

Systemes d'exploitation

Institut Supérieur du Numérique



Année Scolaire 2021/2022

Source Audrey Queudet
Université de Nantes

Plan du cours

 Introduction aux systèmes d'exploitation


 Présentation générale d'UNIX

 **Programmation shell** Processus et

 parallélisme

Le Shell UNIX

(1)

- **Interface en ligne de commande UNIX** (=IHM dans laquelle la communication entre l'utilisateur et l'ordinateur s'effectue en mode texte)
- Le shell est utilisable en conjonction avec un **terminal** 
- Lors du **login**, l'utilisateur est connecté avec un shell défini lors de la création de son compte. Possibilité de le modifier via la commande **chsh**
- 2 modes d'utilisation :
 - Simple interpréteur de commandes (mode interactif) Langage de
 - programmation interprété (scripts)

Le Shell UNIX (2)

- Le shell affiche une invite en début de ligne, appelée **prompt** ('\$' ou '#' ou '%'), pour indiquer à l'utilisateur qu'il attend l'entrée d'une commande



```
root@uberhaxor: /
File Edit View Terminal Tabs Help
root@uberhaxor:/# sh
# echo $0
sh
# ls
afs          dev          lib64        sbin
bin          dirs.utils  lost+found   srv
boot        docs        Makefile     sys
catalog     etc         media        test
cdrom       folding     mnt          tmp
changelog.modules home        opt          usr
changelog.source initrd      pending      var
changelog.utils  initrd.img postinst.modules vmlinuz
control.modules  initrd.img.old proc         vmlinuz.old
control.source  internal   README.Debian
control.utils   lib        root
copyright      lib32      rules
#
```

Pourquoi utiliser un shell : avantages ?

- Dans de nombreux contextes, on ne dispose pas d'interface graphique
- Travail en ligne de commande souvent plus efficace qu'à travers une interface graphique
- Automatisation de tâches répétitives
- Meilleure compréhension du système UNIX (fichiers de configuration...)




Pourquoi ne pas utiliser un shell : inconvénients ?

- Documentation difficile d'accès pour le débutant
- Syntaxe cohérente mais parfois obscure (concision vs. clarté)
- Messages d'erreurs parfois difficilement exploitables
- Relative lenteur



Les différents shells

- Shell de Stephen R. Bourne : Bourne shell : **sh**
Bourne-Again shell : **bash**
- Shell de David Korn : 
- Korn shell : **ksh**
→
C shell : **csh**
- Tenex C shell (version moderne du csh) : **tcsh**
- Shell de Kenneth Almquist prenant peu de place sur le disque :
• Almquist shell : **ash**
Debian Almquist shell : **dash**
- Z Shell (**zsh**), intégrant les fcts les plus pratiques de bash, ksh et tcsh

Scripts shell en bash : les concepts de base

- **Caractères spéciaux**
- **Variables**
 - variables d'environnement
 - variables de l'utilisateur
- **Opérateurs**
- **Structures de contrôle**
 - exécution conditionnelle
 - choix multiple
 - boucle for
 - boucles while et until
- **Expressions régulières**



Caractères spéciaux (ou métacaractères)

<i>Caractère</i>	<i>Description</i>
*	Métacaractère qui remplace n'importe quelle chaîne de caractères (même vide)
?	Métacaractère qui remplace un caractère quelconque
;	Permet de séparer plusieurs commandes écrites sur une même ligne
()	Regroupe des commandes
&	Permet le lancement d'un processus en arrièreplan
 	Permet la communication par tube entre deux commandes
#	Introduit un commentaire. Tout ce qui suit dans une ligne est ignoré par le shell
\	Déspecialise le caractère qui suit
'...'	Définit une chaîne de caractères qui ne sera pas évaluée par le shell
"..."	Définit une chaîne de caractères dont les variables seront évaluées par le shell
`...`	Définit une chaîne de caractères qui sera interprétée comme une commande et remplacée par la chaîne qui serait renvoyée à l'exécution de la dite commande

Variables d'environnement (1)

<i>Variable</i>	<i>Description</i>
PWD	Stocke le chemin et le nom du répertoire courant
HOSTNAME	Nom du serveur
HISTSIZE	Taille de l'historique des dernières commandes passées au shell
LANGUAGE	Suffixe de la langue du système
PS1	Chaîne apparaissant à l'invite du Shell
USER	Nom de l'utilisateur
DISPLAY	Adresse du terminal d'affichage
SHELL	Chemin et nom du programme Shell
HOME	Chemin du répertoire de connexion
PATH	Liste des répertoires où chercher les exécutables des commandes externes

Variables d'environnement (2)

- Les variables d'environnement sont manipulées via les commandes :

→ **printenv** : affiche la liste des variables d'environnement **export**

→ **VARIABLE=VALEUR** : donne une valeur à une variable **echo \$VARIABLE** :

→ affiche la valeur de la variable

- Exemples :

```
printenv
PWD=/home/Olivier
LANG=fr
SHELL=/bin/bash

printenv LANG
fr
```

