

Administration Système sous Linux (Ubuntu Server)

Grégory Morel

2018-2019

CPE Lyon

Deuxième partie

Commandes et scripts Bash

Bash ?

Dans un environnement *serveur*, l'interface avec la machine est (souvent) un **shell**

Plusieurs variantes; les principaux :

- **sh** : Bourne Shell, l'ancêtre de tous les shells
- **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

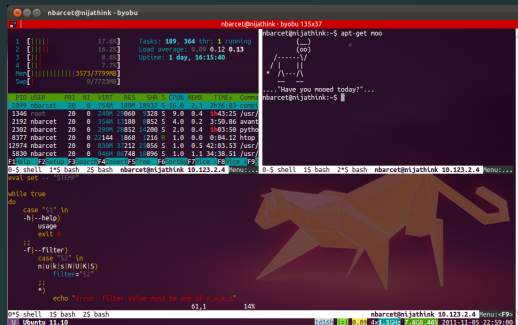
Dans la suite du cours, on utilisera **Bash**.

Travailler avec plusieurs fenêtres

6 consoles virtuelles disponibles, accessibles par **Alt+F1**, **Alt+F2**...

On peut aussi utiliser un **multiplexeur de terminal** :

- **GNU Screen** : ancien, bas niveau
- **tmux** : version moderne de Screen (fenêtres multiples...)
- **Byobu** : surcouche de tmux



Mémento des commandes de base

Utilisation de la console

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

Historique des commandes

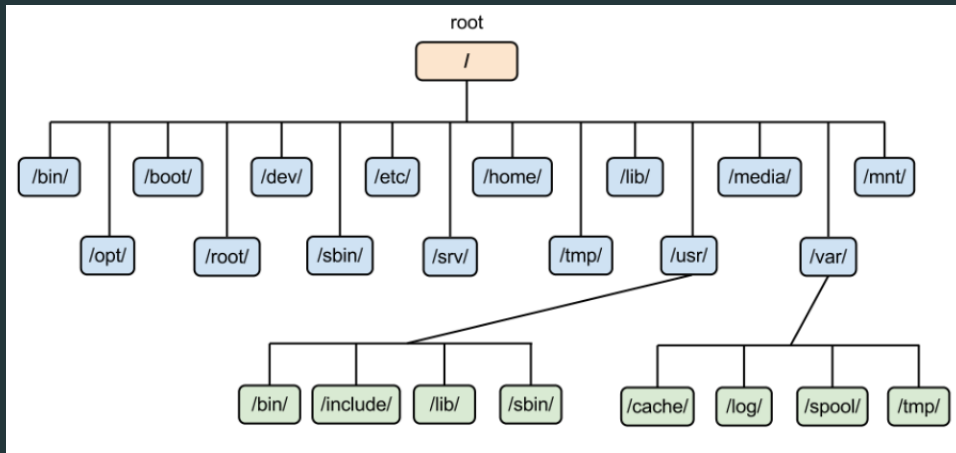
<code>!!</code>	rappelle la dernière commande
<code>history</code>	affiche l'historique des commandes tapées (avec un numéro)
<code>!n</code>	rappelle la commande <code>n</code>
<code>CTRL + R</code>	recherche dans l'historique
<code>ECHAP + .</code>	rappelle le dernier argument de la commande précédente

Navigation dans le système de fichiers

<code>cd</code>	revient au dossier <code>\$HOME</code> (dossier de l'utilisateur)
<code>cd chemin</code>	va dans le dossier spécifié par <i>chemin</i>
<code>cd ..</code>	remonte dans le dossier parent
<code>cd -</code>	revient au dossier dans lequel on était précédemment
<code>~</code>	raccourci pour <code>\$HOME</code> <code>cd ~/musique</code> \Leftrightarrow <code>cd \$HOME/musique</code>
<code>pwd</code>	affiche le dossier courant

Aparté : hiérarchie standard du système de fichiers

Une seule arborescence pour tout le système : FHS¹



1. Filesystem Hierarchy Standard

Aparté : hiérarchie standard du système de fichiers

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

Aparté : hiérarchie standard du système de fichiers

/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 (<i>variable files</i>) : logs, mails, sites web...

