Programmation Unix 1 – cours n°5

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°5

1. Syntaxe avancée
2. Substitutions
3. Tableaux indexés
4. Tableaux associatifs
5. Redirections avancées

1 - Syntaxe avancée

- Exécution en arrière-plan
- Séparateurs entre deux commandes
- Grammaire des commandes

Exécution en arrière-plan

Le shell peut exécuter une commande dans un sous-shell en arrière-plan :

```
commande arguments &
```

- le shell n'attend pas qu'elle soit terminée
- `$?` = 0 (succès) même si la commande n'est pas trouvée
- la commande est détachée du terminal

On peut mettre en arrière-plan la dernière commande lancée :

`^Z` puis `bg`

Remettre en avant-plan la dernière commande en arrière-plan :

```
fg
```

Liste des commandes qui tournent en arrière-plan :

```
jobs -l
```

Séparateurs entre deux commandes

`com1 ; com2`	*séquentiel* : attend la fin de `com1` avant de lancer `com2` (';' ou RC)
`com1 & com2`	*parallèle* : lance `com1` en arrière-plan puis lance tout de suite `com2`
`com1 && com2`	*séquentiel conditionnel* : si `com1` réussi, alors lance `com2`
`com1 || com2`	*séquentiel conditionnel* : si `com1` échoue, alors lance `com2`

Le résultat est celui de la dernière commande exécutée.

Séparateur conditionnel

Attention : `com1 && com2 || com3` n'est pas équivalent à `if com1 ; then com2 ; else com3 ; fi`

```
$ if true ; then false ; else echo "perdu" ; fi
$ true && false || echo "perdu"
perdu
```

Usage typique :

```
(cd progs && make monprog && ./monprog)
```

```
mkdir svg || { echo "Échec." > /dev/stderr ; exit 1;}
```

```
for fichier in * ; do
    test -f "$fichier" || continue
    echo "$fichier"
done
```

Grammaire des commandes

Commande simple :

```
[var=valeur ...] commande [arguments] [redirections]
```

Pipeline de commandes :

```
[time] [!] simple [ | simple ...]
```

Liste de commandes :

```
pipeline [séparateur pipeline ...]
```

où `séparateur` est parmi `;` `&` `&&` `||`

Commande composée :

```
(liste)
{ liste ;}
if liste ; then liste ; fi
while liste ; do liste ; done
```

2 - Substitutions

- Indirections
- Référence de nom
- Passage par référence
- Expansion du tilde
- Ordre des substitutions

Indirections

Il est possible de mémoriser le nom d'une variable, puis de retrouver sa valeur :

```
$ a=12 ; x=a
$ echo "$x"
a
$ echo "$$x"
2158x
$ echo "${$x}"
bash: ${$x} : mauvaise substitution
$ echo "${!x}"
12
```

Dernier argument :

```
$ set ga bu zo ; echo "${!#}"
zo
```

Référence de nom

Nouveauté bash 4.3 : attribut *nameref* (inspiré de `ksh`)

Une variable peut faire référence à une autre variable.

```
declare -n nouvelle=originale
local   -n nouvelle=originale
```

Exemple :

```
$ a=ga
$ declare -n b=a
$ echo $b
ga
$ a=bu ; echo $b
bu
$ b=zo ; echo $a
zo
```

Passage par référence

Ce mécanisme permet de passer un paramètre par référence :

```
afficher_valeur () # var
{
  local -n var="$1"
  echo "$var"
}

modifier_valeur () # var valeur
{
  local -n var="$1" ; local valeur="$2"
  var="$valeur"
}

$ x=lundi
$ modifier_valeur x mardi
$ afficher_valeur x
mardi
```

Expansion du tilde

`~`	$\rightarrow$	répertoire principal
`~user`	$\rightarrow$	répertoire principal du `user`
`+~`	$\rightarrow$	répertoire courant
`-~`	$\rightarrow$	répertoire précédent

Délicat à manipuler :

- ne doit pas être protégé par `""` ou `''`
- hors affectation, doit figurer en tête
- dans une affectation, doit suivre le premier `'='` ou un `':'`

Ordre des substitutions

Les substitutions sont effectuées dans cet ordre :

1)	Expansion des accolades	`{ga,bu}`
2)	Expansion du tilde	`~  ~thiel`
3)	Expansion des arguments	`$1  $#  $*  "$@"`
4)	Expansion des variables	`$foo  ${t[]}`
5)	Substitution des commandes	`$()`
6)	Évaluation arithmétique	`$(())`
7)	Découpage des mots	
8)	Développement noms fichiers	`* ? []`

3 - Tableaux indexés

- Tableaux de chaînes de caractères
- à 1 dimension, indexés par entiers $\geqslant 0$
- élastiques, de taille illimitée
- creux, cases non contigües

`declare -a tab`	crée un tableau `tab`
`declare -a tab[$n]`	idem, taille ignorée
`tab[$i]=valeur`	affectation + création automatique
`${tab[$i]}`	expansion de la valeur "" si non affectée

Accolades obligatoires

Lors de l'expansion de valeur, les accolades sont obligatoires.

