Bash

Anne Cadiou, **Laurent Pouilloux**,
Francis Chavanon, Gilles Galboud, Laurent Carrel

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et Séminaires Pour l'Informatique et le Calcul Scientifique
PMCS2I
Lundi 25 janvier 2021

CC BY SA

C'est quoi ?

`bash` est un langage de **shell**.

Le **shell** est un interpréteur : chaque commande est lue syntaxiquement et exécutée dans un **terminal**

Un **terminal** est programme, émulateur de console permettant d'exécuter des commandes. C'est une interface en ligne de commande (CLI), par opposition aux interfaces graphiques utilisateur (GUI), dans lesquelles les exécutables sont lancés pas un clic de souris.

La console est le programme qui affiche des lignes sur votre écran lorsque vous démarrez votre ordinateur en l'absence de serveur graphique.

Terminal, shell et ligne de commande

Les commandes qui sont saisies au calvier dans un terminal sont interprétées (exécutées) par le shell. Leur syntaxe respecte celle du langage de shell configuré par défaut dans le terminal.

Exemple de terminaux

- rxvt-unicode
- terminator
- xterm
- gnome-terminal
- lxterminal
- XFCE terminal

Se lance sur votre portable linux sous gnome en saisissant la commande dans `ALT + F2` ou en tapant `CTRL+ALT+t`

Pourquoi utiliser la ligne de commande ?

Avantages

- rapide
- permet l'automatisation
- léger
- adapté au travail à distance
- traçable
- reproductible

Inconvénients

- peu intuitif

  mais ça s'apprend vite !
- nécessite de programmer

  ... mais c'est le but, non ?

En pratique on s'en sort facilement avec une petite quinzaine de commandes à connaître... le reste vient après par un effet boule de neige.

Interpréteurs de commande

- shell (interface entre l'utilisateur et l'OS)
- application (exécutable) chargée d'interpréter les commandes des utilisateurs et de les transmettre au système
  - sh (Bourne shell)
  - csh (C shell)
  - **bash** (Bourne again shell)
  - ksh (Korn shell)
  - tcsh (Tenex C shell)
  - zsh (Zero shell)

Actuellement, par défaut on trouve le plus souvent `/bin/bash`

Utilisation du shell

- Se lance dans un terminal : fenêtre avec une invite de commande (prompt)

  `login@machine$`

- Les commandes sont saisies au clavier à la suite du prompt

  `login@machine$ ls -l`

- Manuel associé à la commande :

  `login@machine$ man ls`

- `exit` quitte le terminal
- `CTRL+c` stoppe la commande en cours
- `CTRL+d` termine l'entrée standard

Fichier de configuration associé :
`.bashrc`

Navigation dans le terminal

- flêches haut et bas pour naviguer dans l'historique
- `TAB` permet en bash de compléter automatiquement le nom de la commande ou un chemin vers un fichier
- `history` donne l'historique des commandes lancées en ligne de commandes

```
 999 30/03/17 06:08:00 ls -lrt
1000 30/03/17 06:08:03 exit
1001 30/03/17 07:44:35 history
```

- `!hi` rappelle la dernière commande commençant par `hi` utilisée précédemment
- `!1001` rappelle la commande 1001
- `CTRL+a` ramène au début de la ligne de commande
- `CTRL+e` va à la fin de la ligne de commande
- `CTRL+r` permet de rechercher une commande déjà exécutée

Les entrées-sorties standards

Lors de l'exécution d'une commande, un processus est créé
Celui-ci va ouvrir trois flux :

- stdin entrée standard (clavier, par défaut), identifiée par l'entier 0 (descripteur)
- stdout sortie standard (écran, par défaut), identifiée par 1
- stderr sortie d'erreur standard (écran, par défaut), identifiée par 2

Les redirections

Il est possible de rediriger les flux d'entrée-sortie par des opérateurs spécifiques :

