Shell Unix

Guide du Shell Linux / Unix

Support de cours pour une formation de 3 jours,
organisée et délivrée par Pierre ROYER

Bonne lecture…

Pierre ROYER

Architecte systèmes d'information

INDEX

I. Introduction

A. Présentation

Un shell est un programme ayant pour fonction d'assurer l'interface entre l'utilisateur et un système Unix / Linux. Il est aussi nommé interpréteur de commandes.

Le premier interpréteur de commandes date des années 70 : Bourne Shell (sh). Il a été écrit par Steve Bourne aux laboratoires AT&T, et utilisé depuis 1977 sur les systèmes Unix.

Le Bourne Shell est repris par David Korn en 1983 --> Korn Shell (ksh)

Le C shell (csh ou sa version améliorée, tcsh) est un shell Unix qui a été créé par Bill Joy dans les années 1970, et à été diffusé sur la version 2BSD du système Unix BSD.

Le Bash est fondé sur le Bourne shell, et lui apporte de nombreuses améliorations, provenant notamment du Korn shell et du C shell.

Plusieurs shells sont disponibles sur les plates-formes Unix (/etc/shells).

```
[CentOS@localhost ~]# vi /~/Scripts/MonScript
```

```
#!/bin/sh : Bourne shell
#!/bin/bash : Bourne shell again
#!/bin/ksh : Korn shell
#!/bin/csh : C shell
#!/bin/tcsh : Tenex shell
```

Contrairement aux commandes internes qui sont intégrées au processus shell, les commandes externes sont des fichiers localisés dans l'arborescence.

La différence est entre le type de commande est visible par :

```
[CentOS@localhost ~]# type cd
cd is a shell builtin

[CentOS@localhost ~]# type ls
ls is /usr/bin/ls
```

II. Variables d'environnement

A. Affectations

Affichage de la liste des variables du shell courant :

```
[CentOS@localhost ~]# set
```

Affichage d'une variable :

```
[CentOS@localhost ~]# echo $HOME
```

Spécification des chemins, avec ajout du répertoire courant (.) :

```
[CentOS@localhost ~]# PATH=/usr/bin:/usr/sbin:/bin:/sbin:.
```

Si une variable contient des caractères spéciaux, il faut empêcher le shell d'interpréter ceux-ci en entourant la valeur avec des simples quotes.

```
[CentOS@localhost ~]# variable='/>*<\'
```

Suppression d'une variable :

```
[CentOS@localhost ~]# unset variable
```

B. Principales variables

PWD : répertoire courant

PS1 : prompt de la console :

Différentes variables permettent une personnalisation du prompt :

Nom de l'utilisateur : \u
Nom de la machine : \h
Répertoire courant : \w
L'heure (24) : \T
L'heure (AM-PM) : \@
Code de couleur : : [\033[40;36m\]

```
[CentOS@localhost ~]# PS1="\[\033[0;31m\][\u@\h]\[\033[0;32m\]\w\[\033[0;m\] $ "
```

TERM : type de terminal utilisé.

LOGNAME : nom de l'utilisateur connect

C. Variables locales

Les variables d'environnement qui définissent la disposition du clavier sont définies dans ces fichiers :

- **/etc/locale.conf** sur Red Hat ou CentOS
- **/etc/sysconfig/language** sur SuSE Linux
- **/etc/default/locale** sur Debian et Ubuntu

Le changement de paramètres peut être effectués avec

```
[root@CentOS ~]# localectl set-keymap fr
```

Les autres variables locales sont consultables avec la commande :

```
[root@CentOS ~]# locale
```

D. Exportations

Afin de transmettre une variable aux processus descendants, il faut l'exporter :

```
[CentOS@localhost ~]# export PATH
```

Liste des variables exportées :

```
[CentOS@localhost ~]# env
```

E. Répertoire utilisateur

Le caractère tilde (~) représente le répertoire d'accueil de l'utilisateur courant :

Ces commandes sont identiques :

```
[pierreau@localhost ~]# ls ~
[pierreau@localhost ~]# ls ~pierreau
[pierreau@localhost ~]# ls /home/pierreau
```

F. Scripts de connexion

Le script shell système **`/etc/profile`** contient les paramétrages communs à tous les utilisateurs. Il est exécute lors d'une connexion à un shell. Sur certains systèmes, il est situé dans /etc/rc.d/rc.sysinit, ou bien encore dans /etc/default/init (Unix Solaris).

