Christophe Blaess

Déroulement des scripts

Ecriture de scripts shell

Ce document a été réalisé sur système Linux avec les logiciels libres :

- **Formation+** pour préparer le cours complet,
- **LaTeX** pour produire les transparents et le support de cours imprimé,
- **Xfig** pour les graphiques vectoriels,
- **The Gimp** pour les images et les photos.

Support de cours : version 17

Ecriture de scripts shell

Déroulement des scripts

Christophe Blaess

Commandes simples

Une commande *simple* du shell est décrite par le schéma suivant :

[*affectations de variables*] `commande` [*arguments...*] [*redirections*]

Chaque processus se terminant normalement sous Unix renvoie un *code de retour*, indiquant ses conditions de réussite ou d'échec.

La terminaison anormale d'un processus se manifeste toujours par l'occurrence d'un signal. Le code de retour obtenu est $128+n$ si le processus est tué par le signal numéro n.

Le code de retour est consultable dans le paramètre `$?` du shell après terminaison de la commande.

Voir aussi :
Manuel Unix
– `signal`(7)

A vous. . .

```
$ gvim
$ echo $LANG
$ LANG=en_US gvim
$ echo $LANG
```

```
$ a=123 echo $a
$ echo $a
```

```
$ cd /etc
$ echo $?
```

```
$ cd /azertyuiop
$ echo $?
$ echo $?
```

```
$ sleep 1
$ echo $?
```

```
$ sleep 60
    (Controle-C)
$ echo $?
```

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

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

```
$ grep root /etc/inexistant
$ echo $?
```

Pipelines

Les commandes simples peuvent être organisées en *pipelines* :

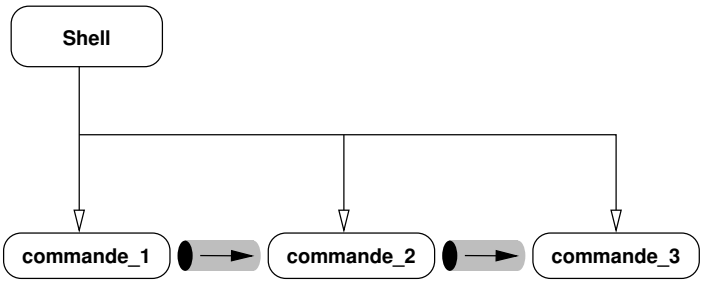
`commande` [| *commandes...*]

La sortie standard de chaque commande – sauf la dernière – est redirigée via un *tube* Unix vers l'entrée standard de la commande suivante.

Les sorties d'erreur ne sont pas concernées.

Le code de retour est celui de la dernière commande à se terminer (en principe la dernière du pipeline).

Le shell exécute généralement chaque commande simple du pipeline dans un processus fils distinct.



Les redirections dans les tubes sont mises en place avant de lancer les commandes. Ces redirections sont transparentes pour les commandes exécutées.

Listes de pipelines

Quatre organisations logiques sont possibles entre les pipelines :

Séquencement ;

Le second pipeline est exécuté une fois le premier terminé, quel que soit le résultat.

Parallélisme &

Les deux pipelines sont exécutés simultanément.

Dépendance &&

Le second pipeline n'est exécuté que si le premier a réussi.

Alternative ||

Le second pipeline n'est exécuté que si le premier a échoué.

Séquencement

```
commande_1 ; commande_2 ; commande_3
```

Le point-virgule peut être remplacé par un saut-de-ligne.

À vous...

Exécutez la ligne suivante :

```
$ echo "début"; sleep 5; echo "fin"
```

Recommencez en pressant *Controle-C* pendant la pause de cinq secondes.

Conclusion ?

Voir aussi :
Manuel Unix
– `sleep`(1)

Parallélisme

```
commande_1 & commande_2 & commande_3
```

Les commandes suivies du symbole `&` sont exécutées à l'arrière-plan.

Eléments de contrôle des jobs :
- `$!` :*PID* du dernier job lancé à l'arrière-plan ;
- *Contrôle-Z* : suspend le job au premier-plan ;
- `bg` : relance à l'arrière-plan un job suspendu ;
- `fg` : passe à l'avant-plan un job suspendu ou à l'arrière-plan ;
- `jobs` : affiche la liste des jobs en cours ;
- `wait pid` : attend la fin d'un job à l'arrière-plan.

