Unix/Bash

Commandes avancées / scripts

Fichier de configuration `.bashrc`

- fichier `.bashrc` : fichier de configuration chargé à chaque fois qu'un **bash** est lancé
- configuration, personnalisation de l'interpréteur de commandes
- par ex. : créer des *alias*, modifier l'affichage du terminal, définir des fonctions `shell`, exporter des variables d'environnement, ...

Fichier de configuration `.bashrc`

Quelques exemples (de base), voir la documentation pour toutes les autres possibilités

```
#alias
alias ll='ls -la'
alias rm='rm -i'
alias mv='mv -i'
alias cl='clear'
#variable
export PATH=$PATH:~/myscripts/ #par exemple
#prompt
export PS1="[\u][\W] "# produit un prompt "[login][currentdir] "
```

1. Vérifier la présence (et au besoin le créer) du fichier `~/.bashrc`
2. Tester les différentes options ci-dessus
3. Pour recharger le fichier de configuration, lancer un nouveau `shell`, ou utiliser la commande `source`
4. Aide : `http://www.explainshell.com`

Quelques commandes (1)

Archivage et compression avec tar

- `-v : mode verbose`

```
tar -cvf  <archive> <dir> : Archive dans le fichier <archive> le
                            répertoire dir
tar -tvf  <archive>       : liste les fichiers dans l'archive
tar -xvf  <archive>       : extraction de l'archive
tar -cvzf <archive> <dir> : archive et compresse avec gzip
                            (extension .tgz)
tar -xvzf <archive>       : extrait une archive compressée avec gzip
tar -cvjf <archive> <dir> : archive et compresse avec bzip2
                            (extension .tbz)
tar -xvjf <archive>       : extrait une archive compressée avec bzip2
```

- **Rq :** possibilité de compresser un fichier seul avec `gzip/gunzip` ou `bzip2/bunzip2`.

Quelques commandes (2)

Mesure de l'espace disque avec `du` et `df`

- `-h (human readable): affiche sous la forme (Kio, Mio) valable pour les deux commandes`
- `df . : espace disque de la partition courante + diverses infos`

```
Filesystem     Size   Used  Avail Capacity  Mounted on
/dev/disk0s2  465Gi  338Gi  127Gi    73%    /
```

- `du <dirs> : espace occupé par la liste de répertoires dirs`

  `du -c : affiche le total de chaque répertoire`
  `du -s : affiche le total de tous les répertoires`

Exercices

Archivage et compression avec `tar`

1. Copier le répertoire `~mfaverge/public_html/IF104` dans un dossier `tmp` sur votre compte
2. Déterminer la place occupée par le répertoire `~/tmp/if104`
3. Créer une archive de ce répertoire nommée `if104-arch.tgz`
4. Lister le contenu de l'archive
5. Vérifier le niveau de compression
6. Extraire l'archive dans un dossier `tmp` et vérifier la présence des fichiers
7. Supprimer l'archive ainsi que le dossier `tmp`
8. Extraire désormais l'archive `~mfaverge/public_html/IF104/testtar.txt` dans un dossier `tmp`, jusqu'à obtention du fichier `README.txt`

Quelques commandes (2)

Filtres, traitement de chaînes

```
* grep <string> <files> :
  recherche du motif string dans les fichiers files
 -i : ignore la casse
 -n : précède chaque ligne par son numéro
 -v : affiche les lignes ne contenant pas string
 -R : recherche récursive dans les répertoires
* sort : trie des lignes dans l'ordre alphabétique
* uniq : trie les doublons
* head : extraction des premières lignes
* tail : extraction des dernières lignes
* set : découpe une chaine de caratères en fonction de délimiteurs.
  les délimiteurs sont connus de bash grâce à la variable IFS
* tr : remplacements de caractères
* sed : remplacements de motifs
* cut : sélection par colonne
```

Quelques commandes (3)

Opérateurs de liste

Enchaîner plusieurs commande à la suite grâce au " ;" ou au pipe `|`. En voici d'autres :