En fonction du type de shell lancé, un autre script est exécute pour initialiser un contexte de session, juste après le login :

- sh ou ksh : `~/.profile`

- bash : `~/.bash_profile` sinon `.~bash_login`, sinon `~/.profile`

Après modification de ce script, sa prise en compte est simple :

```
[pierreau@localhost ~]# $HOME/.bash_profile
```

G. Les alias

Le shell permettent de définir et paramétrer ses propres commandes internes, via des alias.

```
[pierreau@localhost ~]# alias lll='ls -al | sort -n | more'
```

Supprimer un alias :

```
[pierreau@localhost ~]# unalias lll
```

H. Options du shell

Certains paramètres peuvent être activés ou désactivé. Leur liste est visible via :

```
[CentOS@localhost ~]# set -o
```

Par exemple, l'option `ignoreeof` permet d'activer ou non la séquence de touche [CTRL d] pour quitter un shell

```
[CentOS@localhost ~]# set -o ignoreeof
```

Afin de désactiver une option :

```
[CentOS@localhost ~]# set +o ignoreeof
```

III. Redirections, Pipes, /dev/null

A. LES CANAUX D'ENTREES-SORTIES

Le canal d'entrée standard **stdin** (clavier) : **0**

L'injection d'un fichier de paramètres à un programme peut s'effectuer via une redirection du canal d'entrée vers ce programme :

```
[CentOS@localhost ~]# mail pierreau < message.txt
[CentOS@localhost ~]# Commande 0< paramètres
```

PS : une écriture plus simple est envisageable : ***Commande*** **<** parametres

Double redirection en lecture

```
[CentOS@localhost ~]# mail pierreau <<FIN
> Déjeuner confirmé à 13 heures
> Pierre.
> FIN
```

Le canal de sortie standard **stdout** (le terminal) : **1**

Il est utilisé pour l'affichage des résultats des commandes sur l'écran

Le canal des erreurs **stderr** (le terminal) : **2**

Si l'on provoque une erreur, le système utilise le canal 2.

Redirection du canal 2 :

```
[CentOS@localhost ~]# cat /tmp/FichierInexistant 2> /tmp/Resultat
[CentOS@localhost ~]# find / -name passwd 2> erreur
```

Redirection d'un canal de sortie vers un autre canal de sortie : **>&**

Redirection du canal stderr vers stdout :

```
[CentOS@localhost ~]# Commande 2>&1
```

Redirection de stdout et stderr vers un fichier Resultat :

Attention : ***Commande*** > /tmp/Resultat **2>&1**

- **stderr** est redirigé vers la valeur courante de la **stdout**, donc l'**écran**
- **stdout** vers un fichier. Donc **stdout a été redirigé vers ce fichier**.

Alors que ***Commande*** **>** /tmp/Resultat **2>&1**

- **stdout** est redirigée vers un fichier

- **stderr** est redirigée vers la valeur courante sur laquelle pointe **stdout**, donc **le fichier**. En conséquence, **stdout et stderr ont bien été redirigés vers un même fichier**.

B. LES PIPES

Ils consistent à brancher des canaux entre eux dans le but d'effectuer des opérations à la chaîne :

```
[CentOS@localhost ~]# who | wc -l
[CentOS@localhost ~]# cat /etc/passwd | more
[CentOS@localhost ~]# ls /etc | sort -n | grep -v root
[CentOS@localhost ~]# find . -type f -print0 | xargs -0 du -k | sort -nr
[CentOS@localhost ~]# echo "Déjeuner confirmé à 13 heures" | write pierreau
```

C. Le périphérique /dev/null

Il s'agit d'un pseudo périphérique dans lequel les flux ne sont pas traités, perdus à jamais ; l'information est inexistante, à la manière d'un trou noir…

Nous ne voulons pas traiter les messages stdout : `cat FichierInexistant 2> /dev/null`

Nous désirons vider un fichier : `cat /dev/null > FichierPlein`

D.Séparateur de commandes

Le caractère spécial **;** du shell permet d'écrire plusieurs commandes sur une même ligne. Les commandes sont exécutées séquentiellement. Mkdi

```
[pierreau@localhost ~]# mkdir repertoire ; cd repertoire ; pwd
/home/pierreau/repertoire
```

E.Regroupement de commandes

Les parenthèses permettent d'agréger des commandes. Ainsi, ces commandes diffèrent :

```
[CentOS@localhost ~]# date ; ls > fichier.txt
[CentOS@localhost ~]# (date ; ls) > fichier.txt
```