Une commande à l'arrière-plan ne peut pas lire sur son terminal de contrôle, et en général ne peut pas écrire non plus (configurable avec `stty`(1)).

A vous. . .

Exécutez la séquence suivante, en observant bien chaque étape :

```
$ xclock
   (Contrôle-Z)
$ jobs
```

```
$ bg
$ jobs
$ fg
   (Contrôle-C)
```

Dépendance

```
commande_1 && commande_2
```

La seconde commande n'est exécutée que si la première a réussi.

> Sous Unix, par convention, la réussite d'une commande est indiquée par un code de retour **nul**.

Vérification de condition :

```
[ $DEBUG -gt 0 ] && echo "Variable = $VAR"
```

Enchaînement de commandes :

```
cd /home/projet && tar -cf /tmp/projet.tar * && rm -rf *
```

Alternative

```
commande_1 || commande_2
```

La seconde commande n'est exécutée que si la première a échoué.

Symétriquement l'échec d'une commande est indiqué par un code de retour **non-nul**.

Essais successifs :

```
cd /var/tmp || cd /usr/tmp || cd /tmp
```

Vérification d'erreur :

```
mount /mnt/cdrom || { echo "Impossible de monter le CD";exit 1; }
```

```
...
cd ~/                       || exit 1
mount /mnt/cdrom            || { echo "Impossible de monter le CD; exit 1;}
cp /mnt/cdrom/tp.tar /tmp || { echo "Fichier tp.tar absent du CD"; exit 2;}
tar -xf /tmp/tp.tar         || { echo "Installation impossible"; exit 2; }
rm -f /tmp/tp.tar
...
```

Commandes composées

Les accolades permettent d'isoler une portion de liste de pipelines, en ayant priorité sur les opérateurs `;`, `&`, `&&` et `||` se trouvant à l'extérieur des accolades.

```
mount /mnt/cdrom || { echo "Impossible de monter le CD";exit 1; }
```

A vous...

Essayez les commandes suivantes, en comprenant bien les problèmes qui se posent :

```
$ {echo "sans espace"}
$ { echo "avec espace" }
$ { echo "avec espace et un point-virgule"; }
```

Il existe également une construction `(`*commandes...*`)` appelée *sous-shell* qui permet de regrouper des commandes. Elle est rarement utilisée.

Fonctions

Les fonctions permettent de regrouper des commandes que l'on exécutera ensuite avec une invocation simple.

Intérêts :
- appel depuis plusieurs emplacements du script,
- découpage du script en unités fonctionnelles simples,
- possibilité d'invocations itératives (récursivité).

Au sein de la fonction, les arguments sont disponibles dans les paramètres positionnels `$1`, `$2`, etc.

Le paramètre spécial `$#` contient le nombre d'arguments de la fonction.

Définition d'une fonction (SUSv3) :

```
nom_fonction()
{
     contenu de la fonction
}
```

Ou (Korn) :

```
function nom_fonction
{
     contenu de la fonction
}
```

Invocation de la fonction :

```
nom_fonction argument_1 argument_2 ...
```

Le mot-clé `return` *n* permet de mettre fin à une fonction en renvoyant le code de retour *n*.

Au sein du script, le code de retour est disponible dans la variable `$?` après exécution de la fonction.

Dans une fonction les variables sont globales par défaut. Le mot-clé `local` déclare une variable qui ne sera visible que dans la fonction et éventuellement celles qu'elle appelle.

```
choix_fichier()
{
        local n=0
        local f
        local fichier
        for f in *; do
                n=$((n+1))
                fichier[$n]=$f
                echo "$n) $f" >&2
        done
        echo -n "Votre choix :" >&2
        read reponse
        if [ "$reponse" -lt 1 ] || [ "$reponse" -gt $n ] ; then
                return 1
        fi
        echo ${fichier[$reponse]}
        return 0
}

...

fic=$(choix_fichier)
[ $? -ne 0 ] && exit 1
```

Sélections

Deux structures de contrôle pour les sélections :

Le test `if/then/else` permet d'organiser le déroulement du script en fonction du code de retour d'une commande exécutée.

La sélection `case/esac` compare une chaîne avec divers motifs et réagit en conséquence.