Variables de l'utilisateur

- L'utilisateur peut déclarer facilement de nouvelles variables par l'affectation directe d'une valeur (numérique, chaîne de caractères) :

```
ma_variable=valeur
```

- Exemples :

```
EMAIL=audrey.queudet@univ-nantes.fr
moi=audrey
vous=L2
phrase1="Bonjour $vous, moi c'est $moi"
phrase2='Bonjour $vous, moi c'est $moi'
echo $phrase1
Bonjour L2, moi c'est audrey
echo $phrase2
Bonjour $vous, moi c'est $moi

rep=`pwd` echo $rep
/home/queudet/data
```

Opérateurs sur les fichiers

<i>Opérateur</i>	<i>Description</i>
<i>-e filename</i>	Vrai si <i>filename</i> existe
<i>-d filename</i>	Vrai si <i>filename</i> est un répertoire
<i>-f filename</i>	Vrai si <i>filename</i> est un fichier ordinaire
<i>-L filename</i>	Vrai si <i>filename</i> est un lien symbolique
<i>-r filename</i>	Vrai si <i>filename</i> est lisible(r)
<i>-w filename</i>	Vrai si <i>filename</i> est modifiable(w)
<i>-x filename</i>	Vrai si <i>filename</i> est exécutable(x)
<i>file1 -nt file2</i>	Vrai si file1 plus récent que file2
<i>file1 -ot file2</i>	Vrai si file1 plus ancien que file2

Opérateurs sur les chaînes

<i>Opérateur</i>	<i>Description</i>
<i>-z chaîne</i>	Vrai si la <i>chaîne</i> est vide
<i>-n chaîne</i>	Vrai si la <i>chaîne</i> est non vide
<i>chaîne1 = chaîne2</i>	Vrai si les deux <i>chaînes</i> sont égales
<i>Chaîne1 != chaîne2</i>	Vrai si les deux <i>chaînes</i> sont différentes

Opérateurs arithmétiques

<i>Opérateur</i>	<i>Description</i>
+	addition
-	soustraction
*	multiplication
/	division
**	puissance
%	modulo

- Expressions arithmétiques :

```
$(( ... ))  
n=1  
echo $(( 5*n+1 ))
```

Opérateurs de comparaison numérique

<i>Opérateur</i>	<i>Description</i>
<i>num1 -eq num2</i>	égalité
<i>num1 -ne num2</i>	inégalité
<i>num1 -lt num2</i>	inférieur (<)
<i>num1 -le num2</i>	inférieur ou égal (\leq)
<i>num1 -gt num2</i>	supérieur (>)
<i>num1 -ge num2</i>	supérieur ou égal (\geq)

Opérateurs booléens

<i>Opérateur</i>	<i>Description</i>
-a	ET logique
-o	OU logique
!	NON logique

Structures de contrôle : exécution conditionnelle

- L'instruction **if** permet d'exécuter des instructions si une condition est vraie

→ Le bloc **if/then**

```
if [ condition ]  
then  
    actions  
fi
```

→ Le bloc **if/then/else**

```
if [ condition ]  
then  
    action1  
else  
    action2  
fi
```

→ Enchaînement de plusieurs conditions

```
if [ condition1 ]  
then  
    action1  
elif [ condition2 ]  
then  
    action2  
elif [ condition 3 ]  
then  
    action3  
else  
    action4  
fi
```

Structures de contrôle : choix multiple

- L'instruction **case** permet de choisir une suite d'instructions suivant la valeur d'une expression

```
case "$x" in
    case1)
        actions1
        ;;
    case2)
        actions2
        ;;
    ...
    caseN)
        actionsN
        ;;
esac
```

Structures de contrôle :

boucle for

- L'instruction **for** permet une exécution répétitive d'une suite d'instructions

→ Schéma classique

```
for VAR in LISTE
do
    actions done
```

→ Schéma alternatif

```
for ((initialisation de VAR; contrôle de VAR; modification de VAR))
do
    actions done
```

Structures de contrôle : boucles while et until

- L'instruction **while** permet une exécution répétitive d'une suite d'instructions tant qu'une condition est vraie

```
while [ condition ]  
do  
    actions done
```



Condition de continuation de la boucle

- L'instruction **until** permet une exécution répétitive d'une suite d'instructions jusqu'à ce qu'une condition soit vraie

```
until [ condition ]  
do  
    actions done
```



Condition d'arrêt de la boucle

Expressions régulières : définition

- Une **expression régulière** est un patron qui recouvre un ensemble de chaînes de caractères

- Les expressions régulières sont puissantes pour extraire des lignes particulières d'un fichier ou d'un résultat

- Beaucoup de commandes UNIX emploient des expressions régulières :

grep, find, sed, awk. Bash a des fonctionnalités intégrées pour cibler des

patrons et peut reconnaître des classes de caractères et des intervalles.