Exercices

IV. Les filtres

head : affiche entête d'un fichier

head –**n**x : affiche les x premières lignes

tail : affiche la fin d'un fichier

tail –**n**x : affiche les x dernières lignes

tail -f *file* : affiche en temps réel les modifications du fichier

head -15 *file* | tail -5 : affiche les lignes 10 à 15 de file

wc : compte les lignes, les mots et les caractères d'un fichier

diff : compare 2 fichiers

cut : récupère sur chaque ligne un ou plusieurs caractères

cut -c 1-10 : extrait les 10 premiers caractère de chaque ligne

cut -c 1,10 : extrait les 1ères et les 10èmes caractères

cut -d: -f3 : extraction du 3ème champ **d**élimités par « : »

paste *file1 file2* : collage de colonnes de deux fichiers

sort [options] fichier : trie les colonnes (options = -n : tri numérique, -r : décroissant, -k : colonne à trier, -t : séparateur de champs, -f ne tient pas compte de la casse). Les numéros de colonnes commencent à zéro si on n'utilise pas -t.

sort -t: -k 7,7 /etc/passwd : trie sur la 7ème colonne sans tenir compte de la casse

sort *file1* | uniq : élimine les lignes redondantes d'un fichier

tr [a-z] [A-Z] <fichier : remplace les minuscules dans fichier en majuscules

grep '*expression*' *fichier* : recherche les lignes comprenant l'expression

/usr/xpg4/bin/**grep -e** '*string1*' **-e** '*string2*' *file* : recherche string1 et string2 dans file (Solaris)

grep -v '*expression*' *fichier* : extrait les lignes **ne comprenant pas** *expression*

grep -c '*expression*' *fichier* : compte le nombre de ligne correspondant à *expression*

grep -n '*expression*' *fichier* : indique le numéro de ligne correspondant au critère

grep -i : ne tient pas compte de la casse

grep -l *chaine* * : recherche le fichier où se trouve la chaîne recherchée

sed ‘***expression’ 'fichier*** : remplacement de texte

sed -e 's/*exp1/exp2/'* : remplace exp1 par exp2 (une occurrence par ligne)

sed -e 's/*exp1/exp2/g'* : remplace exp1 par exp2 (toutes les occurrences)

sed '/*expr*/d' *fichier* : suppression de la ligne "matchée" par une expression régulière

sed '3d' *fichier* : suppression de la 3ème ligne

sed '$d' *fichier* : suppression de la dernière ligne

sed '/nologin$/d' /etc/passwd : efface les lignes qui se terminent par **nologin**

sed '/nologin$/ !d' /etc/passwd : n'efface pas les lignes qui se terminent par **nologin**

sed -n '/^.\{65\}/p' /etc/passwd : affiche les lignes de plus de 65 caractères

sed '0~3G' /etc/passwd : insérer une ligne blanche toutes les 3 lignes

V. Echo

A. Les caractères d'échappement

Le retour à la ligne :
```
[CentOS@localhost ~]# echo -e "a\nb"
```

Continuité su la même ligne :
```
[CentOS@localhost ~]# echo -e "a\c" ; echo -e "b"
```

La tabulation :
```
[CentOS@localhost ~]# echo -e "a\tb"
```

B. Les caractères de substitution

***** représente une suite de caractères quelconques
? représente un caractères unique.
Les crochets permettent de spécifier une série des caractères à une position précise :

Fichiers dont le nom commence par f ou o et se termine par le caractère . suivi d'une minuscule :
```
[CentOS@localhost ~]# ls [fo]*.[a-z]
```

Fichiers dont le nom comporte en deuxième caractère une majuscule ou un chiffre ou la lettre i. Les deux premiers caractères seront suivis d'une chaîne quelconque :
```
[CentOS@localhost ~]# ls ?[A-Z0-9i]*
```

Noms de fichier ne commençant pas par une minuscule
```
[CentOS@localhost ~]# ls  [!a-z]*
```

C. Les caractères spéciaux

Caractères	Signification
espace - tabulation - saut de ligne	Séparateurs de mots sur la ligne de commande
&	Arrière-plan
| << < > >>	Tube et redirections
() et {}	Regroupement de commandes
;	Séparateur de commandes
* ? [ ] ?() +() *() !() @()	Caractères de génération de noms de fichier
$ et ${ }	Valeur d'une variable
`` $()	Substitution de commandes
' ' " " \	Caractères de protection

D. Les caractères de protection

Les simples quotes (') retirent la signification des caractères spéciaux du shell :

```
[pierreau@localhost ~]# echo $HOME
/home/pierreau

[pierreau@localhost ~]# echo '$HOME'
$HOME
```

