Construction de Scripts Shell Basiques

Programmation Shell sous Linux

Ilias TOUGUI

L'Ecole Supérieure d'Informatique et du Numérique

INF2132 - Systèmes d'Exploitation

PLAN DU COURS

1. Utilisation de Commandes Multiples
2. Création d'un Fichier Script
3. Affichage de Messages
4. Substitution de Commandes
5. Redirection des Entrées et Sorties
6. Pipes
7. Calculs Mathématiques
8. Statut de Sortie

Introduction
Concepts de base

Maintenant que nous avons couvert les bases du système Linux et de la ligne de commande, il est temps de commencer à coder.

Objectifs de ce chapitre :

- Comprendre les bases de l'écriture de scripts shell
- Maîtriser les concepts fondamentaux avant de créer des scripts complexes
- Apprendre à combiner plusieurs commandes

Concept clé : La capacité d'entrer plusieurs commandes et de traiter les résultats de chaque commande, même en passant les résultats d'une commande à une autre.

Utilisation de Commandes Multiples

Chaînage de commandes avec le point-virgule

Le shell permet de chaîner des commandes ensemble en une seule étape.

Méthode : Utiliser le point-virgule (;) pour séparer les commandes

```
$ date ; who
Tue Nov  4 09:30:50 PM +01 2025
ilias    seat0        2025-11-04 10:09
ilias    tty2         2025-11-04 10:09
$
```

Félicitations ! Vous venez d'écrire un script shell !

Utilisation de Commandes Multiples

Limitations du chaînage en ligne

Avantages :

- Rapidité pour de petits scripts
- Possibilité de combiner autant de commandes que souhaité
- Limite maximale de 255 caractères sur la ligne de commande

Inconvénient majeur :

- Vous devez entrer la commande complète à chaque exécution
- Pas pratique pour des scripts réutilisables

Solution : Au lieu d'entrer manuellement les commandes, vous pouvez les combiner dans un simple fichier texte. Quand vous devez exécuter les commandes, exécutez simplement le fichier texte.

Création d'un Fichier Script

Éléments essentiels d'un script

Pour placer des commandes shell dans un fichier texte, vous devez utiliser un éditeur de texte pour créer un fichier et y entrer les commandes.

Ligne obligatoire : Le shebang

```
#!/bin/bash
```

Explication :

- Le symbole # est normalement utilisé pour les commentaires
- La première ligne est un cas spécial : #! indique au shell quel shell utiliser
- Permet d'exécuter un script avec un shell différent de celui en cours

Création d'un Fichier Script

Structure d'un script simple

Exemple de script :

```
#!/bin/bash
# Ce script affiche la date et qui est connecté
date
who
```

Points importants :

- Chaque commande sur une ligne séparée (ou utiliser le point-virgule)
- Le shell traite les commandes dans l'ordre d'apparition
- Les lignes commençant par # (sauf #!/bin/bash) sont des commentaires
- Les commentaires sont ignorés par le shell

Conseil : Laissez des commentaires pour vous rappeler ce que fait le script quand vous y reviendrez plus tard !

Création d'un Fichier Script

Problème 1 : Le PATH

Sauvegardez ce script dans un fichier appelé **test1**. Vous êtes presque prêt !

Première tentative d'exécution :

```
$ test1
bash: test1: command not found
$
```

Problème : Le shell utilise la variable d'environnement PATH pour trouver les commandes.

```
$ echo $PATH
/home/ilias/.local/bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:
/snap/bin:/snap/bin
$
```

Création d'un Fichier Script

Solutions pour le PATH

Deux solutions possibles :

1. Ajouter le répertoire du script à la variable PATH
2. Utiliser un chemin absolu ou relatif pour référencer le script

Nous utilisons la deuxième méthode :

```
$ ./test1
bash: ./test1: Permission denied
$
```

Note : Le point simple (.) représente le répertoire courant dans le shell.

Nouveau problème : Permissions insuffisantes !

Création d'un Fichier Script

Problème 2 : Les permissions

Le shell a trouvé le fichier, mais il y a un autre problème !

Vérification des permissions :

```
$ ls -l test1
-rw-rw-r--    1 user     user    73 Sep 24 19:56 test1
$
```

Explication :

- La valeur umask détermine les permissions par défaut
- Dans Ubuntu, umask est configuré à 002
- Le fichier est créé avec seulement les permissions lecture/écriture
- Pas de permission d'exécution (x) !

