Programmation Bash

Une version à jour et éditable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
https://fr.wikibooks.org/wiki/Programmation_Bash

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la Licence de documentation libre GNU, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans Texte de dernière page de couverture. Une copie de cette licence est incluse dans l'annexe nommée « Licence de documentation libre GNU ».

Sections

- 1 Introduction
- 2 Hello World
  - 2.1 Références
- 3 Notions essentielles du shell bash
  - 3.1 Commandes shell
    - 3.1.1 Syntaxe
    - 3.1.2 Informations sur les commandes
    - 3.1.3 Manuel
  - 3.2 Enchaînements de commandes
    - 3.2.1 Enchaînements simples
      - 3.2.1.1 Exécutions simultanées
      - 3.2.1.2 Exécutions successives
    - 3.2.2 Enchaînements conditionnels
      - 3.2.2.1 Et... et ... et
      - 3.2.2.2 Alternative
  - 3.3 Variables
    - 3.3.1 Affectation
    - 3.3.2 Substitution
    - 3.3.3 Variables prépositionnées
    - 3.3.4 Variables d'environnement
  - 3.4 Quotes, apostrophe, guillemets et apostrophe inversée
    - 3.4.1 Simple quote ou apostrophe
    - 3.4.2 Doubles quotes ou guillemets
    - 3.4.3 Back-quote, apostrophe inversée ou accent grave
- 4 Interactions avec le système de fichiers
  - 4.1 Fichiers et répertoires
    - 4.1.1 Créer
      - 4.1.1.1 Création d'un fichier
      - 4.1.1.2 Création d'un répertoire
    - 4.1.2 Déplacer et renommer
    - 4.1.3 Copier
    - 4.1.4 Supprimer
  - 4.2 Déplacements dans un système de fichier
    - 4.2.1 L'organisation, les chemins
    - 4.2.2 Les commandes cd et pwd
  - 4.3 "Monter" une ressource dans le système de fichiers

- 5 Flux et redirections
  - 5.1 En Bash
  - 5.2 Boîte à outils redirectionnels
  - 5.3 cat tautologique
  - 5.4 Redirection de la sortie standard
  - 5.5 Redirection de la sortie d'erreur standard
  - 5.6 Concaténation dans un fichier
  - 5.7 Tubes
  - 5.8 Résumé des outils de redirection
- 6 Tests
  - 6.1 Conditions
  - 6.2 Test if
  - 6.3 Test case
  - 6.4 Syntaxe du test
    - 6.4.1 Tester une variable
  - 6.5 Tests sur les objets du système de fichiers
  - 6.6 Tests sur les chaînes de caractères
  - 6.7 Tests sur les nombres (entiers)
  - 6.8 Tests et logique
  - 6.9 Un exemple complet
- 7 Structures conditionnelles
  - 7.1 if
    - 7.1.1 Test simple
    - 7.1.2 Test avancé
  - 7.2 case
- 8 Boucles
  - 8.1 Boucle for
    - 8.1.1 Première syntaxe
    - 8.1.2 Seconde syntaxe
  - 8.2 Boucles until et while
- 9 Scripts
  - 9.1 Au commencement, la ligne shebang
  - 9.2 Exécution d'un script
  - 9.3 Exemple de script
  - 9.4 Paramètres
- 10 Enchaînements et scripts
  - 10.1 Scripts
    - 10.1.1 Exécution d'un script
    - 10.1.2 Variables spéciales

    - 10.2 Enchaînements
        - 10.2.1 Enchaînements simples
        - 10.2.2 Enchaînements conditionnels
    - 10.3 Références
- 11 Calculs
    - 11.1 Calcul avancé avec bc
- 12 Interactions avec l'utilisateur
    - 12.1 Lire la saisie d'un utilisateur
    - 12.2 Interaction et case
    - 12.3 Instruction select
- 13 Regex
    - 13.1 Exemple
    - 13.2 Références
- 14 Fonctions
    - 14.1 Déclaration
    - 14.2 Appel et paramètres
        - 14.2.1 Appel
        - 14.2.2 Paramètres passés à la fonction
        - 14.2.3 Fin
    - 14.3 Variables
- 15 Commandes shell
    - 15.1 A
    - 15.2 B
    - 15.3 C
    - 15.4 D
    - 15.5 E
    - 15.6 F
    - 15.7 G
    - 15.8 H
    - 15.9 I
    - 15.10 J
    - 15.11 K
    - 15.12 L
    - 15.13 M
    - 15.14 N
    - 15.15 O
    - 15.16 P
    - 15.17 Q
    - 15.18 R
    - 15.19 S
    - 15.20 T

- 15.21 U
- 15.22 V
- 15.23 W
- 15.24 X
- 15.25 Y
- 15.26 Z
- 16 Commandes ksh
  - 16.1 Commandes d'aide
  - 16.2 Écran
  - 16.3 Système de fichiers
  - 16.4 Recherche
  - 16.5 Gestion de texte
  - 16.6 Permissions
  - 16.7 Processus
  - 16.8 Gestion de disques

Introduction

Le **Bash** est un shell, c'est-à-dire un interpréteur de commandes, écrit pour le projet GNU en 1989.