- `>` redirige la sortie standard (par exemple dans un fichier)
- `<` redirige l'entrée standard
- `>>` redirige la sortie standard et concatène
- `> &` redirige les sorties standard et d'erreur
- `|` redirection de la sortie standard vers l'entrée standard (pipe)

Processus

- `CTRL+z` suspend le processus (fonctionnalité du shell)
- `bg` relance le processus en pause en tâche de fond et le détache du terminal (`&`)
- `fg` montre le processus attaché
- `cmd &` lance la commande `cmd` en tâche de fond

nohup, screen, tmux

permet que le programme continue de tourner après déconnexion (sauf si on éteint la machine...)

```
nohup mon_programme &
```

Multiplexeur de terminaux :

`screen` ou `tmux`

permet

- d'ouvrir plusieurs terminaux en un seul
- de détacher ces terminaux même si on se déconnecte
- de les réattacher ensuite pour reprendre où on en était

`screen -ls` liste les terminaux en cours

`screen -S Aspics` nomme le terminal

`screen -r Aspics` réattache (après détachement)

`CTRL+a c` création

`CTRL+a n` passe au terminal suivant (dans la liste)

`CTRL+a d` détache le terminal

`tmux` est du même genre et utilise `CTRL+b c`...

Script

Fichier en bash permettant d'exécuter une série de commandes

```
#!/bin/bash

pwd
ls -l
```

Le sha-bang (#!) permet de se placer dans le langage bash.

Les commandes sont exécutées séquentiellement.

Le script se lance dans le terminal par
`. ./script.sh`
ou
`./script.sh`
ou
`bash ./script.sh`
suivant les droits associés au fichier.

Assistance au débogage

Lister les commandes et le résultat de leur évolution

`bash -x ./script.sh`

```
#!/bin/bash

set -x
pwd
ls -l
```

La commande `set` permet de définir des variables d'environnement

Pour débugger, il est utile de ne prédéfinir aucune variable et d'activer le plus de commentaires possibles (mode bavard)

```
#!/bin/bash

set -o verbose
set -o nounset

pwd
ls -l
```

Variables

Les variables possèdent un nom et une valeur. Par exemple, le nom du fichier est

```
fichier='script.sh'
```

La variable s'utilise en précédent son nom du symbole $:

```
echo $fichier
```

echo en affiche le contenu.
Les variables peuvent être de type numérique

```
x=15
echo $x
15
pi=3.1415
echo $pi
3.1415
```

Variables d'environnement

Il existe des variables prédéfinies dans l'environnement. Ce sont des variables globales.
`echo $USER`
Ajouter une variable à l'environnement nécessite de l'y exporter :

```
export fichier='script.sh'
```

L'ensemble des variables de l'environnement peuvent être listées par la commande
`env`
Les variables globales peuvent être définies dans le fichier `.bashrc`
Variables de base :

- `PATH` : liste des chemins où le shell cherche les exécutables utilisés
- `SHELL` : le shell de votre session
- `HOME` : répertoire racine de votre compte
- `PWD` : chemin absolu vers votre répertoire local

Guillements, quotes, etc.

Les simples quotes définissent une chaîne de caractères. Elles n'interprêtent pas les caractères qu'elles encadrent :

```
echo '$fichier'
$fichier
```

Les doubles quotes permettent d'interprêter les caractères qu'elles encadrent :

```
echo "$fichier"
script.sh
```

Les accents graves (back quotes) exécutent ce qu'elles encadrent :

```
cmd=`ls -l`
echo "liste des fichiers : $cmd"
-rw-r--r-- 1 acadiou acadiou 23 nov. 2 23:09 script.sh
```

Préférer néanmoins la notation suivante à celle avec les accents graves

```
  cmd=$(ls -l)
```

Paramètres

Des variables peuvent être saisies en paramètres d'un script, par exemple :

```
./script.sh param1 param2 param3
```

Ces paramètres sont passés au script de la manière suivante :

- `$#` : le nombre de paramètre ;
- `$0` : le nom du script (ici `script.sh`) ;
- `$1` : le premier paramètre ;
- `$n` : le n^{e}paramètre ($n > 0$).