Création d'un Fichier Script

Solution : chmod

Solution : Donner la permission d'exécution avec chmod

```
$ chmod u+x test1
$ ./test1
Tue Nov  4 09:40:54 PM +01 2025
ilias    seat0        2025-11-04 10:09
ilias    tty2         2025-11-04 10:09
$
```

Succès ! Tous les éléments sont maintenant en place pour exécuter le nouveau fichier de script shell.

Affichage de Messages

La commande echo

La plupart des commandes shell produisent leur propre sortie, qui s'affiche sur le moniteur de la console. Souvent, vous voudrez ajouter vos propres messages texte pour aider l'utilisateur du script.

La commande echo :

```
$ echo This is a test
This is a test
$
```

Remarque : Par défaut, vous n'avez pas besoin de guillemets pour délimiter la chaîne que vous affichez.

Affichage de Messages

Gestion des guillemets

Problème avec les apostrophes :

```
$ echo Let's see if this'll work
Lets see if thisll work
$
```

Solution : Utiliser des guillemets doubles ou simples

```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
$

$ echo 'Amine says "scripting is easy".'
Amine says "scripting is easy".
$
```

Règle : Si vous utilisez un type de guillemets dans votre texte, utilisez l'autre type pour délimiter la chaîne.

Affichage de Messages

Intégration dans un script

Exemple d'utilisation dans un script :

```
$ cat test1
#!/bin/bash
# Ce script affiche la date et qui est connecté
echo  The time and date are:
date
echo "Let's see who's logged into the system:"
who
$
```

Sortie du script :

```
$ ./test1
The time and date are:
Tue Nov  4 09:40:54 PM +01 2025
Let's see who's logged into the system:
ilias    seat0        2025-11-04 10:09
ilias    tty2         2025-11-04 10:09
```

Affichage de Messages

L'option -n

Afficher sur la même ligne : Le paramètre -n

Changez la première instruction echo en :

```
echo -n "The time and date are: "
```

Résultat :

```
$ ./test1
The time and date are: Mon Feb 21 15:42:23 EST 2014
Let's see who's logged into the system:
Christine tty2         2014-02-21 15:26
Samantha tty3         2014-02-21 15:26
$
```

Important : Utilisez des guillemets autour de la chaîne pour garantir un espace à la fin de la chaîne affichée.

Substitution de Commandes

Concept

Une des fonctionnalités les plus utiles des scripts shell est la capacité d'extraire des informations de la sortie d'une commande et de les affecter à une variable.

Deux syntaxes possibles :

1. Le caractère backtick (‘)

```
testing=‘date‘
```

2. Le format $()

```
testing=$(date)
```

Attention : Le backtick n'est pas une apostrophe normale ! Sur un clavier US, il se trouve généralement sur la même touche que le tilde (~).

Substitution de Commandes

Exemple simple

Exemple de base :

```
$ cat test5
#!/bin/bash
testing=$(date)
echo "The date and time are: " $testing
$
```

Sortie :

```
$ chmod u+x test5
$ ./test5
The date and time are:  Mon Jan 31 20:23:25 EDT 2014
$
```

Remarque : La variable **testing** reçoit la sortie de la commande date. Une fois capturée, vous pouvez faire ce que vous voulez avec cette valeur !

Substitution de Commandes

Exemple pratique : fichier de log

Cas d'usage populaire : Créer des noms de fichiers uniques avec la date

```
#!/bin/bash
# Copier le listing du répertoire /usr/bin dans un fichier log
today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
```

Format de date : +%y%m%d = année-mois-jour sur 2 chiffres

```
$ date +%y%m%d
251104
$
```

Résultat : Création d'un fichier **log.251104**

```
-rw-rw-r-- 1 ilias ilias 118607 Nov  4 22:29 log.251104
```

Substitution de Commandes

Notion de sous-shell

Important : La substitution de commandes crée un sous-shell

Qu'est-ce qu'un sous-shell ?

- Un shell enfant séparé généré depuis le shell qui exécute le script
- Les variables créées dans le script ne sont pas disponibles pour le sous-shell

Création de sous-shells :

- Exécution d'une commande avec ./chemin depuis la ligne de commande
- Substitution de commandes

Pas de sous-shell :

- Exécution sans chemin
- Commandes intégrées au shell (built-in)

Conseil : Soyez prudent lors de l'exécution de scripts depuis la ligne de commande !

Redirection des Entrées et Sorties

Introduction

Parfois, vous voulez sauvegarder la sortie d'une commande au lieu de simplement l'afficher sur le moniteur. Le shell bash fournit plusieurs opérateurs qui permettent de rediriger la sortie d'une commande vers un emplacement alternatif (comme un fichier).

