Support de cours

SYSTÈMES D'EXPLOITATION

CHAPITRE V: PROGRAMMATION SHELL

Plan

- **Introduction au Shell**
- **Le scripts Shell**
- **Programmation Shell**
  - **Les variables**
  - **Les commandes : echo , read**
  - **Les structures de contrôle**
  - **Opérateurs de comparaison**

Introduction au Shell

Définition

Le **Shell** est un **interpréteur de commande**. Son rôle est d'analyser la commande tapée afin de faire réagir le système pour qu'il réponde aux besoins de l'utilisateur. C'est le premier langage de commandes développé sur Unix par **Steve Bourne**.

Les différents types de Shell:

- **sh** (bourne shell) Le bourne shell est le plus ancien shell unix, et il est présent sur la majorité des systèmes Unix.
- **bash** (bourne again shell) Bash est une version étendue du bourne shell et a été créé sur linux.
- **csh** (c-shell) Le c-shell est un shell écrit en C et est plus convivial que le bourne shell. La syntaxe de ses scripts est très similaire a celle de C. Il provient de la distribution de Berkeley et a été écrite par Bill Joy.
- **tcsh** (c-shell) Le tcsh est une version plus conviviale au niveau de l'utilisateur du csh.
- **ksh** (Korn Shell) pour lequel deux versions majeurs sont aujourd'hui couramment utilisées (ksh 88 et ksh 93) .
- .....

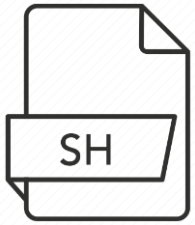
Introduction au Shell

- Un programme shell appelé aussi « script » est un outil facile à utiliser pour construire des applications en regroupant des appels système, outils, utilitaires et programmes compilés.

- Il existe deux moyens de « programmer » en Shell:

  → Le premier est dit en « ***direct*** ». L'utilisateur tape « directement » la ou les commandes qu'il veut lancer.

  → Le second est dit en « ***script*** », appelé aussi « ***batch*** » ou « ***source Shell*** ». L'utilisateur crée un fichier texte par l'éditeur de son choix (par exemple : « vi »). Il met dans ce script toutes les commandes qu'il voudra lui faire exécuter ; en respectant la règle de base de ne mettre **qu'une seule commande par ligne**.



Le script Shell

→ Structure et exécution d'un script

- Toutes les instructions et commandes sont regroupées au sein d'un script. Lors de sont exécution, chaque ligne sera lue une à une et exécutée.
- Une ligne peut se composer de commandes, de commentaires ou être vide.
- Plusieurs instructions par lignes sont possibles, séparées par le « **;** » ou liées conditionnellement par « **&&** » ou « || ». Le « ; » est l'équivalent d'un **saut de ligne**.
- Par convention l' extension des scripts ***Shell*** se terminent généralement (pas obligatoirement) par « **.sh** » pour le *Bourne Shell* et *le Bourne Again Shell*, par « **.ksh** » pour le *Korn Shell* et par « **.csh** » pour *le C-Shell*.
- Quand un script est lancé, un nouveau Shell « fils » est créé qui va exécuter chacune des commandes.

Le script Shell

- Tout script aura comme première ligne, la description du Shell à lancer. On aura donc :

Shell	Première ligne
Bourne Shell	#!/bin/sh
C Shell	#!/bin/csh
Korn Shell	#!/bin/ksh
TC Shell	#!/bin/tcsh
Bourne Another Shell	#!/bin/bash

- Pour rendre un script **exécutable** directement : $ **chmod u+x** monscript
- **Pour l'exécuter** : $ ./monscript
- Pour éviter le ./ :
  $ PATH=$PATH:.
  $ monscript

Le script Shell

- Lors de l'ouverture d'une session, le Shell exécute des fichiers de configuration, qui peuvent contenir des commandes quelconques et sont généralement utilisées pour définir des variables d'environnement et des alias.

