

## Langage de commandes Shell

### Généralités :

Le Shell est à la fois un *langage de commandes* (shell script) et l'interpréteur de ce langage. Il assure l'interface externe entre les utilisateurs et le système, mais ne fait pas partie du noyau. Principaux langages de commandes sont disponibles sur les systèmes UNIX et GNU/Linux : csh (C-Shell), ksh (Korn-Shell), bash (Bourne Again-Shell). L'interpréteur du shell a la possibilité, soit d'exécuter lui-même la commande demandée par l'utilisateur (commande interne), soit de lancer l'exécution de programmes existant par ailleurs pour ce faire (commandes externes). A chacune des commandes externes correspond un fichier exécutable ayant comme nom le "nom de la commande". Ce fichier se trouve souvent dans le répertoire `/bin`. L'exécution d'une commande externe entraîne la création d'un nouveau processus exécutant le programme correspondant à la commande. Une commande externe peut être appelée à partir de n'importe lequel des langages de commandes disponibles, mais aussi à partir d'un programme utilisateur. Chacun des langages de commandes dispose de ses propres commandes qui sont les commandes internes.

### Mécanisme d'exécution d'une commande :

Vous avez déjà vu qu'UNIX propose une vaste gamme de commandes et qu'il est possible d'en définir de nouvelles. Cela différencie radicalement de beaucoup d'autres os, *Operating System* (système d'exploitation). Une commande sous un système de type UNIX (GNU/Linux et \*BSD) est un simple fichier. La commande `ls` par exemple existe sous la forme d'un fichier binaire située dans le répertoire `/bin`. Par conséquent au lieu de taper `ls` vous pourriez entrer `/bin/ls`. Les commandes entrées par un utilisateur, aussi bien celles entrées au clavier que celles lancées par un *programme* (script) permettant d'exécuter des *travaux* ou *tâches de fond* (job batch), sont toujours reçues par un *interpréteur* (shell). Le shell est vu par les systèmes comme un processus. Une commande reçue par le shell peut être exécutée par le shell ; sinon, elle est exécutée dans un autre processus qui est créé par le shell. Le nom de la commande Bourne Again Shell est `bash` ; c'est un interpréteur de commandes avec une syntaxe similaire au langage C, un mécanisme d'*historique* (history) et une facilité de contrôle de *travaux* (job).

Le code de retour d'une commande est aussi un mécanisme fourni par le shell (quel qu'il soit) qui signale à l'utilisateur si l'exécution de cette commande s'est bien déroulée ou bien s'il y a eu un problème quelconque. Le code de retour est un petit entier positif ou nul, toujours compris entre 0 et 255. Par convention, un code de retour égal à 0 sans erreur (attention, c'est l'inverse du langage C !) signifie que la commande s'est exécutée correctement. Un code différent de 0 signifie soit une erreur d'exécution, soit une erreur syntaxique. Une variable système spéciale `$?` contient toujours la valeur de retour de la précédente commande (on peut afficher cette valeur avec la commande `echo`) : le paramètre spécial `?` du shell (à ne pas confondre avec le caractère générique `?` du shell) contient le code de retour de la dernière commande exécutée de manière séquentielle ; on emploie également l'expression *exécution synchrone*. L'exécution séquentielle signifie que le shell n'exécute une commande que lorsque la précédente s'est terminée. C'est bien sûr le mode d'exécution par défaut d'une commande :

ex : pwd

/home/toto affiche le répertoire courant de travail

echo \$? vérification du code de retour

0 affichage du code de retour 0 : la commande s'est exécutée correctement

ls -l vi

ls: vi: Aucun fichier ou répertoire de ce type affiche le message d'erreur

echo \$?

1 affichage du code de retour : erreur d'exécution !

Dans l'exemple précédent, la commande `ls` ne trouve pas le fichier correspondant à l'éditeur de texte `vi` dans le répertoire courant (ce qui est tout à fait normal !) et positionne un code de retour à 1.

En `bash`, il est possible d'obtenir la négation d'un code de retour d'une commande en plaçant “! ” devant celle-ci. Cela signifie que si le code de retour de la commande est égal à 0, alors le code de retour de “! ” est égal à 1 :

ex : ! ls fichier le code de retour de ls fichier est égal à 0

fichier affiche le fichier s'il existe

echo \$?

1 le code de retour est égal à 1 car “! ” est placé devant la commande `ls`

Le code de retour d'une suite de commandes est le code de retour de la dernière commande exécutée. Le code de retour de la suite de commandes `cmd1; cmd2; cmd3` est le code de retour de la commande `cmd3` :

ex : pwd; ls vi; echo bonjour

/home/toto

ls: vi: Aucun fichier ou répertoire de ce type

bonjour

echo \$?