- cmd_1 `;` cmd_2 : **enchaînement séquentiel** ; cmd_2 est exécutée lorsque cmd_1 est terminée
- cmd_1 `&` cmd_2 : **enchaînement parallèle** ; cmd_1 et cmd_2 sont lancées en parallèle
- cmd_1 `&&` cmd_2 : **et** ; cmd_2 est exécutée si cmd_1 retourne *vrai*
- cmd_1 `||` cmd_2 : **et** ; cmd_2 est exécutée si cmd_1 retourne *faux*

Exercices

À partir du fichier `/etc/passwd` et des commandes de traitements de chaînes, trouver la ligne de commandes permettant de

1. Lister tous les utilisateurs
2. Trier les utilisateurs dans l'ordre alphabétique et afficher les deux premiers
3. Afficher `rOOt`
4. Afficher tous les utilisateurs ayant bash pour shell, l'affichage se fera sous la forme `<user> uses /bin/bash`
5. Afficher les 5 plus gros répertoires d'un répertoire passé en paramètre qui font au moins un Mo, les trier dans l'ordre décroissant (les plus gros d'abord), et si ils en existent, afficher cette liste suivi de la chaine "Répertoire à supprimer". A tester sur `/usr/lib`

Scripts Shell

- Programme écrit dans un langage interprétable par un interpréteur de commande (pas de phase de compilation)
- **`bash`** est à la fois le nom de l'interpréteur de commande et le langage de programmation interprété .
- Un script shell simple peut être une suite de commandes écrite dans un fichier qui est exécutable par le système
- Un script shell permet de créer des programmes exécutant une série de commande qui pourrait être pénible et répétitif à saisir au clavier en ligne de commande.

Exécution

- Une manière d'exécuter un programme shell est de le lancer en ligne de commande et de spécifier quel interpréteur est capable de lire ce langage de programmation (on parle d'interprétation).
- Script : `myscript.sh` (convention `.sh`) alors

```
~$ echo "echo Bonjour" > myscript.sh
~$ bash myscript.sh
Bonjour
```

- Pour éviter d'indiquer explicitement l'interpréteur, le script shell peut commencer par une ligne spéciale qui va indiquer au shell quel interpréteur de commande utiliser.
- Cette ligne est le *shebang* (contraction de *shell* et de *bang* :!)
- Un script commence toujours par `#!cheminverslinterpreteur`.
- Rendre le script exécutable, i.e., lui donner le droit d'exécution (`x`).

Exécution (Résumé)

```
~$ which bash
/bin/bash
~$ echo \#\!/bin/bash > myscript.sh
~$ echo "echo Bonjour" >> myscript.sh
~$ chmod 700 myscript.sh
~$ ls -l myscript.sh
-rwx------ 1 user group 27 sep 19  2012 myscript.sh
~$ ./myscript.sh
Bonjour
```

Arguments

- Un script shell peut prendre en paramètre des arguments.
- Lors de l'évaluation de la commande par le shell, le shell coupe les chaînes de caractères selon différents séparateurs, le plus courant étant l'espace
- Il stocke ensuite les différents token dans un vecteur. Le programmeur peut accéder aux éléments de ce vecteur.

Les motifs d'accès sont les suivants

- `$#` : le nombre d'élément courant dans le vecteur
- `$1 ... $9` : accès au premier, deuxième, ... neuvième
- `$*` : liste de tous les éléments

Comme le nombre d'arguments max est 9, la commande `shift` permet de palier à ce problème.

Il existe d'autres motifs associés à `$`

- `$0` : le nom de la commande courante
- `$?` : valeur de retour de la dernière commande
- `$$` : pid de la commande courante

Structures de contrôle

La programmation en shell, comme tout langage de programmation, nécessite des structures de contrôle avec des syntaxes particulières, i.e., la façon d'écrire ces structures.

