Le shell UNIX

Table des matières

Un shell Unix est un interpréteur de commandes destiné aux systèmes d'exploitation de type Unix qui permet d'accéder aux fonctionnalités internes du système d'exploitation. Il se présente sous la forme d'une interface en ligne de commande accessible depuis la console ou un terminal.

1. Premiers pas

1.1. Qu'est-ce qu'un shell ?

Un shell[1] est une couche logicielle qui fournit l'interface utilisateur d'un système d'exploitation. Il correspond à la couche la plus externe de ce dernier. L'interface système est utilisé comme diminutif de l'interface utilisateur du système d'exploitation.

Il existe deux environnements très différents sous Unix :

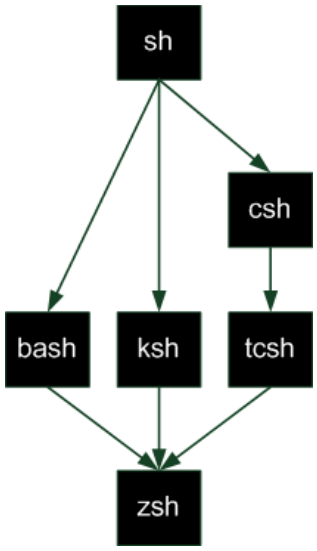
- l'environnement console
- l'environnement graphique

Et il existe plusieurs environnements console : les shells.

La console a toujours un fond noir et un texte blanc. En revanche, les fonctionnalités offertes par l'invite de commandes peuvent varier en fonction du shell que l'on utilise.

Voici les noms de quelques-uns des principaux shells :

- sh : Bourne Shell. L'ancêtre de tous les shells.
- bash : Bourne Again Shell. Une amélioration du Bourne Shell, disponible par défaut sous GNU/Linux et Mac OS X.
- ksh : Korn Shell. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
- csh : C Shell. Un shell utilisant une syntaxe proche du langage C.
- tcsh : Tenex C Shell. Amélioration du C Shell.
- zsh : Z Shell. Shell assez récent reprenant les meilleures idées de bash, ksh et tcsh.



Le bash est le shell par défaut. Cependant, sh reste toujours plus répandu que bash.

Beaucoup de solutions sont données en ligne de commande, non pas qu'Unix n'ait pas d'interface graphique, mais pour certaines tâches, l'utilisation de la ligne de commande s'avère bien plus pratique et plus puissante que la souris.

1.2. Ecrire un script

Un script shell permet d'automatiser une série d'opérations. Il se présente sous la forme d'un fichier contenant une ou plusieurs commandes qui seront exécutées de manière séquentielle.

Si vous voulez écrire un programme sh, vous avez deux possibilités :

- soit vous tapez dans un shell toutes les commandes
- ou alors vous rassemblez toutes les instructions dans un fichier .sh

Pour créer un script, ouvrez l'éditeur Vi en mode console en lui donnant le nom du nouveau fichier à

1 interface système

créer :

```
$ vi test.sh
```

Si test.sh n'existe pas, il sera créé automatiquement.

La première chose à faire dans un script shell est d'indiquer quel shell est utilisé. Pour utiliser la syntaxe de bash, plus complet que sh, il faut indiquer où se trouve le programme bash :

```
#!/bin/bash
```

Le #! est appelé le sha-bang[2].

NB : /bin/bash peut être remplacé par /bin/sh pour coder avec sh, /bin/ksh avec ksh, etc.

Bien que non indispensable, cette ligne permet de s'assurer que le script est bien exécuté avec le bon shell. En l'absence de cette ligne, c'est le shell de l'utilisateur qui sera chargé.

Voici le contenu de test.sh que nous écrirons :

```
#!/bin/bash
# indique au système que l'argument qui suit est le programme utilisé pour exécuter ce fichier.
# En cas général les "#" servent à faire des commentaires comme ici
echo Liste des fichiers :
ls -la
```

Enregistrez le fichier et fermez Vi en tapant :wq (write quit) ou encore :x.

Vous retrouvez alors l'invite de commandes.

1.3. Exécuter un script

Il suffit de se placer dans le dossier où est le script, et de lancer :

```
bash test.sh
```

Pour l'exécuter grâce au raccourci « ./ », il faut le rendre exécutable avec chmod. Pour ceci tapez la commande qui suit :

```
chmod +x test.sh
```

Puis vous pouvez exécuter le script :

```
./test.sh
```

Actuellement, le script doit être lancé via ./test.sh et il faut être dans le bon répertoire.

Pour pouvoir être exécutés depuis n'importe quel répertoire sans « ./ », il faut ajouter un répertoire au "PATH". Quand on tape une commande ("ls" par exemple), le shell regarde dans le PATH qui lui indique où chercher le code de la commande.

Pour voir à quoi ressemble votre PATH, tapez dans la console :

```
echo $PATH
```

Cette commande chez moi donnait initialement :

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

C'est à dire que le shell va aller voir si la définition de la commande tapée ("ls" pour continuer sur le même exemple) se trouve dans /usr/local/bin puis dans /usr/bin… jusqu'à ce qu'il la trouve.

