Dernière mise-à-jour : 2020/01/30 03:28

SO105 - La Ligne de Commande

Le Shell

Un shell est un **interpréteur de commandes** ou en anglais un **Command Line Interpreter (C.L.I)**. Il est utilisé comme interface pour donner des instructions ou **commandes** au système d'exploitation.

Le mot shell est générique. Il existe de nombreux shells dans le monde Unix, par exemple :

Shell	Nom	Date de Sortie	Inventeur	Commande	Commentaires
tsh	Thompson Shell	1971	Ken Thompson	sh	Le premier shell
sh	Bourne Shell	1977	Stephen Bourne	sh	Le shell commun à tous les Unix. Sous Solaris : /usr/bin/sh
csh	C-Shell	1978	Bill Joy	csh	Le shell BSD. Sous Solaris : /usr/bin/csh
tcsh	Tenex C-Shell	1979	Ken Greer	tcsh	Un dérivé du shell csh. Sous Solaris : /usr/bin/tcsh
ksh	Korn Shell	1980	David Korn	ksh	Uniquement libre depuis 2005. Sous Solaris : /usr/bin/ksh
bash	Bourne Again Shell	1987	Brian Fox	bash	Le shell par défaut de Linux et de MacOS X. Sous Solaris : /usr/bin/bash
zsh	Z Shell	1990	Paul Falstad	zsh	Zsh est plutôt orienté pour l'interactivité avec l'utilisateur.

Cette unité concerne l'utilisation du shell **ksh** sous Unix. Cependant, il peut aussi être utile aux utilisateurs de **bash** sous UNIX car les commandes sont pratiquement identiques.

Le shell **/bin/ksh** permet de:

- Rappeler des commandes
- Générer la fin de noms de fichiers
- Utiliser des alias
- Utiliser les variables tableaux
- Utiliser les variables numériques et l'arithmétique du langage C
- Gérer des chaines de caractères

- Utiliser les fonctions

Une commande commence toujours par un mot clef. Ce mot clef est interpréter par le shell selon le type de commande et dans l'ordre qui suit :

1. Les alias
2. Les fonctions
3. Les commandes internes au shell
4. Les commandes externes au shell

Le shell par défaut de root sous Solaris est **/bin/sh**.

Le shell des utilisateurs est défini par **root** dans le fichier **/etc/passwd** :

```
# cat /etc/passwd
root:x:0:0:Super-User:/:/sbin/sh
daemon:x:1:1::/:
bin:x:2:2::/usr/bin:
sys:x:3:3::/:
adm:x:4:4:Admin:/var/adm:
lp:x:71:8:Line Printer Admin:/usr/spool/lp:
uucp:x:5:5:uucp Admin:/usr/lib/uucp:
nuucp:x:9:9:uucp Admin:/var/spool/uucppublic:/usr/lib/uucp/uucico
smmsp:x:25:25:SendMail Message Submission Program:/:
listen:x:37:4:Network Admin:/usr/net/nls:
gdm:x:50:50:GDM Reserved UID:/:
webservd:x:80:80:WebServer Reserved UID:/:
postgres:x:90:90:PostgreSQL Reserved UID:/:/usr/bin/pfksh
svctag:x:95:12:Service Tag UID:/:
nobody:x:60001:60001:NFS Anonymous Access User:/:
noaccess:x:60002:60002:No Access User:/:
nobody4:x:65534:65534:SunOS 4.x NFS Anonymous Access User:/:
test:x:100:1:Test:/export/home/test:/bin/ksh
```

Ne modifiez **jamais** le shell de root dans le fichier **/etc/passwd**.

Devenez maintenant l'utilisateur test grâce à la commande **su -** :

```
# su - test
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
$
```

Vérifiez que le shell de l'utilisateur **test** soit bien **ksh** grâce à la consultation de la valeur de la variable système **SHELL** :

```
$ echo $SHELL
/bin/ksh
```

Il convient maintenant de modifier le shell de **test** en **bash** :

```
$ which bash
/usr/bin/bash
```

passez en tant que **root** grâce à la commande **exit** puis éditez le fichier **/etc/passwd** :

```
test:x:101:1:Test:/export/home/test:/usr/bin/bash
```

Ensuite connectez-vous de nouveau en tant que l'utilisateur test et utilisez la commande **echo** pour visualiser le contenu de la variable système **SHELL** :

```
# su - test
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
-bash-3.00$ echo $SHELL
/usr/bin/bash
```

Définissez de nouveau le CLI de l'utilisateur **test** en **/bin/ksh** avant de poursuivre.

Une commande tapée dans un CLI commence toujours par un mot clef. Ce mot clef est interprétée par le shell selon le type de commande et dans l'ordre qui suit :

1. Les alias
2. Les fonctions
3. Les commandes internes au shell
4. Les commandes externes au shell

La suite de ce cours concerne l'utilisation du shell **ksh** sous Solaris. Le shell **ksh** permet de:

- Rappeler des commandes
- Générer la fin de noms de fichiers
- Utiliser des alias
- Utiliser les variables tableaux
- Utiliser les variables numériques et l'arithmétique du langage C
- Gérer des chaînes de caractères
- Utiliser les fonctions

Defenissez le shell courant de root en tant que **/bin/ksh** :

```
# /bin/ksh
#
```

Les Commandes Internes et Externes au shell

Les commandes internes au shell sont des commandes telles **cd**. Pour vérifier le type de commande, il faut utiliser la commande **type** en tant que **root** :

```
# type cd
cd est une commande prédéfinie du shell
```

Les commandes externes au shell sont des binaires exécutables ou des scripts, généralement situés dans /usr/sbin, /usr/bin, /usr/openwin/bin ou /usr/ucb :

```
# type ifconfig
ifconfig est /usr/sbin/ifconfig
```

Les Aliases

Les alias sont des noms permettant de désigner une commande ou une suite de commandes et ne sont spécifiques qu'au shell qui les a créés ainsi qu'à l'environnement de l'utilisateur :

```
# type stop
stop est un alias exporté pour kill -STOP
```

ou

```
# su - test
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
$ type ls
ls est un alias suivi pour /usr/bin/ls
```

Notez que dans ce cas l'alias **ls** est en effet un alias qui utilise la **commande** ls elle-même.