La redirection peut être utilisée pour :

- Sortie : envoyer la sortie d'une commande vers un fichier
- Entrée : rediriger un fichier vers une commande pour l'entrée

Objectif : Apprendre à utiliser la redirection dans vos scripts shell

Redirection de Sortie

Opérateur de redirection de base

Syntaxe : Le symbole supérieur à (>)

```
commande > fichier_sortie
```

Exemple :

```
$ date > test6
$ ls -l test6
-rw-r--r--    1 user     user    29 Nov  4 22:28 test6
$ cat test6
Tue Nov  4 10:34:25 PM +01 2025
$
```

Important : L'opérateur de redirection crée le fichier (avec les paramètres umask par défaut) et redirige la sortie de la commande date vers le fichier test6.

Redirection de Sortie

Écrasement vs Ajout

Attention : Si le fichier de sortie existe déjà, > écrase le contenu !

```
$ who > test6
$ cat test6
ilias    seat0        2025-11-04 10:09
ilias    tty2         2025-11-04 10:09
```

Solution : Utiliser » pour ajouter au lieu d'écraser

```
$ date >> test6
$ cat test6
ilias    seat0        2025-11-04 10:09
ilias    tty2         2025-11-04 10:09
Tue Nov  4 10:39:40 PM +01 2025
```

Cas d'usage : Très utile pour créer des fichiers de log où vous documentez une action sur le système.

Redirection d'Entrée

Opérateur de redirection d'entrée

Syntaxe : Le symbole inférieur à (<)

```
commande < fichier_entrée
```

Moyen mnémotechnique : Le symbole pointe dans la direction du flux de données.

Exemple avec wc :

```
$ wc < test6
      3  15 110
$
```

La commande wc fournit trois valeurs :

- Nombre de lignes dans le texte
- Nombre de mots dans le texte
- Nombre d'octets dans le texte

Redirection d'Entrée

Redirection d'entrée inline

Syntaxe : Le symbole double inférieur à («)

```
commande << marqueur
données
marqueur
```

Exemple :

```
$ wc << EOF
> test string 1
> test string 2
> EOF
      3       9      42
```

Explication :

- Permet de spécifier les données directement sur la ligne de commande
- Le shell invite avec le prompt secondaire (PS2)
- Continue jusqu'à rencontrer le marqueur de fin

Pipes

Concept de base

Parfois, vous devez envoyer la sortie d'une commande à l'entrée d'une autre commande. C'est possible avec la redirection, mais c'est un peu maladroit.

Méthode maladroite :

```
$ ls /usr/bin > bin.list
$ sort < bin.list
2ping
a5toa4
aa-enabled
aa-exec
aa-features-abi
[...]
```

Problème : Nécessite un fichier intermédiaire.

Solution : Utiliser les pipes (|) pour rediriger directement la sortie d'une commande vers une autre !

Pipes

Syntaxe et utilisation

Syntaxe : Le symbole pipe (|) - deux barres verticales

```
commande1 | commande2
```

Important : Ne pensez pas aux pipes comme deux commandes exécutées l'une après l'autre. Le système Linux exécute les deux commandes **en même temps**, en les reliant en interne.

Exemple :

```
$ ls /usr/bin | sort
2ping
a5toa4
aa-enabled
aa-exec
aa-features-abi
[...]
```

Avantage : Pas de fichier intermédiaire ni de zone tampon !

Pipes
Chaînage multiple

Puissance des pipes : Il n'y a pas de limite au nombre de pipes !

Exemple avec pagination :

```
$ ls /usr/bin  | sort | more
```

Séquence de traitement :

1. La commande `ls /usr/bin` produit une liste des executables
2. sort trie cette liste alphabétiquement en temps réel
3. more affiche les données, en s'arrêtant à chaque écran

Résultat : Vous pouvez maintenant lire confortablement la sortie, en faisant des pauses à chaque écran d'information.

Pipes

Combinaison avec la redirection

Encore plus sophistiqué : Combiner pipes et redirection !

```
$ ls /usr/bin | sort > bin.list
$ more bin.list
2ping
a5toa4
aa-enabled
aa-exec
aa-features-abi
[...]
```

Résultat : Les données dans le fichier rpm.list sont maintenant triées !

Usage populaire : Pipe vers more pour les commandes produisant une longue sortie

```
$ ls -l | more
```

Calculs Mathématiques

Introduction

Une autre fonctionnalité cruciale pour tout langage de programmation est la capacité de manipuler des nombres. Malheureusement, pour les scripts shell, ce processus est un peu maladroit.