2 ou shebang : en-tête d'un fichier texte qui indique au système d'exploitation que ce fichier n'est pas un binaire

Ajouter un répertoire au PATH peut donc être très pratique. Par convention, ce répertoire s'appelle bin et se place dans votre répertoire personnel. Si votre répertoire personnel est /home/toto, ce répertoire sera donc /home/toto/bin. Pour pouvoir utiliser mes scripts en tapant directement leur nom (sans le "./") depuis n'importe quel répertoire de mon ordinateur, il suffit d'indiquer au shell de chercher aussi dans ce nouveau dossier en l'ajoutant au PATH :

```
export PATH=$PATH:$HOME/bin
```

La commande :

```
echo $PATH
```

retourne maintenant :

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/toto/bin
```

et on peut lancer le script appelé "test.sh" situé dans "/home/toto/bin" en tapant directement :

```
test.sh
```

Cette procédure est pour une modification temporaire du PATH et qui sera donc effacée à la fin de la session. Pour rendre la modification permanente, ajouter la commande dans le fichier texte caché .bashrc se trouvant dans votre dossier personnel ainsi que dans le dossier /root.

Remarque : pour lancer le script en mode débogage, il faut ajouter le paramètre -x à la suite de la commande du shell.

```
bash -x test.sh
```

Le shell affiche alors le détail de l'exécution du script, ce qui peut aider à retrouver la cause des erreurs.

Le paramètre -u montrera les variables qui n'ont pas été utilisées pendant l'exécution du programme.

2. Les variables

2.1. Déclarer et afficher une variable

Pour affecter une valeur à une variable, il suffit de donner un nom à une variable et de l'initialiser avec le symbole égal « = » :

```
message=Salut
```

NB : ne pas mettre d'espaces autour du symbole égal « = ».

Il est possible d'utiliser des quotes pour délimiter un texte contenant des espaces. Il existe trois types de quotes :

- les apostrophes ' ' (simples quotes)
- les guillemets " " (doubles quotes)
- les accents graves ` ` (back quotes), qui s'insèrent avec Alt Gr + 7 sur un clavier AZERTY français

```
message="Salut tout le monde"
```

Pour insérer un apostrophe (ou une apostrophe) dans la valeur de la variable, il faut la faire précéder d'un antislash \. Ce caractère d'échappement permet également :

- d'afficher un saut de ligne : \n
- d'afficher une tabulation : \t

Pour afficher la valeur d'une variable, il suffit de la faire précéder d'un $:

```
echo $message
```

Remarque : selon le type de quotes utilisé, la réaction de bash ne sera pas la même.

Avec de simples quotes, la variable n'est pas analysée et le $ est affiché tel quel.

```
message='Bonjour tout le monde'
echo 'Le message est : $message'
```

donnera :

```
Le message est : $message
```

Avec des doubles quotes la variable est analysée et son contenu affiché :

```
Le message est : Bonjour tout le monde
```

Les back quotes demandent à bash d'exécuter ce qui se trouve à l'intérieur :

```
message=`pwd`
echo "Vous êtes dans le dossier $message"
```

donnera :

```
Vous êtes dans le dossier /home/toto/bin
```

La commande pwd a été exécutée et son contenu inséré dans la variable message.

2.2. Demander une saisie

Vous pouvez demander à l'utilisateur de saisir du texte avec la commande read. Ce texte sera immédiatement stocké dans une variable.

On peut demander de saisir autant de variables d'affilée que l'on souhaite :

```
#!/bin/bash
read prenom nom
echo "Bonjour $prenom $nom !"
```

donnera :

```
John Doe
Bonjour John Doe !
```

read lit ce que vous tapez mot par mot (en considérant que les mots sont séparés par des espaces). Il assigne chaque mot à une variable différente, d'où le fait que le nom et le prénom ont été correctement et respectivement assignés à $nom et $prenom.

Remarque : si vous rentrez plus de mots au clavier que vous n'avez prévu de variables pour en stocker, la dernière variable de la liste récupérera tous les mots restants. En clair, si j'avais tapé pour le programme précédent « John Doe Wayne », la variable $prenom aurait eu pour valeur « Doe Wayne ».

Pour que l'utilisateur sache quoi faire, utilisez l'option -p de read :

```
#!/bin/bash
```

```
read -p 'Entrez votre nom : ' nom
echo "Bonjour $nom !"
```

Remarque : le message 'Entrez votre nom' a été entouré de quotes. Si on ne l'avait pas fait, le bash aurait considéré que chaque mot était un paramètre différent !

Pour limiter le nombre de caractères, utilisez l'option -n :

```
#!/bin/bash
read -p 'Entrez votre login (5 caractères max) : ' -n 5 nom
echo "\nBonjour $nom !"
```

Bash coupera automatiquement au bout de 5 caractères sans avoir besoin d'appuyer sur la touche Entrée. Pour que le message ne s'affiche pas sur la même ligne, on affiche avec \n.

Pour limiter le temps autorisé pour saisir un message, on peut définir un timeout avec -t en secondes au bout duquel read s'arrêtera :

```
#!/bin/bash
read -p 'Entrez le code de désamorçage de la bombe (vous avez 5 secondes) : '  -t 5 code
echo -e "\nBoum !"
```

Pour ne pas afficher le texte saisi, utilisez l'option -s :

```
#!/bin/bash
read -p 'Entrez votre mot de passe : ' -s pass
echo -e "\nMerci !"
```

2.3. Opérations mathématiques

En bash, les variables sont toutes des chaînes de caractères. En soi, le bash n'est pas vraiment capable de manipuler des nombres ; il n'est donc pas capable d'effectuer des opérations.

Il faudra utiliser la commande let :

```
let "a = 5"
let "b = 2"
let "c = a + b"
```

Ou l'instruction : c=$(($a+$b))

Les opérations utilisables sont :

- l'addition : +
- la soustraction : -
- a multiplication : *
- la division entière : /
- la puissance : **
- le modulo (renvoie le reste de la division entière) : %

Exemples :

```
let "a = 5 * 3"          # $a = 15
let "a = 4 ** 2"         # $a = 16 (4 au carré)
let "a = 8 / 2"          # $a = 4
```