La liste des alias définis peut être visualisée en utilisant la commande **alias** :

```
$ alias
autoload='typeset -fu'
command='command '
functions='typeset -f'
history='fc -l'
integer='typeset -i'
local=typeset
```

```
ls=/usr/bin/ls
nohup='nohup '
r='fc -e -'
stop='kill -STOP'
suspend='kill -STOP $$'
```

Un alias se définit en utilisant de nouveau la commande **alias** :

```
$ alias dir='ls -l'
$ alias
autoload='typeset -fu'
command='command '
dir='ls -l'
functions='typeset -f'
history='fc -l'
integer='typeset -i'
local=typeset
ls=/usr/bin/ls
nohup='nohup '
r='fc -e -'
stop='kill -STOP'
suspend='kill -STOP $$'
$ dir
total 6
-rw-r--r--   1 test     other        136 août 15 10:19 local.cshrc
-rw-r--r--   1 test     other        157 août 15 10:19 local.login
-rw-r--r--   1 test     other        174 août 15 10:19 local.profile
```

Notez que la liste des alias contient, sans distinction, les alias définis dans les fichiers de démarrage du système ainsi que l'alias **dir** créé par **test** qui n'est que disponible à **test** dans le terminal courant.

Pour supprimer un alias, il convient d'utiliser la commande **unalias** :

```
$ unalias dir
$ alias
autoload='typeset -fu'
cat=/usr/bin/cat
command='command '
functions='typeset -f'
history='fc -l'
integer='typeset -i'
local=typeset
ls=/usr/bin/ls
nohup='nohup '
r='fc -e -'
stop='kill -STOP'
suspend='kill -STOP $$'
```

Pour forcer l'exécution d'une commande et non l'alias il faut faire précéder la commande par le caractère **** :

```
$ alias ls='ls -l'
$ ls
total 6
-rw-r--r--   1 test     other        136 août 15 10:19 local.cshrc
-rw-r--r--   1 test     other        157 août 15 10:19 local.login
-rw-r--r--   1 test     other        174 août 15 10:19 local.profile
$ \ls
local.cshrc    local.login    local.profile
```

Dans l'exemple ci-dessus, la première commande étant un alias, la sortie est celle de la commande **ls -l** tandis que la deuxième est celle de la commande **ls** simple.

Le Prompt

Le prompt d'un utilisateur dépend de son statut :

- **$** pour un utilisateur normal
- **#** pour root

Rappeler des Commandes

Le shell **ksh** permet le rappel des dernières commandes saisies. Afin de connaître la liste des commandes mémorisées, utilisez la commande **history** en tant que l'utilisateur **test** :

```
$ history
5       exit
6       type ls
7       alias
8       alias dir='ls -l'
9       alias
10      cat /etc/shells
11      dir
12      \dir
13      unalias dir
14      alias
15      alias ls='ls -l'
16      ls
17      \ls
18      which chsh
19      ls -a
20      history
```

Rappelez-vous que la commande **history** est un alias :

```
$ alias
...
history='fc -l'
...
```

En réalité donc c'est la commande **fc** qui est appelée.

L'historique des commandes est en mode **emacs** par défaut. Afin de pouvoir rappeler les dernières commandes saisies, il convient de passer en mode **emacs** :

```
$ set -o emacs
```

Le rappel de la dernière commande se fait en utilisant les touches **[CTRL]-[P]** et le rappel de la commande suivante se fait en utilisant les touches **[CTRL]-[N]** :

Caractère de Contrôle	**Définition**
[CTRL]-[P]	Rappelle la commande précédente
[CTRL]-[N]	Rappelle la commande suivante

Le paramétrage de la fonction du rappel des commandes est fait en définissant des variables système.

Afin de faciliter la définition de ces variables, créez le fichier **/etc/kshrc** en tant que **root** :

```
#
# This file is the common Environment setup for Korn Shell
#
HISTSIZE=256
HISTFILE=$HOME/.sh_history
export HISTFILE HISTSIZE
set -o emacs
```

Vous noterez que dans ce fichier, la valeur de **HISTSIZE** est de **256**. Ceci implique que les dernières 256 commandes sont mémorisées.

Devenez maintenant l'utilisateur standard **test** :

```
# su - test
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
$
```

Les commandes mémorisées sont stockées dans le fichier **$HOME/.sh_history** ou **$HOME** indique le répertoire personnel de l'utilisateur concerné :

```
$ cat $HOME/.sh_history
□□echo $SHELL
whereis bash
which bash
exit
exit
type ls
alias
alias dir='ls -l'
alias
cat /etc/shells
dir
\dir
unalias dir
alias
alias ls='ls -l'
ls
\ls
ls -a
history
alias
set -o emacs
exit
cat $HOME/.sh_history
```

La comparaison du contenu de ce fichier avec la sortie de la commande **history** démontre que les deux sont identiques, à part bien

évidemment, la dernière commande saisie, soit history.

Vous pouvez rappeler une commande spécifique de l'historique en utilisant la commande **fc -e -** suivi du numéro de la commande à rappeler :

```
$ history
...
52      ls
53      pwd
54      vmstat
55      df
56      history
$ fc -e - 52
ls
local.cshrc    local.login    local.profile
```

Afin de faire appel à au fichier **/etc/kshrc**, éditez maintenant le fichier **/etc/profile** :

```
$ exit
# vi /etc/profile
```

en y ajoutant les lignes suivantes :

```
ENV=/etc/kshrc
export ENV
```

Vous obtiendrez une fenêtre similaire à celle-ci :

```
#ident  "@(#)profile    1.19    01/03/13 SMI"   /* SVr4.0 1.3   */

# The profile that all logins get before using their own .profile.

trap ""  2 3
```

```
export LOGNAME PATH

ENV=/etc/kshrc
export ENV

if [ "$TERM" = "" ]
then
        if /bin/i386
        then
                TERM=sun-color
        else
                TERM=sun
        fi
        export TERM
fi

#       Login and -su shells get /etc/profile services.
#       -rsh is given its environment in its .profile.
```

Connectez-vous de nouveau en tant que test :

```
# su - test
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
```

Vérifiez que vous pouvez rappeler les dernières commandes dans votre console.

Générer les fins de noms de fichiers

Le shell **ksh** permet la génération des fins de noms de fichiers. Celle-ci est accomplie grâce à l'utilisation de la touche **[echap]**. Dans l'exemple qui suit, la commande saisie est :

```
$ ls [echap] [echap]
```

Vous obtiendrez :

```
$ ls local.
```

Vous noterez que le shell propose **local.**. En effet, sans plus d'information, le shell ne sait pas quel fichier parmi les trois présents doit être ouvert :

```
$ ls | grep local.
local.cshrc
local.login
local.profile
```

Par contre si vous ajoutez la lettre **c** :

```
$ ls local.c [echap] [echap]
```

vous noterez que le nom du fichier est complété automatiquement. Ceci est du au fait qu'il n'existe qu'un seul élément dans le répertoire commençant par la chaîne **local.c** :

```
$ ls local.cshrc
```

Le shell interactif

Lors de l'utilisation du shell, nous avons souvent besoin d'exécuter une commande sur plusieurs fichiers au lieu de les traiter individuellement. A cette fin nous pouvons utiliser les caractères spéciaux.

