Introduction à l'environnement UNIX

R&I 1

Antoine Rollet

Bordeaux INP - Enseirb-Matmeca

Unix/Bash

Scripts - Scripts avancés

Avant propos

Lien vers le cours :

https://rollet.zzz.bordeaux-inp.fr/

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

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 commandes qui pourraient être pénibles et répétitives à saisir au clavier

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 (Exemple)

- Ecrire dans un fichier `myscript.sh` le code suivant (utiliser par exemple l'éditeur de texte `nano`) :

```
#!/bin/bash
echo "Bonjour"
```

- Ajouter les droits d'exécution : `chmod +x myscript.sh`
- Exécuter le script : `./myscript.sh`
  → Affiche `Bonjour`

Arguments

- Un script `shell` peut prendre en paramètre des arguments.
- Lors de l'évaluation de la commande par le `shell`, ce dernier 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'éléments courant dans le vecteur
- `$1 ... $9` : accès au premier, deuxième, . . .neuvième
- `$*` : liste de tous les éléments

La commande `shift` permet de décaler la liste des arguments d'une position vers la gauche.

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

Exercice : Ecrire un script `commands.sh` qui affiche toutes ces valeurs. Le tester avec `commands.sh Hello my friends` puis `commands.sh Hello "my friends"`. Expliquer.

Commande `test`

La commande `test`, possède deux syntaxes :
`test arg_1 arg_2 ...` et `[ arg_1 arg_2 ... ]` (⚠ne pas oublier les espaces)

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`
   - Chaîne vide : `-z chn`
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` : $\leq$

Négation : `! EXPRESSION` → `EXPRESSION` est fausse
Pour plus de détails cf le man : `man test`

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 ?
Indice : Essayer l'exemple précédent avec "$1" ou $1, avec les paramètres :

- tic tac
- "tic tac"

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 commandes identiques à une liste de variables données 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
```

⚠En cas d'utilisation combinée de `for` avec `cat`, l'espace fait partie des séparateurs. Cf après.

Exemples

```
for A in 1 2 3 4 5
do
  echo $A
done

for B in *
do
  cp $B $B.old
done

LISTE="1 2 3 4"
for C in $( echo $LISTE ); do ...; done

for D in $( cat inputFile ); do ...; done
```

Itérations (2)

Structures de contrôle permettant d'appliquer une série de commandes identiques à une liste de variables données 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
```

Dans l'exemple, 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. Ici, on affiche au final l'argument qui suit immédiatement `-o` (s'il existe).

Exemple

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

Itérations (3)

Autre type de boucle : la boucle `select` → menus numérotés

Syntaxe

```
select var in str1 str2 ... strN
do
   Statement(s) to be executed
done
```

Le script affiche un menu numéroté et attend une saisie clavier stockée dans la variable `var`. L'exemple de droite affiche :

```
1) quit
2) bonjour
3) bonsoir
#?
```

Example

```
select CHOIX in "quit" "bonjour" "bonsoir"
do
   case $CHOIX in
      "quit")
         break
      ;;
      "bonjour"|"bonsoir")
         echo "Hello!"
      ;;
      *)
         echo "Bad choice"
      ;;
   esac
done
```

Trois façons usuelles de sortir d'une boucle `select` :

- `continue` : passer à l'itération suivante
- `break` : sortir de l'itération
- `exit [n]` : sortir du programme avec la valeur `n` (*convention* : 0 si correct)

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
my_function valeur #autre appel
                   #avec un arg
```

Exemple

