Système d'exploitation

Système d'exploitation - Utilisation

Mickael Rouvier

CERI – Avignon Université
mickael.rouvier@univ-avignon.fr

AVIGNON UNIVERSITÉ

Section 1

Programmation en bash

Introduction

- Jusqu'à présent on a vu comment lancer des commandes
- **Problème** :
  - Comment **relancer** une séquence de commande ?
  - Comment rendre des séquences de commandes **générique** ?
- Langage de programmation intégré au bash (variable, boucle, test conditionnelle...)
- Avantage
  - Permet d'automatiser des tâches (sauvegarder vos données, surveillance de votre réseau...)
  - Rien à installer
  - Très peu de nouvelles choses à apprendre. On peut utiliser les commandes du bash simplement (*ls*, *cut*, *grep*...)
- Inconvénient
  - Pas aussi complet que le langage de programmation C/C++ ou autre (exception, pointeur...)

Création d'un script bash - Hello World

Exemple de création d'un script bash :

- Fichier texte (l'extension du fichier est .sh)
- Hashbang représenté par **#!** : en-tête du fichier qui permet de définir l'interpréteur à utiliser

```python
#!/bin/python

print "Hello world !"
```

Listing 1 – "Exemple de script en python : main.py"

```bash
#!/bin/bash

echo "Hello world !"
```

Listing 2 – "Exemple de script bash : main.sh"

- Plusieurs façon d'éxécuter un script :

```
invite/> chmod u+x main.py
invite/> ./main.py
```

```
invite/> python main.py
```

```
invite/> chmod u+x main.sh
invite/> ./main.sh
```

```
invite/> sh main.sh
```

Variables : affectation - Partie 1

Affectation basique de variable

- Typage dynamique (pas nécessaire de spécifier le type)
- **Attention** : pas d'espace avant et après le =
- La variable peut contenir des chiffres, des lettres et des symboles souligné mais ne doit pas commencer par un chiffre
- La portée de la variable est globale

```cpp
int integer = 5;
float flt = 4.5;
string str =
    "bonjour";
```

Listing 3 – "exemple de typage de variable en C++"

```python
integer = 5
flt = 4.5
str = "bonjour"
```

Listing 4 – "exemple de typage de variable en python"

```bash
integer=5
flt=4.5
str="bonjour"
```

Listing 5 – "exemple de typage de variable en bash"

Variables : substitution - Partie 1

Afficher le contenu d'une variable :

- On peut afficher le contenu d'une variable à l'aide de la commande **echo**
- **$** permet d'accéder au contenu de la variable
- La substitution est le mécanisme qui différencie la variable de son contenu

```
1 int a = 5;
2 cout<< a <<endl;
```

Listing 6 – "Exemple pour afficher le contenu d'une variable en C++"

```
1 a = 5
2 print a
```

Listing 7 – "Exemple pour afficher le contenu d'une variable en pyton"

```
1 a=5
2 echo $a
```

Listing 8 – "Exemple pour afficher le contenu d'une variable en bash"

Variables : substitution - Partie 2

Afficher le contenu d'une variable :

- **simple quote ou apostrophe** délimitent une chaîne de caractères. **Attention** : les variables ne seront pas interprétées
- **doubles quotes ou guillemets** délimitent une chaîne de caractère, les variables sont interprétés par le shell. Utile pour générer des messages dynamiques au sein d'un script.
- **${}** permet de délimiter le début et la fin d'une variable

```
1 var=paul
2 echo 'bonjour $var'
```

```
bonjour $var
```

```
1 var=paul
2 echo "bonjour $var"
3
```

```
bonjour paul
```

```
var=fichier
echo "$var_conf.txt"
echo
    "${var}_conf.txt"
```

Variables : affectation - Partie 2

Affectation de variable élaborée :

- Affectation de variable à partir d'une autre variable

```
1 toto=bonjour
2 variable=$toto
3 toto=salut
4 echo $variable
5 echo $toto
```

```
bonjour
salut
```

```
1 extension=.txt
2 variable=fichier$toto
3 echo $variable
```

```
fichier.txt
```

Variables : affectation - Partie 3

Affectation de variable élaborée :

- Affectation de variable élaborée : affecte le résultat d'une commande

```
1 file=/etc/apache2/apache.conf
2 variable=`basename $file .conf`
3 echo $variable
```

```
apache
```

```
1 variable=`ls *.txt`
2 echo $variable
```

```
file1.txt .... fileN.txt
```

```
1 variable=`cat /etc/passwd | grep root | cut -f5 -d" "`
2 echo $variable
```

```
password
```

- Les back-quotes, apostrophes inversées ou accents graves permettent délimitent une commande à exécuter.