L'antislash inhibe la signification spéciale du caractère qui le suit.

```
[pierreau@localhost ~]# echo L\'antislash protege la quote
[pierreau@localhost ~]# echo "L\t\'antislash protege...\nla quote"
```

Les guillemets retirent la signification de tous les caractères spéciaux du shell sauf :

- $
- ``
- $()
- \
- ""

```
[pierreau@localhost ~]# echo "$(logname)"
pierreau

[pierreau@localhost ~]# echo '$(logname)'
$(logname)
```

VI. Premier script

Par convention, les scripts se reconnaissent par leur extension.
Par obligation, les scripts ont au moins un statut eXécutable.
Les lignes de commentaires commencent par le caractère dièse.

Choisir et définir en première ligne du script l'un des shells disponibles dans `/etc/shells`
Préciser son chemin absolu, juste après le Shebang (**#!**)

```
[CentOS@localhost ~]# vi ~/Scripts/MonScript.sh
```

```
#!/bin/bash : Bourne shell again
```

```
[CentOS@localhost ~]# chmod u+x ~/Scripts/MonScript.sh
```

A. Paramètres positionnels

Les scripts peuvent être appelés avec des options (9 au maximum)

Variables :
$0 : nom du script
$1 à **$9** : paramètres envoyés
$# : nombre de paramètres
$@ : liste de tous les paramètres
$* : tous les paramètres individuels

```
#!/bin/bash
# recherche si des paramètres sont passés
if [ $# -gt 0 ]
then
   echo "Il y a $# paramètre(s)"
   echo "$@"
   echo "$*"
   echo Quelques paramètre :$1 $3 $5 $7
else
   echo "Il n'y a aucun paramètre !"
fi
exit 0
```

$? : renvoie le résultat d'une commande (chiffre de 0 à 255)

B. Read

Saisie d'une variable au clavier :

```
#!/bin/bash
echo -n "Connaissez-vous Linux (Oui/Non/autre choix ?)"
read reponse
echo "${reponse}"
exit 0
```

C. Vérifier et corriger

Un script peut être validé par plusieurs options :

Lecture des commandes sans les exécuter :
```
[CentOS@localhost ~]# bash -n ~/Scripts/MonScript.sh
```

Mode verbose : exécute et affiche les commandes :
```
[CentOS@localhost ~]# bash -v ~/Scripts/MonScript.sh
```

Affichage des lignes valides et incorrectes :
```
[CentOS@localhost ~]# bash -x ~/Scripts/MonScript.sh
```

VII. Les tests

A. Les fichiers

Expression	Implémenté en Bourne Shell ?	Code de retour **VRAI** si :
	Tests sur l'existence et la taille du fichier	
-e	non	le fichier existe
-s	oui	le fichier n'est pas vide
	Tests sur le type du fichier	
-f	oui	le fichier est de type ordinaire
-d	oui	le fichier est de type répertoire
-h	oui	le fichier est de type lien symbolique
-L	non	le fichier est de type lien symbolique
-b	oui	le fichier est de type spécial bloc
-c	oui	le fichier est de type spécial caractère
-p	oui	le fichier est de type tube nommé
-S	non	le fichier est de type socket
	Tests sur les permissions du fichier	
-r	oui	le fichier est accessible en lecture
-w	oui	le fichier est accessible en écriture
-x	oui	le fichier possède le droit d'exécution
-u	oui	le fichier possède le setuid-bit
-g	oui	le fichier possède le setgid-bit
-k	oui	le fichier possède le sticky-bit
	Divers	
nomfic1 -nt nomfic2	non	le fichier **`nomfic1`** est plus récent que le fichier **`nomfic2`**
nomfic1 -ot nomfic2	non	le fichier **`nomfic1`** est plus ancien que le fichier **`nomfic2`**
nomfic1 -ef nomfic2	non	les fichiers **`nomfic1`** et **`nomfic2`** référencent la même inode (liens physiques)
-O nomfic	non	l'utilisateur est propriétaire du fichier
-G nomfic	non	l'utilisateur appartient au groupe propriétaire du fichier
-t [desc]	oui	le descripteur (1 par défaut) est associé à un terminal

Exemples :

Recherche si les fichiers passés en argument existent

```
for i in "$@"
do
  if [ -f $i ]
  then
    echo "Le fichier $i existe"
  else
    echo "Le fichier $i n'existe pas"
  fi
done
exit 0
```