Deux approches différentes :

1. La commande expr (ancienne méthode)
2. Les crochets bash (méthode moderne)

Limitation : Le shell bash ne supporte que l'arithmétique entière par défaut !

Solution pour les calculs en virgule flottante : Utiliser la calculatrice bash (bc)

La Commande expr

Syntaxe de base

À l'origine, le shell Bourne fournissait une commande spéciale pour traiter les équations mathématiques.

Exemple simple :

```
$ expr 1 + 5
6
```

Opérateurs reconnus par expr :

- Arithmétiques : +, -, *, /, %
- Comparaison : <, <=, =, !=, >=, >
- Logiques : |, &
- Chaînes : substr, index, length, match

Important : La commande expr existe toujours pour la compatibilité avec le shell Bourne.

La Commande expr

Problèmes avec les caractères spéciaux

Problème : Beaucoup d'opérateurs ont d'autres significations dans le shell !

Exemple d'erreur :

```
$ expr 5 * 2
expr: syntax error
$
```

Solution : Utiliser le caractère d'échappement (backslash)

```
$ expr 5 \* 2
10
$
```

Conséquence : L'utilisation de la commande expr devient vraiment lourde !

La Commande expr

Utilisation dans les scripts

Dans un script, expr est tout aussi maladroit :

```
$ cat test6
#!/bin/bash
# Exemple d'utilisation de la commande expr
var1=10
var2=20
var3=$(expr $var2 / $var1)
echo The result is $var3
```

Sortie :

```
$ chmod u+x test6
$ ./test6
The result is 2
```

Note : Vous devez utiliser la substitution de commandes pour extraire la sortie de expr !

Utilisation des Crochets

Méthode moderne bash

Le shell bash inclut expr pour la compatibilité, mais il fournit aussi une méthode **beaucoup plus facile** pour effectuer des équations mathématiques.

Syntaxe : $[opération]

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2=$[$var1 * 2]
$ echo $var2
12
```

Avantages :

- Syntaxe beaucoup plus simple
- Pas besoin d'échapper les caractères spéciaux
- Le shell sait que ce n'est pas un wildcard car c'est dans les crochets

Utilisation des Crochets

Exemple dans un script

Script avec crochets :

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo The final result is $var4
```

Sortie :

```
$ chmod u+x test7
$ ./test7
The final result is 500
```

Remarque : Vous pouvez utiliser des parenthèses pour contrôler l'ordre des opérations !

Limitation : Arithmétique Entière

Problème avec les divisions

Limitation majeure : Le shell bash ne supporte que l'arithmétique entière !

```
$ cat test8
#!/bin/bash
var1=100
var2=45
var3=$[$var1 / $var2]
echo The final result is $var3
```

Résultat :

```
$ chmod u+x test8
$ ./test8
The final result is 2
```

Problème : 100/45 = 2.22... mais le résultat est tronqué à 2 !

Solution : La Calculatrice bash (bc)

Introduction à bc

Solution la plus populaire : Utiliser la calculatrice bash intégrée (bc)

bc est en fait un langage de programmation ! Il permet :

- Nombres (entiers et virgule flottante)
- Variables (simples et tableaux)
- Commentaires (# ou /* */)
- Expressions
- Instructions de programmation (if-then, etc.)
- Fonctions

Accès depuis le shell :

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006
This is free software with ABSOLUTELY NO WARRANTY.
```

Bases de bc

Utilisation interactive

Exemple d'utilisation :

```
$ bc
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$
```

Pour quitter : Entrez quit

Important : L'arithmétique en virgule flottante est contrôlée par la variable intégrée **scale**. Vous devez définir cette valeur au nombre de décimales souhaité !

Bases de bc

Contrôle de la précision avec scale

Variable scale : Définit le nombre de décimales

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
$
```

Explication :

- Par défaut, scale = 0 (pas de décimales)
- Après scale=4, bc affiche 4 décimales
- L'option -q supprime la bannière de bienvenue

Bases de bc

Variables dans bc

bc comprend aussi les variables :

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```

Fonctionnalités :

- Définir des variables avec =
- Utiliser les variables dans les calculs
- L'instruction print affiche les variables et les nombres
- Les variables restent disponibles durant toute la session bc

Utilisation de bc dans les Scripts

Format de base

Comment utiliser bc dans un script ?