Bash est l'acronyme de *Bourne-Again SHell*, un calembour sur le shell Bourne sh, qui a été historiquement le premier shell d'Unix. Le shell Bourne original fut écrit par Stephen Bourne. Bash a été principalement écrit par Brian Fox et Chet Ramey. La syntaxe de Bash est compatible avec sh et inclut des développements issus de csh et ksh.

En effet, le Korn shell (ksh) est un shell Unix développé par David Korn. Il est compatible avec le Bash et inclut également un grand nombre de fonctionnalités du C shell, mais il permet également des fonctions de scripting avancées utilisées dans des langages de programmation plus évolués comme awk, sed, et Perl. La liste de ses commandes sera détaillée à la fin de ce livre.

Bash est le shell par défaut de la plupart des systèmes GNU/Linux, il est distribué sous la licence libre GNU GPL. Il fonctionne sur la plupart des systèmes d'exploitation de type Unix, et a également été porté sous Windows par Cygwin, puis sous windows 10 build 14316 par Microsoft.

Si un langage de programmation permet de dire à un processeur ce qu'il doit faire, **un interpréteur de commandes permet de dire à un système d'exploitation, un service ou à une application ce qu'ils doivent faire.** Cette nuance permet aux anglo-saxons de distinguer les notions de *programming* lorsqu'il s'agit d'écrire un code pour créer une application et de *scripting* lorsqu'il s'agit de programmer un système d'exploitation, un service ou une application. Le jeu d'instructions de Bash est donc spécialisé afin d'utiliser des systèmes Unix.

Hello World

Pour créer une variable :

```
V1='Hello World'
echo $V1
```

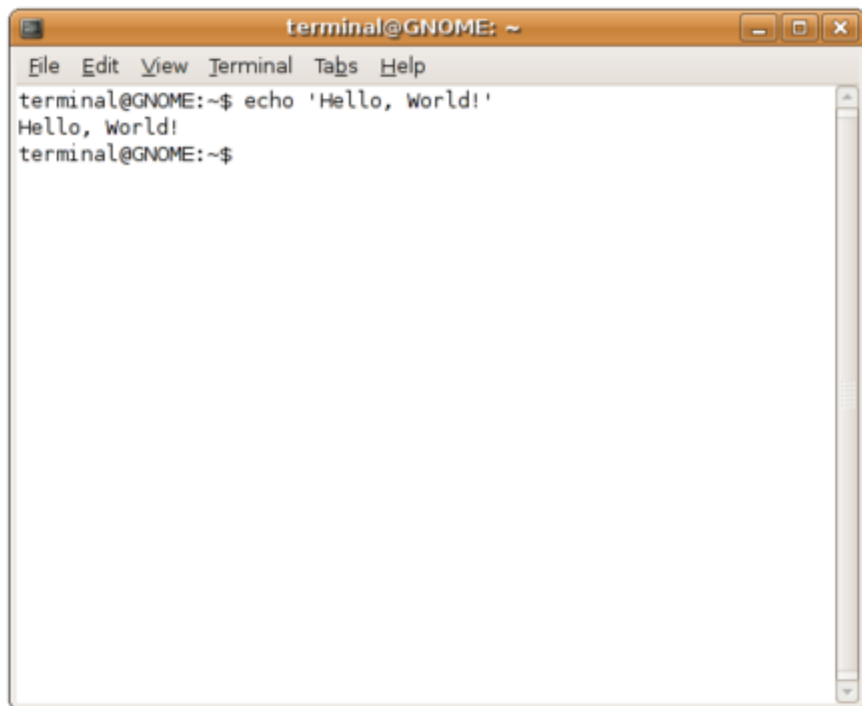
Un tableau[1] :

```
T[0]='Hello'
T[1]='World'

echo ${T[0]} ${T[1]}

# ou
echo ${T[*]}

# ou
for V in ${T[*]}
 do echo $V
done
```



Exemple de commande Bash dans un terminal.

Références

1. http://www.thegeekstuff.com/2010/06/bash-array-tutorial/

Notions essentielles du shell bash

Commandes shell

Définition

Une commande shell est une chaîne de caractères en minuscules qui peut être invoquée au travers d'une invite de commande ou d'un script. Des options et des arguments peuvent la compléter. Ceux-ci sont généralement appelés paramètres de la commande.

Exemples de commande bash :

```
ls
ls -l /tmp
cd /tmp
cp liste.pdf /pub/pdf
cp -r repertoire/ /pub/
```

Syntaxe

La syntaxe générique d'une commande shell est :

```
commande [options [argument1 [argument2 [... ]]]]
```

où une chaîne de caractères entre crochets est optionnelle.

Le nombre d'argument peut être limité (se reporter au manuel de la commande).

Les options de la commande peuvent être :

- des options courtes formées par un tiret suivi d'une lettre. Exemple : `-s`
- des options longues formées par deux tirets suivis d'un mot. Exemple : `--size`
- des options courtes ou longues qui exigent un argument. Exemples : `--speed 50` ou `-s 50`

Informations sur les commandes

La commande `whereis` permet de rechercher les fichiers exécutables, les sources et les pages de manuel d'une commande Par exemple :

```
$ whereis python
```

```
python: /usr/bin/python2.3 /usr/bin/python /etc/python2.3 /usr/lib/python2.3 /usr/lib
/python2.4
/usr/local/lib/python2.3 /usr/include/python2.3 /usr/share/man/man1/python.1.gz
```

signifie qu'il existe sur le système plusieurs commandes python dans différents répertoire. La page du manuel se trouve dans `/usr/share/man/man1/`.