  → **sh** exécute le fichier ***~/.profile***

  → **bash** exécute le fichier ***~/.bash_profile*** ou par défaut le fichier ***~/.profile***

  → **csh** exécute le fichier ***~/.cshrc***

  → **tcsh** exécute le fichier ***~/.cshrc***

- On remarque bien que ces fichiers de configuration sont des fichiers cachés.

Remarque : Chaque utilisateur peut ajouter des commandes Shell au profil personnel ***~/.bash_profile***

Le script Shell

→ Les caractères spéciaux Shell

Caractère	Description
*?[] [^]	Substitution les noms de fichiers **Exemple** : cp * DATA copie tous les fichiers dans le répertoire DATA.
&& , ||, !	Opérateurs booléens
;	Permet de séparer plusieurs commandes écrites sur une même ligne.
( )	Regroupe des commandes. **Exemple** (echo "Liste :"; ls) > liste.txt écrit la chaîne Liste : et la liste des fichiers du répertoire courant dans le fichier liste.txt.
&	Permet le lancement d'un processus en arrière plan. Cela permet d'exécuter d'autres commandes pendant qu'un processus est en marche. **Exemple** : netscape&.
|	Permet la communication par tube entre deux commandes.
> , >> , <, <<, 2>, 2>>	Les opérateurs de redirections entrées/sorties

→ Les caractères spéciaux Shell

Caractère	Description
#	Introduit un commentaire. Donc tout ce qui suit ce caractère dans une ligne est ignoré par le Shell. **Exemple** : # ceci est un commentaire.
\	Déspécialise le caractère qui suit. C'est-à-dire que si le caractère qui suit celui là est un caractère spécial alors le Shell l'ignorera. **Exemple** : echo Bon*jour affiche bon*jour à l'écran.
'...'	Défini une chaîne de caractères qui ne sera pas évaluée par le Shell. **Exemple** : echo '*?&' affiche sur la sortie standard les caractères spéciaux *?& sans les interpréter.
"..."	Défini une chaîne de caractères dont les variables seront évaluées par le Shell. **Exemple** : echo "Vous êtes $USER."
`...`	Défini une chaîne de caractères qui sera interprétée comme une commande et remplacée par la chaîne qui serait renvoyée sur la sortie standard à l'exécution de la dite commande. **Exemple** : echo `pwd` >> liste.txt écrit à la fin du fichier le chemin et le nom du répertoire courant. Le caractère spécial utilisé s'obtient par la combinaison de touche : AltGr + 7 (c'est l'accent grave).

→ Quotation

- ' " ` \ :changent la façon dont le Shell interprète les caractères spéciaux

Symbole	Signification
' (single-quote)	le shell ignore tout caractère spéciaux entre deux '
" (double-quote)	le shell ignore tout caractère spéciaux entre deux ", à l'exception de $ et \ et '
\ (antislash ou backslash) ` (backquote ou antiquote)	le shell ignore le caractère spécial suivant le \ le shell exécute ce qu'il y a entre deux `

→ Les commentaires

Une ligne de commentaire commence toujours par le caractère « # ». Un commentaire peut être placé en fin d'une ligne comportant déjà des commandes.

Exemple:

```
# La ligne suivante effectue un ls
ls # La ligne en question
```

→ Les variables d'environnement

→ Le Shell possède des variables d'environnement qui permettent de garder en mémoire des informations importantes telles que le login de l'utilisateur (stocké dans la variable **$USER**) ainsi que son répertoire de connexion (**$HOME**), la liste des répertoires dans lesquels aller chercher les exécutables des commandes externes (**$PATH**), et bien d'autres encore...

→ La commande **$ env** affiche la liste de toutes les variables d'environnement du **Shell** avec leurs valeurs.