L'argument est un fichier ou un répertoire ? Si la réponse est nul, on envoie un message

```
if [ -z $1 ] ; then
   echo "Vous n'avez pas entré de paramètres"
   echo "Le mode d'utilisation du script est $0 NomDuFichier"
   exit 0
fi

if [ -f $1 ] ; then
   echo "$1 est un fichier"
   exit 0
fi

if [ -d $1 ] ; then
   echo "$1 est un répertoire"
   exit 0
fi

echo "$1 n'est ni un fichier ni un répertoire ou inexistant."
exit 0
```

On peut effectuer des tests inversés avec l'opérateur !

```
if ! [[ -s "$fichier" ]]
```

B. Les chaînes de caractères

Expression	Code de retour **VRAI** si :
-z ch1	la chaîne est de longueur 0 (-z:zero)
-n ch1	la chaîne n'est pas de longueur 0 (-n:non zero)
ch1 = ch2	les deux chaînes sont égales
ch1 != ch2	les deux chaînes sont différentes
ch1	la chaîne n'est pas vide

Exemples :

En ligne de commande

```
[CentOS@localhost ~]# ch1=root
[CentOS@localhost ~]# ch2=pierreau
[CentOS@localhost ~]# [ "$ch1" = "$ch2" ]
[CentOS@localhost ~]# echo $?
1
```

echo $? renvoie la valeur de retour de la dernière expression ou script.

C. Les nombres

Expression	Code de retour **VRAI** si :
nb1 -eq nb2	**nb1** est égal à **nb2**
nb1 -ne nb2	**nb1** est différent de **nb2**
nb1 -lt nb2	**nb1** est strictement inférieur à **nb2**
nb1 -le nb2	**nb1** est inférieur ou égal à **nb2**
nb1 -gt nb2	**nb1** est strictement supérieur à **nb2**
nb1 -ge nb2	**nb1** est supérieur ou égal à **nb2**

Exemples :

```
#! /usr/bin/bash

# Test du nombre d'arguments
if [ $# -ne 2 ] ; then
 echo "Mauvais nombre d'arguments"
 echo "Usage: $0 nb1 nb2"
 exit 1
fi

if [[ $1 -gt $2 ]]
then
  echo "$1 est plus grand que $2"
else
  echo "$2 est plus grand que $1"
fi
exit 0
```

D. Les opérateurs

Opérateur de la commande **test** (par ordre de priorité décroissante)	Signification
!	Négation
-a	ET
-o	OU

E. Les opérateurs du shell

Opérateur	
&&	ET logique
||	OU logique

1. L'opérateur $$

```
[CentOS@localhost ~]# commande1 && commande2
```
La deuxième commande est exécutée uniquement si la première commande renvoie un code vrai.

L'expression globale est vraie si les deux commandes renvoient vrai.

2. L'opérateur ||

```
[CentOS@localhost ~]# commande1 || commande2
```
La deuxième commande est exécutée uniquement si la première commande renvoie un code faux.

L'expression globale est vraie si au moins l'une des deux commandes renvoie vrai.

F. Les opérateurs arithmétiques

Opérateurs	Signification
	Opérateurs arithmétiques
nb1 + nb2	Addition
nb1 - nb2	Soustraction
nb1 * nb2	Multiplication
nb1 / nb2	Division
nb1 % nb2	Modulo
	Opérateurs de comparaison, VRAI si :
nb1 \> nb2	**nb1** est strictement supérieur à **nb2**
nb1 \>= nb2	**nb1** est supérieur ou égal à **nb2**
nb1 \< nb2	**nb1** est strictement inférieur à **nb2**
nb1 \<= nb2	**nb1** est inférieur ou égal à **nb2**
nb1 = nb2	**nb1** est égal à **nb2**
nb1 != nb2	**nb1** est différent de **nb2**
	Opérateurs logiques, VRAI si :
chaine1 \& chaine2	les deux chaînes sont vraies (valeur différente de chaîne nulle et 0)
chaine1 | chaine2	l'une des chaînes est vraie (valeur différente de chaîne nulle et 0)

Exemples :

1. Syntaxe avec expr

```
[CentOS@localhost ~]# x=2
[CentOS@localhost ~]# resultat=`expr $x \* 3 + 1`
[CentOS@localhost ~]# echo $resultat
7
```

2. Syntaxe avec (())

```
[CentOS@localhost ~]# x=2
[CentOS@localhost ~]# ((x=x*3+1))
[CentOS@localhost ~]# echo $x
7
```