0 code de retour de echo bonjour

Il en est de même pour le *tube* (pipeline) `cmd1 | cmd2 | cmd3`. Le code de retour sera celui de `cmd3` :

ex : cat /etc/passwd | grep daemon

daemon:x:2:2:daemon:/sbin:/sbin/nologin

echo \$?

0 code de retour de grep daemon

(\*) Note : un problème qui se pose au débutant lorsqu'il utilise le code de retour d'une commande, sous les systèmes UNIX : c'est que chaque commande positionne « à sa manière » les codes de retour différents de 0. Ainsi, un code de retour égal à 1 positionné par la commande `ls` n'a pas la même signification qu'un code de retour à 1 positionné par la commande `grep`. Il n'existe qu'une seule solution à ce problème : lire le manuel correspondant à la commande.

## Evaluation de commandes :

`bash` est le shell le plus utilisé avec GNU/Linux. Le shell permet de donner des instructions au système en ligne de commande. Lorsque l'on tape une instruction, le premier mot est la commande elle-même. Ensuite viennent les paramètres éventuels. Le shell va alors essayer de trouver à quoi la commande correspond dans l'ordre suivant. Tout d'abord si la commande contient une spécification d'emplacement, le programme se trouvant dans ce répertoire est exécuté s'il s'y trouve et si l'utilisateur a les droits adéquats. Sinon une erreur est affichée. Ensuite `bash` regarde quels `alias` sont définis. Un `alias` est un moyen d'exécuter une commande par un raccourci. Pour que par exemple le fait de taper `mon_alias` exécute `ma_commande`, il faudra déclarer l'`alias` comme ceci :

```
ex: alias mon_alias='ma_commande'
```

Les guillemets ne sont utiles que si la commande a des paramètres (c'est à dire qu'elle contient des espaces). Le contenu de `ma_commande` sera évalué selon le même schéma pour trouver le programme à exécuter :

```
ex: mon_alias paramètre1 paramètre2
```

Tout se passe comme si `ma_commande` avait été tapé à la place de `mon_alias`. Il va donc aussi avoir les paramètres. Si aucun `alias` n'est trouvé, ce sont alors les fonctions qui sont examinées. Une fonction est un moyen de regrouper plusieurs commandes. Et de faire des traitements plus complexe qu'avec un `alias`. Une *fonction* (function) se déclare comme ceci :

```
ex: fonction ma_fonction() { ma_commande1; ma_commande2; }
```

Pour la définition de la *fonction* (function), on peut omettre le mot-clé `fonction` ou les parenthèses, mais au moins un des deux doit être présent.

Le point-virgule (;) permet de séparer les commandes et aussi de les terminer. A l'intérieur de la *fonction* (function), des variables positionnelles (`$1`, `$2`, ...) sont définies automatiquement par le shell. Elles correspondent aux paramètres passés à la fonction :

```
ex: fonction mon_affichage { echo "SAIT : $1"; }
    mon_affichage Bonjour appel de la fonction
    SAIT : Bonjour affichage du résultat
```

Dans le cas où aucune fonction correspondant n'existe, le shell consulte ses *commandes intégrées* (shell builtin commands). Avec `bash`, on peut citer comme exemple de commandes intégrées `cd` (pour changer le répertoire courant) ou `echo` vu précédemment.

Enfin le shell parcourt les répertoires contenus dans la variable d'environnement `$PATH` pour y chercher un programme du nom tapé. Si celui-ci est trouvé, il est exécuté, sinon une erreur est affichée :

```
ex: echo $PATH
    /home/toto/bin:/usr/local/bin:/usr/bin:/bin
```

Si deux programmes avec le même nom se situent dans `/usr/local/bin` et `/bin`, c'est le premier qui sera exécuté. En résumé voici l'ordre d'évaluation si la commande est exécutée sans spécification de chemin : alias, fonction, commandes intégrées, exécutables dans un des répertoires de `$PATH`. Si tout cela échoue, un message d'erreur sera affiché. La commande `type` permet de savoir à quelle catégorie appartient une commande. Dans les deux premiers cas, on a la correspondance de l'alias ou de la *fonction* (fonction). Les commandes intégrées sont indiquées par un texte générique et pour le dernier cas, le chemin de l'exécutable est affiché.