Programmation Shell: Les variables d'environnement

Variable=Valeur	Description
PWD=/home/hugo	Stocke le chemin et le nom du répertoire courant.
HOSTNAME=localhost.ld	Nom du serveur.
LANGUAGE=fr	Suffixe de la langue du système.
USER=hugo	Nom de l'utilisateur.
DISPLAY=unix:0.0	Adresse du terminal d'affichage.
SHELL=/bin/bash	Chemin et nom du programme Shell (il en existe plusieurs différents).
HOME=/home/hugo	Chemin du répertoire de connexion.
PATH=:/usr/local/bin: /bin:/usr/bin: /usr/X11R6/bin	Liste des répertoires où chercher les exécutables des commandes externes.

→ Autres variables

Variable	Description
$$	PID du processus Shell en cours.
$!	PID du dernier processu lancé en background.
$?	Code erreur de retour de la dernière commande (0:vrai; sinon faux).

- **Exemples**

```
$ echo "le répertoire de connexion est : $HOME"
 #affiche le nom du répertoire personnel de l'utilisateur,
#mémorisé par la variable HOME
```

→ **Le répertoire de connexion est : /home/usms**

→ Les noms de variables

Les noms des variables peuvent être composés:

- D'une suite de lettres, de chiffres et du caractère _ : ce cas correspond aux variables créées par l'utilisateur
- D'un chiffre : ce cas correspond aux paramètres des fichiers de commandes
- De l'un des caractères quelconques * @ # ? - $! ce cas correspond à un ensemble de variables gérées par le Shell

→Déclaration

Il n'est pas nécessaire de déclarer une variable avant de l'utiliser. Les objets possédés par les variables sont d'un seul type des chaînes de caractères.

→ Référencement du contenu d'une variable

Pour référencer la valeur d'une variable, on utilise la notation consistant à écrire le signe **$** suivi **du nom de la variable**.

Exemple

```
i=2; echo $i;
```

→ **2**

→ Affectation de valeurs aux variables

Pour affecter une valeur à une variable, il suffit de taper le nom de variable, suivi du signe égal, suivi de la valeur que l'on désire affecter à la variable. La syntaxe d'une affectation est la suivante :

nom-de-variable = chaîne-de-caractères

Exemple

```
Var1="bonjour"
PERSONNE=$USER
```

- On peut affecter une chaîne vide à une variable de 3 manières différentes:

```
V1=
V2=' '
V3=" "
```

Si la chaîne-de-caractères à affecter à la variable comporte des blancs, il faut l'entourer à l'aide des quotes et il ne faut surtout pas laisser des espaces autour du signe =

MESSAGE='hello everybody'

Programmation Shell: Les variables utilisateurs

→ Evaluation de commandes

Il est possible de stocker le résultat d'une commande dans une variable en utilisant:

- les back quottes (anti-apostrophes qu'on obtient en tapant Altgr+7)
- les parenthèses

Exemple

```
Fichiers=$(ls /usr/include/*.h | grep std)
#on envoie le flux de sortie de la commande ls comme flux
#d'entrée pour la commande grep
echo $Fichiers
```

→ /usr/include/stdint.h /usr/include/stdio_ext.h
→ /usr/include/stdio.h
→ /usr/include/stdlib.h /usr/include/unistd.h

```
$ n=`ls | wc -l`
$ echo $n
```

→ 50

Programmation Shell: Les variables utilisateurs

→ Les variables spéciales $

caractères	significations
$#	Nombre de paramètres
$1 , $2 , etc	Paramètre 1, paramètre 2, etc.
$0	Nom du programme
$*	Tous les paramètres de position, vus comme un seul mot (doit être entre guillemets).
$@	référence la liste des paramètres sous la forme “$1“, “$2“,…., “$9“
$?	Fourni le code de retour.
$_	Le dernier argument de la commande « dernière commande ». Donc $_ retour de la sortie standard (attention avec echo)
$$	PID du processus
$PPID	PID du processus parent
$!	PID de la dernière commande lancée en arrière plan

→ Les variables spéciales $

Exemples

On suppose que l'utilisateur ***toto*** n'est pas défini dans le système :

```
grep toto /etc/passwd
echo $?
```

→ **1**

```
grep usms /etc/passwd
```

usms:x:77:227:utilisateur usms:/users/ usms:/bin/bash

```
echo $?
```

→ **0**

- Si le code retour de la commande **est nul**, ça signifie que la commande ou le programme a été bien exécuté sans erreur.
- Si le code retour est **non nul**, ceci signifie que la commande a rencontré des erreurs lors de son exécution.