Exemple de script :

```
#!/bin/bash

echo "Lancement du script $0 avec $# parametres"
```

```
./script.sh p1 p2
Lancement du script ./exemples/script.sh avec 2 parametres
```

Manipulation des chaînes de caractères

Déclaration et affectation

```
${variable = 'valeur'}
```

Retourne la longueur de la chaîne de caractères

```
${#variable}
```

Extraction d'une séquence

```
${variable:position:length}
```

Supprime la sous-chaîne `cut` à partir de la fin de `variable`

```
${variable%cut}
```

Supprime la sous-chaîne `cut` à partir du début de `variable`

```
${variable#cut}
```

Exemple

```
#!/bin/bash

dir=$1
mkdir -p $dir

ext='png'
EXT='gif'

for file in Images/*.$ext; do
  ls -l $file
  convert $file $dir/${$file%.$ext}.$EXT
done
```

Opérations mathématiques

Les opérations s'effectuent avec la commande `let` et ont une syntaxe proche sur C.

- `+` : addition ;
- `-` : soustraction ;
- `*` : multiplication ;
- `/` : division ;
- `**` : puissance ;
- `%` : modulo (reste de la division entière).

```
#!/bin/bash

let "n = 10 % 3" # $n = 1
echo "n = $n"
n = 1
```

N.B. : les commentaires sont précédés d'un #

if

Syntaxes :

```
if [ test ]
then
       echo "Le test est
            vrai"
elif [ condition2 ]
then
       echo "Le test est
            vrai"
else
       echo "Aucun des tests
             ne passe"
fi
```

```
if [ test ]; then
       echo "Le test est
            vrai"
fi
```

Tests sur les chaînes de caractères

Identité :

```
$chaine1 == $chaine2
```

Différences :

```
$chaine1 != $chaine2
```

Vérifie si la chaîne de caractères est vide

```
-z $chaine
```

Teste si la chaîne de caractères n'est pas vide

```
-n $chaine
```

Exemple

```
#!/bin/bash

if [ -z $1 ]
then
       echo "No arguments"
else
       echo "The command has some arguments"
fi
```

```
./test.sh
No arguments
```

```
./test.sh arg1 arg2
The command has some arguments
```

Tests sur les nombres

Identité :

```
$num1 -eq $num2
```

Différence :

```
$num1 -ne $num2
```

Inégalités :
Inférieur (ou égal)

```
$num1 -lt $num2
$num1 -le $num2
```

Supérieur (ou égal)

```
$num1 -gt $num2
$num1 -ge $num2
```

Exemple

```
#!/bin/bash

if [ $# -eq 0 ]; then
  echo "Usage: $0 run#";
fi
```

```
./test.sh
Usage: ./test.sh run#
```

Tests sur les fichiers

Vérifie si le fichier existe

```
-e $nomfichier
```

Vérifie si le fichier est un répertoire (resp. fichier)

```
-d $nomfichier #directory type
-f $nomfichier #file type
```

Vérifie si le fichier est un lien symbolique

```
-L $nomfichier
```

Vérifie les droits sur le fichier

```
-r $nomfichier #acces ouvert en lecture
-w $nomfichier #acces ouvert en ecriture
-x $nomfichier #executable
```

Vérifie l'antériorité

```
$fichier1 -nt $fichier2 #newer than, plus recent que
$fichier1 -ot $fichier2 #older than, plus vieux que
```

Exemple

```
#!/bin/bash

dir=$1
if [ ! -d $dir ]; then
   mkdir $dir
fi

# ou encore ...
mkdir -p $dir
```