```
let "a = 10 / 3"          # $a = 3
let "a = 10 % 3"          # $a = 1
```

2.4. Les variables d'environnement

Les variables d'environnement sont des variables que l'on peut utiliser dans n'importe quel programme. On parle aussi parfois de variables globales. Vous pouvez afficher toutes celles que vous avez actuellement en mémoire avec la commande env :

```
$ env
TERM=xterm
SHELL=/bin/bash
USER=toto
PATH=/home/toto/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin: /usr/bin:/sbin:/bin:/usr/games
PWD=/home/toto/bin
EDITOR=vi
HOME=/home/toto
OLDPWD=/home/toto
[ ... ]
```

- SHELL : indique quel type de shell est en cours d'utilisation (sh, bash, ksh…) ;
- PATH : une liste des répertoires qui contiennent des exécutables que vous pouvez lancer sans indiquer leur répertoire.
- EDITOR : l'éditeur de texte par défaut qui s'ouvre lorsque cela est nécessaire.
- HOME : la position du dossier home.
- PWD : le dossier dans lequel vous vous trouvez.
- OLDPWD : le dossier dans lequel vous vous trouviez auparavant.
- USER contient le login d'utilisateur
- HOSTNAME contient le nom de la machine
- HISTSIZE contient la taille maximale des commandes exécutées contenues dans le fichier historique
- PS1 contient les paramètres d'affichage de l'invite de commande (le prompt)

Remarque : les noms de ces variables sont, par convention, écrits en majuscules.

Pour utiliser ces variables, il suffit de les appeler par leur nom :

```
#!/bin/bash
echo "Votre éditeur par défaut est $EDITOR"
```

Pour définir votre propre variable d'environnement, utilisez la commande export dans votre .bashrc.

2.5. Les variables des paramètres

Il existe des variables spéciales :

Nom	**fonction**
$*	contient tous les arguments passés à la fonction

$#	contient le nombre d'argument
$?	contient le code de retour de la dernière opération
$0	contient le 1er paramètre (nom du script)
$1	contient le 2ème paramètre
...	
$n	contient l'argument n, n étant un nombre
$!	contient le PID de la dernière commande lancée

Exemple : créer le fichier arg.sh avec le contenu qui suit :

```
#!/bin/bash
echo "Nombre d'argument "$#
echo "Les arguments sont "$*
echo "Le second argument est "$2
echo "Et le code de retour du dernier echo est "$?
```

Lancez ce script avec un ou plusieurs arguments et vous aurez :

```
./arg.sh 1 2 3
Nombre d'argument 3
Les arguments sont 1 2 3
Le second argument est 2
Et le code de retour du dernier echo est 0
```

2.6. Les tableaux

Le bash gère également les variables « tableaux ». Un tableau est défini de la façon suivante :

```
tableau=('valeur0' 'valeur1' 'valeur2')
```

Pour accéder à une cellule du tableau, il faut utiliser la syntaxe suivante :

```
${tableau[2]}
```

On peut aussi définir le contenu du tableau élément par élément:

```
#!/bin/bash
tableau[0]='valeur0'
tableau[1]='valeur1'
tableau[2]='valeur2'
echo ${tableau[1]}
```

L'index d'un tableau comme à partir de 0.

- Pour compter le nombre d'éléments du tableau : len=${#tableau[*]}
- Pour afficher un élément : echo ${tableau[1]}
- Pour afficher tous les éléments : echo ${tableau[*]}

3. Les conditions

Une condition peut être écritebash sous différentes formes. On parle de structures conditionnelles.

3.1. if… else

Les symboles suivant permettent de faire des comparaisons simples ou multiples pour les conditions.

Symbole	Signification
=	Vérifie si les deux chaînes sont identiques. Notez que bash est sensible à la casse : « b » est donc différent de « B ». Il est aussi possible d'écrire « == ».
!=	Vérifie si les deux chaînes sont différentes.
-z	Vérifie si la chaîne est vide.
-n	Vérifie si la chaîne est non vide.
-eq	Vérifie si les nombres sont égaux (**eq***ual*). À ne pas confondre avec le « = » qui, lui, compare deux chaînes de caractères.
-ne	Vérifie si les nombres sont différents (**n***on***e***qual*). Encore une fois, ne confondez pas avec « != » qui est censé être utilisé sur des chaînes de caractères.
-lt	Vérifie si num1 est inférieur (<) à num2 (**l***ower***t***han*).
-le	Vérifie si num1 est inférieur ou égal (<=) à num2 (**l***owero***e***qual*).
-gt	Vérifie si num1 est supérieur (>) à num2 (**g***reater***t***han*).
-ge	Vérifie si num1 est supérieur ou égal (>=) à num2 (**g***reatero***e***qual*).

Symbole	Signification
&&	ET logique
||	OU logique
!	NON logique

Pour introduire une condition :

1. on utilise le mot if, qui en anglais signifie « si ».
2. on ajoute à la suite entre crochets [] la condition en elle-même.
3. on utilise le mot then, suivi des instructions à exécuter si la condition est **vraie**.
4. **Facultativement** : on utilise le mot else, qui en anglais signifie « sinon », suivi des des instructions à exécuter si la condition est **fausse**.
5. on termine le test avec le mot fi.

```
#!/bin/bash
age=8                          # initialisation de la variable

if [ age -lt 12 ] then                  # c'est très important d'avoir des espaces entre les crochets
        echo "Salut gamin !"            # action si condition vraie
else    # SINON
        …                               # action si condition fausse
```

```
fi
```

Dans le cas des conditions multiples, on utilise les opérateurs logiques.