Attention : la programmation en shell nécessite beaucoup de rigueur du programmeur au niveau de la syntaxe. Le bash est très sensible aux espaces (dont il se sert comme délimiteur) et aux sauts de ligne

Commande `test`

La commande **`test`**, possède deux syntaxes :
`test arg_1 arg_2 ...` et **`[ arg_1 arg_2 ... ]`**

Elle permet de tester :

1. l'existence et la nature d'un fichier, par exemple
   - **`-e␣chemin`** : si chemin est un fichier existant
   - **`-f␣chemin`** : si chemin est un fichier existant et est un fichier ordinaire
   - **`-d␣chemin`** si chemin est un fichier existant et est un répertoire
   - **`-r␣chemin`** : si chemin est un fichier existant et est accessible en lecture
   - **`-w␣chemin`** : si chemin est un fichier existant et est accessible en écriture
2. test d'égalité de deux chaînes de caractères
   - **`chn_1␣=␣chn_2`**
   - **`chn_1␣!=␣chn_2`**
3. comparaison de nombres
   - **`chn_1␣-eq␣chn_2`** : $=$
   - **`chn_1␣-ne␣chn_2`** : $\neq$
   - **`chn_1␣-gt␣chn_2`** : $>$
   - **`chn_1␣-ge␣chn_2`** : $\geq$
   - **`chn_1␣-lt␣chn_2`** : $<$
   - **`chn_1␣-le␣chn_2`** : $\leq$

pour plus de détails voir le man.

Conditions (if)

Structures de contrôle de type conditionnelle

- **Si** quelque-chose-de-vrai **alors :** execute-bloc-1 **sinon :** execute-bloc-2
- et du type aiguillage par rapport à un motif reconnu

([ ] = optionnel, `<liste>` liste de commande, pas forcément un test)

Syntaxe

```
if <liste>
then
  <liste>
[ elif <liste>
then
  <liste>
        ]
[ else
    <liste> ]
fi
```

Exemple

```
if [ $USER = "$1" ]
then
  echo $USER c'est moi
else
  echo c'est pas moi
fi
```

Conditions (if) - Remarques

Il est possible d'écrire tout ou partie du test sur une seule ligne

Syntaxe

```
if <liste>; then; <liste>; [ elif <liste>; then; <liste>; ]
[ else ; <liste>; ] fi
```

Exemple

```
if [ $USER = "$1" ]; then; echo $USER c'est moi;
else; echo c'est pas moi; fi
```

Notez aussi les guillemets autour de `$1`. A quoi servent-ils ?

Conditions (case)

Structure conditionnelle test de type `case` à la syntaxe suivante (aiguillage de motif)

Syntaxe

```
case <mot> in
  <motif> [ | <motif> ...] )
    <liste>
    ...
    ;;
  ...
esac
```

Exemple

```
case $N in
  mbox )
    echo $N: boxes
    ;;
  * )
    echo $N: unknown type
    ;;
esac
```

Itérations (1)

Structures de contrôle permettant d'appliquer une série de commande identique à une liste de variables donnée en argument.

1. de type `for`, i.e. : pour chaque élément de la liste : appliquer un traitement à cet élément.

Syntaxe

```
for <nom>  in <mot> ...
do
  <liste>
  ...
done
```

Exemple

```
for N in *
do
  cp $N $N.old
done

for N in ‘cat $LISTE‘; do ...; done
```

Itérations (2)

Structures de contrôle permettant d'appliquer une série de commande identique à une liste de variables donnée en argument.

2. de type `while`, i.e. : tant qu'il y a un élément ou tant que la condition est vrai : appliquer un traitement

Syntaxe

```
while <liste_1>
do
  <liste_2>
  ...
done
```

Ici la commande `shift` permet de décaler la liste des arguments vers la gauche, le premier étant perdu, la boucle itère (est répétée) tant que le nombre de paramètres est positif.

Exemple

```
while [ $# -gt 0 ]
do
  case $1 in
    -o )
      shift
      echo $1
      ;;
  esac
  shift
done
```

Itérations (3)

Il existe 3 types d'échappement en `bash` pour sortir d'une boucle :

- `continue` : passer à l'itération suivante
- `break` : sortir de l'itération
- `exit [n]` : sortir du programme avec la valeur `n`

Déclaration de fonctions

Un programme peut être décomposé en fonctions ; afin de

- modulariser le code
- factoriser le code (éviter de réécrire du code inutile)
- faciliter la maintenance du code

Syntaxe

```
my_function()
{
  #corps de la fonctions
}

my_function #ici on l'appelle
```