Lorsque l'on exécute une commande, `bash` crée un sous-shell pour le lancer. C'est-à-dire que cela se passe comme si on avait lancé une nouvelle fois `bash`. Pour modifier ce comportement, on peut placer un point (`.`) en début de ligne. Cela provoquera alors l'exécution dans le shell courant :

ex : `. ma_commande` exécute la commande (ne pas oublier l'espace entre le point et la commande)

### Historique des commandes :

`bash` conserve les dernières commandes tapées afin de pouvoir facilement les retrouver. Elles sont sauvegardées en mémoire pour le shell courant si plusieurs sont lancés. Et lorsque celui-ci est quitté, elles sont écrites dans le fichier `.bash_history` qui se trouve à la racine du répertoire personnel de l'utilisateur (`~`). Ce fichier est celui par défaut, son nom peut être changé par la variable d'environnement `$HISTFILE`. Le nombre de précédentes commandes sauvegardées en mémoire est défini par la variable `$HISTSIZE` (par défaut 1000) et le nombre devant être sauvegardées dans le fichier par la variable `$HISTFILESIZE` (également 1000 par défaut). Les commandes sont sauvegardées avant la substitution éventuelle faite (si des caractères `*`, `?` ou `[ ]` sont présents).

Pour parcourir l'historique, il suffit d'utiliser les touches fléchées (à gauche du pavé numérique). La flèche vers le haut permet de parcourir les commandes dans l'ordre inverse de leur exécution et la flèche vers le bas dans l'autre sens. Peuvent également être utilisées respectivement les combinaisons `[Ctrl-P]` (previous) et `[Ctrl-N]` (next). Pour les habitués de l'éditeur de texte Emacs, ce sont les combinaisons de touches de ce logiciel qui peuvent être utilisées pour éditer les commandes entrées.

De même que dans cet éditeur, on peut rechercher parmi les commandes précédentes. Pour cela il suffit de maintenir les touches `[Ctrl-R]`. Apparaît alors une invite permettant la recherche de la forme suivante :

ex : `(reverse-i-search)`':`

Qui remplace alors l'invite de commande habituelle. La recherche se fait de manière incrémentale. C'est à dire qu'au fur et à mesure que l'on tape des caractères, la plus récente ligne de l'historique trouvée sera affichée. Pour interrompre la recherche, on peut appuyer simultanément `[Ctrl-G]` ou une des touches fléchées (gauche et droite permettant alors de l'éditer). La touche `[Enter]` permet d'exécuter la commande affichée.

## Le Shell script (programmation Shell)

Les diverses commandes, que nous avons vu jusqu'à présent saisies sur une ligne de commande, peuvent aussi être stockées dans un fichier. Après avoir doté le fichier de lignes regroupées de commandes shell, il faut les autorisations d'exécution (x) à l'aide de la commande `chmod`, de cette manière le nom du fichier peut-être utilisé comme une commande normale :

ex : `vi` `status` contiendra les lignes suivantes :

```
# Début de fichier : un commentaire est précédé par le caractère "#"  
pwd      # indique le répertoire de travail  
ls -l    # liste les fichiers  
who      # montre qui est connecté  
:wq      appuyez [Esc] puis enregistrez (w) le fichier en quittant (q) l'éditeur vi  
chmod u=rwx,go=x status donne la permission d'exécuter le script à tous les utilisateurs
```

Si vous exécutez la commande `status` dans le shell vous constaterez que l'erreur 127 est levée :

ex : `status` exécution de la commande

```
bash: status: command not found message d'erreur : la commande n'est pas trouvée  
echo $? code de retour différent de 0 : 127
```

Cela signifie simplement que le shell cherche ce fichier au mauvais endroit dans l'arborescence. Il ne cherche pas obligatoirement dans le répertoire courant. Il faut spécifier explicitement ce répertoire source, c'est à dire soit en spécifiant le chemin absolu plus le script à exécuter, soit en se plaçant dans son répertoire courant (dans ce cas on l'appelle par `./`) :

ex : `/home/toto/status` spécifie le chemin absolu du fichier à exécuter

```
./status appel de la commande depuis son répertoire courant toto
```

Le shell interprète les scripts à l'exécution commande après commande. Le shell commence par vérifier s'il s'agit d'une commande interne ou externe. Si c'est une commande externe, le shell cherchera à des endroits bien précis de l'arborescence, pour trouver le fichier de même nom. S'il le trouve un second test est effectué. Au cours de ce second test, le shell vérifie s'il s'agit d'un fichier binaire ou d'un script (\*). Dans tous les cas un nouveau shell est lancé. Le shell charge le fichier binaire si la commande est un nom de programme. Dans le cas d'un script le nouveau shell interprétera les commandes les unes après les autres.