VIII. Les structures de contrôle

A. If / elif / fi

```
# Conditions sur saisie
echo "Entrez un code postal : \c"
read cp
if [[ $cp = 75[0-9][0-9][0-9] ]]
  then
  echo "$cp est un code postal parisien"
elif [[ $cp = @(7[78]|9[1-5])[0-9][0-9][0-9] ]]
# @ : Au moins le code département doit etre présent (2 chiffres)
  then
  echo "$cp est un code postal de la région parisienne"
elif [[ $cp = @([0-9][0-9][0-9][0-9][0-9]) ]]
# @ : Code postal obligatoire sur 5 chiffres
  then
  echo "$cp est un code postal de province (Métropole)"
else
  echo "$cp n'est pas un code postal de France Métropolitaine"
  exit 1
fi
exit 0
```

B. case

```
# Choix avec case
echo -n "Connaissez-vous Linux (Oui/Non/autre choix ?)"
read reponse
case "${reponse}" in
   O*|o*) echo "Tant mieux"
          exit 0
   ;;
   N*|n*) echo "Il n'est jamais trop tard  !"
          exit 1
   ;;
   *) echo "Comment est-ce possible ?"
      exit -1
   ;;
esac
exit 0
```

C. for / in / do / done

```
# Rempli un tableau avec des noms de fichiers
#!/bin/bash
# Boucle For
n=0
for fichier in *
do
        if [[ -e "$fichier" ]]
        then
                tableau[$n]=$fichier
                echo "${tableau[n]}"
        else
                echo "Aucun fichier"
        fi
((n+=1))
done
```

D. while / do / done

```
# Boucles sur saisies : addition de nombres
somme=0
echo "Saisir un nombre par ligne, [CTRL]d pour sortir de la
boucle"
while read nombre
do
  # On attend 0 à 1 fois un caractère + ou -
  if [[ "$nombre" != ?([+-])+([0-9]) ]]
  then
     echo "La valeur saisie n'est pas un nombre"
     continue
  fi
    # Equivalences :
     # - ((somme=somme+nombre))
     # - en Bourne Shell: somme=`expr $somme + $nombre`
    ((somme+=nombre))
done
echo "Somme: $somme"
exit 0
```

E. until / do / done

```
# Boucles sur saisies : nombre à trouver
a_trouver=$(($RANDOM % 10) + 1))
echo "Saisissez un nombre compris entre 1 et 10"
read nombre
until [ "$nombre" -eq "$a_trouver" ]
do
  if [ "$nombre" -lt "$a_trouver" ]; then
    echo "Trop petit :("
  else
    echo "Trop grand !"
  fi
  read nombre
done
echo "Bravo !"
exit 0
```

F. select item

La commande **`select`** permet de boucler sur un menu. Les variables **`REPLY`** et **`PS3`** sont réservées à l'usage de cette commande.

```
#! /usr/bin/bash
PS3="Votre choix : "
select item in "- Choix 1" "- Choix 2" "- Fin"
do
  echo -e "\n Vous avez choisi l'item $REPLY: $item"
  case "$REPLY" in
    1)
       echo -e "\n Premier choix"
       ;;
    2)
       echo -e "\n Deuxième choix"
```

```
        ;;
    3) echo -e "\n Au revoir"
       exit 0
       ;;
    *) echo -e "\n Saisie incorrecte"
       ;;
  esac
done
```

G. break - continue

Les commandes **break** et continue permettent de gérer ou non la sortie des boucles **for**, **while**, **until** et **select**.

IX. Notions avancées

A. Les tableaux

```
#!/bin/bash
# Enregistre le nom des fichiers du répertoire courant
n=0
for fichier in *
do
  tableau[n]=$fichier
  echo "Fichier $n : ${tableau[n]}"
  ((n+=1))
done
echo "${#tableau[*]} fichiers dans le tableau :"
echo "${tableau[*]}"
```

B. Substitution de variables

Le shell nous offre la possibilité de retirer certains champs d'une variable, sans pour autant en modifier le contenu.

```
[CentOS@localhost ~]# ligne="champ1:champ2:champ3"
```

Retire le plus petit fragment à gauche de la chaîne :
```
[CentOS@localhost ~]# echo ${ligne#*:}
champ2:champ3
```

Retire le plus grand fragment de gauche :
```
[CentOS@localhost ~]# echo ${ligne##*:}
champ3
```

Retire le plus petit fragment de droite :
```
[CentOS@localhost ~]# echo ${ligne%:*}
champ1:champ2
```

Retire le plus grand fragment de droite :
```
[CentOS@localhost ~]# echo ${ligne%%:*}
champ1
```