Tester une condition

Permet de tester [condition] :

renvoient le code retour 0 si l'expression est vrai

```
1 [ 2 = 2 ]
2 echo $?
```

```
0
```

renvoient le code retour 1 si l'expression est fausse

```
1 [ 2 = 1 ]
2 echo $?
```

```
1
```

Test - chaîne de caractère

Les opérateurs de tests disponibles pour les chaînes de caractères sont :

- **c1 = c2** : vrai si c1 et c2 sont égaux
- **c1 != c2** : vrai si c1 et c2 sont différents
- **-z c** : vrai si c est une chaine vide
- **-n c** : vrai si c n'est pas une chaîne vide

```
1 variable=john
2 [ "john" = ${variable} ]
3 echo $?
```

```
0
```

Test - nombre

Les opérateurs de tests disponibles pour les nombres sont :

- **n1 –eq n2** : vrai si n1 et n2 sont égaux (equal)
- **n1 -ne n2** : vrai si n1 et n2 sont différents (not equal)
- **n1 -lt n2** : vrai si n1 est strictement inférieur à n2 (less than)
- **n1 -le n2** : vrai si n1 est inférieur ou égal à n2 (less equal)
- **n1 –gt n2** : vrai si n1 est strictement supérieur à n2 (greater than)
- **n1 –ge n2** : vrai si n1 et supérieur ou égal à n2 (greater equal)

```
1 variable=5
2 [ 5 -eq ${variable} ]
3 echo $?
```

```
0
```

Test - objets du système

Les opérateurs de tests disponibles pour les objets du système sont :

- **-e $FILE** : vrai si l'objet désigné par $FILE existe
- **-s $FILE** : vrai si l'objet désigné par $FILE existe et sa taille est supérieure à zéro
- **-f $FILE** : vrai si l'objet désigné par $FILE est un fichier dans le répertoire courant

```
1 [ -e "fichier.conf" ]
2 echo $?
```

```
0
```

Test - expressions régulières

- Les opérateurs de tests disponibles sont, pour les expressions régulières :
  - **! e** : vrai si e est faux
  - **e1 –a e2** : vrai si e1 et e2 sont vrais
  - **e1 –o e2** : vrai si e1 ou e2 est vrai

La structure conditionnelle if - Partie 1

- L'instruction **if** permet d'effectuer les instructions si la condition est réalisé (VRAI) :

```
1 if condition
2 then
3     instructions
4 fi
```

- Exemple :

```
1 a=5
2 if [ $a -eq "5" ]
3 then
4     echo "a is equal to 5"
5 fi
```

```
a is equal to 5
```

La structure conditionnelle if - Partie 2

L'instruction **if** peut aussi inclure une instruction else permettant d'exécuter des instructions dans le cas ou la condition n'est pas réalisée (FAUX) :

```
1 if condition
2 then
3     instructions
4 else
5     instructions
6 fi
```

La structure conditionnelle if - Partie 3

Exemple :

```
1 a=7
2 if [ $a -eq "5" ]
3 then
4     echo "a is equal to 5"
5 else
6     echo "a is not equal to 5"
7 fi
```

```
a is not equal to 5
```

La structure conditionnelle if - Partie 4

Il est bien sur possible d'imbriquer des if dans d'autres if :

```
1  if condition
2  then
3      instructions
4  else
5      if condition
6      then
7          instructions
8      else
9          instructions
10     fi
11 fi
```