Exemple

```
usage()
{
  echo "Usage: $0 [-abcd] files..."
}

if [ $# -eq 0 ]
then
  usage
  exit 0
fi
```

Exercices : `mykill.sh`

- Il est souvent fastidieux de trouver le pid d'un processus à tuer
- Écrire un script shell qui recherche un processus par son nom
- Récupère le (ou les) pid correspondant
- Envoie le signal sigkill a l'ensemble de ces processus
- Exemple d'exécution :

```
~$ ./mykill.sh xeyes
```

Exercices : `liste.sh`

- Écrire un script shell qui liste les noms de toutes les personnes dont le home est stocké au même endroit (même groupe). Par défaut, elle listera les personnes dans le même groupe que soit même.
- Exemple d'exécution :

```
~$ ./liste.sh edao
ABBASSI Asmaa
...
ZOUBAIRI Saad
~$ ./liste.sh
ABGRALL Remi
...
TA Vinh-Thong
```

- On utilisera la commande **`finger`** et la commande **`brain`** sans la faire surchauffer :)

Exercices : `sum.sh`

- Écrire un script shell qui permet de réaliser la somme des entiers entrés en paramètres sur la ligne de commande et affiche les étapes du calculs
- Indication : utiliser une boucle **`for`** et la commande **`shift`**
- Exemple d'exécution :

```
~$ ./sum.sh 1 2 3 4 5 6 7 8 9 10
1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
10 + 5 = 15
15 + 6 = 21
21 + 7 = 28
28 + 8 = 36
36 + 9 = 45
45 + 10 = 55
```

- Testez les commandes :

```
~$ ./sum.sh $(seq 1 8)
~$ ./sum.sh $(seq 1 10000)
```

Exercices : `skeleton.sh` (1)

Écrire un script shell qui permet de réaliser des squelettes de fichiers LaTeX ou `bash`

1. Faire une première version à base d'`echo` qui génère un fichier LaTeX, dont le nom est donné en paramètre (sans l'extension)
2. Faire une fonction usage permettant d'afficher l'aide de ce programme
   `Usage: skeleton filename`
3. Faire un test permettant de savoir si le nombre de paramètres donnés dans la ligne de commande est correct, sinon affichage de `usage`.
4. Faire une version permettant de produire un squelette de script `bash`.
5. améliorer le script en mettant en place un aiguillage sur le paramètre `type`
6. on peut remarquer que l'entête est la même quelque soit le type de fichier, seul le symbole de commentaire change. Factoriser cette partie en implantant une fonction qui prend en paramètre le symbole désiré.
7. améliorer le script en séparant la création du corps des squelettes dans des fonctions séparées.
8. améliorer le script en utilisation des variables, l'évaluation du chemin pour aller chercher bash, la date, le nom d'utilisateur

Exercices : `skeleton.sh` (2)

Exemple d'éxecution : `skeleton.sh latex test-latex` produit le fichier `test-latex.tex` contenant

test-latex.tex

```
%
% file: test-latex.tex
% date: mardi 20 septembre 2012, 15:41:25 (UTC+0200)
% author: vta <vta@enseib-matmeca.fr>
% description:
%

\documentclass{article}
%\usepackage{}

\begin{document}

\end{document}
```

Exercices : `skeleton.sh` (3)

Exemple d'éxecution : **`skeleton.sh bash test-bash`** produit le fichier **`test-bash.sh`** et donne les droits d'exécution.

test-bash.sh

```
#
# file: test-bash.sh
# date: mardi 20 septembre 2012, 15:44:57 (UTC+0200)
# author: vta <vta@enseib-matmeca.fr>
# description:
#

#!/bin/bash
CMD=‘basename $0‘

usage()
{
  echo "Usage: $CMD ..."
}

if [ $# -lt ... ]
then
  usage
  exit 1
fi
```

Exercices : backup.sh

Écrire le un script shell qui

- affiche les n fichiers les plus récemment modifier,
- demande à l'utilisateur s'il veut créer une archive à partir de ces fichiers (y/n).
- en cas de réponse positif : demande le nom de l'archive,
- en cas de réponse négative : quit
- tout autres réponse : message d'erreur et arrête le programme.