```
#!/bin/bash
age=8                                   # initialisation de la variable

if [ age -lt 12] && [ $1 = "garcon" ] then      # si j'ai moins de 12 ans et que je suis un garçon
        echo "Bonjour Jeune homme"              # action si condition vraie
elif [ age < 12 ] && [ $1 != "garcon" ]         # si j'ai moins de 12 ans et que je suis pas un garçon
        echo "Bonjour Mademoiselle"
fi
```

Le script « test.sh » affiche Jeune homme ou Mademoiselle en fonction du paramètre passé.

```
./test.sh garcon
```

Le mot clé elif est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si ».

L'avantages de bash est depouvoir facilement faire des tests sur des fichiers :

Condition	**Signification**
-e $nomfichier	Vérifie si le fichier existe.
-d $nomfichier	Vérifie si le fichier est un répertoire.
-f $nomfichier	Vérifie si le fichier est un… fichier.
-L $nomfichier	Vérifie si le fichier est un lien symbolique (raccourci).
-r $nomfichier	Vérifie si le fichier est lisible (r).
-w $nomfichier	Vérifie si le fichier est modifiable (w).
-x $nomfichier	Vérifie si le fichier est exécutable (x).
$fichier1 -nt $fichier2	Vérifie si fichier1 est plus récent que fichier2 (newerthan).
$fichier1 -ot $fichier	Vérifie si fichier1 est plus vieux que fichier2 (olderthan).

```
#!/bin/bash
fichier="/home/toto/essai.sh"                   # initialisation de la variable

if [ ! -e $fichier ]
then
      echo "Le fichier $fichier n'existe pas"
fi
```

3.2. case... in

Les structures à base de if… else suffisent pour traiter n'importe quelle condition.

Cependant, pour une imbrication de if... else trop importante, il existe une autre structure plus souple : case... in.

```
#!/bin/bash
```

```
case $1 in        # on teste la valeur passée en paramètre au script (n jour de la semaine)
        "lundi")        # dans le cas où c'est le début de la semaine
        "mardi")        # on peut tester deux valeurs à la suite
                echo "courage !!!"
                ;;                # ne pas oublier ;; sinon bash continue

        "mercredi")     # dans le cas où c'est le début de la semaine
                echo "c'est le jour des enfants";
                ;;

        "jeudi")        #  dans le cas où c'est la fin de la semaine
        "vendredi")
                echo "bientôt le we !";
                ;;

        *)      # il faut traiter les autres jours (cas par défaut)
                echo "vive le week end !";
                ;;
esac
```

La valeur * est le traitement par défaut quelle que soit la valeur de la variable.

Bash traite les instructions in à la suite. Pour interrompre le traitement, il faut utiliser ;;.

Voici une autre syntaxe qui peut être utlisée avec case..in :

```
#!/bin/sh
while [ 1 ]
do
        echo -n "Etes-vous fatigué ? "
        read on
        case "$on" in
                oui | o | O | Oui | OUI ) echo "Allez faire du café !";;
                non | n | N | Non | NON ) echo "Programmez !";;
                quit ) echo "Au revoir !" break;;
                * ) echo "Ah bon ?";;
        esac
done
```

Ce script boucle tant que le mot quit n'a pas été entré au clavier.

4. Les boucles

4.1. Tant que : while

Une boucle permet de répéter des instructions plusieurs fois.

- les instructions sont d'abord exécutées dans l'ordre, de haut en bas
- à la fin des instructions, on retourne à la première
- et ainsi de suite...

Tant que la condition est remplie, les instructions sont réexécutées. Dès que la condition n'est plus remplie, on sort de la boucle.

```
#!/bin/bash
lignes=1

# on boucle tant que on n'est pas arrivé à 100 lignes
while [ $lignes -le 100 ]
do
        echo "ligne n° $lignes\n"             # affiche le n° de ligne
        let "lignes = lignes + 1"             # incrément du nombre de ligne
done
```

Il faut TOUJOURS s'assurer que la condition sera fausse au moins une fois. Si elle ne l'est jamais, alors la boucle s'exécutera à l'infini !

5.2. Répéter : until... do

Les boucles until-do ressemblent beaucoup aux boucles while, mais en inversant. C'est-à-dire qu'elle exécute le bloc jusqu'à ce que la condition soit vraie, donc elle s'emploie exactement comme la commande `while`.

```
#!/bin/bash
lignes=1

# on boucle tant que on n'est pas arrivé à 100 lignes
until [ $lignes -eq 100 ]
do
        echo "ligne n° $lignes\n"             # affiche le n° de ligne
        let "lignes = lignes + 1"             # incrément du nombre de ligne
done
```

5.3. Boucle : for

La boucle for permet de parcourir une liste de valeurs et de boucler autant de fois qu'il y a de valeurs.

```
#!/bin/bash

for fichier in `ls`
do
        echo "$fichier\n"             # affiche le fichier
        mv $fichier $fichier.old      # renomme le fichier
done
```

Toutefois, on peut utiliser la boucle for pour parcourir une liste de nombres grâce à la commande seq.

seq initialisation incrémentation limite

1. initialisation. C'est la valeur que l'on donne au départ à la variable.
2. incrémentation, qui vous met à jour la variable à chaque tour de boucle.

3. limite. Tant que la limite n'est pas atteinte, la boucle est réexécutée.

```
#!/bin/bash

for lignes in `seq 1 1 100`
do
        echo "ligne n° $lignes\n"             # affiche le n° de ligne
done
```

Cette boucle est utilisée lorsque l'on connait à l'avance le nombre d'instructions à répéter.

6. Les fonctions

Les fonctions sont indispensables pour bien structurer un programme mais aussi pouvoir le simplifier, créer une tâche, la rappeler… Voici la syntaxe générale de 'déclaration' d'une fonction :

```
nom_fonction(){
        # instructions
}
```

Cette partie ne fait rien en elle même, elle dit juste que quand on appellera nom_fonction, elle fera instruction.

Exemple :

```
#!/bin/sh

# Definition de ma fonction
mafonction(){
        echo 'La liste des fichiers de ce répertoire'
        ls -l
}
#fin de la définition de ma fonction

echo 'Vous allez voir la liste des fichiers de ce répertoire:'
mafonction      #appel de ma fonction
exit 0
```

Les fonctions peuvent être définies n'importe ou dans le code du moment qu'elle sont définies avant d'être utilisées. Même si en bash les variables sont globales, il est possible de les déclarer comme locales au sein d'un fonction en la précédant du mot clé local: local ma_fonction

Exemple: Un sleep interactif.