La structure conditionnelle case

- L'instruction **case** permet de comparer une valeur avec une liste d'autres valeurs et d'exécuter un bloc d'instructions lorsqu'une des valeurs de la liste correspond.

```
1 case valeur_testee in
2     valeur1) instruction(s);;
3     valeur2) instruction(s);;
4     valeur3) instruction(s);;
5 esac
```

- Les valeurs peuvent contenir des caractéres joker, ainsi :

```
1 case $a in
2     t?t?) echo "ok";;
3     *) exit 1;;
4 esac
```

- affichera ok si la variable $a a une valeur correspondant au motif t ?t ?, comme par exemple toto ou tata (tonton ne marchera pas).

Boucle for - Partie 1

- La boucle for permet de parcourir une liste de valeurs (elle effectue donc un nombre de tours de boucle qui est connu à l'avance)

```
1 for variable in liste_valeurs
2 do
3     instructions
4 done
```

- Exemple en mettant des valeurs :

```
1 for valeur in e1 e2 e3
2 do
3     echo $valeur
4 done
```

Boucle for - Partie 2

- Exemple en utilisant la commande seq :

```
1 for valeur in `seq 1 1 5`
2 do
3     echo "boucle $valeur"
4 done
```

- Exemple en utilisant la commande ls :

```
1 for valeur in `ls a*`
2 do
3     echo "fichier $valeur"
4 done
```

Boucle for - Partie 3

- Exemple en utilisant les arguments de script :

```
1 for valeur in $@
2 do
3     echo "argument $valeur"
4 done
```

Boucle for - Partie 4

- Dans laquelle e1, e2 et e3 sont des expressions arithmétiques. Une telle boucle commence par exécuter l'expression e1, puis tant que l'expression e2 est différente de zéro le bloc de l'instruction est exécutée et l'expression e3 évaluée

```
1 for ((e1; e2; e3))
2 do
3     instruction
4 done
```

- Exemple deuxième syntaxe

```
1 for ((i=0; 10 - $i, i++))
2 do
3     echo "iteration $i"
4 done
```

Boucle while - Partie 1

- La boucle while exécute un bloc d'instructions tant qu'une certaine condition est satisfaisante, lorsque cette condition devient fausse la boucle se termine. Cette boucle permet donc de faire un nombre indéterminé de tours de boucle, voir infini si la condition ne devient jamais fausse.
- Exemple :

```
1 while condition
2 do
3     instruction
4 done
```

Boucle while - Partie 2

- Exemple d'utilisation :

```
1  atrouver=50
2  echo "entrez un nombre compris entre 1 et 100"
3  read i
4  while [ "$i" -ne "$atrouver" ]
5      if [ "$i" -lt "$atrouver" ]
6      then
7          echo "trop petit, entrez un nombre compris entre 1
               et 100"
8      else
9          echo "trop grand, entrez un nombre compris entre 1
               et 100"
10     fi
11     read i
12 done
```

Récupérer les arguments passés dans un script - Partie 1

- Plusieurs variables spéciales sont disponibles lors de l'exécution d'un script.
  - **$0** a pour valeur le nom du script
  - **$1** jusqu'à **$9** ont respectivement pour valeur les neuf premiers
  - **$#** a pour valeur le nombre d'arguments passés au script
  - **$@** contient la liste de tous les arguments du script
- Exemple pour afficher les arguments

```
1 echo $1
2 echo $2
```

```
invite/> ./programme.sh toto tata
toto
tata
```

Récupérer les arguments passés dans un script - Partie 2

- Exemple pour vérifier le nombre d'argument

```
1  if [ "$#" != "2" ]
2  then
3      echo "$0 <(i) param_1> <(i) param_2>"
4      echo "param_1 : parametre 1"
5      echo "param_2 : parametre 2"
6      exit -1
7  fi
8
9  echo $1
10 echo $2
```

Calcul - Partie 1

Problèmatique : Comment effectuer des calculs via le bash ?

