell utilisé par défaut sous Ubuntu est bash, mais il existe aussi ksh, zsh, etc.
- Il est possible d'automatiser une série de commandes. On crée pour cela un fichier contenant la liste des commandes à exécuter, appelé *script shell*. On dit que l'on fait de la programmation shell.

- En fonction du shell utilisé, on dispose de différents outils pour créer son script shell. Nous utiliserons ici bash, donc notre fichier de script doit commencer par la ligne `#!/bin/bash`.

- Dans le fichier de script, il suffit d'écrire les commandes à exécuter les unes après les autres, chacune sur une ligne différente.

- Pour exécuter le script (et donc exécuter la liste des commandes qu'il contient) il faut donner les droits d'exécution au fichier (`chmod +x script.sh`) et lancer l'exécution du script avec la commande `./script.sh`.