Expressions régulières : les opérateurs

<i>Opérateur</i>	<i>Description</i>
.	Correspond a tout caractère
?	L'élément précédent est optionnel et sera présent au plus une fois
*	L'élément précédent sera présent zéro fois ou plus
+	L'élément précédent sera présent une fois ou plus
{N}	L'élément précédent sera présent exactement N fois
{N, }	L'élément précédent sera présent N ou plus de fois
{N,M}	L'élément précédent sera présent au moins N fois, mais pas plus de M fois
-	Représente l'intervalle s'il n'est pas le premier ou le dernier dans une liste
^	Correspond à une chaîne vide au début de la ligne; Représente aussi les caractères ne se trouvant pas dans l'intervalle d'une liste
\$	Correspond à la chaîne vide à la fin d'une ligne
\b	Correspond à la chaîne vide au début ou à la fin d'un mot

Structure d'un script shell

- Un script bash est un simple fichier texte exécutable (droit x) dont la première ligne doit obligatoirement être **#!/bin/bash**

#! Sur la première ligne : interpréteur du présent script (# ! suivi du chemin complet du shell utilisé plus d'éventuels arguments)

#commentaires Les ligne de commentaire sont précédées de #

- Dans un éditeur de texte, écrivons le script suivant :

```
#!/bin/bash #  
# Shell-script affichant "bonjour" sur la sortie standard  
#  
message='bonjour'  echo $message
```

- Enregistrons ce script sous le nom **bonjour.sh**

Exécution de scripts bash

- Dans un terminal, en ligne de commande, rendons le script exécutable :

```
chmod u+x bonjour.sh
```

- Exécutons le script (plusieurs solutions) :

```
bonjour.sh
```

ou

```
. bonjour.sh
```

ou

```
sh bonjour.sh
```

 ajouter le chemin qui contient le script à la variable PATH
export PATH=\$PATH:/home/mes_scripts

ou

```
exec bonjour.sh
```

Passage de paramètres à un script

- Il est possible d'exécuter un script en lui passant un certain nombre de paramètres (ou arguments), comme pour n'importe quelle autre commande :

```
mon_script.sh arg1 arg2 ... argN
```

- En bash, les arguments de la ligne de commande sont stockées dans des variables spéciales :

<i>Variable</i>	<i>Description</i>
\$#	Le nombre de paramètres passés au script shell
\$* et \$@	Tous les paramètres passés au script shell
\$0	Nom de la commande
\$1	Valeur du premier paramètre
\$i	Valeur du ième paramètre si i compris entre 1 et 9
\$9	Valeur du neuvième paramètre

Passage de paramètres : exemple

```
# !/bin/sh
# Mon programme qui affiche les parametres de #la ligne
de commande
echo "* Le nom du programme est : $0"
echo "* Le troisieme parametre est : $3"
echo "* Le nombre de parametre est : $#"
```

(The following lines are not visible in the image but are part of the script content shown in the next block)

```
echo "* Tous les parametres (mots individuels) : $*"
echo "* Tous les parametres : $@"
exit 0
$
```

\$ **./script2.sh un "deux" "trois quatre" cinq**

```
* Le nom du programme est : ./script2.sh
* Le troisieme parametre est : trois quatre
* Le nombre de parametre est : 4
* Tous les parametres (mots individuels) : un deux trois
quatre cinq
* Tous les parametres : un deux trois quatre cinq
$
```

Passage de paramètres : précaution

```
# !/bin/sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12
exit 0

$ ./script3 faux.sh un deux trois quatre cinq six
sept huit neuf dix onze douze
un deux trois quatre cinq six sept huit neuf un0 un1
un2
$
# !/bin/sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12}
exit 0

$ ./script3 vrai.sh un deux trois quatre cinq six
sept huit neuf dix onze douze
un deux trois quatre cinq six sept huit neuf dix onze
douze
$
```

Opérateurs logiques du shell : &&, ||




- Souvent utilisé pour une forme compacte et élégante.

ET logique :

```
cmd1 && cmd2    if cmd1  
then cmd2  
fi
```

Si la commande 1 réussit, alors faire la commande 2.

- OU logique

```
cmd1 || cmd2    if ! cmd1  
then cmd2  
fi
```

Si la commande 1 a échoué, alors faire la commande 2.

```
cat toto || touch toto
```

Fonction

- On peut regrouper les commandes au sein d'une fonction.
Une fonction se définit de la manière suivante :

```
nom_fonction ()  
{  
  liste-commandes  
}
```

- Les paramètres au sein de la fonction sont accessibles via \$1, \$2,... \$@, \$#.
- L'appel d'une fonction se fait de la manière suivante :
 nom_fonction parametre1 parametre2...
- Une fonction doit être déclarée avant de pouvoir être exécutée.

Code de retour : return, exit

- **return n** :Renvoie une valeur de retour pour la fonction shell.
- **exit n**: Provoque l'arrêt du shell courant avec un code retour de n si celui-ci est spécifié. S'il n'est pas spécifié, il s'agira de la valeur de retour de la dernière commande exécutée.

Petits calculs numériques : expr

- `expr` chaine: évalue la chaine de caractère représentant des opérations.

```
$ titi=3
$ echo $titi
3
$ titi=$titi+1
$ echo $titi
3+1
$ tutu=3
$ tutu='expr $tutu + 1'
$ echo $tutu
4
$
```