Pensez à consulter le manuel : **man hier**

Opérations de base sur les fichiers

<code>ls</code>	liste les fichiers d'un dossier <code>-a</code> : liste aussi les fichiers cachés <code>-l</code> : affiche les détails
<code>ll</code>	alias pour <code>ls -aLF</code>
<code>cat fichier</code>	affiche le contenu d'un fichier
<code>more fichier</code>	affiche page par page
<code>less fichier</code>	semblable à <code>more</code> mais plus élaboré
<code>head -n</code>	affiche les n premières lignes
<code>tail -n</code>	affiche les n dernières lignes
<code>touch fichier</code>	modifie l'horodatage de <i>fichier</i> , ou <i>crée fichier</i> s'il n'existe pas
<code>tar</code>	crée une archive de plusieurs fichiers
<code>gzip / gunzip</code>	compresse / décompresse
<code>zcat, zless</code>	id. <code>cat</code> et <code>less</code> , 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...)
iconv	change l'encodage d'un fichier
grep	recherche par expressions rationnelles (regex)
awk	commande de manipulation très puissante

Chercher des fichiers

<code>locate</code>	recherche dans la base de données des fichiers indexés
<code>sudo updatedb</code>	force la mise à jour de l'index
<code>find</code>	recherche par nom, date, taille...

Redirection de flux

<code>c1 c2</code>	relie la sortie de la première commande à l'entrée de la deuxième
<code>></code>	redirige la sortie (mais pas les erreurs) dans un fichier (qui est écrasé s'il existe déjà)
<code>>></code>	redirige la sortie (mais pas les erreurs) à la fin d'un fichier
<code><</code>	prend un fichier en entrée
<code><<</code>	prend en entrée le clavier au fur et à mesure
<code>2>, 2>></code>	redirige uniquement les erreurs dans un fichier
<code>2>&1</code>	envoie la sortie d'erreurs sur la sortie standard

Multitâches

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

Autres commandes utiles

<code>echo</code>	affiche ce qui lui est passé en argument
<code>xargs</code>	convertit l'entrée standard en arguments pour une commande
<code>diff</code>	compare le contenu de deux fichiers
<code>which</code>	localise une commande, un binaire
<code>whereis</code>	localise une commande et sa page de manuel

Où trouver de l'aide?

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 <code>/dev</code>)
5	Formats des fichiers (ex. : <code>/etc/passwd</code>)
6	Jeux, économiseurs d'écran, gadgets...
7	Divers
8	Commandes d'administration

"RTFM !!!"

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*

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 qu'il propose : `help commande`

+ livres, web...

Variables d'environnement

Variables d'environnement

Les **variables d'environnement** (v.e.) sont des variables de configuration **globales**, utilisées par les programmes pour modifier certains comportements

Exemple : la variable d'environnement **LANG** détermine la langue que les logiciels utilisent pour communiquer avec l'utilisateur

Pour afficher le contenu d'une variable, on utilise **printenv variable**¹ :


```
$ batman@gotham:~$ printenv LANG  
fr_FR.UTF-8
```

1. Si on ne spécifie aucune variable, elles sont toutes affichées

Variables d'environnement

Pour récupérer la valeur d'une v.e., on la fait précéder du symbole **\$**

```
$ batman@gotham:~$ echo USER
USER
$ batman@gotham:~$ echo $USER
batman
```

 La variable est immédiatement remplacée par sa valeur :

```
$ batman@gotham:~$ $USER
La commande « batman » n'a pas été trouvée
```

Variables d'environnement

Pour **créer** une valeur d'environnement, il faut utiliser **export** :

```
$ batman@gotham:~$ export VAR="abcdef" ; printenv VAR  
abcdef
```