La commande `which` permet de savoir quel est le fichier exécuté lorsque l'on entre le nom d'une commande. Par exemple :

```
$ which python
/usr/bin/python
```

signifie que le code de la commande `python` utilisée par le système d'exploitation se trouve dans le répertoire `/usr/bin/`.

Enfin, `whatis` permet de savoir rapidement à quoi sert une commande. Exemple :

```
$ whatis python
python (1)          - an interpreted, interactive, object-oriented programming
language
```

Manuel

Pour obtenir des informations plus complètes sur l'usage et l'utilité d'une commande, il suffit d'invoquer le manuel en ligne de commande : `man nom_de_la_commande`. Pour sortir du manuel, composer la combinaison de touche `esc : q`.

```
$ man echo
Remise en forme de echo(1), attendez SVP...
ECHO(1)                                     Manuel de l'utilisateur Linux
ECHO(1)

NOM
       echo - Afficher une ligne de texte.

SYNOPSIS
       echo [-neE] [message ...]
       echo {--help,--version}

DESCRIPTION
       Cette page de manuel documente la version GNU de echo.

       La plupart des shells ont une commande interne ayant le même nom et les mêmes
fonctionnalités.

       echo  écrit  chaque  message  sur  la sortie standard, avec une espace entre
chacun d'eux,

```

```
 et un saut de ligne après le dernier.

   OPTIONS
       -n     Ne pas effectuer le saut de ligne final.

       -e     Interprète les séquences de caractères précédées d'un backslash '\'
suivantes :
              \a     alerte (sonnerie)
              \b     retour en arrière d'un caractère
              \c     supprimer le saut de ligne final
              \f     saut de page
              \n     saut de ligne
              \r     retour chariot
              \t     tabulation horizontale
              \v     tabulation verticale
              \\     backslash
              \nnn   le caractère de code ASCII nnn (en octal)

       -E     Désactiver les interprétations des séquences spéciales.

   OPTIONS GNU
       Quand la version GNU de echo est appelée avec exactement un argument, les
options suivantes sont reconnues :

       --help Afficher un message d'aide sur la sortie standard et se terminer
normalement.

       --version
              Afficher un numéro de version sur la sortie standard et se terminer
normalement.

TRADUCTION
       Christophe Blaess, 1996-2003.
```

Exemple : consultation du manuel pour la commande `echo`

Enchaînements de commandes

Enchaînements simples

Exécutions simultanées

```
com1 & com2 & com3 & ... & comN
```

Les commandes `com1` jusqu'à `comN` sont exécutées parallèlement.

Exécutions successives

```
com1 ; com2 ; ... ; comN
```

Les commandes `com1` jusqu'à `comN` sont exécutées successivement, les unes après les autres.

Enchaînements conditionnels

Et... et ... et

```
com1 && com2 && ... && comN
```

Dans ce cas, la commande `comP` ne s'exécute que si `com(P-1)` s'est soldée par un succès. Exemple :

```
$ echo toto && echo jojo && echo sidonie
toto
jojo
sidonie
$  [ 1 -gt 2 ] && echo jojo
$
```

Dans le premier cas `echo toto && echo jojo && echo sidonie`, toutes les propositions sont justes, donc elles sont toutes exécutées. Dans le second `[ 1 -gt 2 ] && echo jojo`, la première partie `[ 1 -gt 2 ]` teste si 1 est plus grand que 2. Comme cette assertion est fausse, la deuxième partie n'est pas exécutée.

Cette structure est souvent utilisée pour programmer plus rapidement une structure conditionnelle de type if. Par exemple, les deux séries de commandes suivantes produisent le même résultat :

```
[ $note -gt 10 ] && echo "tu as la moyenne"
```

```
if [ $note -gt 10 ]
then
echo "tu as la moyenne"
fi
```

Alternative

```
com1 || com2 || ... || comN
```

Dans cet exemple les commandes `com1` jusqu'à `comN` sont exécutées successivement tant qu'aucune ne se termine correctement. Dès qu'une des commandes se solde par un succès, la commande alternative est terminée.

Exemple :

```
$ echo toto || echo jojo || echo sidonie
toto
$ [ 1 -gt 2 ] || echo jojo
jojo
```

Variables

Affectation

Pour affecter une valeur à une variable, la syntaxe est la suivante :

```
variable=valeur
```

place la `valeur` dans la `variable`. La portée de cette variable est locale au processus shell qui l'a définie.

Il faut faire très attention à ne pas placer d'espaces ni avant ni après le signe égal lors d'une affectation en bash. Ainsi, l'expression `variable =valeur` engendrera l'erreur `variable: not found` et `variable= valeur` donnera l'erreur `valeur: not found`.

Bash utilise le mécanisme de typage dynamique : il n'est donc pas nécessaire de spécifier un type pour une nouvelle variable.

Un nom de variable peut contenir des chiffres, des lettres et des symboles souligné mais ne doit pas commencer par un chiffre. La plupart du temps, les variables du shell bash sont des chaînes de caractères.