Programmation Shell: Les variables utilisateurs

Exemple

```
#!/bin/bash
echo 'programme :' $0
echo 'argument 1 :' $1
echo 'argument 2 :' $2
echo 'argument 3 :' $3
echo 'argument 4 :' $4
echo "nombre d'arguments :" $#
echo "tous:" $*
```

`$ ./arguments un deux trois`

→

programme : ./arguments
argument 1 : un
argument 2 : deux
argument 3 : trois
argument 4 :
nombre d'arguments : 3
tous: un deux trois

Programmation Shell: Les expressions mathématiques

→ Les expressions mathématiques

- Le Shell peut évaluer des expressions arithmétiques délimitées par **$(( ))**

Exemple

```
$ n=1
$ echo $(( n + 1 ))
$ p = $(( n * 5 / 2 ))
$ echo $p
```

Programmation Shell: Les commandes

→ La commande: `echo`

- La commande **echo** permet d'afficher l'expression donnée en paramètre. Cette dernière peut être :
  - → Soit une variable
  - → Soit une chaîne
  - → Soit une expression composée de chaines et de variables.

→ La commande: `read`

- La commande **read** lit la saisie de l'utilisateur à partir du canal d'entrée standard (clavier) et stocke ces données dans des variables du Shell.
- Read lit une ligne entrée, la découpe en mots séparés par des espaces et affecte chaque mot aux variables de la liste-de-variables.
- Les noms de ces variables sont transmis comme paramètres de ***read*** dont la syntaxe est la suivante : **read var1 [var2 …]**.
- Lorsque **read** est traité, le Shell attend une entrée de la part de l'utilisateur.

→ Autre commandes

- **Commande `shift`** : La commande shift permet donc d'accéder aux paramètres qui sont au-delà du neuvième paramètre
- **Commande `exit`** : permet de terminer un Shell en transmettant un code de retour. Si exit est invoqué avec un paramètre, ce paramètre qui est pris comme code de retour. Sinon, le code de retour sera le code de retour de la dernière commande exécutée.
- **Commande `expr`:** cette commande évalue les arguments comme une expression. Le résultat est envoyé sur la sortie standard. *arguments* est une expression comprenant des opérateurs.
- **Commande `test`:** permet d'exprimer des prédicats sur les chaînes de caractères, sur les entiers et sur les fichiers. La valeur de ces prédicats peut ensuite être testée dans une structure **if**, **while** ou **until**. Syntaxe: **`test` prédicat** ou **`test` [prédicat]**
- **Commande `let` et `(( ))`** : Les commandes let et les (( )) sont équivalentes, elles servent à :
  - Affecter des variables numériques. : **let x=1 ; ((x=4))**
  - Faire des calculs : **let x=x+1 ; ((i +=1))**

→ Structure if

Syntaxe 1	Syntaxe 2	Syntaxe 3
if [condition] **then** action1 **fi**	**if** [condition] **then** action1 **else** action2 **fi**	**if** [condition] **then** action1 **elif** [condition] **then** action2 **elif** [condition] **then** action3 **else** action4 **fi**

→ Structure if

Exemple

```
if test -d $1
then echo "$1 est un répertoire"
elif test -f $1
then echo "$1 est un fichier
elif test -L $1
then echo "$1 est un lien symbolique"
else echo "$1 autre ..."
fi
```