- Utiliser la substitution de commande
- Combiner avec la redirection inline
- Capturer le résultat dans une variable

Format général :

```
variable=$(echo "options; expression" | bc)
```

Exemple simple :

```
$ cat test9
#!/bin/bash
var1=$(echo "scale=4; 3.44 / 5" | bc)
echo The answer is $var1
```

Utilisation de bc dans les Scripts

Résultats et exemples avancés

Résultat :

```
$ chmod u+x test9
$ ./test9
The answer is .6880
```

Exemple avec variables :

```
$ cat test10
#!/bin/bash
var1=100
var2=45
var3=$(echo "scale=4; $var1 / $var2" | bc)
echo The answer for this is $var3
```

Sortie :

```
$ ./test10
The answer for this is 2.2222
```

Statut de Sortie

Concept et utilité

Qu'est-ce que le statut de sortie ?

- Chaque commande renvoie un code de sortie à la fin de son exécution
- Valeur entière entre 0 et 255
- Indique si la commande s'est terminée avec succès
- Si échec, indique la raison potentielle

Accès au statut de sortie :

- Variable spéciale : `$?`
- Contient le code de sortie de la dernière commande

```
$ date
Tue Nov 11 01:24:11 PM +01 2025
$ echo $?
0
```

Statut de Sortie

Interprétation des codes

Convention :

Code	Description
0	Réussite de la commande
1	Erreur générale inconnue
2	Utilisation abusive des commandes shell
126	La commande ne peut pas s'exécuter
127	Commande introuvable
128	Argument de sortie invalide
128+x	Erreur fatale avec le signal x de Linux
130	Commande terminée par Ctrl+C
255	État de sortie hors plage

Statut de Sortie

Interprétation des codes

Convention :

- Code 0 : commande terminée avec succès
- Code non-zéro : erreur survenue

Exemple d'erreur :

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
```

Exemple d'erreur :

```
$ mkdir
-bash: mkdir: missing operand
$ echo $?
1
```

Statut de Sortie

Commande exit

La commande exit :

- Permet de terminer le script à tout moment
- Spécifie un code de sortie personnalisé
- Syntaxe : `exit code`

Exemple :

```
$ cat test13
#!/bin/bash
# Test de la commande exit
var1=10
var2=30
var3=$[$var1 + $var2]
echo The answer is $var3
exit 5
```

Statut de Sortie

Résultat de exit

Exécution et vérification :

```
$ chmod u+x test13
$ ./test13
The answer is 40
$ echo $?
5
```

Le code de sortie est bien 5, comme spécifié dans le script.

Utilisation de variables :

```
$ cat test14
#!/bin/bash
# Test de exit avec une variable
var1=10
var2=30
var3=$[$var1 + $var2]
exit $var3
```

Statut de Sortie

Limitation des codes de sortie

Résultat :

```
$ ./test14
$ echo $?
40
```

Problème avec valeurs > 255 :

```
$ cat test14b
#!/bin/bash
var1=10
var2=30
var3=$[$var1 * $var2]
echo The value is $var3
exit $var3
```

Statut de Sortie

Limitation des codes de sortie > 255

Problème avec valeurs > 255 :

```
$ cat test14b
#!/bin/bash
var1=10
var2=30
var3=$[$var1 * $var2]
echo The value is $var3
exit $var3
```

Sortie :

```
$ ./test14b
The value is 300
$ echo $?
44
```

Le code est réduit par arithmétique modulo : 300 % 256 = 44

Résumé du Chapitre

Points clés

Concepts couverts :

- Chaînage de commandes multiples avec `;`
- Création de fichiers scripts avec shebang `#!/bin/bash`
- Gestion des permissions d'exécution
- Affichage de messages avec `echo`
- Variables d'environnement et variables utilisateur
- Substitution de commande : `$(commande)`
- Redirections : `>`, `>>`, `<`, `<<`
- Pipes pour chaîner les commandes : `|`
- Opérations mathématiques : `$[...]`, `expr`, `bc`
- Statuts de sortie et commande `exit`

Prochaines Étapes

Au-delà des bases

Limitations actuelles :

- Les scripts s'exécutent de manière linéaire
- Toutes les commandes sont exécutées dans l'ordre
- Pas de logique conditionnelle

Prochainement :

- Contrôle de flux logique
- Instructions conditionnelles (if-then)
- Altérer l'exécution selon des conditions
- Vérifier les statuts de sortie pour détecter les erreurs
- Rendre les scripts plus intelligents et adaptatifs

Références

Source principale :

- Richard Blum et Christine Bresnahan (2015)
- *Linux Command Line and Shell Scripting Bible*
- 3ème édition
- Chapitre 11 : *Basic Script Building*
- Wiley Publishing

Contenu adapté :

Ce support de cours a été adapté et traduit en français à partir du matériel source pour un usage pédagogique.