💡 Sinon, on crée une *variable de Shell* **locale** (i.e. connue du Shell courant seulement, et donc uniquement pour la session en cours)

```
$ batman@gotham:~$ VAR="abcdef" ; printenv VAR  
$ batman@gotham:~$  
$ batman@gotham:~$ echo $VAR  
abcdef
```

💡 La commande **set** liste **toutes** les variables (locales ou d'environnement) du shell courant

Variables d'environnement

Pour **modifier** la valeur d'une variable **existante** :

```
$ batman@gotham:~$ printenv LANG  
fr_FR.UTF-8  
$ batman@gotham:~$ LANG="en_US.UTF-8"  
$ batman@gotham:~$ printenv LANG  
en_US.UTF-8
```

⚠ Ne pas mettre d'espace entre la variable et le signe '='

⚠ La modification est **temporaire**, et n'est effective que pour la session courante

Variables d'environnement

Pour que la création ou la modification d'une variable soit **permanente**, il faut ajouter la commande au fichier `~/.bashrc`¹ qui est lu à chaque démarrage de bash

⚠ Ce fichier n'est lu qu'au *démarrage* de bash ; pour forcer bash à le relire immédiatement, il faut le **sourcer** :

```
source ~/.bashrc
```

💡 **.bashrc** ne concerne que l'utilisateur courant; si on veut toucher *tous* les utilisateurs, il faut modifier le fichier *global* `/etc/bash_bashrc`

1. Les fichiers dont le nom se terminent par **rc** sont très souvent des fichiers de configuration

Variables d'environnement

Exemple : la variable **PATH**

Elle indique à **bash** où trouver les commandes tapées par l'utilisateur.

```
$ batman@gotham:~$ printenv PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
```

Quand on tape une commande, **bash** regarde successivement dans chacun de ces dossiers jusqu'à la trouver.

Si on veut utiliser une commande se trouvant dans un dossier ne figurant pas dans `$PATH`, on a deux possibilités :

- indiquer à chaque fois le chemin complet vers commande
- modifier la variable `PATH` :

```
PATH=$PATH:~/mondossier
```

Pour que le changement soit permanent, on rajoute cette ligne à la fin du fichier `~/.bashrc`

Pour **supprimer** une variable, on utilise la commande **unset** :

```
$ batman@gotham:~$ export VAR=toto ; printenv VAR  
toto  
$ batman@gotham:~$ unset VAR ; printenv VAR  
$ batman@gotham:~$
```

Introduction aux scripts Bash

Éditeurs de texte

- **nano** : rudimentaire, mais peut afficher la coloration syntaxique; utile pour éditer rapidement des fichiers textes
- **vim** : éditeur très puissant, extensible, peut être utilisé comme un véritable environnement de développement
- **emacs** : concurrent de vim ; pas installé par défaut, très puissant, extensible et personnalisable

⚠ Ces éditeurs sont très différents du "bloc-note" de Windows, en particulier au niveau des raccourcis clavier

<https://doc.ubuntu-fr.org/{nano|vim|emacs}>

Hello World!

Un script Bash commence toujours par la ligne suivante¹ :

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

La séquence `#!` est appelée **shebang** : elle indique à l'OS que le fichier est un script. Juste derrière, on indique le chemin vers un **interpréteur** capable d'exécuter le script

La suite du script est la liste des commandes à exécuter :

```
echo "Hello, World!"
```

Par convention, on donne une extension **.sh** aux scripts

1. Non obligatoire, mais permet de s'assurer que le script est exécuté par le bon interpréteur

Hello World!

On a ensuite besoin de rendre le script **exécutable** :

```
$ batman@gotham:~$ chmod u+x hello.sh
```

On peut enfin exécuter notre script :

```
$ batman@gotham:~$ ./hello.sh  
Hello, World!
```

? Que signifie **./** avant le nom de la commande?
Pour quelle raison Linux impose-t-il ce mécanisme?