Substitution

On peut vérifier le contenu d'une variable à l'aide de la commande `echo` :

```
$ variable=5
$ echo $variable
5
```

Ce mécanisme qui différencie la variable de son contenu est nommé substitution. On peut en faire l'expérience grâce aux commandes suivantes :

```
$ variable=5
$ echo variable
variable
$ echo $variable
5
```

Par convention, le symbole `$` représente l'invite de commande.

Variables prépositionnées

Certaines variables ont une signification spéciale réservée. Ces variables sont très utilisée lors la création de scripts :

- pour récupérer les paramètres transmis sur la ligne de commande,

- pour savoir si une commande a échoué ou réussi,
- pour automatiser le traitement de tous paramètres.

Par exemple, la variable `$?` permet de connaître le code de retour de la dernière commande effectuée qui vaut généralement 0 si cette commande s'est bien déroulée et autre chose sinon. Voici une liste de ces variables prépositionnées :

- `$0` : nom du script. Plus précisément, il s'agit du paramètre 0 de la ligne de commande, équivalent de `argv[0]`
- `$1`, `$2`, ..., `$9` : respectivement premier, deuxième, ..., neuvième paramètre de la ligne de commande
- `$*` : tous les paramètres vus comme un seul mot
- `$@` : tous les paramètres vus comme des mots séparés : `"$@"` équivaut à `"$1" "$2"` ...
- `$#` : nombre de paramètres sur la ligne de commande
- `$-` : options du shell
- `$?` : code de retour de la dernière commande
- `$$` : PID du shell
- `$!` : PID du dernier processus lancé en arrière-plan
- `$_` : dernier argument de la commande précédente

Variables d'environnement

Une variable d'environnement est une variable accessible par tous les processus fils du shell courant. Pour créer une variable d'environnement, on *exporte* la valeur d'une variable avec la commande `export`.

```
export variable
```

Pour illustrer la différence entre une variable locale et une variable d'environnement, il suffit de créer une variable d'environnement, de lancer un shell fils et d'afficher la variable.

```
$ ma_variable=toto
$ export ma_variable
$ bash
$ echo $ma_variable
toto
$ exit
```

Dans ce cas, la valeur de la variable est accessible depuis le shell fils. Si on essaye de faire la même chose sans exporter la variable, la valeur ne sera pas accessible.

```
$ ma_variable=toto
$ bash
$ echo $ma_variable

$ exit
```

La commande `env` permet de récupérer la liste des variables d'environnement du système d'exploitation. D'une

manière générale, les variables d'environnement sont notées en majuscules.

```
$ env
REMOTEHOST=pcloin
HOST=machine.univ-autonome.fr
TERM=xterm-color
SHELL=/bin/bash
SSH_CLIENT=::ffff:197.123.57.3 55747 22
SSH_TTY=/dev/pts/0
http_proxy=http://proxy.univ-autonome.fr:3128/
GROUP=shadogroup
USER=shadouser
temp=2.6.20.1
HOSTTYPE=i386-linux
MAIL=/var/mail/shadouser
PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/game
PWD=/home/shadouser
EDITOR=/bin/vi
LANG=fr_FR@euro
temp1=Linux
PS1=\[\033[0;31m\]-->\T<--\[\033[0;34m\][\u@\h]\w[$]\[\033[0;32m\] -->
\[\033[0;34m\]
PS2=>
SHLVL=2
HOME=/home/shadouser
OSTYPE=linux
VENDOR=intel
MACHTYPE=i386
LOGNAME=patin
SSH_CONNECTION=::ffff:197.123.57.3 55747 ::ffff:199.1.1.62 22
_=/usr/bin/env
```

Utilisation de la commande env : consultation des variables d'environnement sur station linux

Quotes, apostrophe, guillemets et apostrophe inversée

Simple quote ou apostrophe

Les simples quotes délimitent une chaîne de caractères. Même si cette chaîne contient des commandes ou des variables shell, celles-ci ne seront pas interprétées. Par exemple :

```
$ variable="secret"
$ echo 'Mon mot de passe est $variable.'
Mon mot de passe est $variable.
```

Doubles quotes ou guillemets

Les doubles quotes délimitent une chaîne de caractères, mais les noms de variable sont interprétés par le shell. Par

exemple :

```
$ variable="secret"
$ echo "Mon mot de passe est $variable."
Mon mot de passe est secret.
```

Ceci est utile pour générer des messages dynamiques au sein d'un script.

Back-quote, apostrophe inversée ou accent grave

Bash considère que les Back-quotes délimitent une commande à exécuter. Les noms de variable et les commandes sont donc interprétés. Par exemple :

```
$ echo  `variable="connu"; echo "Mon mot de passe est $variable."`
Mon mot de passe est connu.
```

Autre exemple :

```
echo `ls`
```

Cette commande affiche le contenu du répertoire courant à l'écran. Elle est strictement équivalente à `ls`.

Le caractère apostrophe inversée est obtenu sur les clavier PC français par la combinaison Alt Gr + 7 et se trouve à gauche de la touche retour sur les clavier Macintosh.