La commande `test` permet de vérifier des conditions sur des chaînes, des nombres ou des fichiers.

La notation `! condition` permet d'inverser le résultat logique de la condition.

Structure `if-then-else`

```
if condition
then
    commandes...
elif condition
then
    commandes...
else
    commandes...
fi
```

```
if condition ; then
    commandes...
elif condition ; then
    commandes...
else
    commandes...
fi
```

```
if condition ; then commandes... ; fi
```

Exemples de tests

```
if ! ping -c 1 $nom_serveur 2>/dev/null
then
    echo "Serveur $nom_serveur injoignable" >&2
    exit 1
fi
```

```
if ! cd $rep_appli 2>/dev/null
then
    if ! mkdir -p $rep_appli 2>/dev/null; then
        echo "Impossible de créer le répertoire $rep_appli" >&2
        exit 1
    fi
    cd $rep_appli
fi
```

```
if mount '\\serveur\Disque' /mnt/srv -t cifs
then
    echo "Disque serveur disponible dans /mnt/srv"
fi
```

`test` *expression*
`[` *expression* `]`

Expression	vraie si...
`[ !` *condition* `]`	la *condition* est fausse
`[` *condition_1* `-a` *condition_2* `]`	les deux conditions sont vraies
`[` *condition_1* `-o` *condition_2* `]`	une au moins des conditions est vraie

Toutefois, on préfère en général les constructions :
- `! [` *condition* `]`
- `[` *condition_1* `] && [` *condition_2* `]`
- `[` *condition_1* `] || [` *condition_2* `]`

qui sont plus lisibles.

Voir aussi :
Manuel Unix
- `test`(1)

Tests sur les chaînes de caractères

Expression	vraie si...
[*chaîne*]	la chaîne n'est pas vide
[-n *chaine*]	la longueur de la chaîne est non-nulle
[-z *chaine*]	la longueur de la chaîne est nulle
[*chaine_1* = *chaine_2*]	les deux chaînes sont égales
[*chaine_1* != *chaine_2*]	les deux chaînes sont différentes
[*chaine_1* < *chaine_2*]	la première chaîne précède la seconde
[*chaine_1* > *chaine_2*]	la première chaîne suit la seconde

Exemple

```
if [ "$reponse" = "o" ] || [ "$reponse" = "O" ]; then ...
```

Tests sur les nombres

Expression	vraie si...
`[` *valeur_1* `-eq` *valeur_2* `]`	les deux valeurs sont égales
`[` *valeur_1* `-ne` *valeur_2* `]`	les deux valeurs sont différentes
`[` *valeur_1* `-lt` *valeur_2* `]`	la première valeur est strictement inférieure à la seconde
`[` *valeur_1* `-le` *valeur_2* `]`	la première valeur est inférieure ou égale à la seconde
`[` *valeur_1* `-gt` *valeur_2* `]`	la première valeur est strictement supérieure à la seconde
`[` *valeur_1* `-ge` *valeur_2* `]`	la première valeur est supérieure ou égale à la seconde

Les tests numériques ne portent que sur des valeurs **entières**.

Exemple

```
if [ $age -ge 18 ] && [ $age -lt 30 ]; then ...
```

Tests sur les fichiers

Expression	vraie si...
`[ -e` *fichier* `]`	le fichier existe
`[ -f` *fichier* `]`	le fichier est un fichier normal
`[ -h` *fichier* `]`	le fichier est un lien symbolique
`[ -L` *fichier* `]`	le fichier est un lien symbolique
`[ -b` *fichier* `]`	le fichier représente un périphérique en mode bloc
`[ -c` *fichier* `]`	le fichier représente un périphérique en mode caractère
`[ -p` *fichier* `]`	le fichier est un tube nommé (*fifo*)
`[ -S` *fichier* `]`	le fichier est une *socket*
`[ -d` *fichier* `]`	le fichier est un répertoire
`[ -O` *fichier* `]`	le fichier nous appartient
`[ -G` *fichier* `]`	le fichier appartient à notre groupe
`[ -r` *fichier* `]`	nous pouvons lire le fichier
`[ -w` *fichier* `]`	nous pouvons écrire dans le fichier
`[ -x` *fichier* `]`	nous pouvons exécuter le fichier
`[ -s` *fichier* `]`	la taille du fichier est non-nulle
`[ -u` *fichier* `]`	le bit Set-UID du fichier est activé
`[ -g` *fichier* `]`	le bit Set-GID du fichier est activé