Les guillemets

Guillemets simples : chaîne littérale (le contenu n'est pas interprété)

```
var=toto
echo 'contenu de var : $var'
-----
contenu de var : $var
```

Guillemets doubles : chaîne interprétée

```
var=toto
echo "contenu de var : $var"
-----
contenu de var : toto
```

Exécuter une commande

Pour exécuter une commande dans un script, on la place dans `$(...)`:

```
res=$(pwd)
echo "vous êtes dans le dossier : $res"
-----
vous êtes dans le dossier : /home/batman
```

Demander de saisir une valeur

On peut demander à l'utilisateur de saisir des valeurs avec **read** :

```
read -p 'Saisissez deux valeurs a et b : ' a b
```

La commande **read** a de nombreuses options intéressantes :

- **-p** : affiche un message
- **-t** : limite de temps
- **-s** : n'affiche pas le texte saisi (pour des mots de passe par exemple)
- **-n** : limite le nombre de caractères

On peut aussi passer des paramètres directement sur la ligne de commande :

```
$ batman@gotham:~$ ./mon_script.sh param1 param2
```

Dans le script, les variables suivantes permettent de manipuler les paramètres :

- \$# : nombre de paramètres
- \$0 : nom du script
- \$1 : premier paramètre
- \$2 : second paramètre
- etc.

Shift

Imaginons maintenant le cas suivant :

```
$ batman@gotham:~$ ./mon_script.sh param1 param2 ... param15
```

Pour éviter d'avoir à gérer 15 variables \$1...\$15, on peut utiliser **shift** :

```
while ((" $#")); do  
    echo $1  
    shift  
done
```

```
-----  
$ batman@gotham:~$ ./mon_script.sh param1 param2 ... param15  
param1  
param2  
...  
param15
```


Paramètres spéciaux

`$*` permet de récupérer l'ensemble des paramètres sous la forme d'un seul argument :

```
for param in $*; do  
    echo $param  
done
```

```
-----
```

```
$ batman@gotham:~$ mon_script abc def ghi  
abc  
def  
ghi
```

Paramètres spéciaux

⚠ Problème : si un paramètre contient des espaces, il est scindé en plusieurs arguments :

```
for param in $*; do
    echo $param
done

echo $1
-----
$ batman@gotham:~$ mon_script "abc def ghi"
abc
def
ghi
abc def ghi
```

Paramètres spéciaux

Et si on encadrait `$*` par des guillemets ?

```
for param in "$*"; do
    echo $param
done
-----
$ batman@gotham:~$ mon_script "abc def ghi"
abc def ghi
```

Cette fois ça marche ! Enfin, presque... :

```
for param in "$*"; do
    echo $param
done
-----
$ batman@gotham:~$ mon_script abc def ghi
abc def ghi
```

Paramètres spéciaux

En effet, on ne peut plus distinguer les différents paramètres :

```
#!/bin/bash
ls -l "$*"
-----
$ batman@gotham:~$ touch toto titi
$ batman@gotham:~$ ls toto titi
toto titi      <-- ça fonctionne
$ batman@gotham:~$ mon_script toto titi
ls: erreur 'toto titi': fichier ou dossier inexistant
```

Paramètres spéciaux

Il nous faudrait un paramètre spécial qui :

- fournisse autant d'arguments qu'à l'origine lorsqu'il est entre guillemets
- protège chacun des arguments par des guillemets (pour préserver les espaces)

Ce paramètre existe : c'est `$@`

```
#!/bin/bash
ls -l "$@"
-----
$ batman@gotham:~$ mon_script toto titi
-rw-r--r-- 1 batman batman 0 févr.  1 11:39 titi
-rw-r--r-- 1 batman batman 0 févr.  1 11:39 toto
```

Comment faire pour afficher le *i*-ème paramètre, quand *i* est elle-même une variable ?

```
for i in $(seq 1 3); do
    echo $i
done
-----
$ batman@gotham:~$ mon_script a b c
1
2
3
```

Solution : on **déréférence** la variable, avec la syntaxe **`${!i}`** :

```
for i in $(seq 1 3); do
    echo ${!i}
done
-----
$ batman@gotham:~$ mon_script a b c
a
b
c
```