Interactions avec le système de fichiers

Fichiers et répertoires

Créer

Création d'un fichier

Pour créer un fichier texte, le plus simple est d'utiliser l'éditeur de texte vi. Pour créer un fichier vide, il suffit d'utiliser la commande `touch`. La commande `ls` permet de recenser le contenu d'un répertoire. Exemple :

```
$ ls
$ touch vide
$ ls -l
total 0
-rw-r--r--  1 shadouser shadogroup 0 2007-03-03 10:20 vide
```

Création d'un répertoire

Pour créer un répertoire, utiliser la commande `mkdir`. Exemple :

```
$ ls
$ mkdir dossier
$ ls -l
total 4
drwxr-xr-x  2 shadouser shadogroup 4096 2007-03-03 10:28 dossier
```

Déplacer et renommer

Pour déplacer un objet du système de fichier, utiliser la commande `mv`. Exemple :

```
$ pwd
/home/shadouser/test
$ ls -l
total 4
drwxr-xr-x  2 shadouser shadogroup 4096 2007-03-03 10:28 dossier
-rw-r--r--  1 shadouser shadogroup 0 2007-03-03 10:20 vide
$ ls -l ~/
total 0
$ mv vide ..
$ ls
dossier
$ ls ..
vide
```

Pour renommer un fichier, il suffit de le déplacer vers un autre nom de fichier. Exemple :

```
$ ls
vide
$ mv vide plein
$ ls
plein
```

Copier

Pour copier un fichier, utiliser la commande `cp` comme suit :

```
$ pwd
/home/shadouser/test
$ ls
dossier vide
$ cp vide dossier/
$ ls dossier/
vide
$ ls
dossier  vide
```

Pour copier un répertoire, utiliser l'option `-r`.

Supprimer

La commande `rm` détruit un fichier. Attention, ce fichier ne pourra pas être récupéré ou difficilement. Exemple :

```
$ ls
plein
$ rm plein
$ ls
```

Pour supprimer un répertoire, utiliser l'option `-r`. Pour forcer la suppression, utiliser l'option `-f`

Déplacements dans un système de fichier

L'organisation, les chemins

Il faut toujours se souvenir que du point de vue d'un système de type UNIX, tout est fichier. Cela signifie que, pour le système, les périphériques - disques durs, écrans, imprimantes etc... - sont représentés par des fichiers. Les répertoires sont eux aussi des fichiers, d'un type particulier, puisqu'il "contiennent" d'autres fichiers. Le lecteur comprendra donc qu'il est important de maîtriser les notions de bases concernant la manipulation de fichiers.

Les fichiers d'un tel système sont organisés sous forme d'une arborescence dont la racine est représentée par le slash : `/` C'est par ce signe que débute tout chemin absolu menant à un fichier, sous réserve, bien entendu, de son existence. Pour indiquer un nœud dans ce chemin, une bifurcation, on utilise comme séparateur le slash.

Exemple :

```
/home/aniko/latex/rapport.tex
```

Ce chemin mène à un fichier baptisé "rapport.tex" qui se trouve dans un répertoire "latex", lui-même dans le répertoire "aniko", inclus dans "home" qui est un répertoire directement accessible depuis la racine. Cette organisation permet une grande souplesse, puisqu'il est aisé de détacher une partie de l'arborescence pour la greffer ailleurs. En outre cela permet de compartimenter clairement l'organisation du système, puisque par le jeu de différents droits sur les fichiers, tous ne sont pas lisibles, modifiables ou exécutables par tous les utilisateurs.

Les commandes `cd` et `pwd`

La commande `cd`, i.e. *common directory* (répertoire courant), permet de se déplacer dans le système de fichiers. La commande `pwd`, i.e. *present working directory*, affiche l'endroit où l'on se trouve actuellement dans le système de fichiers.

Exemple :

```
$ pwd
/home/shadouser
$ ls -l
total 4,0K
drwxr-xr-x   4 shadouser shadogroup     4096 2007-03-03 10:31 tmp
$ cd tmp
$ pwd
/home/shadouser/tmp
$ cd ..
$ pwd
/home/shadouser
$ cd .
$ pwd
/home/shadouser
$ cd tmp
$ pwd
/home/shadouser/tmp
$ cd ~/
$ pwd
/home/shadouser
```

Des symboles sont utilisés pour représenter les répertoires couramment utilisés :

- . désigne le répertoire courant, celui dans lequel on se trouve,
- .. désigne le répertoire parent de celui dans lequel on se trouve,
- ~ désigne le répertoire personnel de l'utilisateur, dans l'exemple précédent `/home/shadouser`. La variable d'environnement `$HOME` détient l'information sur le répertoire personnel de l'utilisateur.

"Monter" une ressource dans le système de fichiers

Les systèmes de fichiers Unix ne comprennent, comme nous le décrivons plus haut, qu'une seule arborescence logique. La racine de cette arborescence est nommée `/`. Il est possible d'intégrer sous cette arborescence tout type de périphérique : souris, disque dur, lecteur dvd, ou clé usb. La commande `mount` permet d'insérer une ressource dans cette arborescence.

Exemple :

```
$ mount -t ext3 /dev/hda2 /tmp
```

Cette commande permet de monter la ressource identifiée par le système comme `/dev/hda2` dans le répertoire `/tmp`. Il s'agit probablement d'une partition de disque dur dont le format est ext3. Cela signifie que pour accéder à cette ressource il suffit d'aller dans ce répertoire.

Exemple :

```
mount -t nfs pc33:/pub/ /home/sidonie/import/
```

Cette commande permet de monter le répertoire `/pub` du système de fichier du serveur distant `pc33` dans l'arborescence de la machine à travers le réseau en utilisant NFS.