- La syntaxe **$((operation))**

```
1 a=1
2 a=$(($a+1))
3 echo $a
```

Calcul - Partie 2

- La commande **let**

```
1 a=1
2 let "a=$a + 1"
3 echo $a
```

Calcul - Partie 3

- La commande **bc**

```
1 a=1
2 a=`echo "$a+1" | bc`
3 echo $a
```

- En natif, bash ne propose que des fonctionnalités de calcul limités (additions simples...). bc permet des calculs plus complexes, avec gestion des décimales (ne pas oublier l'option –l).

```
1 echo "1/3" | bc -l
2 echo "sqrt(2)" | bc -l
```

Fonction - Déclaration de fonction

- Comme pour toute autre langage, l'utilisation de fonctions facilite le développement de scripts et la structuration de ceux-ci.
  - **Déclaration** - pour déclarer une fonction, on utilise la syntaxe suivante :

```
1 MaFunction() {
2     instructions
3 }
```

  - **Appel et paramètres** - pour appeler une fonction, on utilise la syntaxe suivante :

```
1 MaFunction "param_1" "param_2" ... "param_N"
```

```
1 MaFunction() {
2     param1=$1
3     param2=$2
4 }
```

Fonction : Exemples

- Exemple - déclarer et appeler fonction sans argument :

```
1 MaFunction () {
2     echo "on entre dans ma function"
3 }
4
5 MaFunction
```

- Exemple - déclarer et appeler fonction avec argument :

```
1 MaFunction () {
2    param1=$1
3    param2=$2
4    param3=$3
5    echo "on entre dans ma function"
6 }
7
8 MaFunction "param_1" "param_2" "param_3"
```

Fonction - Retour de function

Problèmatique : Comment récupérer des informations retourner par une fonction ?

- Solution 1 :

```
1 res=""
2 MaFunction () {
3    res="salut"
4 }
5
6 MaFunction
7 echo $res
```

- Solution 2 :

```
1 MaFunction () {
2    echo "results"
3 }
4
5 res=`MaFunction`
6 echo $res
```

Fonction - variable

Problèmatique : Comment traiter les variables locales ?

```
1 MaFunction () {
2    local res="salut"
3 }
4
5 MaFunction
6 echo $res
```

Enchaînements de commandes

Enchaînement simple : les commandes sont lancés les unes à la suite des autres (exécution successives)

```
1 #!/bin/bash
2
3 cmd_1
4 cmd_2
5 cmd_2
6
7 echo "fini"
```

Listing 9 – "Enchaînement simple de plusieurs commandes"

Enchaînement simultanées : les programmes sont lancés en parallèle

```
1 #!/bin/bash
2
3 cmd_1 &
4 cmd_2 &
5 cmd_3 &
6
7 wait
8 echo "fini"
```

Listing 10 – "Enchaînement simultanées de plusieurs commandes"

Enchaînements conditionnels - Partie 1

Enchaînement conditionnels et : Dans ce cas, la commande comP ne s'exécute que si com(P-1) s'est soldée par un succès.

```
1 #!/bin/bash
2
3 cmd_1 && cmd_2 && cmd_2
```

Enchaînement conditionnels alternative : les commandes com1 jusqu'à comN sont exécutées successivement tant qu'aucune ne se termine correctement. Dès qu'une des commandes se solde par un succès, la commande alternative est terminée.

```
1 #!/bin/bash
2
3 cmd_1 || cmd_2 || cmd_3
```

Enchaînements conditionnels - Partie 2

Enchaînement conditionnels et

```
1 #!/bin/bash
2
3 echo "toto" && echo "tata" &&
      echo "titi"
```

```
toto
tata
titi
```

Enchaînement conditionnels alternative

```
1 #!/bin/bash
2
3 echo "toto" || echo "tata" ||
      echo "titi"
```

```
toto
```

Enchaînements conditionnels - Partie 3

Enchaînement conditionnels et

```
1 #!/bin/bash
2
3 note=8
4 [ $note -gt 10 ] && echo "vous
      avez la moyenne"
```

Enchaînement conditionnels alternative

```
1 #!/bin/bash
2
3 note=8
4 [ $note -gt 10 ] || echo "vous
      n'avez pas la moyenne"
```

```
vous n'avez pas la moyenne
```