Exemple d'exécution :

```
~$ ./backup.sh 4
backup.sh sum.sh templates.sh myscript.sh
Do you want to archive and compress these files (y/n)?
n
~$ ./backup.sh 4
backup.sh sum.sh templates.sh myscript.sh
Do you want to archive and compress these files (y/n)?
u
unkwon answer, quit
~$ ./backup.sh 4
Do you want to archive and compress these files (y/n)?
y
Enter the archive name:
arxiv
Save in arxiv.tgz
~$
```

Exercices : pdflatex.sh

Avez vous bien compris le procéder de compilation utilisé par pdflatex ?

Écrire le un script shell qui

- prend un fichier `.tex` en entrée
- compile ce fichier autant de fois que nécéssaire et pas une de plus.
- ressort les erreurs rencontrées si il y en a
- Avancé : Prendre en compte la bibliographie si elle est présente.

Exemple d'exécution :

```
~$ ./pdflatex.sh document.tex
=== compile document.tex ===
=== compile document.tex ===
=== document.pdf generated ===
~$
```

Arithmetic Operators

```
$ var=$(( 20 + 5 ))
$ expr 1 + 3   # 4
$ expr 2 - 1   # 1
$ expr 10 / 3  # 3
$ expr 20 % 3  # 2 (remainder)
$ expr 10 \* 3 # 30 (multiply)
```

String Operators

Expression	Meaning
`${#str}`	Length of $str
`${str:pos}`	Extract substring from $str at $pos
`${str:pos:len}`	Extract $len chars from $str at $pos
`${str/sub/rep}`	Replace first match of $sub with $rep
`${str//sub/rep}`	Replace all matches of $sub with $rep
`${str/#sub/rep}`	If $sub matches front end of $str, substitute $rep for $sub
`${str/%sub/rep}`	If $sub matches back end of $str, substitute $rep for $sub

Relational Operators

Num	String	Test
-eq	=	Equal to
	==	Equal to
-ne	!=	Not equal to
-lt	\<	Less than
-le		Less than or equal to
-gt	\>	Greater than
-ge		Greater than or equal to
	-z	is empty
	-n	is not empty

File Operators

	True if file exists and...
-f file	...is a regular file
-r file	...is readable
-w file	...is writable
-x file	...is executable
-d file	...is a directory
-s file	...has a size greater than zero.

Control Structures

```
if [ condition ] # true = 0
  then
# condition is true
elif [ condition1 ]
  then
# condition1 is true
 elif condition2
   then
# condition2 is true
else
# None of the conditions is true
fi

case expression in
  pattern1) execute commands ;;
  pattern2) execute commands ;;
esac

while [ true ]
do
#  execute commands
done

until [ false ]
do
#   execute commands
done

for x in 1 2 3 4 5 # or for x in {1..5}
do
    echo "The value of x is $x";
done

LIMIT=10
for ((x=1; x <= LIMIT ; x++))
do
  echo -n "$x "
done

for file in *~
do
  echo "$file"
done

break [n] # exit n levels of loop
continue [n] # go to next iteration of loop n up
```

Function Usage

```
function-name arg1 arg2 arg3 argN
```

n.b. functions must be defined before use...

Function Definition

```
function function-name ()
{
# statement1
# statement2
# statementN
  return [integer] # optional
}
```

Functions have access to script variables, and may have local variables:

```
$ local var=value
```

Arrays

```
$ vars[2]=”two” # declare an array
$ echo ${vars[2]} # access an element
$ fruits=(apples oranges pears) # populate array
$ echo ${fruits[0]} # apples - index from 0
$ declare -a fruits # creates an array

echo "Enter your favourite fruits: "
read -a fruits
echo You entered ${#fruits[@]} fruits
for f in "${fruits[@]}"
do
  echo "$f"
done

$ array=( "${fruits[@]}" "grapes" ) # add to end
$ copy="${fruits[@]}" # copy an array
$ unset fruits[1] # delete one element
$ unset fruits # delete array
```