Caractère Spéciaux	**Description**
*	Représente un ou une suite de caractères
?	Représente un caractères
[abc]	Représente un caractère parmi ceux entre crochets
[!abc]	Représente un caractère ne trouvant pas parmi ceux entre crochets
?(expression1|expression2| ...)	Représente 0 ou 1 fois l'expression1 ou 0 ou 1 fois l'expression2 ...
***(expression1|expression2| ...)**	Représente 0 à x fois l'expression1 ou 0 à x fois l'expression2 ...

Caractère Spéciaux	Description
+(expression1|expression2| ...)	Représente 1 à x fois l'expression1 ou 1 à x fois l'expression2 ...
@(expression1|expression2| ...)	Représente 1 fois l'expression1 ou 1 fois l'expression2 ...
!(expression1|expression2| ...)	Représente 0 fois l'expression1 ou 0 fois l'expression2 ...

Caractère *

Dans votre répertoire individuel, créez un répertoire **formation**. Ensuite créez dans ce répertoire 5 fichiers nommés respectivement f1, f2, f3, f4 et f5 :

```
$ pwd
/export/home/test
$ mkdir formation
$ cd formation
$ touch f1 f2 f3 f4 f5
$ ls -l
total 0
-rw-r--r--   1 test     other          0 août 15 16:00 f1
-rw-r--r--   1 test     other          0 août 15 16:00 f2
-rw-r--r--   1 test     other          0 août 15 16:00 f3
-rw-r--r--   1 test     other          0 août 15 16:00 f4
-rw-r--r--   1 test     other          0 août 15 16:00 f5
```

Afin de démontrer l'utilisation du caractère spécial *, saisissez la commande suivante :

```
$ echo f*
f1 f2 f3 f4 f5
```

Notez que le caractère * remplace un caractère ou une suite de caractères.

Caractère ?

Créez maintenant les fichiers f52 et f62 :

```
$ touch f52 f62
```

Saisissez ensuite la commande suivante :

```
$ echo f?2
f52 f62
```

Notez que le caractère **?** remplace **un seul** caractère.

Caractères [ ]

L'utilisation peut prendre plusieurs formes différentes :

Joker	Description
[xyz]	Représente le caractère x ou y ou z
[m-t]	Représente le caractère m ou n .... t
[!xyz]	Représente un caractère autre que x ou y ou z
[!m-t]	Représente un caractère autre que m ou n .... t

Afin de démontrer l'utilisation des caractères **[** et **]**, créez le fichier a100 :

```
$ touch a100
```

Ensuite saisissez les commandes suivantes et notez le résultat :

```
$ echo [a-f]*
a100 f1 f2 f3 f4 f5 f52 f62
$ echo [af]*
a100 f1 f2 f3 f4 f5 f52 f62
```

Notez ici que tous les fichiers commençant par les lettres **a**, **b**, **c**, **d**, **e** ou **f** sont affichés à l'écran.

```
$ echo [!a]*
f1 f2 f3 f4 f5 f52 f62
```

Notez ici que tous les fichiers sont affichés à l'écran, à l'exception d'un fichier commençant par la lettre **a** .

```
$ echo [a-b]*
a100
```

Notez ici que seul le fichier commençant par la lettre **a** est affiché à l'écran car il n'existe pas de fichiers commençant par la lettre **b**.

```
$ echo [a-f]
[a-f]
```

Notez que dans ce cas, il n'existe pas de fichiers dénommés **a**, **b**, **c**, **d**, **e** ou **f**. Pour cette raison, n'ayons trouvé aucune correspondance entre le filtre utilisé et les objets dans le répertoire courant, le commande **echo** retourne le filtre passé en argument, c'est-à-dire **[a-f]**.

?(expression)

Créez les fichiers f, f.txt, f123.txt, f123123.txt, f123123123.txt :

```
$ touch f f.txt f123.txt f123123.txt f123123123.txt
```

Saisissez la commande suivante :

```
$ ls f?(123).txt
f.txt     f123.txt
```

Notez ici que la commande affiche les fichiers ayant un nom contenant 0 ou 1 occurence de la chaîne **123**.

***(expression)**

Saisissez la commande suivante :

```
$ ls f*(123).txt
f.txt           f123.txt        f123123.txt     f123123123.txt
```

Notez ici que la commande affiche les fichiers ayant un nom contenant de 0 jusqu'à x occurences de la chaîne **123**.

+(expression)

Saisissez la commande suivante :

```
$ ls f+(123).txt
f123.txt        f123123.txt     f123123123.txt
```

Notez ici que la commande affiche les fichiers ayant un nom contenant entre 1 et x occurences de la chaîne **123**.

@(expression)

Saisissez la commande suivante :

```
$ ls f@(123).txt
f123.txt
```

Notez ici que la commande affiche les fichiers ayant un nom contenant 1 seule occurence de la chaîne **123**.

!(expression)

Saisissez la commande suivante :

```
$ ls f!(123).txt
f.txt           f123123.txt     f123123123.txt
```

Notez ici que la commande n'affiche que les fichiers ayant un nom qui ne contient **pas** la chaîne **123**.

Caractères d'Échappement

Afin d'utiliser un caractère spécial dans un contexte littéral, il faut utiliser un caractère d'échappement. Il existe trois caractères d'échappement :

Caractère	Description
\	Protège le caractère qui le suit
' '	Protège tout caractère, à l'exception du caractère ' lui-même, se trouvant entre les deux '
“ ”	Protège tout caractère, à l'exception des caractères **“** lui-même, **$**, **** et **'**, se trouvant entre les deux **“**

Afin d'illustrer l'utilisation des caractères d'échappement, considérons la commande suivante :

```
$ echo * est un caractère spécial [Entrée]
```

Lors de la saisie de cette commande dans votre répertoire **formation**, vous obtiendrez une fenêtre similaire à celle-ci :

```
$ echo * est un caractère spécial
a100 f1 f2 f3 f4 f5 f52 f62 est un caractère spécial
```

Vous noterez que le caractère spécial * a bien été interprété par le shell.

Afin de protéger le caractère *, nous devons utiliser un caractère d'échappement. Commençons par l'utilisation du caractère **** :

```
$ echo \* est un caractère spécial
* est un caractère spécial
```

Vous noterez que le caractère spécial * n'a pas été interprété par le shell.