Exemple :

```
$ t[0]=ga ; t[1]=bu
$ echo $t                    # incorrect
ga
$ echo $t[1]                 # incorrect
ga[1]
$ echo ${t[1]}
bu
```

Expansion arithmétique de l'indice

Bash fait une expansion arithmétique de l'indice :

- même syntaxe entre `[ ]` que dans `(( ))`
- on peut omettre les `$`.

Exemple :

```
$ i=2
$ x[i]="ga" ; x[i+1]="bu"
$ echo "${x[i]} ${x[i+1]}"
ga bu
```

Liste

On peut exprimer les éléments d'un tableau par une liste :

`tab=()`	vide le tableau
`tab=(val1 val2 .. valn)`	écrase le tableau, indices `0..n-1`
`tab=([2]=val1 [5]=val2 ..)`	idem, pour les indices donnés
`tab+=(valx ..)`	concatène la liste à la fin de `tab`

Exemple :

```
$ t=(ba bu)
$ t+=(zo meu)
$ declare -p t
declare -a t='([0]="ba" [1]="bu" [2]="zo" [3]="meu")'
```

Valeur des cases

Récupérer la liste des valeurs :

`${tab[*]}` ou `"${tab[@]}"` séparées par ␣ ou "␣"

Exemple :

```
$ t=(ga bu zo meu)
$ echo "${t[1]}"
bu
$ echo "${t[*]}"
ga bu zo meu
```

Itérer sur les cases

Utiliser `${tab[*]}` ou `"${tab[@]}"` ?

```
$ t=(ga "bu zo" meu)

$ for e in ${t[*]}; do echo -n "'$e' "; done
'ga' 'bu' 'zo' 'meu'

$ for e in "${t[*]}"; do echo -n "'$e' "; done
'ga bu zo meu'

$ for e in ${t[@]}; do echo -n "'$e' "; done
'ga' 'bu' 'zo' 'meu'

$ for e in "${t[@]}"; do echo -n "'$e' "; done
'ga' 'bu zo' 'meu'
```

Affectation mixte

```
$ t=(zero un deux [5]=cinq [4]=quatre)
$ echo ${t[*]}
zero un deux quatre cinq

$ t=(zero un deux [5]=cinq [4]=quatre ga)
$ echo ${t[*]}
zero un deux quatre ga
```

$\rightarrow$ Élément inséré au dernier index utilisé + 1

Spécifié ainsi dans le `man` de bash ; à éviter.

Nombre de cases

Longueur d'une case : `${#tab[i]}`

Nombre d'éléments : `${#tab[*]}`

Exemple :

```
$ t=(ga "bu zo") ; echo "${#t[*]}"
2
$ t[6]="meu" ; echo "${#t[*]}"
3
$ declare -p t
declare -a t='([0]="ga" [1]="bu zo" [6]="meu")'
$ echo "${#t[1]}"
5
```

Suppression

Supprimer le tableau : `unset tab`

une case : `unset tab[i]`

```
$ t=(ga bu zo meu)
$ unset t[1] t[2]
$ echo "${t[*]}"
ga meu
$ echo "${#t[*]}"
2
$ declare -p t
declare -a t='([0]="ga" [3]="meu")'   # tableau creux
```

Ne pas se servir du nombre d'éléments pour itérer si le tableau n'est pas tassé.

Liste des indices

Liste des indices des éléments : `${!tab[*]}`

Exemple :

```
$ t=([2]=ga [8]=bu [5]=zo)
$ echo ${!t[*]}
2 5 8
```

Boucle sur les indices d'un tableau creux :

```
$ for i in ${!t[*]}; do
    echo "t[$i] = ${t[i]}"
  done
t[2] = ga
t[5] = zo
t[8] = bu
```

Compléments

Tableau local à une fonction :

```
local -a tab
```

Lire une ligne sur l'entrée standard puis mémoriser les mots dans un tableau :

```
read -a tab
```

Placer les arguments dans un tableau :

```
args=("$@")
```

Recopier un tableau et tasser :

```
B=("${A[@]}")
```

Concaténer 2 tableaux et tasser :

```
C=("${A[@]}" "${B[@]}")
```

Passage de tableau en paramètre

Les paramètres sont des chaînes de caractères.