Array elements do not have to be sequential - indices are listed in `{!fruits[@]}`:

```
for i in ${!fruits[@]}
do
  echo fruits[$i]=${fruits[i]}
done
```

All variables are single element arrays:

```
$ var=”The quick brown fox”
$ echo {var[0]} #  The quick brown fox
```

String operators can be applied to all the string elements in an array using ${name[@] ... } notation, e.g.:

```
$ echo ${arrayZ[@]//abc/xyz} # Replace all
occurrences of abc with xyz
```

User Interaction

```
echo -n "Prompt: "
read
echo "You typed $REPLY."

echo -n "Prompt: "
read response
echo "You typed $response."

PS3="Choose a fruit: "
select fruit in "apples" "oranges" "pears"
do
  if [ -n "$fruit" ]
  then
    break
  fi
  echo "Invalid choice"
done

$ dialog --menu "Choose" 10 20 4 1 apples 2 \
oranges 3 pears 4 bananas 2>/tmp/ans
$ fruit=`cat /tmp/ans`
$ echo $fruit

$ zenity --list --radiolist --column "Choose" \
--column "Fruit" 0 Apples 0 Oranges 0 Pears 0 \
Bananas > /tmp/ans
$ fruit=`cat /tmp/ans`
$ echo $fruit
```

Reading Input from a File

```
exec 6<&0              # 'Park' stdin on #6
exec < temp.txt        # stdin=file "temp.txt"
read                   # from stdin
until [ -z "$REPLY" ]
do
  echo "$REPLY"        # lists temp.txt
  read
done
exec 0<&6 6<&-         # restore stdin
echo -n "Press any key to continue"
read
```

Trapping Exceptions

```
TMPFILE=`mktemp`
on_break()
{
  rm -f $TMPFILE
  exit 1
}
trap on_break 2 # catches Ctrl+C
```

Data and Time

```
$ start=`date +%s`
$ end=`date +%s`
$ echo That took $((end-start)) seconds
$ date +"%c" -d19540409
Fri 09 Apr 1954 12:00:00 AM GMT
```

Case Conversion

```
$ in="The quick brown fox"
$ out=`echo $in | tr [:lower:] [:upper:]`
$ echo "$out"
THE QUICK BROWN FOX
```

Preset Variables

$HOME	User's home directory
$HOSTNAME	Name of host
$HOSTTYPE	Type of host (e.g. i486)
$PWD	Current directory
$REPLY	default variable for READ and SELECT
$SECONDS	Elapsed time of script
$TMOUT	Max. script elapsed time or wait time for read

References

Linux Shell Scripting Tutorial - A Beginner's handbook
http://www.cyberciti.biz/nixcraft/linux/docs/uniqlinuxfeatures/lsst/
BASH Programming Introduction, Mike G
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
Advanced BASH Scripting Guide, Mendel Cooper
http://tldp.org/LDP/abs/html/

Copyright & Licence

This version dated:

BASH Quick Reference Card

"All the useful stuff on a single card"

```
#!/bin/bash
$ chmod ugo+x shell_script.sh

$ bash [options] [file]
```

Options
-x show execution of [file]
-v echo lines as they are read

Variables

```
$ var="some value" # declare a variable
$ echo $var # access contents of variable
$ echo ${var} # access contents of variable
$ echo ${var:-"default value"} # with default
$ var= # delete a variable
$ unset var # delete a variable
```

Quoting - "$variable" - preserves whitespace

Positional Variables

$0	Name of script
$1-$9	Positional parameters #1 - #9
${10}	to access positional parameter #10 onwards
$#	Number of positional parameters
"$*"	All the positional parameters (as a single word) *
"$@"	All the positional parameters (as separate strings)
$?	Return value

`set [values]` - sets positional params to `[values]`
`set -- -` deletes all positional parameters
`shift [n]`- move positional params n places to the left

Command Substitution

```
$ var=`ls *.txt` # Variable contains output
$ var=$(ls *.txt)   # Alternative form
$ cat myfile >/dev/null # suppress stdout
$ rm nofile 2>/dev/null # suppress stderr
$ cat nofile 2>/dev/null >/dev/null # suppress
both
```