Exemple :

```
mount -t vfat /dev/sda /home/sidonie/cle/
```

Cette commande permet de monter une clé usb.

Umount :

```
umount /home/sidonie/cle/
```

Cette commande permet de démonter le lecteur monté sur /home/sidonie/cle.

Flux et redirections

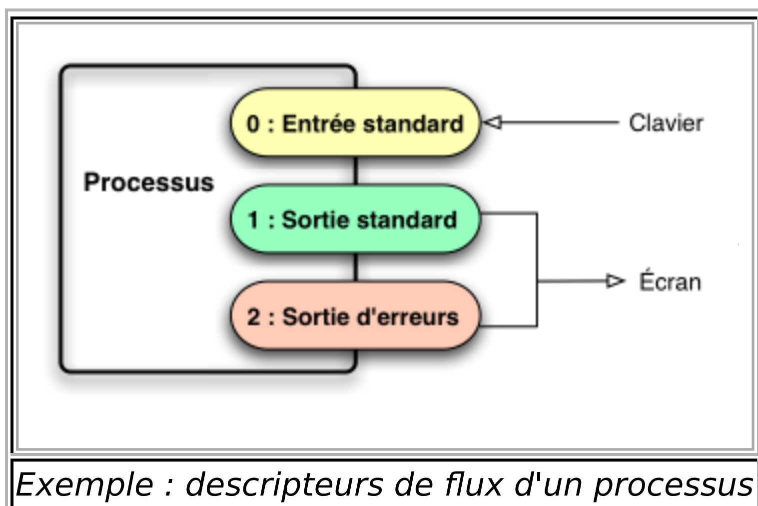
En Bash

Chaque application lancée en Bash a les flux d'entrée/sortie (stdin, stdout, stderr) que lui confère l'environnement bash (cf man : ENVIRONNEMENT D'EXÉCUTION DES COMMANDES), en plus des flux stocké (fichier) ou en transit(ex : tube nommé)

Sous les systèmes Unix, chaque processus possédera trois descripteurs de flux :

- l'entrée standard, qui permet d'envoyer des données au programme,
- la sortie standard, qui est utilisée pour afficher les résultats d'un programme,
- la sortie standard des erreurs, qui permet d'afficher les messages correspondant aux erreurs survenues lors de l'exécution du programme.

Par défaut les deux flux de sortie sont envoyés sur le terminal de l'utilisateur (écran) et l'entrée prend ses données depuis le clavier. Tout comme avec les fichiers standards, il est possible de lire (sortie) et d'écrire (entrée) sur les descripteurs de flux.



Exemple : descripteurs de flux d'un processus

Boîte à outils redirectionnels

Les trois flux standards peuvent être redirigés vers d'autres sources autres que le clavier ou l'écran. Par exemple, on peut ordonner à un processus de diriger sa sortie standard vers un fichier. Pour cela, les numéros des descripteurs de flux sont utilisés. Les outils pour réaliser ces redirections sont les suivants :

- > redirige le flux de sortie de la commande pour la placer dans un fichier. Par défaut, si rien n'est précisé, le flux redirigé est la sortie standard, i.e. > est équivalent à 1>. Pour rediriger la sortie d'erreur standard, on utilise 2>.
- < redirige le flux d'entrée de la commande pour la prendre dans un fichier,
- | redirige la sortie standard de la commande de gauche sur l’entrée standard de la commande de droite,
- >> redirige le flux de sortie de la commande pour l’ajouter à la fin d’un fichier existant.

cat tautologique

La commande `cat` recopie l'entrée standard sur la sortie standard. Pour quitter cette commande, utiliser la combinaison de touches CTRL D. Par exemple :

```
$ cat
je pense
je pense
donc
donc
je suis
je suis
```

Par défaut, `cat` prend ses données en entrée ligne par ligne. Ce qui explique qu'à chaque fois que l'on tape entrée, les caractères inscrits sur l'entrée standard via le clavier sont recopiés sur la sortie standard. Cette commande peut prendre un fichier comme entrée standard. Exemple :

```
$ cat monfichier
Affichage du contenu de mon fichier.
$
```

Redirection de la sortie standard

On peut utiliser cat pour créer un fichier texte rapidement, en utilisant la redirection de la sortie standard vers un fichier. Exemple :

```
$ cat > journal
Voici les premières lignes de mon journal.
Riches de sens, elles sont le fruit d'un intense travail.
$ cat journal
Voici les premières lignes de mon journal.
Riches de sens, elles sont le fruit d'un intense travail.
```

Souvent, dans les scripts bash, on utilise l'astuce suivante pour créer un fichier texte dynamiquement :

```
$ cat <<FIN > fichier
Ce fichier contient des données très importantes.
Son contenu est généré dynamiquement via un script.
cat cesse d'enregistrer lorsque les caractères stipulant la fin sont donnés à l'entrée
standard.
Dans ce cas, il s'agit de  :
FIN
$ cat fichier
Ce fichier contient des données très importantes.
Son contenu est généré dynamiquement via un script.
cat cesse d'enregistrer lorsque les caractères stipulant la fin sont donnés à l'entrée
standard.
Dans ce cas, il s'agit de  :
$
```

Pour ajouter du contenu à un fichier, il suffit d'utiliser >>.