Faire plusieurs tests à la fois

et :

```
&&
```

ou :

```
||
```

inverser un test :

```
!
```

Exemple

```
#!/bin/bash

if [ $# -ge 1 ] && [ $1 = 'pingouin' ]
then
        echo "C'est le bon argument"
else
        echo "Dommage, ce n'est pas le bon argument."
fi
```

case

```
#!/bin/bash

COMPIL=$1
case $COMPIL in
    newton_ompi_foss16)
        modules="foss/2016a"
        ;;
    newton_ompi_foss18 | p2chpd_ompi)
        modules="foss/2018a"
        ;;
    newton_impi_15)
        modules="intel/2015a"
        ;;
    turing)
        modules="lapack/3.4.2 fftw/3.3.3 idrmemmpi"
        ;;
    *)
        echo "bad COMPIL=$COMPIL"
        ;;
esac
```

while

Syntaxes :

```
while [ test ]
do
        echo "Action repetee
             tant que le test
             est vrai"
done
```

```
while [ test ]; do
        echo "Le test est
             vrai"
done
```

À l'inverse, utiliser la condition `until`, qui permet de boucle jusqu'à ce que la test soit vrai.

Exemple

```
#!/bin/bash

while [ -z $reponse ] || [ $reponse != 'oui' ]
do
       read -p 'Dites oui : ' reponse
done
```

```
$ ./while.sh
Dites oui : non
Dites oui : bla
Dites oui : oui
```

for

Permet de parcourir une liste de valeurs

Syntaxe :

```
for variable in valeur1 valeur2
do
        echo "variable $variable"
done
```

Exemples :

```
#!/bin/bash

for fichier in $(ls)
do
        echo "Fichier : $fichier"
done
```

```
#!/bin/bash

for fichier in *.sh
do
        echo "Fichier : $fichier"
done
```

Exemple numérique

```
#!/bin/bash

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

```
1
2
3
4
5
6
7
8
9
10
```

`$(seq 1 2 10)` permet de ne retenir que les nombres impairs.

Exemple de soumission multiples 1/2

- un programme `a.out` prenant deux paramètres *Ra* et *Pr*
- un script de soumission slurm associé

```
#!/bin/bash
#SBATCH --partition=mononode
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --time=2:00:00
$HOME/bin/a.out RA PR
```

pour tout lancer d'un coup :

```
for i in $(seq 1 7); do
  for pr in 0.1 1 10; do
    let "ra = 10 ** $i"
    rundir="Ra-$ra-Pr-$pr"
    mkdir -p $rundir
    cp slurm.sh $rundir
    cd $rundir
    sed -i "s/RA/$ra/g" slurm.sh
    sed -i "s/PR/$pr/g" slurm.sh
    sbatch slurm.sh
    cd ..
  done
done
```

Exemple de soumission multiples 1/2

- un programme `a.out` prenant deux paramètres *Ra* et *Pr*
- un script de soumission slurm associé

```
#!/bin/bash
#SBATCH --partition=mononode
#SBATCH --nodes=1
#SBATCH --cpus-per-task=16
#SBATCH --time=2:00:00
$HOME/bin/a.out RA PR
```

pour tout lancer d'un coup :

```
for i in $(seq 1 7); do
  for pr in 0.1 1 10; do
    let "ra = 10 ** $i"
    rundir="Ra-$ra-Pr-$pr"
    mkdir -p $rundir
    cp slurm.sh $rundir
    cd $rundir
    sed -i "s/RA/$ra/g" slurm.sh
    sed -i "s/PR/$pr/g" slurm.sh
    sbatch slurm.sh
    cd ..
  done
done
```

Exemple de soumissions multiples 2/2

Modification de tous mes jobs

```
for j in $(squeue -h -u $USER); do
  scontrol update Jobid=$j partition=test
done
```

Annulation de tous mes jobs

```
for j in $(squeue -h -u $USER); do
  scancel $j
done
```