Tableaux

Un tableau se définit comme ceci :

```
tab=(elem1 elem2 elem3 ...)
```

Et on accède à une valeur comme ceci :

```
${tab[0]}
```

 Les accolades sont obligatoires (pourquoi?); par ailleurs les indices commencent à 0

On peut attribuer directement une valeur à une case :

```
tab[5]=42
```

Et afficher tout le contenu du tableau :

```
$tab[*]
```


Conditions

Réaliser un test :

```
if [ $nom = "Astérix" ]; then
    echo "Idéfix"
elif [ $nom = "Tintin" ]; then
    echo "Milou"
else
    echo "Autre"
fi
```

⚠ Les espaces entre les crochets et le test sont obligatoires !

On peut combiner plusieurs tests à l'aide des opérateurs logiques classiques :

- **&&** : ET
- **||** : OU
- **!** : NON

Conditions

Tests possibles sur des chaînes de caractères :

Condition	Signification
<code>"\$chaine1" =¹ "\$chaine2"</code>	Teste si les deux chaînes sont identiques (sensible à la casse)
<code>"\$chaine1" != "\$chaine2"</code>	Teste si les deux chaînes sont différentes
<code>-z "\$chaine"</code>	Teste si la chaîne est vide
<code>-n "\$chaine"</code>	Teste si la chaîne est non vide

1. Pour les habitués des langages de programmation, il est possible d'utiliser `==`

Conditions

Tests possibles sur des nombres :

Condition	Signification
<code>\$num1 -eq \$num2</code>	Teste si les deux nombres sont égaux
<code>\$num1 -ne \$num2</code>	Teste si les deux nombres sont différents
<code>\$num1 -lt \$num2</code>	Teste si $\text{num1} < \text{num2}$
<code>\$num1 -le \$num2</code>	Teste si $\text{num1} \leq \text{num2}$
<code>\$num1 -gt \$num2</code>	Teste si $\text{num1} > \text{num2}$
<code>\$num1 -ge \$num2</code>	Teste si $\text{num1} \geq \text{num2}$

Boucles

Boucle **while** :

```
while [ test ]  
do  
    echo 'Action en boucle'  
done
```

Boucle **for** sur une liste :

```
for fichier in $(ls)  
do  
    cp $fichier $fichier.bak  
done
```

Boucle **for** sur une suite de nombres :

```
for i in $(seq 1 10)  
do  
    echo $i  
done
```

Calcul numérique

Les opérations arithmétiques sont réalisées par l'opérateur `((...))` :

```
echo $((2 + 3 * (5 ** 2) ))
```

```
-----
```

```
77
```

⚠ Cet opérateur ne travaille qu'avec des entiers. Pour des opérations plus complexes, on utilise `bc` (*bash calculator*), qui nécessite une syntaxe particulière :

```
LC_NUMERIC=C          # pour gérer le point décimal
```

```
...
```

```
half=$(echo "1 / 2" | bc -l)
```

```
printf '%.3f\n' $half    # affichage avec 3 décimales
```

```
-----
```

```
0.500
```

Fonctions : syntaxe

Il existe deux syntaxes pour déclarer une fonction en bash :

```
ma_fonction () {  
    <instructions>  
}
```

ou :


```
function ma_fonction {  
    <instructions>  
}
```

? Comment passer des paramètres avec la deuxième syntaxe ?

Fonctions : paramètres

En fait, en bash, les paramètres des fonctions se passent et se récupèrent comme sur la ligne de commande :

```
function ma_fonction {  
    echo $1 $2  
}  
...  
ma_fonction arg1 arg2
```

 On doit toujours définir une fonction **avant** de l'utiliser

Fonctions : valeur de retour

Les fonctions ne peuvent renvoyer qu'un entier, correspondant à un *statut* :

- **0** si la fonction s'est terminée normalement
- **un entier positif** sinon

Pour exploiter le résultat d'une fonction, on utilise la *substitution de commande* :

```
lines () {  
    cat $1 | wc -l  
}  
  
echo "Le fichier $1 contient $( lines $1 ) lignes."
```