Redirection de la sortie d'erreur standard

La bonne gestion des messages d’erreur est une des clés de la réussite d’un script. Il est possible de collecter proprement ces messages grâce à la redirection. Lorsqu'on lance la commande `cat` sur un fichier qui n'existe pas, on obtient un message d'erreur `No such file or directory`. On peut vérifier que ce message est un bien un message d'erreur en redirigeant la sortie d'erreur standard vers un fichier.

```
$ cat toto
cat: toto: No such file or directory
$ cat toto 2>erreur.log
$ ls
erreur.log
$ cat erreur.log
cat: toto: No such file or directory
```

Concaténation dans un fichier

Par défaut, lorsque l'on redirige la sortie standard d'un processus dans un fichier, ce fichier est écrasé par le nouveau contenu. Dans le cas d'un journal d'erreur, ceci peut s'avérer fâcheux, car la journalisation est écrasée à chaque nouveau message. Pour concaténer un message au contenu existant d'un fichier, on utilise >>. En poursuivant l'exemple du paragraphe précédent, on obtient :

```
$ cat titi 2>>erreur.log
$ cat erreur.log
cat: toto: No such file or directory
cat: titi: No such file or directory
```

Tubes

Un tube permet de rediriger la sortie standard d'une commande vers l'entrée standard d'une autre commande.

```
commande1 | commande2
```

ou pour connecter en plus la sortie d'erreur de la commande1 sur l'entrée de la commande2:

```
commande1 |& command2
```

La commande suivante `ls|grep "conf$"` liste le contenu d'un répertoire et ne sélectionne que les fichiers dont le

nom se termine par `conf`. La figure suivante illustre la redirection de la sortie standard de `ls` (la liste des fichiers d'un répertoire) vers l'entrée standard de la commande `grep`.

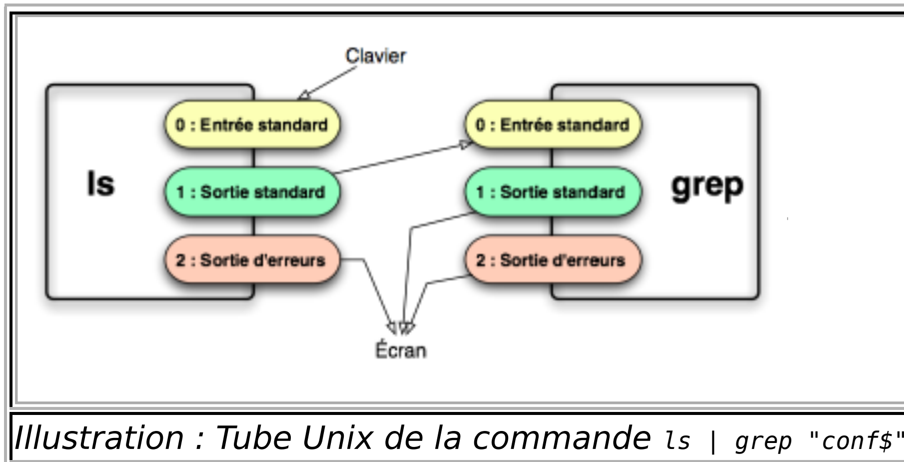


Illustration : Tube Unix de la commande `ls | grep "conf$"`

Résumé des outils de redirection

- `com > fic` redirige la sortie standard de `com` dans le fichier `fic`,
- `com 2> fic` redirige la sortie des erreurs de `com` dans le fichier `fic`,
- `com 2>&1` redirige la sortie des erreurs de `com` vers la sortie standard de `com`,
- `com < fic` redirige l'entrée standard de `com` dans le fichier `fic`,
- `com1 | com2` redirige la sortie standard de la commande `com1` vers l'entrée standard de `com2`.
- `com1 |& com2` branche ("connecte" selon le manuel bash) la sortie standard et la sortie d'erreur de `com1` sur l'entrée de `com2`

Tests

Conditions

Deux syntaxes équivalentes permettent de tester des expressions : `[ expression ]` et `test expression`. Elles renvoient toutes les deux un code de retour valant 0 si l'expression est vraie et 1 si l'expression est fausse. Attention en shell (bin bash) on ne met pas de if avec des = , on utilise les valeurs eq, lt etc...)

```
$ [ 2 = 2 ]
$ echo $?
0
$ [ 2 = 3 ]
$ echo $?
1
```

La commande `test` fonctionne de manière complétement équivalente :

```
$ test 2 = 2
$ echo $?
0
$ test 2 = 3
$ echo $?
1
```

Les opérateurs de tests disponibles sont, pour les chaînes :

- `c1 = c2`, vrai si c1 et c2 sont égaux ;
- `c1 != c2`, vrai si c1 et c2 sont différents ;
- `-z c`, vrai si c est la chaîne vide ;
- `-n c`, vrai si c n'est pas la chaîne vide.

Pour les nombres :

- `n1 -eq n2`, vrai si n1 et n2 sont égaux (equal) ;
- `n1 -ne n2`, vrai si n1 et n2 sont différents (non equal);
- `n1 -lt n2`, vrai si n1 est strictement inférieur à n2 (lower than);
- `n1 -le n2`, vrai si n1 est inférieur ou égal à n2 (lower or equal);
- `n1 -gt n2`, vrai si n1 est strictement supérieur à n2 (greater than) ;
- `n1 -ge n2`, vrai si n1 est supérieur ou égal à n2 (greater or equal).