```
#!/bin/bash
function info(){
        echo -e "$1\nBye"
        exit
}

test -z "$1" && info "requiert 1 argument pour le temps d'attente..." || PRINT=$(($1*500))
test -z $(echo "$1" | grep -e "^[0-9]*$") && info "'$1' est un mauvais argument"
test $1 -gt 0 || info "Je ne prends que les entiers > 0"
```

```
function print_until_sleep(){
        local COUNT=0
        while [ -d /proc/$1 ]
        do
                # équivalent à [ ($COUNT % $2) -eq 0 ] && [echo -n "*"]
                test $(($COUNT % $2)) -eq 0 && echo -n "*"

                # équivalent à let COUNT = COUNT + 1
                COUNT=$(($COUNT+1))
        done
}

sleep $1 & print_until_sleep $! $PRINT
echo -e "\nBye"
```

7. Les commandes de base

cat	Lit (concatène) un ou plusieurs fichier(s), affichage sur la sortie standard
cd	ChangeDirectory, change de répertoire
chmod	CHangeMODe - change le mode d'accès (permissions d'accès) d'un ou plusieurs fichier(s)
chown	CHangeOWNer - change le propriétaire d'un ou de plusieurs fichier(s)
cp	copier des fichiers
crontab	planification de tâches
cut	Retire des parties précises de texte dans chaque ligne d'un fichier
date	Affiche la date selon le format demandé
dd	DevicetoDevice - Recopie octet par octet tout ou partie du contenu d'un périphérique (habituellement de stockage) vers un autre péripherique.
df	affichage de la quantité d'espace libre disponible sur tous les systèmes de fichiers
du	DiksUsage - l'utilisation de disque
echo	Affiche du texte sur la sortie standard (à l'écran)
exit	arrête l'exécution du shell
find	recherche de fichiers
fsck	FileSystemChecK - vérification d'intégralité de système de fichiers
grep	recherche dans un ou plusieurs fichiers les lignes qui correspondent à un motif

groupadd	Ajouter un groupe d'utilisateurs
gunzip	décompression de fichiers
gzip	compression de fichiers
head	affiche les premières lignes (par défaut 10) d'un fichier
help	affiche une aide sur les commandes internes de bash
kill	envoyer un signal à un processus
less	programme d'affichage à l'écran
ln	création de liens
ls	liste le contenu des répertoires
man	affiche les pages de manuel
mkdir	MaKeDIRectory - crée un répertoire
mkfs	MaKeFileSystem - création de systèmes de fichiers
more	programme d'affichage à l'écran
mount	monter un système de fichiers
mv	déplacer, renommer un fichier
ps	affiche les processus en cours d'exécution
pwd	Print name of current/working directory - affiche le chemin complet du repertoire courant