```
usage()
{
    echo "Erreur de type" $1
    echo "Usage: $(basename $0) [-ab] file"
}

if [ $# -eq 0 ]
then
  usage "Parameter required"
  exit -1
fi
```

- les variables `$1, $2,`... contiennent les valeurs passées en paramètre à la fonction
- ⚠ Il faut déclarer la fonction **avant** de l'appeler

Retour sur l'IFS

- En bash, les mots sont découpés selon un délimiteur défini dans la variable `IFS` (Internal Field Separator).
- Par défaut, il contient espace, tabulation et saut de ligne : `IFS=$' \t\n'` (remarquer la notation `$'...'` pour gérer les `\`)
- Il est possible de le modifier → exemple : `IFS=$'\n'` (classique pour lire des fichiers) ou `IFS=";"` (plus rare). Dans ce cas, il est recommandé de sauvergarder sa valeur et de la remettre ensuite : `oldIFS="$IFS" ... IFS="$oldIFS"`

scriptIFS.sh

```
oldIFS="$IFS"
IFS=(v1 ou v2, cf ci-dessous)
for i in $(cat data.txt)
do
    echo [$i]
done
IFS="$oldIFS"
```

data.txt

```
1 2

3 4
```

- v1 : `IFS=' '` affiche :
  ```
  [1]
  [2
  3]
  [4]
  ```
- v2 : `IFS=$'\n'` affiche :
  ```
  [1 2]
  [3 4]
  ```

Lire un fichier ligne par ligne

- Il est fréquent de prendre un fichier en entrée, et d'appliquer un traitement à chaque ligne
- Possibilité : `for i in $(cat input.file); do ... done`
  → attention à la gestion du séparateur (espace vs ligne, ...)
  → exemple : `IFS=$'\n'` (classique) ou `IFS=";"` (plus rare)
  cf slide précédent.
- Autre méthode : `while read line; do COMMAND; done < input.file`
  ⚠ avec `read`, la dernière ligne du fichier `input.file` doit terminer par un saut de ligne pour être prise en compte (il existe des moyens de le prendre en compte, non abordé ici). ⚠ C'est subtil... la plupart des éditeurs de texte rajoutent un `\n` en fin de ligne en sauvegardant au format `.txt` (ou assimilé).
- `-r` n'autorise pas les échappements de caractères (backslashs), par exemple une ligne avec `\n` au milieu sera considérée comme une seule ligne par `read`
- `IFS=` pour effacer le séparateur de champs (seule la fin de la ligne sera prise en compte par le `read`)
- Ce qui donne :
  `while IFS= read -r line; do COMMAND_on $line; done < input.file`
- Adaptable pour lire prendre en entrée une commande dont le résultat contient plusieurs lignes à traiter (utilisation d'un *"here string"*) :
  `while IFS= read -r line; do COMMAND_on $line; done <<< $(COMMAND)`

Exercices : `displayArgs.sh`

- Ecrire un script bash qui affiche tous les arguments passés en paramètre sauf le premier
- Idem, mais qui affiche tous les arguments passés en paramètre sauf le dernier

Exercices : `decompte.sh`

- Ecrire un script bash prenant deux paramètres (*délai* et *début*) qui réalise un décompte jusque à 0 toutes les *délai* secondes, en partant de *début*.
- Ecrire une version avec une boucle `for` et une autre avec boucle `while`.

Exercices : `check_pdf.sh`

- Ecrire un script bash prenant un répertoire en paramètre, et qui pour tous les fichiers terminant par `.pdf` de ce répertoire (recherche non récursive), s'assure qu'il s'agit bien de fichiers pdf. Le script affiche une alerte uniquement en cas de détection de fichier corrompu.
- Ecrire une version avec une boucle `for`.
- Contrainte : ne pas utiliser la commande `find`
- On pourra supposer que les noms de fichiers ne contiennent pas d'espaces (sinon, penser à utiliser des guillements + gestion de l'`IFS`)

Exercices : `notes.sh`

- Ecrire un script qui saisisse en boucle au clavier un nom et une note (format `Nom:note`) sous forme de nombre entier, qui affiche à chaque fois à l'écran la mention correspondante (Ajourné, Passable, Assez Bien, Bien, Très Bien), et sauvegarde l'ensemble dans un fichier `my_notes.csv` (à la suite si le fichier existe déjà). Taper la touche `q` pour quitter.

Exercices : montage audio

- Ecrire un script bash permettant de découper un fichier audio en plusieurs sous-parties. Il prend 3 paramètres :
  - un fichier audio : le fichier à découper
  - un fichier au format csv, comprenant des lignes de la forme `beginning;end;title_of_the_subpart`, les timings sont exprimés sous la forme `HH:MM:SS`
  - une date au format `YYMMDD`
- Pour chaque ligne du fichier csv, il génère un fichier audio au format `.wav`, démarrant et terminant aux instants spécifiés, ayant le nom `YYMMDD_title_of_subpart.wav`
- Indice : utiliser la commande `ffmpeg`
- Contrainte : utiliser une boucle `for` [1]
- Vous pouvez tester votre script avec une interview de Richard Stallman [2] disponible à https://jayandjef.github.io/programming/featured/2025/02/15/an-interview-with-richard-stallman/, et vous pouvez par exemple utiliser le fichier audio à https://jayandjef.github.io/assets/stallman/chocolate.wav avec le fichier `chocolate.csv` fourni ici : https://rollet.zzz.bordeaux-inp.fr/IntroUnix/ressources/chocolate.csv
- Ajouter des contrôle de validité du fichier csv, et du nombre de fichiers créés

1. pour des raisons techniques, la boucle `while` est plus difficile à écrire
2. https://www.stallman.org/

Exercices : `random_files.sh`

- Ecrire un script bash qui prend un répertoire en paramètre, choisit aléatoirement (récursivement) un fichier `.txt` dans ce répertoire, et propose à l'utilisateur de l'ouvrir automatiquement avec un éditeur de texte, de recommencer, ou de quitter.
- Indice : utiliser l'opérateur `select`
- Remarque : on pourrait envisager ce genre d'application avec des fichiers audios, ou vidéos.

Exercices : `mykill.sh`

Il est souvent fastidieux de trouver le pid d'un processus à tuer

- Écrire un script bash 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 oneko
```