- On peut passer un tableau par *recopie* avec `"${tab[@]}"`
- Depuis bash 4.3 on peut passer des tableaux par *référence* :

```
afficher_case () # tableau indice
{
    local -n tab="$1"        # -n = attribut nameref
    local ind="$2"
    echo "${tab[ind]}"
}

$ ga=(le "lundi au" soleil)
$ afficher_case ga 1
lundi au
```

Modification d'un tableau en paramètre

C'est réellement un passage par référence :

```
modifier_case () # tableau indice valeur
{
    local -n tab="$1"
    local ind="$2" val="$3"
    tab[ind]="$val"
}

$ ga=(bu zo)
$ modifier_case ga 5 meu
$ declare -p ga
declare -a ga='([0]="bu" [1]="zo" [5]="meu")'
```

Tableaux d'entiers

On peut utiliser les expressions arithmétiques :

```
$ t=(7 9 6 4 11 8)
$ n=${#t[*]}
$ for ((i=1; i<n; i++)); do
      if ((t[i] > t[i-1])); then
        echo $((t[i]))
      fi
    done
9
11
```

4 - Tableaux associatifs

Tableau dont les indices sont des chaînes (non vides) :
encore appelés "hash table" ou "dictionnaire".

Associe des "clés" à des "valeurs".

Permet de représenter des tables de bases de données
$\rightarrow$ outil très puissant (Javascript, Php, Perl, Python, Ruby, ...)

Déclaration obligatoire : `declare -A dico`

puis même syntaxe que pour les tableaux.

Incompatibilité avec tableaux normaux $\rightarrow$ `unset` avant

Exemple

```
$ declare -A dico
$ dico=([lundi]=poisson [mardi]=viande)

$ echo ${!dico[*]}
lundi mardi                                  # clés

$ echo ${dico[*]}
poisson viande                               # valeurs

$ for cle in "${!dico[@]}"; do
    echo "dico[$cle] = '${dico[$cle]}'"
  done
dico[lundi] = 'poisson'
dico[mardi] = 'viande'
```

Passage de tableau en paramètre

Passage par *référence* possible depuis bash 4.3 :

```
afficher_case () # tableau cle
{
    local -n tab="$1"          # attribut nameref
    local cle="$2"
    echo "${tab[$cle]}"        # $cle car non entier
}

$ declare -A dico=([lundi]=poisson [mardi]=viande)
$ afficher_case dico mardi
viande
```

Modification d'un tableau en paramètre

Test du passage par référence :

```
modifier_case () # tableau cle valeur
{
    local -n tab="$1"
    local cle="$2" val="$3"
    tab[$cle]="$val"        # $cle car non entier
}
```

```
$ modifier_case dico jeudi oeuf
$ declare -p dico
declare -A dico='([lundi]="poisson" [jeudi]="oeuf" \
                  [mardi]="viande" )'
```

5 - Redirections avancées

On a vu :

<		redirection stdin
>		redirection stdout
>>		append
>|		écrasement forcé
<<	<<<	document en ligne
2>		redirection stderr

Redirection sur un descripteur existant :

- `[n]<&k` en entrée
- `[n]>&k` en sortie

→ Le descripteur `n` devient une copie du descripteur `k` existant (fonction C : `dup2`).

Exemples de redirections

Redirection de la sortie d'erreur dans le fichier :

```
ls -R / > liste.txt 2>&1
```

Ordre important : 2 $\rightarrow$ terminal puis 1 $\rightarrow$ fichier

```
ls -R / 2>&1 > liste.txt
```

Pour écrire sur stderr :

```
echo "message" > /dev/stderr
```

ou

```
echo "message" >&2
```

préféré

Nouveaux descripteurs

On peut créer des descripteurs :

```
$ echo "foo" 3>| trace.txt 1>&3
$ cat trace.txt
foo
```

On peut rediriger le script ou le terminal de façon permanente :

```
$ exec 4>| trace.txt
$ echo "bar" >&4
$ cat trace.txt
bar
$ exec 4>&-  # ferme le fichier
```

Exemple : lecture en parallèle

```
while read ligne1 <&3 && read ligne2 <&4
do
    echo "Lu '$ligne1' et '$ligne2'"
done 3< fichier1 4< fichier2
```

Cette construction permet :

- de lire à chaque itération une ligne dans chaque fichier ;
- de s'arrêter dès que la fin de l'un des deux fichiers est atteinte.

Voir chapitre 20 de l'ABSG.