→ Structure `case`

Syntaxe

case valeur_de_variable **in**
val1)
commandes
;;
val2)
commandes
;;
...
*)
commandes
esac

→ **Structure case**

Exemple

```
case $# in
0) echo "aucun parametre " ;;
echo "Syntaxe : $0 <nom d'utilisateur>";;
1) echo "1 parametre passe au programme : $1";;
2) echo "2 parametres passes au programme : $1 et
$2";;
*) echo "TROP DE PARAMETRES !"
esac
```

→ La boucle `for`

Syntaxe

for var **in** liste

do

commandes

done

Exemple

```
for file in *.sh
do
cat $file
done
```

→ La boucle `while`

Syntaxe

while [condition]

do

commandes

done

Exemple

```
while [ "$var1" != "fin" ]
do
echo "Variable d'entrée #1 (quitte avec fin) "
read var1
echo "variable #1 = $var1"
echo
done
```

Programmation Shell: Opérateurs de comparaison

→ Tests sur les fichiers (et sur les répertoires)

- **-e fichier** :Vrai si le *fichier*/*répertoire* existe
- **-s fichier** :Vrai si le *fichier* à une taille supérieure à 0
- **-z fichier** :Vrai si le *fichier* fait 0 octet (donc si il est vide)
- **-r fichier** :Vrai si le *fichier/répertoire* est lisible
- **-w fichier** :Vrai si le *fichier/répertoire* est modifiable
- **-x fichier** :Vrai si le *fichier* est exécutable ou si le répertoire est accessible
- **-O fichier** :Vrai si le *fichier/répertoire* appartient à l'utilisateur
- **-G fichier** :Vrai si le *fichier/répertoire* appartient au groupe de l'utilisateur
- **-b nom** :Vrai si *nom* représente un périphérique (pseudo-fichier) de type bloc (disques et partitions de disques généralement)

Programmation Shell: Opérateurs de comparaison

→ Tests sur les fichiers (et sur les répertoires)

- **-c *nom*** **:**Vrai si *nom* représente un périphérique (pseudo-fichier) de type caractère (terminaux, modems et port parallèles par exemple)
- **-d *nom* :**Vrai si *nom* représente un répertoire
- **-f *nom* :**Vrai si *nom* représente un fichier
- **-L *nom* :**Vrai si *nom* représente un lien symbolique
- **-p *nom* :**Vrai si *nom* représente un tube nommé
- ***f1* -nt *f2* :**Vrai si les deux fichiers existent et si *f1* est plus récent que *f2*
- ***f1* -ot *f2* :**Vrai si les deux fichiers existent et si *f1* est plus ancien que *f2*
- ***f1* -ef *f2* :**Vrai si les deux fichiers représentent un seul et même fichier

Programmation Shell: Opérateurs de comparaison

→ Tests sur les entiers

- ***entier1* -eq *entier2* :**Vrai si *entier1* est égal à *entier2*
- ***entier1* -ge *entier2* :**Vrai si *entier1* est supérieur ou égal à *entier2*
- ***entier1* -gt *entier2*** :Vrai si *entier1* est strictement supérieur à *entier2*
- ***entier1* -le *entier2*** :Vrai si *entier1* est inferieur ou égal à *entier2*
- ***entier1* -lt *entier2*** :Vrai si *entier1* est strictement inferieur à *entier2*
- ***entier1* -ne *entier2*** :Vrai si *entier1* est diffèrent de *entier2*

Programmation Shell: Opérateurs de comparaison

→Tests sur les chaînes de caractères

- **-n ”chaîne”** :Vrai si la chaîne n’est pas vide
- **-z ”chaîne”** :Vrai si la chaîne est vide
- **”chaine1” = ”chaine2”** :Vrai si les deux chaînes sont identiques
- **”chaine1” != ”chaine2”** :Vrai si les deux chaînes sont différentes

Support de cours

SYSTÈMES D'EXPLOITATION

CHAPITRE V: PROGRAMMATION SHELL