rm	suppression de fichiers
rmdir	Remove empty directories - suppression d'un dossier vide
tail	affiche les 10 dernières lignes d'un fichier
tar	création d'archives
su	Substitute User identity ou Switch User - prendre l'identité d'un utilisateur
uname	Affiche des informations sur le système.
useradd	ajouter un utilisateur
whereis	localiser une commande

8. La racine

Dans les systèmes de la famille Unix, la racine représente le sommet de l'arborescence des répertoires.

Elle est représentée par le caractère / (slash) et signifie "root"[3].

Tous les répertoires de votre système sont liés à la racine de façon directe ou indirecte.

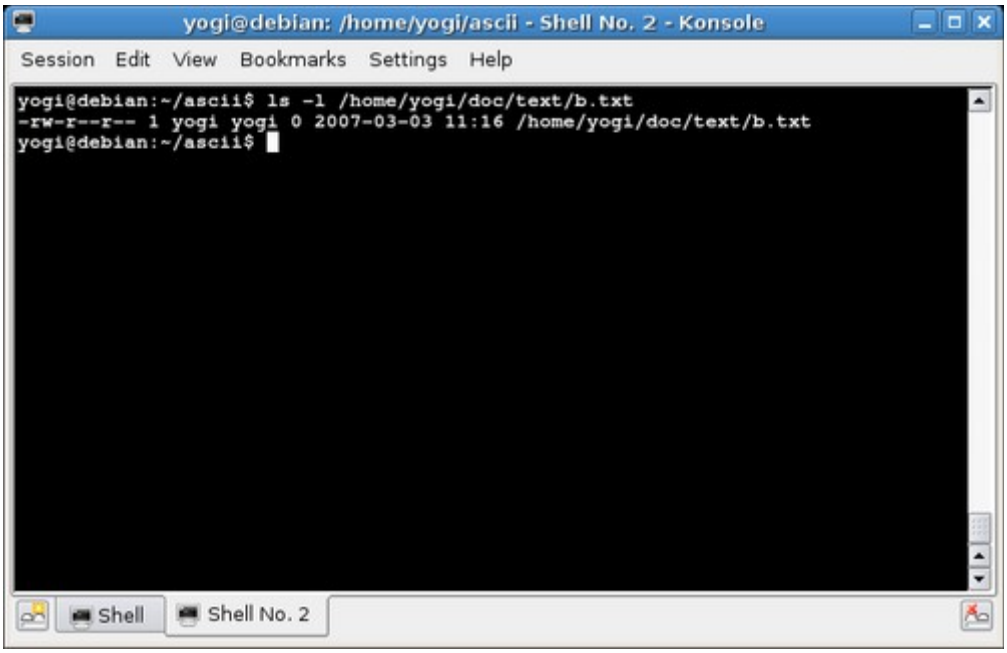
- . indique le répertoire courant
- .. indique le répertoire parent

Une chose très importante à savoir quand on est connecté dans un shell, c'est de savoir où on se trouve dans l'arborescence :

La commande pwd (PrintWorkingDirectory) affiche votre localisation dans l'arborescence.

Le chemin absolu représente l'arborescence complète de fichiers, en partant de la racine.

Exemple :



- Le fichier b.txt se trouve dans /home/user/doc/text
- Vous vous trouvez dans /home/user/ascii
- Le chemin absolu vers b.txt est /home/user/doc/text/b.txt

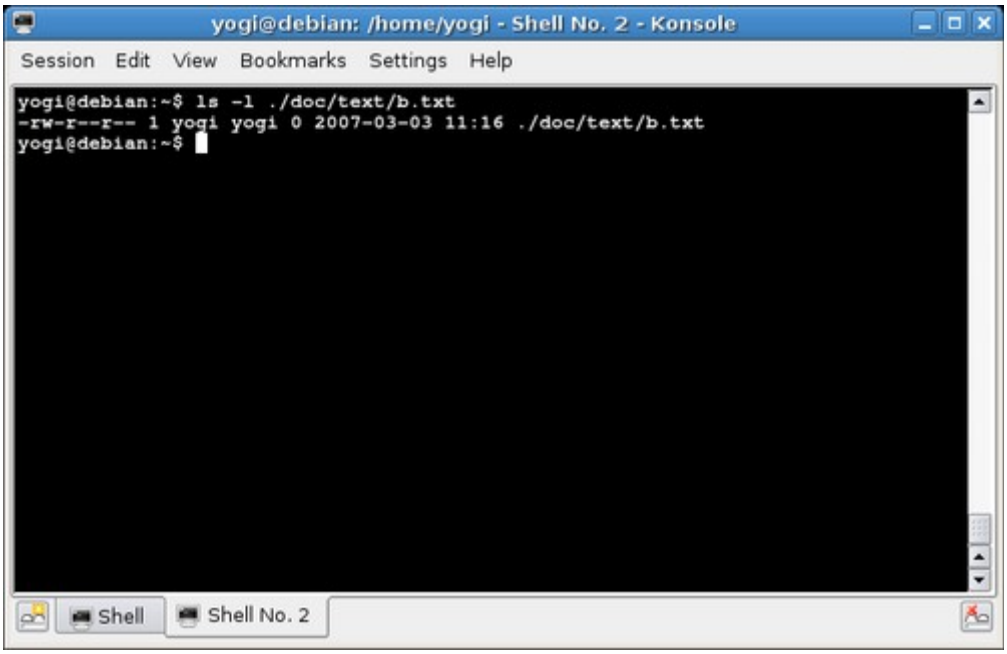
Quelque soit votre localisation dans l'arborescence l'utilisation du chemin absolu est le moyen le plus sûr pour accéder au fichier désiré.

Le chemin relatif pour accéder à un fichier c'est l'arborescence rapporté à votre localisation dans le shell. On utilise les notations . et / ou ..

- . nous permet de descendre dans l'arborescence du répertoire courant
- .. nous permet dans un 1er temps de monter en arborescence dans le but d'atteindre d'autres répertoires

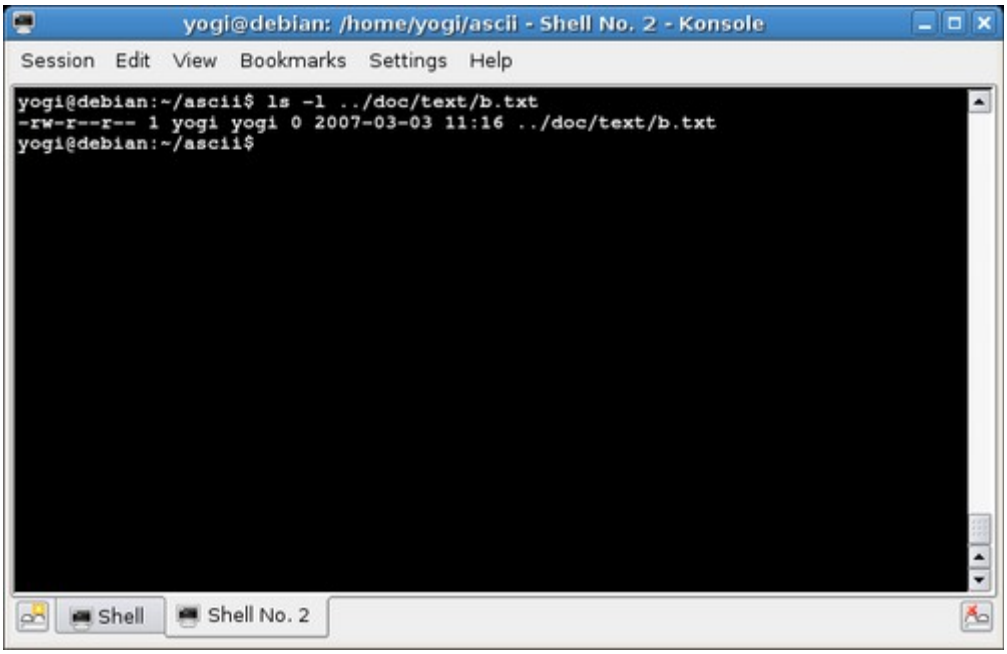
3 racine en français

Exemple : le répertoire courant .



- Le fichier b.txt se trouve dans /home/user/doc/text
- Vous vous trouvez dans /home/user
- Le chemin relatif vers b.txt est ./doc/text/b.txt

Exemple : le répertoire parent ..



- Le fichier b.txt se trouve dans /home/user/doc/text
- Vous vous trouvez dans /home/user/ascii
- Le chemin relatif vers b.txt est ../doc/text/b.txt

Le répertoire ascii se trouve dans /home/yogi donc en écrivant .. je vais utiliser le répertoire

parent /home/yogi comme point de départ puis ensuite je vais dans doc/text (remarqué que je n'ai pas dit /doc/text - qui aurait lié doc de la racine /)

Pour ce déplacer dans l'arborescence utiliser la commande cd :