Les scripts shell externes sont des fichiers de configuration, des programmes pouvant exécuter des tâches particulières : *parefeu* (firewall), *sauvegarde* (backup), *redirection* (forward) et services ou *démons* (daemons) :

ex : `ls -l /etc/init.d` répertoire contenant les scripts shell permettant d'initialiser des services lancés par le processus `init`

`ls -l /etc/profile.d` répertoire spécial permettant de stocker des shell scripts appelés par le fichier de configuration `profile`

```
vi /etc/profile visualise le fichier de configuration
```

(\*) Note : les quatre premiers caractères d'un fichier binaire contiennent un code d'identification spécial (une espèce de nombre magique). Les shell scripts n'ont pas ce code d'identification.

## Exécution et structure conditionnelle (création d'un script shell) : test, [ ], if et else

Une *procédure* (script shell) est une suite de commandes shell. Elle peut accepter des paramètres :

ex : commande paramètre1 paramètre2 ... paramètren (nième paramètre).

Le premier paramètre est référencé par \$1, le second par \$2, ..., le neuvième par \$9 :

ex : \$1, \$2, ..., \$9 sont des paramètres de commande

La création d'un script commence en général par la ligne `#!/bin/bash` déclarant le type de shell qui sera appelé pour interpréter les commandes. Dans ce cas nous utilisons le bash :

```
ex:#!/bin/bash
# Essai de script : ce script permet de tester la présence des
# paramètres de la commande
# Cette commande externe (script shell) doit comporter deux arguments
PATH=$PATH:/home/toto/myscript; # Chemin de recherche du nom de la
# commande
export PATH # Exporte la variable du chemin

usage() { # Nom de la procédure
    echo "usage: $1 $2" # Affiche les paramètres
}

main() { # Procédure principale
    if [ $# = 0 ]; then # Teste si nombre de paramètres = 0
        usage `basename $0` "fichier"; # appel de la procédure usage
        exit 1 # Code retour 1
    fi # Fin de la condition if
}
main $*
```

### Exécution conditionnelle :

Si la commande `test` permet de vérifier la réalisation d'une condition en renvoyant un code de retour (valeur de retour). La commande `test` peut être remplacée par des crochets `[ ]` :

ex : `test -r xyz && "Le fichier xyz est lisible"` enchaînement avec `&&`

`echo $?`

`[-r xyz] && "Le fichier xyz est lisible"` idem

En relation avec `test` mais cette fois pour piloter l'ordre d'exécution des commandes dans un fichier de script shell, nous utilisons des structures de contrôle. Ces structures de contrôle permettent de décider de l'exécution d'une commande en fonction de la réalisation d'une condition, c'est notamment le cas des structure de type `if`. L'instruction `if` permet de tester une suite de commandes, mais aussi une expression placée entre `[ ]` afin d'exécuter des instructions si une condition est vraie. L'élément important est la valeur de retour de la dernière commande de cette suite. S'il s'agit de 0 toutes les commandes placées derrière le mot clé `then` sont exécutées. Si la valeur de retour est différente de 0, ce sont les commandes placées derrière le mot clé `else` qui seront exécutées :

Condition simple :

- suite de commandes

```
if suite_commandes then
    commandes
fi
```

ex:if grep "PATH" /etc/profile > /dev/null 2>&1; then

```
    echo "Chemin absolu (recherche PATH exécution de commande) : $PATH"
fi
```

- avec paramètres

ex:if grep "\$1" /etc/profile > /dev/null 2>&1; then

```
    echo "$1 trouvé"
fi
```

- expression/condition entre crochets

```
if [ expression_condition ] then
    action
fi
```

ex:if [ \$# = 0 ]; then

```
    echo "Erreur : Appel : $0 Critère" >&2
fi
if grep "$1" /etc/profile; then
    echo "$1 trouvé"
fi
```

action est une suite de commandes quelconques. L'indentation (4 espaces) n'est pas obligatoire mais très fortement recommandée pour la lisibilité du code. On peut aussi utiliser la forme complète :

Condition avec alternative :

- suite de commandes

```
if suite_commandes then
    commandes
else
    commandes
fi
```

ex:if grep "PATH" /etc/profile; then

```
    echo "Chemin absolu (recherche PATH exécution de commande) : $PATH"
else
    echo "Aucun chemin"
fi
```

- paramètres

ex:if grep "\$1" /etc/profile; then

```
    echo "$1 trouvé"
else
    echo "$1 non trouvé"
fi
```

- avec expression/condition entre crochets

```
if [ expression_condition ] then
    action
else
    action
fi
```

ex:if [ \$# = 0 ]; then

```
    echo "Erreur : Appel : $0 Critère : aucun : Nombre : $# " >&2
    exit 1
```

```
else
```

```
    echo "Succès : Appel : $0 Critère : $1 : Nombre : $# "
    exit 0
```

```
fi
```