Le même résultat peut être obtenu en utilisant ainsi :

```
$ echo "* est un caractère spécial"
* est un caractère spécial
$ echo '* est un caractère spécial'
* est un caractère spécial
```

Codes Retour

Chaque commande retourne un code à la fin de son exécution. La variable spéciale **$?** sert à stocker le code retour de la dernière commande exécutée.

Par exemple :

```
$ cd ..
$ mkdir codes
$ echo $?
0
$ touch codes/retour
$ rmdir codes
rmdir: échec de suppression de « codes »: Le dossier n'est pas vide
$ echo $?
2
```

Dans cette exemple la création du répertoire **codes** s'est bien déroulée. Le code retour stocké dans la variable $? est un zéro.

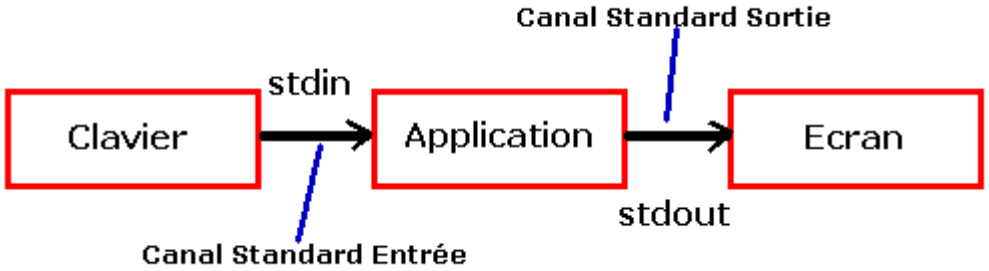
La suppression du répertoire a rencontré une erreur car **codes** contenait le fichier **retour**. Le code retour stocké dans la variable $? est un **deux**.

Si le code retour est **zéro** la dernière commande s'est déroulée sans erreur.

Si le code retour est **autre que zéro** la dernière commande s'est déroulée avec une erreur.

Redirections

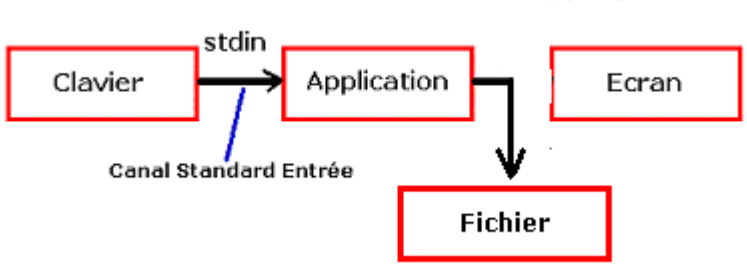
Votre dialogue avec le système Solaris utilise des canaux d’entrée et de sortie. On appelle le clavier, le **canal d’entrée standard** et l’écran, le **canal de sortie standard** :



;#; ;#;

Autrement dit, en tapant une commande sur le clavier, vous voyez le résultat de cette commande à l'écran.

Parfois, cependant il est utile de re-diriger le canal de sortie standard vers un fichier. De cette façon, le résultat d'une commande telle **free** peut être stocké dans un fichier pour une consultation ultérieure :



;#; ;#;

Cet effet est obtenu en utilisant une **redirection** :

```
$ vmstat > fichier [Entrée]
```

Si le fichier cible n'existe pas, il est créé et son contenu sera le résultat de la commande vmstat. Par contre si le fichier existe déjà, il sera écrasé.

Pour ajouter des données supplémentaires au même fichier cible, il faut utiliser une **double redirection** :

```
$ date >> fichier [Entrée]
```

De cette façon, la date du jour sera rajoutée à la fin de votre fichier après les informations de la commande free.

Pour visualiser le contenu du fichier fichier nous utilisons la commande **cat** :

```
$ vmstat > fichier
$ date >> fichier
$ cat fichier
 kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 -- --   in   sy   cs us sy id
 0 0 0 907680 469372 139 1580 8 0  0  0 14  3 -1  0  0  454 2712  367  2  4 94
mercredi 15 août 2012 17 h 33 CEST
```

Notez que la sortie standard ne peut être redirigée que dans **une seule direction**.

Les canaux d'entrées et de sorties sont numérotés :

- 0 = Le Canal d'entrée Standard
- 1 = Le Canal de Sortie Standard
- 2 = Le Canal d'erreur

La commande suivante créera un fichier nommé **erreurlog** qui contient les messages d'erreur de l'exécution de la commande **rmdir** :

```
$ rmdir formation/ 2> erreurlog
$ cat erreurlog
rmdir : répertoire "formation/" : Répertoire non vide
```

En effet l'erreur est générée parce que le répertoire **formation** n'est pas vide.

Nous pouvons également réunir des canaux. Pour mettre en application ceci, il faut comprendre que le shell traite les commandes de **gauche à droite**.

Dans l'exemple suivant, nous réunissons le canal de sortie et le canal d'erreurs :

```
$ vmstat > nom_du_fichier 2>&1
```

La syntaxe **2>&1** envoie la sortie du canal 2 au même endroit que le canal 1, à savoir le fichier dénommé **fichier**.

Il est possible de modifier le canal d'entrée standard afin de lire des informations à partir d'un fichier. Dans ce cas la redirection est obtenue en utilisant le caractère **<** :

```
$ wc -w < erreurlog
       8
```

Tubes

Il est aussi possible de relier des commandes avec un tube **|** .

Dans ce cas, le canal de sortie de la commande à gauche du tube est envoyé au canal d'entrée de la commande à droite du tube :

```
$ ls | wc -w
       6
```

Veuillez noter qu'il est possible de relier plusieurs tubes dans la même commande.

Rappelez-vous que la sortie standard ne peut être redirigée que dans une seule direction. Afin de pouvoir rediriger la sortie standard vers un fichier **et** le visualiser à l'écran, nous devons utiliser la commande **tee** avec un pipe :

```
$ date | tee fichier1
mercredi 15 août 2012 17 h 31 CEST
$ cat fichier1
mercredi 15 août 2012 17 h 31 CEST
```

Cette même technique nous permet de créer **deux fichiers** :

```
$ date | tee fichier1 > fichier2
$ cat fichier1
mercredi 15 août 2012 17 h 31 CEST
$ cat fichier2
mercredi 15 août 2012 17 h 31 CEST
```

Substitutions de Commandes

Il est parfois intéressant, notamment dans les scripts, de remplacer une commande par la valeur de sa sortie. Afin d'illustrer ce point, considérons la commande suivante :

```
$echo $(date) [Entrée]
```

```
$ echo date
date
$ echo $(date)
mercredi 15 août 2012 17 h 22 CEST
$ echo `date`
mercredi 15 août 2012 17 h 22 CEST
```

Notez le format de chaque substitution **$(commande)** ou **`commande`**. Sur un clavier français, l'anti-côte est accessible en utilisant les touches Alt Gr et 77.