```
cd /chemin/vers/répertoire
```

Avec pwd vous pouvez vérifier votre nouvelle localisation dans la racine.

9. Les redirections et les pipelines

D'abord on va commencer avec une petite explication concernant les descripteurs des "entrées - sorties" :

- tout ce que vous écrivez dans le shell s'appelle STDIN (StandarDINput)
- tout ce que vous voyez à l'écran peut être :

  - STDOUT (STandarDOUTput)

  - STDERR (STandarDERRor)

Ces descripteurs sont numérotés comme suit :

0: entrée standard (STDIN) ← clavier

1: sortie standard (STDOUT) → écran

2: sortie erreurs (STDERR) → écran

9.1. Les redirections

Une redirection est la possibilité de diriger le résultat d'une commande en utilisant d'autres destinations que les descripteurs standards.

Pour réaliser une redirection on utilise :

commande > fichier - redirection en mode écriture vers le fichier

le fichier est créé s'il n'existe pas

son contenu sera remplacé par le nouveau si le fichier existe déjà

commande >> fichier - redirection en mode ajout vers le fichier

le fichier est créé s'il n'existe pas

le résultat sera ajouté à la fin de fichier

commande < fichier - la commande lit depuis le fichier

Exemples de redirections :

1. envoyer le contenu de fichier1 dans le fichier2

```
$ cat fichier1 > fichier2
```

si le fichier2 existe son contenu d'origine sera supprimé, le fichier2 est créé s'il n'existe pas

2. envoyer le contenu de fichier1 dans le fichier2 - mode ajout

```
$ cat fichier1 >> fichier2
```

si le fichier2 existe, le contenu du fichier1 est ajouté à la fin de fichier2, si le fichier2 n'existe

pas, il sera créé

3. recherche dans la racine le fichier appelé fichier.txt, les erreurs au lieu d'être envoyées sur STDERR (à l'écran) sont envoyées dans /dev/null (sorte de poubelle sans fin)

```
$ find / -name 'fichier.txt' 2>/dev/null
```

4. recherche dans la racine le fichier appellé fichier.txt, les erreurs au lieu d'être envoyées sur STDERR (à l'écran) sont envoyées dans le fichiers erreur.txt

```
$ find / -name 'fichier.txt' 2>erreur.txt
```

9.2. Les pipelines

Les shell des systèmes d'exploitation de type Unix disposent d'un mécanisme appelé pipeline ou pipe[4]. Ce mécanisme permet de chaîner des processus de sorte que la sortie d'un processus (stdout) alimente directement l'entrée (stdin) du suivant. Chaque connexion est implantée par un tube anonyme.

commande1 | commande2	le résultat de la commande1 est utilisé par la commande2
commande1 & commande2	les commandes sont exécutées simultanément, commande1 s'exécutant en arrière-plan
commande1 && commande2	si la commande1 réussi la commande2 est executée
commande1 || commande2	la commande2 s'exécute seulement si la commande1 échoue
commande1;commande2	les commandes sont exécutées dans l'ordre

Exemple de pipelines :

1. le tube | (pipe)

```
$ perl -ne 'print unless /^\s*$/' guideshell | wc -l
```

Dans un premier temps on exécute perl -ne 'print unless /^\s*$/' guideshell dans le but d'afficher le fichier à l'écran, les lignes vides étant eliminées.

Au lieu d'afficher à l'écran on utilise | pour passer le résultat à la commande wc qui va compter le nombre de lignes de ce fichier

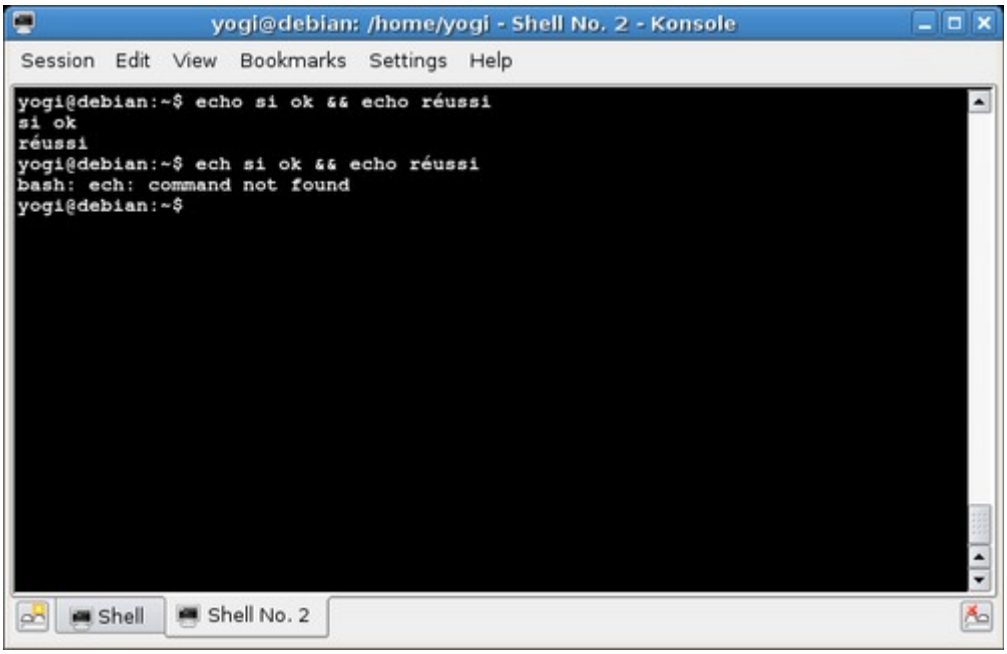
2. le parallélisme &