Condition multiple :

- suite de commandes

```
if suite_commandes then
    commandes
elif commande then
    commandes
else
    commandes
fi
```

- avec expression/condition entre crochets

ex:if [ expression1\_condition1 ]; then

```
    action1
```

```
    elif [ expression2_condition2 ]; then
```

```
        action2
```

```
    else [ expression3_condition3 ]
```

```
        action3
```

```
fi
```

En vous inspirant du modèle de la condition multiple ci-dessus simplifiez le script shell ci-dessous :

```
#!/bin/bash
if [ $# = 0 ]
then
    echo "Erreur : Appel : $0 Critère : aucun " >&2
    exit 1
else
    if grep "$1" $2 > /dev/null 2>&1
    then
        echo "$1 trouvé dans $2"
    else
        echo "$1 non trouvé"
    fi
    exit 0
fi
```

## Opérateurs de comparaison :

Le shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Ci-dessous un aperçu des principaux opérateurs :

Opérateurs sur les fichiers :

`-e nom_fichier` vrai si `nom_fichier` existe

ex : [ `-e /etc/shadow` ]

`-d nom_fichier` vrai si `nom_fichier` est un répertoire

ex : [ `-d /tmp/trash` ]

`-f nom_fichier` vrai si `nom_fichier` est un fichier ordinaire

ex : [ `-f /tmp/glop` ]

`-L nom_fichier` vrai si `nom_fichier` est un lien symbolique

ex : [ `-L /home` ]

`-r nom_fichier` vrai si `nom_fichier` est lisible (r)

ex : [ `-r /boot/vmlinuz` ]

`-w nom_fichier` vrai si `nom_fichier` est modifiable (w)

ex : [ `-w /var/log` ]

`-x nom_fichier` vrai si `nom_fichier` est exécutable (x)

ex : [ `-x /sbin/halt` ]

`fichier1 -nt fichier2` vrai si `fichier1` plus récent que `fichier2`

ex : [ `/tmp/foo -nt /tmp/bar` ]

`fichier1 -o fichier2` vrai si `fichier1` plus ancien que `fichier2`

ex : [ `/tmp/foo -ot /tmp/bar` ]

## Opérateurs sur les chaînes :

`-z chaine` vrai si la chaine est vide

ex : [ `-z "$VAR"` ]

`-n chaine` vrai si la chaine est non vide

ex : [ `-n "$VAR"` ]

`chaine1 = chaine2` vrai si les deux chaînes sont égales

ex : [ `"$VAR" = "toto"` ]

`chaine1 != chaine2` vrai si les deux chaînes sont différentes

ex : [ `"$VAR" != "titi"` ]

## Opérateurs de comparaison numérique :

num1 -eq num2 égalité

ex : [ \$nombre -eq 37 ]

num1 -ne num2 inégalité

ex : [ \$nombre -ne 37 ]

num1 -lt num2 inférieur (<)

ex : [ \$nombre -lt 37 ]

num1 -le num2 inférieur ou égal (<=)

ex : [ \$nombre -le 37 ]

num1 -gt num2 supérieur (>)

ex : [ \$nombre -gt 37 ]

num1 -ge num2 supérieur ou égal (>=)

ex : [ \$nombre -ge 37 ]

## Expressions arithmétiques :

On peut facilement évaluer des expressions arithmétiques avec `bash`. Pour cela, on utilise la notation `$(( ))` qui sera remplacée par le résultat du calcul mathématique. Voici les opérations pouvant être utilisées par ordre de priorité inverse (extrait de la page de manuel de `bash`) :

- + plus et moins unaire
- ! ~ négations logique et binaire
- \* / % multiplication, division, reste
- + - addition, soustraction
- << >> décalage arithmétique à gauche et à droite
- <= >= < > comparaisons
- == != égalité et différence
- & ET binaire, ^ OU exclusif binaire, | OU binaire,
- && ET logique, || OU logique
- = \*= /= %= += -= <<= >>= &= ^= |= assignations

`bash` effectuera le remplacement par la valeur avant l'exécution des commandes éventuelles. La commande suivante est une erreur :

ex : `$((variable=3+2))`

`bash: 5: command not found` la variable contiendra bien la valeur 5 à la suite de cette action. Mais `bash` a remplacé l'expression par sa valeur, donc tout se passe comme si on avait uniquement saisi 5 à l'invite (qui se trouve être un programme inconnu).

Une autre notation est possible en mettant `$ [ ]` autour de l'expression à la place de `$(( ))`.