Chainage de Commandes

Il est possible de regrouper des commandes à l'aide d'un sous-shell :

```
$ (ls -l; ps; who) > liste
```

Le syntaxe de cette commande est :

```
Commande1 && Commande2
```

Par exemple :

```
$ cat liste
total 10
drwxr-xr-x   2 test     other        512 août 15 16:10 codes
drwxr-xr-x   2 test     other        512 août 15 16:03 formation
-rw-r--r--   1 test     other          0 août 15 17:25 liste
-rw-r--r--   1 test     other        136 août 15 10:19 local.cshrc
-rw-r--r--   1 test     other        157 août 15 10:19 local.login
-rw-r--r--   1 test     other        174 août 15 10:19 local.profile
```

et :

```
$ rmdir formation && pwd
rmdir : répertoire "formation" : Répertoire non vide
```

Dans ce cas, Commande 2 est exécutée uniquement dans le cas où Commande1 s'est exécuté sans erreur
Cet exemple envoie le résultat des trois commandes vers le fichier liste en les traitant en sous-tâches (en tâches de fond).
Ou :
Rappelez-vous que chaque commande transmet la façon dont elle a été exécutée (Code Retour). Dans le cas d'un traitement sans erreurs, la valeur du Code retour est 0, sinon elle est autre que 0.

```
Commande1 || Commande2
```

De cette façon, en utilisant les caractères **&&** ou **||**, les commandes peuvent être chainées en fonction du Code Retour de la commande précédente.
Dans ce cas, Commande2 est exécuté si Commande1 a rencontré une erreur.

&& est utilisé afin de s'assurer que la deuxième commande s'exécute dans le cas où la valeur du Code Retour de la première commande est 0, autrement dit qu'il n'y a pas eu d'erreurs.
Par exemple :

```
$ ls || pwd
codes          liste          local.login
formation      local.cshrc    local.profile
```

et :

```
$ rmdir formation || pwd
rmdir : répertoire "formation" : Répertoire non vide
/export/home/test
```

Affichage des variables du shell

Une variable du shell peut être affichée grâce à la commande :

```
$echo $nom_de_la_variable [Entrée]
```

Les variables principales

Variable	Description
PWD	Le répertoire courant.
OLDPWD	Le répertoire avant la dernière commande cd. Même chose que la commande **cd -**.
RANDOM	Un nombre aléatoire entre 0 et 32767
SECONDS	Le nombre de secondes écoules depuis le lancement du shell
HISTFILE	Le fichier historique
HISTSIZE	Le nombre de commandes mémorisées dans le fichier historique
HOME	Le répertoire de connexion.
MAIL	Le fichier contenant le courrier.
MAILCHECK	La fréquence de vérification du courrier en secondes.
PATH	Le chemin de recherche des commandes.
PS1	Le prompt par défaut.
PS2	Le deuxième prompt par défaut
PS3	Le troisième prompt par défaut
PS4	Le quatrième prompt par défaut
SHELL	Le shell de préférence.
TMOUT	Le nombre de secondes moins 60 d'inactivité avant que le shell exécute la commande **exit**.
LANG	La valeur générique pour l'internationalisation
LC_MONETARY	Le symbole monétaire
LC_TIME	Le format de l'heure et de la date
LC_COLLATE	Le jeu de caractères utilisé dans les tris.
LC_NUMERIC	Le code pour le symbole décimal et le séparateur de milliers.

Variable	Description
LC_CTYPE	Le code pour la classification des caractères.
ERRNO	Code de retour du dernier appel système
ENV	Variable contenant le nom du script à exécuter à chaque démarrage d'un shell intéractif
FCEDIT	Editeur de texte utilisé par la commande **fc**

Les Variables de Régionalisation et d'Internationalisation

L'**Internationalisation**, aussi appelé **i18n** car il y a 18 lettres entre la lettre **I** et la lettre **n** dans le mot *Internationalization*, consiste à adapter un logiciel aux paramètres variant d'une région à l'autre :

- longueur des mots,
- accents,
- écriture de gauche à droite ou de droite à gauche,
- unité monétaire,
- styles typographiques et modèles rédactionnels,
- unités de mesures,
- affichage des dates et des heures,
- formats d'impression,
- format du clavier,
- etc ...

Le **Régionalisation**, aussi appelé **i10n** car il y a 10 lettres entre la lettre **L** et la lettre n du mot *Localisation*, consiste à modifier l'internalisation en fonction d'une région spécifique.

Le code pays complet prend la forme suivante : **langue-PAYS.jeu_de_caractères**. Par exemple, pour la langue française les valeurs de langue-PAYS sont :

- fr-BE = la Belgique francophone,
- fr-CA = le Québec,
- fr-FR = la France,
- fr-LU = le Luxembourg,
- fr-MC = Monaco,
- fr-CH = la Suisse francophone.

Les variables système les plus importants contenant les informations concernant le régionalisation sont :

Variable	Description
LC_ALL	Avec une valeur non nulle, celle-ci prend le dessus sur la valeur de toutes les autres variables d'internationalisation
LANG	Fournit une valeur par défaut pour les variables d'environnement dont la valeur est nulle ou non définie.
LC_CTYPE	Détermine les paramètres régionaux pour l'interprétation de séquence d'octets de données texte en caractères.

Par exemple :

```
$ echo $LC_ALL

$ echo $LC_CTYPE
fr_FR.ISO8859-1
$ echo $LANG
fr_FR.ISO8859-1
```

Les variables spéciales

Variable	Description
$LINENO	Contient le numéro de la ligne courante du script ou de la fonction
$$	Contient le PID du shell en cours
$PPID	Contient le PID du processus parent du shell en cours
$0	Contient le nom du script en cours tel que ce nom ait été saisi sur la ligne de commande
$1, $2 ...	Contient respectivement le premier argument, deuxième argument etc passés au script
$#	Contient le nombre d'arguments passés au script
$*	Contient l'ensemble des arguments passés au script
$@	Contient l'ensemble des arguments passés au script

Options du Shell ksh

Pour visualiser les options du shell ksh, il convient d'utiliser la commande **set** :

```
$ set -o
Paramétrage courant des options
allexport       off
bgnice          on
emacs           on
errexit         off
gmacs           off
ignoreeof       off
interactive     on
keyword         off
markdirs        off
monitor         on
noexec          off
noclobber       off
noglob          off
nolog           off
notify          off
nounset         off
privileged      off
restricted      off
trackall        off
verbose         off
vi              off
viraw           off
xtrace          off
```

Pour activer une option il convient de nouveau à utiliser la commande **set** :

```
$ set -o allexport
$ set -o
Paramétrage courant des options
allexport       on
bgnice          on
emacs           on
```

```
errexit         off
gmacs           off
ignoreeof       off
interactive     on
keyword         off
markdirs        off
monitor         on
noexec          off
noclobber       off
noglob          off
nolog           off
notify          off
nounset         off
privileged      off
restricted      off
trackall        off
verbose         off
vi              off
viraw           off
xtrace          off
```

Notez que l'option **allexport** a été activée.