Exercices : `liste.sh`

- Écrire un script bash qui prenne en entrée un fichier contenant une liste de logins et crée en sortie un fichier avec les noms et prénoms correspondants
- Exemple d'exécution :

  ```
  ~$ ./liste.sh logins.csv resultat.txt
  ```

- Indice : utiliser la commande `finger`. Essayer le script sur la machine `ssh.enseirb-matmeca.fr` (`finger` n'est pas installé sur les machines de la salle).
- Adapter pour écrire une nouvelle version qui fasse la même chose avec les utilisateurs connectés (command `who`)

Exercices : `valid_url.sh`

- Ecrire un script bash qui prend en paramètre un fichier contenant des sites web (des `url` de noms de domaine, sans le `http` devant, ou des adresses IP), et affiche à l'écran s'ils/elles ne sont pas atteignables
- Indice : utiliser la commande `ping` avec l'option `-c1`
- Indice : penser au code de retour de la fonction `ping`

Exercices : `backup.sh`

Écrire un script bash qui

- affiche les n fichiers les plus récemment modifiés dans le répertoire courant (non récursivement)
- 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 : `printer_ready.sh`

- Ecrire un script bash qui prend en paramètre le nom d'une imprimante de l'Enseirb-Matmeca, considérant par défaut l'imprimante `pontos`, et vous alerte quand cette dernière est prête à imprimer. Pour cela, le script interrogera l'imprimante toutes les 10 secondes.
- Indice : utiliser la commande `lpq`
- Contrainte : ne pas utiliser l'option `+intervalle` de la commande `lpq`
- Pour simplifier, on supposera que les messages sont `"... is ready"` ou `"... is printing"`

Exercices

- Quelques exercices pour vous entraîner / réviser : rendez-vous sur la page `https://cours-mf.gitlabpages.inria.fr/if104/exercices` pour faire les exercices associés au shell