C. Les fonctions

Les **functions** sont déclarées avant leur appel :

```
#!/bin/bash
# Recherche sur saisie la présence de $user dans /etc/passwd
function exist_user {
  echo "Saisir le nom d'un utilisateur : \c"
  read user
  if grep -q "^$user": /etc/passwd
  :wthen
     return 0
  fi
  return 1
}
if exist_user
then
```

```
  echo "L'utilisateur $user est défini"
  exit 0
else
  echo "L'utilisateur $user n'est pas defini"
  exit 1
fi
```

D. Portée des variables

La variable **$user** du script ci-dessus à par défaut une portée publique. La déclaration d'une variable locale s'effectue avec typeset :

```
#!/bin/bash
function exist_user {
  typeset variableLocale
  variableLocale=1
  return 0
}
```

E. Redirection des entrées/sorties

Il est possible de rediriger les canaux dans un script avec l'instruction exec :

```
#!/bin/bash
exec 1> Messages.txt 2> Erreurs.txt

ls -l /
ls ufdiousqryufgdsyq
```

La sortie standard placées après **exec** sera redirigée vers le fichier `Messages.txt`
La sortie d'erreurs placées après **exec** sera redirigée vers le fichier `Erreurs.txt`

F. Les paramètres

Les scripts peuvent être appelés avec des paramètres (et options). La fonction **getopts** permet de traiter les paramètres valides lors de l'appel du script :

```
[CentOS@localhost ~]# vi ~./Scripts/parametres.sh
```

```
#!/bin/bash
while getopts "d:fiv" OPTNAME
do
   case $OPTNAME in
      d) echo "Option $OPTNAME : directory = $OPTARG" ;;
      f) echo "Option $OPTNAME : force" ;;
      i) echo "Option $OPTNAME : interactive" ;;
      v) echo "Option $OPTNAME : verbrose" ;;
      *) echo "Option $OPTNAME inconnue" ;;
   esac
done
```

```
[CentOS@localhost ~]# ~/Scripts/parametres.sh -i
[CentOS@localhost ~]# ~/Scripts/parametres.sh -d répertoire
```

G. Le contenu des fichiers

1. Accéder à des lignes

Avant et après avoir accéder au contenu d'un fichier, il faut l'ouvrir, soit en lecture, soit en écriture. On va utiliser un descripteur entre 3 et 9 car ceux de 0 à 2 sont déjà réservés :

```
#!/bin/bash
# Ouverture de passwd en lecture (descripteur 3)
# Ouverture de out.txt en écriure (descripteur 4)
exec 3</etc/passwd 4>out.txt
read -u3 ligne
echo $ligne >&4
# Ligne suivante
read -u3 ligne
echo $ligne >&4
# Fermeture des fichiers
exec 3<&-
exec 4>&-
```

2. Accéder à des champs

Nous allons utiliser le délimiteur de champ IFS (Internal Field Separator) :

```
#!/bin/bash
echo "username : userUID : userGID : reste"
cat /etc/passwd | while IFS=';' read username passwd userUID
userGID reste
do
  echo "$username - $userUID - $userGID $reste"
done
```

3. Sommes sur des données

Saisir au clavier une suite de deux nombres (séparés par un espace) pour avoir leur somme.

```
#!/bin/bash
while read n1 n2
do
  echo "${n1} + ${n2} = `expr ${n1} + ${n2}`"
done
```

Le même exemple avec le fichier somme.txt suivant :

```
[CentOS@localhost ~]# vi ./somme.txt
```

```
37  7
3  13
1  71
```

```
#!/bin/bash
while read n1 n2
do
        echo "${n1} + ${n2} = `expr ${n1} + ${n2}`"
done < donnees.txt
```

X. Expressions régulières

- ou RegExp -

Les commandes ed, vi, ex, sed, awk, expr et grep utilisent les expressions régulières.

Le méta caractère **^** identifie un début de ligne.
Le méta caractère **$** identifie une fin de ligne.
L'expression régulière **^$** identifie une ligne vide.
^J : premier caractère de la ligne commençant par un J.
^[^J] : premier caractère de la ligne ne commençant pas par un J.