Bash, Ksh, mais pas SUSv3 :

Expression	vraie si...
`[` *fichier_1* `-ef` *fichier_2* `]`	Les deux noms pour le même fichier
`[` *fichier_1* `-nt` *fichier_2* `]`	Premier fichier plus récent que le second
`[` *fichier_1* `-ot` *fichier_2* `]`	Second fichier plus récent que le premier

Structure `case-esac`

```
case chaine in
   motif ) actions... ;;
   motif ) actions... ;;
   motif ) actions... ;;
esac
```

Les *motifs* peuvent contenir des caractères génériques comme `?` ou `*`.

Les *motifs* peuvent être associés avec un OU logique `|`.

Exemple

```
case $nom_fichier in
    *.gif | *.jpeg | *.tif )
        echo "Fichier graphique" ;;

    *.txt | *.html )
        echo "Fichier texte" ;;

    * ) echo "Format de fichier inconnu..."
        echo "Continuer quand même ?"
        read reponse
        case $reponse in
            O* | o* | Y* | y* )
                echo "Ok" ;;
            * ) exit 1 ;;
        esac ;;
esac
```

Itérations

Il existe trois structures d'itération :

La boucle `for` répète un bloc de code en donnant successivement à une variable toutes les valeurs d'une liste.

La boucle `while` répète une portion de script tant qu'une condition est vraie.

La boucle `until` répète une portion de script tant qu'une condition est fausse.

Les instructions `break` et `continue` permettent de modifier l'exécution d'une structure de boucle en provoquant une sortie ou une reprise prématurées.

Structure `for-do-done`

```
for variable in liste
do
    commandes...
done
```

```
for variable in liste ; do
    commandes...
done
```

```
for variable in liste ; do commandes...; done
```

A vous

```
$ for i in abc def ghi jkl; do echo $i; done
```

```
$ for i in *; do echo $i; done
```

```
$ for i in *.html; do echo $i; done
```

```
$ for i in *.rien; do echo $i; done
```

```
$ for i in $(seq 5 15); do echo $i; done
```

Conclusion ?

Voir aussi :
Manuel Unix
– `seq`(1)

Attention, `seq` n'est pas normalisé par SUSv3.

Structure `while-do-done`

```
while condition
do
   commandes...
done
```

```
while condition ; do
   commandes...
done
```

```
while condition ; do commandes...; done
```

A vous...

```
$ i=1
$ while [ $i -le 10 ]; do echo $i; i=$((i+1)); done
```

```
$ while true; do date; sleep 1; done
```

Voir aussi :

Manuel Unix

- `true`(1)
- `date`(1)
- `sleep`(1)

Structure `until-do-done`

```
until condition
do
   commandes...
done
```

```
until condition ; do
   commandes...
done
```

```
until condition ; do commandes...; done
```

A vous...

```
$ until [ "$rep" = "o" ] || [ "$rep" = "n" ]
> do echo "Votre réponse ?"
> read rep
> done
```

```
$ until false; do date; sleep 1; done
```

Voir aussi :
Manuel Unix
– `false`(1)

Ruptures de séquences

L'instruction `break` permet de sortir immédiatement d'une boucle `for`, `while` ou `until`.

L'instruction `continue` fait passer immédiatement à l'itération suivante d'une boucle `for`, `while` ou `until`.

Exemples...

```
while true; do
...
   if [ "$reponse" = "quitter" ] ; then break; fi
done
```

```
for fichier in *; do
   if [ ! -f "$fichier" ] ; then continue; fi
...
done
```

Travaux pratiques

Exercices

Exercice numéro 1

Structures de boucle for-do-done

Exercice numéro 2

Structures de boucle while-do-done

Exercice numéro 3

Comptage à rebours

Exercice numéro 4

Parallélisme et priorité

Exercice numéro 5

Boucles imbriquées

Exercice numéro 6

Tests des caractéristiques d'un fichier

Exercice numéro 7

Mise en correspondance avec case-esac

Exercice numéro 8

Fonctions récursives

Index