Pour désactiver une option, on utilise la commande **set** avec l'option **+o** :

```
$ set +o allexport
$ set -o
Paramétrage courant des options
allexport       off
bgnice          on
emacs           on
errexit         off
gmacs           off
ignoreeof       off
```

```
interactive     on
keyword         off
markdirs        off
monitor         on
noexec          off
noclobber       off
noglob          off
nolog           off
notify          off
nounset         off
privileged      off
restricted      off
trackall        off
verbose         off
vi              off
viraw           off
xtrace          off
```

Parmi les options, voici la description des plus intéressantes :

Option	**Valeur par Défaut**	**Description**
allexport	off	Le shell export automatiquement toute variable
emacs	off	L'édition de la ligne de commande est au style emacs
noclobber	off	Les simples redirections n'ecrasent pas le fichier de destination
noglob	off	Désactive l'expansion des caractères génériques
nounset	off	Le shell retourne une erreur lors de l'expansion d'une variable inconnue
verbose	off	Affiche les lignes de commandes saisies
vi	on	L'édition de la ligne de commande est au style vi

Exemples

noclobber

```
$ set -o noclobber
$ pwd > file
$ pwd > file
ksh: file: ce fichier existe déjà
$ pwd >| file
$ set +o noclobber
```

noglob

```
$ set -o noglob
$ echo *
*
$ set +o noglob
$ echo *
core Desktop Documents erreurlog fichier1 fichier2 file formation kshrc kshrc1 liste local.cshrc local.login
local.profile nom_du_fichier
```

nounset

```
$ set -o nounset
$ echo $FENESTROS
ksh: FENESTROS: paramètre non défini
$ set +o nounset
$ echo $FENESTROS

$
```

verbose

```
$ set -o verbose
$ echo fenestros
echo fenestros
fenestros
$ set +o verbose
set +o verbose
$ echo fenestros
fenestros
```

Les Scripts Shell

Le but de la suite de cette unité est de vous amener au point où vous êtes capable de comprendre et de déchiffrer les scripts, notamment les scripts de démarrage ainsi que les scripts de contrôle des services.

Écrire des scripts compliqués est en dehors de la portée de cette unité car il nécessite une approche programmation qui ne peut être adressée que lors d'une formation dédiée à l'écriture des scripts.

Exécution

Un script shell est un fichier dont le contenu est lu en entrée standard par le shell. Le contenu du fichier est lu et exécuté d'une manière séquentielle. Afin qu'un script soit exécuté, il suffit qu'il puisse être lu au quel cas le script est exécuté par un shell fils soit en l'appelant en argument à l'appel du shell :

/usr/bin/ksh monscript

soit en redirigeant son entrée standard :

/usr/bin/ksh < monscript

Dans le cas où le droit d'exécution est positionné sur le fichier script et à condition que celui-ci se trouve dans un répertoire spécifié dans le PATH de l'utilisateur qui le lance, le script peut être lancé en l'appelant simplement par son nom :

monscript

Dans le cas où le script doit être exécuté par le shell courant, dans les mêmes conditions que l'exemple précédent, mais qu'il se trouve dans un répertoire autre qu'un des répertoires spécifiés dans le PATH de l'utilisateur qui le lance, il convient de naviguer au répertoire concerné et de lancer la commande suivante :

./monscript

Dans le cas où le script doit être exécuté par le shell courant, dans les mêmes conditions que l'exemple précédent, et non par un shell fils, il convient de le lancer ainsi :

. monscript

Dans un script il est fortement conseillé d'inclure des commentaires. Les commentaires permettent à d'autres personnes de comprendre le script. Toute ligne de commentaire commence avec le caractère **#**.

Il existe aussi un **pseudo commentaire** qui est placé au début du script. Ce pseudo commentaire permet de stipuler quel shell doit être utilisé pour l'exécution du script. L'exécution du script est ainsi rendu indépendant du shell de l'utilisateur qui le lance. Le pseudo commentaire commence avec les caractères **#!**. Chaque script commence donc par une ligne similaire à celle-ci :

```
#!/bin/sh
```

Pour illustrer l'écriture et l'exécution d'un script, créez le fichier **monscript** avec **vi** :

```
$ vi monscript [Entrée]
```

Editez votre fichier ainsi :

```
#debut du fichier monscript
pwd
ls
```

```
#fin du fichier monscript
```

Sauvegardez votre fichier et sortez du programme vi. Lancez ensuite votre script en passant le nom du fichier en argument à /bin/ksh :

```
$ /usr/bin/ksh monscript
/export/home/test
codes          fichier1       liste          local.profile
erreurlog      fichier2       local.cshrc    monscript
fichier        formation      local.login
```

Lancez ensuite le script en redirigeant son entrée standard :

```
$ /usr/bin/ksh < monscript
/export/home/test
codes          fichier1       liste          local.profile
erreurlog      fichier2       local.cshrc    monscript
fichier        formation      local.login
```

Pour lancer le script en l'appelant simplement par son nom, son chemin doit être inclus dans votre PATH.

Saisissez les commandes suivantes :

```
$ pwd
/export/home/test
$ mkdir bin
$ PATH=$PATH:$HOME/bin
$ export PATH
$ echo $PATH
/usr/bin::/export/home/test/bin
```

Vous constaterez que le répertoire **/export/home/test/bin** a été rajouté à votre PATH grâce à la ligne **PATH=$PATH:$HOME/bin**. Déplacez votre script dans ce répertoire, vérifiez les permissions de votre script, rendez-le exécutable pour votre utilisateur et vérifiez qu'il est bien exécutable:

```
$ mv monscript ~/bin
```

```
$ cd ~/bin
$ chmod u+x monscript
$ ls -l
total 2
-rwxr--r--   1 test     other         61 août 15 17:45 monscript
```

Positionnez-vous dans le répertoire /tmp et exécutez maintenant votre script en l'appelant par son nom :

```
$ cd /tmp
$ monscript
/tmp
breg_business_logic_20120812104901282.log
breg_business_logic_20120812104901282.log.lck
hsperfdata_noaccess
hsperfdata_root
ogl_select270
rootswup.trc
sh6826.1
```

Dans les trois cas précédents, le script a été exécuté dans un shell fils. Pour l'exécuter dans le shell en cours, rappelez-vous que la commande est :

```
$ . monscript
/tmp
breg_business_logic_20120812104901282.log
breg_business_logic_20120812104901282.log.lck
hsperfdata_noaccess
hsperfdata_root
ogl_select270
rootswup.trc
sh6826.1
```

La commande read

Vous êtes actuellement connecté en tant que l'utilisateur **test**. Devenez maintenant **root** grâce à la commande **exit**.