Les méta-caractères **[ ]** permettent de désigner des caractères compris dans un certain intervalle de valeur à une position déterminée d'une chaîne de caractères.
[A-D] : caractères compris entre A et D.
[2-5] : caractères compris entre 2 et 5.
[A-Za-z0-9] : caractère minuscule, ou majuscule, ou chiffre
[^a-z] : caractère non minuscule

Exemple pour l'affichage de fichiers commençant par FIC avec des lettres en minuscules ou / et majuscules :

Recherche classique avec grep :
```
[CentOS@localhost ~]# ls -l | grep -i 'fic'
```

Recherche avec les expressions régulières :
```
[CentOS@localhost ~]# ls -l [fF][iI][cC]*
```

Si l'on veut chercher une chaîne de caractère au sein de laquelle se trouve un méta caractère, il faut la faire précéder d'un backslash. Ce principe fonctionne pour les expressions non comprises entre crochets (comme ci-dessus) !

Le méta caractère **.** (point) remplace un caractère unique, à l'exception du caractère retour chariot (**\n**).

Les méta-caractères **{ }** permettent de désigner nombre d'occurrence de la chaîne recherchée :

La syntaxe à utiliser pour les exemples ci-dessous est :

```
[CentOS@localhost ~]# grep -E 'expression' fichier
```

ab{2} : chaîne de caractère composée d'un **a** suivit d'exactement deux **b** (en clair, seule la chaîne "abb" passe à travers ce modèle).
ab{2,} : chaîne de caractère composée d'un a suivi d'au moins deux **b** (par exemple, "abb" ou "abbb").
ab{3,5} : chaîne de caractère composée d'un a suivi de trois à cinq **b** ("abbb", "abbbb" et "abbbbb" sont les seules chaînes qui respectent ce modèle).

Les méta-caractères **( )** permettent de quantifier une chaîne de caractères :
a(bc)* : chaîne de caractères commençant par un **a** suivi d'aucune ou de plusieurs séquence de caractères "**bc**"

a(bc){1,5}* : chaîne de caractères commençant par un **a** suivi d'une à cinq fois la séquence de caractères "**bc**"

Le méta caractère ***** répète 0 à n fois le caractère qui le précède.
^.*[0-9] : lignes commençant par 0 ou n caractères quelconque(s) suivis d'un chiffre.
^.*137 : lignes commençant par 0 ou n caractères quelconque(s) suivis de 137.

Le méta caractère **+** répète 1 à n fois le caractère qui le précède.
Le méta caractère **?** répète 0 à 1 fois le caractère qui le précède.

Le méta caractère **|** représente le booléen **OR** :
(b|cd)ef : chaîne de caractères qui contient la séquence de caractères "bef" ou bien la séquence de caractères "cdef".

XI. AWK

A. En ligne de commande

AWK est un langage de traitement de fichiers textes dont le nom provient de ses trois créateurs : Alfred V. **A**ho, Peter J. **W**einberger et Brian W. **K**ernighan.

Voici quelques commandes :

-F : détermine le séparateur de champs : **`awk -F ';' '{script awk}'`** ***fichier***

`$0` représente toute la ligne, **`$1`** représente la première colonne...

Affiche le champ2 et le champ1 : **`awk -F ';' '{print $2 " " $1}'`** ***fichier***

Affiche le nom et le répertoire home des utilisateurs :

```
cat passwd | gawk -F":" '{print "Le répertoire personnel de " $5 " est " $6}'
```

Quelques variables réservées :

FS : caractère de délimitation de champs

NF : nombre de champs sur la ligne courante

NR : numéro de la ligne courante

RS : caractère séparateur de lignes

OFS : caractère de séparateur de champs en sortie

Affiche le nombre de lignes du fichier : **`awk 'END {print NR}'`** ***fichier***

Affiche la 10ième ligne d'un fichier : **`awk '{if (NR==10) print $0;}`** ***fichier***

Affiche le dernier champ de chaque ligne : **`awk '{print $NF}'`** ***fichier***

Fonctions prédéfinies :

Affiche les lignes de plus de 75 caractères : **`awk 'length($0)>75 {print}'`** ***fichier***

Sur la chaîne **t**, remplace toutes les occurrences de **r** par **s** : **`gsub(r,s,t)`**

Comme gsub, mais remplace uniquement la première occurrence : **`sub(r,s,t)`**

Retourne la sous chaîne de **s** commençant en **i** et de taille **n** : **`substr(s,i,n)`**

B. En script

```
awk 'BEGIN {
        FS=":";
```

```
        variable=10
        print "Liste des répertoires par utilisateur du fichier
/etc/passwd"
        }
        # Affiche la première ligne
        NR <= 1 { print NR, $0 }
        # Affiche les 10 premières lignes
        NR <= variable {
         print "Le répertoire personnel de " $5 " est " $6
        }
        END   { print "Fin" }
' /etc/passwd
```