```
$ echo a & echo b
```

Les 2 commandes s'exécutent simultanément.

3. la dépendance &&

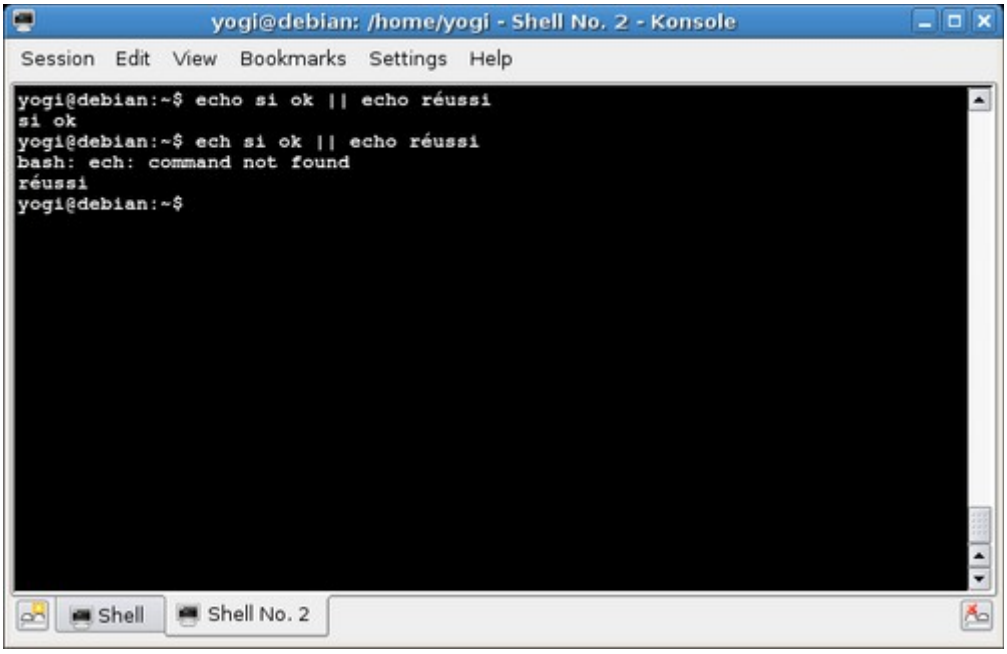
4 tube



Dans le 1er cas, les 2 commandes s'exécutent.

En revanche dans le 2ème cas, où il une erreur de syntaxe pour la 1ère commande, le shell ne regarde même pas le 2ème commande. Il s'arrête et signale que ech n'est pas une commande connue.

4. l'alternative ||



Dans le 1er cas, seule la 1ère commande s'exécute.

Dans le 2ème cas, le shell affiche une erreur pour la 1ère commande mais il exécute quand même la 2ème.

5. le séquencement ;

```
$ echo a ; sleep 1 ; echo b ; sleep 2 ; echo c
```

echo a s'exécute

on attend 1 seconde

echo b s'exécute

on attend 2 secondes

echo c s'exécute

10. Écrire un script correct

- Des vérifications approfondies doivent être effectuées sur TOUTES les commandes utilisées.
- Des commentaires détaillés doivent apparaître lors de chaque étape. De même, chaque étape doit être suivie d'un "echo <voici ce que je fais>" (particulièrement utile notamment lors du débogage).
- Lors d'une mise à jour, un fil de discussion doit être précisé pour tracer les bugs éventuels.
- Commencer par :

```
#!/bin/bash
# Version du script
```

- Écrire les variables en majuscule et NE PAS choisir des noms de commandes (ping , ls, …) de même pour les noms de fonctions
- À la fin de vos scripts, ajouter impérativement :

```
exit 0;
```

  Ce qui indique que votre script s'est exécuté correctement.

- Créer des fonctions pour des actions précises :

```
nom_de_la_fonction()
{
...
}
```

- Utiliser des chemins absolu pour les dossiers et des chemins relatif pour les nom de fichiers

```
$CHEMIN_DU_DOSSIER/$NOM_DU_FICHIER
```

- Utiliser les entrées de commandes pour les fonctions :

```
nom_de_la_fonction $1 $2 $3 ....
```

- Si votre script doit s'arrêter à cause d'une erreur, d'une variable qui ne correspond pas a vos attentes utiliser des numéros exit différents :

```
exit 100;
exit 101;
exit 102;
....
```

Ça permettra d'identifier d'où vient l'erreur.

- Utiliser le tableau ${PIPESTATUS[@]} pour récupérer les états des autres commandes.
- On peut écrire une fonction d'erreur du type :

```
erreur()
{
tab=( ${PIPESTATUS[@]} )

      for (( i=0; i < ${#tab[@]}; i++ )); do ((i+=i)); done

      if ((i > 0)); then
         zenity --error --title="Une erreur est survenue" --text="Une erreur est survenue "
         exit 100
      fi
}
```

ainsi après chaque commande vous pouvez donner des codes d'exécutions différents.