La commande **read** lit son entrée standard et affecte les mots saisis dans la ou les variable(s) passée(s) en argument :

```
# read var1 var2
fenestros edu
# echo $var1
fenestros
# echo $var2
edu
```

La séparation entre le contenu des variables est l'espace. Par conséquent il est intéressant de noter les exmples suivants :

```
# read var1 var2 var3 var4
fenestros edu est super!
# echo $var1
fenestros
# echo $var2
edu
# echo $var3
est
# echo $var4
super!
```

```
# read var1 var2
fenestros edu est super!
# echo $var1
fenestros
# echo $var2
```

```
edu est super!
```

Code de retour

La commande **read** renvoie un code de retour de **0** dans le cas où elle ne reçoit pas l'information **fin de fichier** matérialisée par les touches Ctrl+D :

```
# read var
fenestros
# echo $?
0
# echo $var
fenestros
```

Le contenu de la variable **var** peut être vide et la valeur du code de retour **0** grâce à l'utilisation de la touche Entrée :

```
# read var

# echo $?
0
# echo $var

#
```

Le contenu de la variable **var** peut être vide et la valeur du code de retour **autre que 0** grâce à l'utilisation des touches Ctrl+D :

```
# read var
^D
# echo $?
1
# echo $var

#
```

La variable IFS

La variable IFS contient par défaut les caractères Espace, Tab et Entrée :

```
# echo "$IFS" | od -c
0000000      \t  \n  \n
0000004
```

La valeur de cette variable définit donc le séparateur de mots lors de la saisie des contenus des variables avec la commande **read**. La valeur de la variable **IFS** peut être modifiée :

```
# OLDIFS="$IFS"
# echo "$OLDIFS" | od -c
0000000      \t  \n  \n
0000004
# IFS=":"
# echo "$IFS" | od -c
0000000   :  \n
0000002
```

De cette façon l'espace redevient un caractère normal :

```
# read var1 var2 var3
fenestros:edu est:super!
# echo $var1
fenestros
# echo $var2
edu est
# echo $var3
super!
```

Restaurez l'ancienne valeur de IFS avec la commande IFS="$OLDIFS"

La commande test

La commande **test** peut être utilisée avec deux syntaxes :

test *expression*

ou

[Espace*expression*Espace**]**

Tests de Fichiers

Test	Description
-f fichier	Retourne vrai si fichier est d'un type standard
-d fichier	Retourne vrai si fichier est d'un type répertoire
-r fichier	Retourne vrai si l'utilisateur peut lire fichier
-w fichier	Retourne vrai si l'utilisateur peut modifier fichier
-x fichier	Retourne vrai si l'utilisateur peut exécuter fichier
-e fichier	Retourne vrai si fichier existe
-s fichier	Retourne vrai si fichier n'est pas vide
fichier1 -nt fichier2	Retourne vrai si fichier1 est plus récent que fichier2
fichier1 -ot fichier2	Retourne vrai si fichier1 est plus ancien que fichier2
fichier1 -ef fichier2	Retourne vrai si fichier1 est identique à fichier2

Exemples

Testez si le fichier **a100** est un fichier ordinaire :

```
# cd formation
# test -f a100
# echo $?
```

```
0
# [ -f a100 ]
# echo $?
0
```

Testez si le fichier a101 existe :

```
# [ -f a101 ]
# echo $?
1
```

Testez si /export/home/test/formation est un répertoire :

```
# [ -d /export/home/test/formation ]
# echo $?
0
```

Tests de chaînes de caractère

Test	Description
-n chaîne	Retourne vrai si chaîne n'est pas de longueur 0
-z chaîne	Retourne vrai si chaîne est de longueur 0
chaîne1 = chaîne2	Retourne vrai si chaîne1 est égale à chaîne2
chaîne1 != chaîne2	Retourne vrai si chaîne1 est différente de chaîne2
chaîne1	Retourne vrai si chaîne1 n'est pas vide

Exemples

Testez si les deux chaînes sont égales :

```
# chaine1="root"
# chaine2="fenestros"
```

```
# [ "chaine1" = "chaine2" ]
# echo $?
1
```

Testez si la chaîne1 n'a pas de longueur 0 :

```
# [ -n "chaine1" ]
# echo $?
0
```

Testez si la chaîne1 a une longueur de 0 :

```
# [ -z "chaine1" ]
# echo $?
1
```

Tests sur des nombres

Test	Description
valeur1 -eq valeur2	Retourne vrai si valeur1 est égale à valeur2
valeur1 -ne valeur2	Retourne vrai si valeur1 n'est pas égale à valeur2
valeur1 -lt valeur2	Retourne vrai si valeur1 est inférieure à valeur2
valeur1 -le valeur2	Retourne vrai si valeur1 est inférieur ou égale à valeur2
valeur1 -gt valeur2	Retourne vrai si valeur1 est supérieure à valeur2
valeur1 -ge valeur2	Retourne vrai si valeur1 est supérieure ou égale à valeur2

Exemple

Comparez les deux nombres **nombre1** et **nombre2** :

```
# read nombre1
35
```

```
# read nombre2
21
# [ $nombre1 -lt $nombre2 ]
# echo $?
1
# [ $nombre2 -lt $nombre1 ]
# echo $?
0
# [ $nombre2 -eq $nombre1 ]
# echo $?
1
```

Les opérateurs

Test	Description
!expression	Retourne vrai si expression est fausse
expression1 -a expression2	Représente un et logique entre expression1 et expression2
expression1 -o expression2	Représente un ou logique entre expression1 et expression2
\(expression\)	Les parenthèses permettent de regrouper des expressions

Exemples

Testez si $fichier n'est pas un répertoire :

```
# fichier=a100
# [ ! -d $fichier ]
# echo $?
0
```

Testez si $repertoire est un répertoire **et** si l'utilisateur à le droit de le traverser :

```
# repertoire=/usr
```

```
# [ -d $repertoire -a -x $repertoire ]
# echo $?
0
```

Testez si l'utilisateur peut écrire dans le fichier a100 **et** /usr est un répertoire **ou** /tmp est un répertoire :

```
# [ -w a100 -a \( -d /usr -o -d /tmp \) ]
# echo $?
1
# cd formation
# [ -w a100 -a \( -d /usr -o -d /tmp \) ]
# echo $?
0
```

Tests d'environnement utilisateur

Test	Description
-o option	Retourne vrai si l'option du shell “option” est activée

Exemples

```
# [ -o allexport ]
# echo $?
1
# [ -o interactive-comments ]
# echo $?
0
```

La commande [[expression]]

La commande **[[Espace expression Espace]]** est une amélioration de la commande **test**. Les opérateurs de la commande test sont compatibles avec

la commande [[expression]] sauf **-a** et **-o** qui sont remplacés par **&&** et **||** respectivement :

Test	**Description**
!expression	Retourne vrai si expression est fausse
expression1 && expression2	Représente un et logique entre expression1 et expression2
expression1 || expression2	Représente un ou logique entre expression1 et expression2
(expression)	Les parenthèses permettent de regrouper des expressions

D'autres opérateurs ont été ajoutés :

Test	**Description**
chaine = modele	Retourne vrai si chaîne correspond au modèle
chaine != modele	Retourne vrai si chaîne ne correspond pas au modèle
chaine1 < chaine2	Retourne vrai si chaîne1 est lexicographiquement avant chaîne2
chaine1 > chaine2	Retourne vrai si chaîne1 est lexicographiquement après chaîne2

Exemples

Testez si l'utilisateur peut écrire dans le fichier a100 **et** /usr est un répertoire **ou** /tmp est un répertoire :

```
# [[ -w a100 && ( -d /usr || -d /tmp ) ]]
# echo $?
0
```

Opérateurs du shell

Opérateur	**Description**
Commande1 && Commande2	Commande 2 est exécutée si la première commande renvoie un code vrai
Commande1 || Commande2	Commande 2 est exécutée si la première commande renvoie un code faux

Exemples :

```
# [[ -d /tmp ]] && echo "Répertoire tmp existe"
Répertoire tmp existe
# [[ -d /tmp ]] || echo "Répertoire tmp existe"

#
```

L'arithmétique

La commande expr

La commande **expr** prend la forme :

expr Espace nombre1 Espace *opérateur* Espace nombre2 Entrée

ou

expr Tab nombre1 Tab *opérateur* Tab nombre2 Entrée

ou

expr Espace chaîne Espace : Espace *expression_régulière* Entrée

ou

expr Tab chaîne Tab : Tab *expression_régulière* Entrée

Opérateurs Arithmétiques

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication

Opérateur	Description
/	Division
%	Modulo
\(\)	Parenthèses

Opérateurs de Comparaison

Opérateur	Description
\<	Inférieur
\<=	Inférieur ou égal
\>	Supérieur
\>=	Supérieur ou égal
=	égal
!=	inégal

Opérateurs Logiques

Opérateur	Description
|	ou logique
\&	et logique

Exemples

Ajoutez 2 à la valeur de $x :

```
$ x=2
$ expr $x + 2
4
```

Si les espaces sont retirés, le résultat est tout autre :

```
$ expr $x+2
```

```
2+2
```

Les opérateurs doivent être protégés :

```
$ expr $x * 2
expr: syntax error
$ expr $x \* 2
4
```

Mettez le résultat d'un calcul dans un variable :

```
$ resultat=`expr $x + 10`
$ echo $resultat
12
```

La commande let

La commande let est l'équivalent de la commande ((expression)). La commande ((expression)) est une amélioration de la commande **expr** :

- plus grand nombre d'opérateurs
- pas besoin d'espaces ou de tabulations entre les arguments
- pas besoin de préfixer les variables d'un $
- les caractères spéciaux du shell n'ont pas besoin d'être protégés
- les affectations se font dans la commande
- exécution plus rapide

Opérateurs Arithmétiques

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication

Opérateur	Description
/	Division
%	Modulo
*	Puissance

Opérateurs de comparaison

Opérateur	Description
<	Inférieur
<=	Inférieur ou égal
>	Supérieur
>=	Supérieur ou égal
==	égal
!=	inégal

Opérateurs Logiques

Opérateur	Description
&&	et logique
||	ou logique
!	négation logique

Opérateurs travaillant sur les bits

Opérateur	Description
~	négation binaire
>>	décalage binaire à droite
<<	décalage binaire à gauche
&	et binaire
|	ou binaire
^	ou exclusif binaire

Exemples

```
$ x=2
$ ((x=$x+10))
$ echo $x
12
$ ((x=x+20))
$ echo $x
32
```

Structures de contrôle

If

La syntaxe de la commande If est la suivante :

```
if condition
then
    commande(s)
else
    commande(s)
fi
```

ou :

```
if condition
then
    commande(s)
    commande(s)
fi
```

ou encore :

```
if condition
then
     commande(s)
elif condition
then
     commande(s)
elif condition
then
     commande(s)
else
    commande(s)

fi
```

case

La syntaxe de la commande case est la suivante :

```
case $variable in
modele1) commande
   ...
   ;;
modele2) commande
   ...
   ;;
modele3 | modele4 | modele5 ) commande
   ...
   ;;
esac
```

Boucles

for

La syntaxe de la commande for est la suivante :

```
for variable in liste_variables
do
    commande(s)
done
```

while

La syntaxe de la commande while est la suivante :

```
while condition
do
    commande(s)
done
```

Scripts de Démarrage

Lors de chaque connexion au système, un script de démarrage est exécuté automatiquement. Ce script se trouve dans le répertoire personnel de l'utilisateur et porte un nom différent selon le shell utilisé:

- **.profile** pour le ksh
- **.bash_profile** pour le bash

Par exemple :

```
$ ls -a
total 16
drwxr-xr-x   2 test     other        512 août 15 11:50 .
drwxr-xr-x   4 root     root         512 août 15 10:19 ..
-rw-------   1 test     other         17 août 15 11:50 .bash_history
-rw-r--r--   1 test     other        144 août 15 10:19 .profile
-rw-------   1 test     other        200 août 15 12:16 .sh_history
-rw-r--r--   1 test     other        136 août 15 10:19 local.cshrc
-rw-r--r--   1 test     other        157 août 15 10:19 local.login
-rw-r--r--   1 test     other        174 août 15 10:19 local.profile
```

L'étude du fichier **.profile** pour votre utilisateur démontrera un fichier similaire à celui-ci :

```
$ cat .profile
#       This is the default standard profile provided to a user.
#       They are expected to edit it to meet their own needs.

MAIL=/usr/mail/${LOGNAME:?}
```

Dans ce fichier nous pouvons noter la définition de la variable **MAIL**.