Pour les expressions :

- `! e`, vrai si e est faux ;
- `e1 -a e2`, vrai si e1 et e2 sont vrais ;
- `e1 -o e2`, vrai si e1 ou e2 est vrai.

Test `if`

L'instruction `if` permet d'effectuer des opérations si une condition est réalisée.

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

L'instruction `if` peut aussi inclure une instruction else permettant d'exécuter des instructions dans le cas ou la condition n'est pas réalisée.

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

Il est bien sur possible d'imbriquer des `if` dans d'autres `if` et notamment des constructions telles que celle ci sont assez courantes :

```
if condition1
    then instruction(s)
else
    if condition2
        then instruction(s)
    else
        if condition3
            ...
        fi
    fi
fi
```

Pour permettre d'alléger ce type de code, ksh fournit un raccourci d'écriture : `elif`. Le code précédent pourrait être réécrit ainsi :

```
if condition1
    then instruction(s)
elif condition2
    then instruction(s)
elif condition3
    ...
fi
```

Test `case`

L'instruction `case` permet de comparer une valeur avec une liste d'autres valeurs et d'exécuter un bloc d'instructions lorsque une des valeurs de la liste correpsond.

```
case valeur_testee in
valeur1) instruction(s);;
valeur2) instruction(s);;
```

```
valeur3) instruction(s);;
* ) instruction_else(s);;
...
esac
```

Ce code est équivalent à :

```
if [ valeur_teste = valeur1 ]
    then instruction(s)
elif [ valeur_testee = valeur2 ]
    then instruction(s)
elif [ valeur_testee = valeur3 ]
    then instruction(s)
...
else
    instruction_else(s)
fi
```

Syntaxe du test

Deux syntaxes équivalentes permettent de réaliser des tests sur des opérandes:

```
[ expression ]
```

ou

```
test expression
```

Ces deux commandes renvoient un code de retour valant 0 si l'expression est vraie et 1 si l'expression est fausse. Exemple :

```
$ [ "salut" = "salut" ]
$ echo $?
0
$ [ 2 -eq 3 ]
$ echo $?
1
$ [ -f /tmp/fichier ]
$ echo "file exist"
```

La commande `test` fonctionne de manière complètement équivalente :

```
$ test "salut" = "salut"
```

```
$ echo $?
0
$ test 2 -eq 3
$ echo $?
1
```

mais certains lancements peuvent être fait sous certaine condition système:

```
pidof api && do_some_thing || exit
```

Tester une variable

Il est tout à fait possible de tester le contenu d'une variable avec les commandes `test` ou `[` :

```
[ $a = toto ]
```

la substitution de la variable par sa valeur est alors effectuée et le test est vrai si la variable contient la valeur `toto` et faux sinon. Par contre, si la variable `a` n'est pas définie lors du test, la substitution de la ligne sera :

```
[  = toto ]
```

ce qui provoquera une erreur. Il est donc préférable de toujours protéger une variable lors d'un test soit avec des guillemets :

```
[ "$a" = toto ]
```

soit avec un préfixe :

```
[ x$a = xtoto ]
```

Attention, dans ce cas un caractère espace dans $a pose quand même un problème s'il est substitué. Il faut donc préférer la solution précédente.

Tests sur les objets du système de fichiers

Les opérateurs de tests disponibles sont, pour les objets du système de fichiers :

- `[ -e $FILE ]`

```
vrai si l'objet désigné par $FILE existe dans le répertoire courant,
```

- `[ -s $FILE ]`

```
vrai si l'objet désigné par $FILE existe dans le répertoire courant et si sa taille
```

```
est supérieure à zéro,
```

- `[ -f $FILE ]`

```
vrai si l'objet désigné par $FILE est un fichier dans le répertoire courant,
```

- `[ -r $FILE ]`

```
vrai si l'objet désigné par $FILE est un fichier lisible dans le répertoire courant,
```

- `[ -w $FILE ]`

```
vrai si l'objet désigné par $FILE est un fichier inscriptible dans le répertoire
courant,
```

- `[ -x $FILE ]`

```
vrai si l'objet désigné par $FILE est un fichier exécutable dans le répertoire
courant,
```

- `[ -d $FILE ]`

```
vrai si l'objet désigné par $FILE est un répertoire dans le répertoire courant,
```

- `[ -N $FILE ]`

```
vrai si l'objet désigné par $FILE à  été modifié depuis la dernière lecture.
```

Tests sur les chaînes de caractères

Les opérateurs de tests disponibles sont, pour les chaînes :

- `[ c1 = c2 ]`

```
vrai si c1 et c2 sont égaux,
```

- `[ c1 != c2 ]`

```
vrai si c1 et c2 sont différents,
```

- `[ -z c ]`

```
vrai si c est une chaîne vide (Zero),
```

- `[ -n c ]`

```
vrai si c n'est pas une chaîne vide (Non zero).
```

Tests sur les nombres (entiers)

Pour les nombres :

- `[ n1 -eq n2 ]`

```
vrai si n1 et n2 sont égaux (EQual),
```

- `[ n1 -ne n2 ]`

```
vrai si n1 et n2 sont différents (Not Equal),
```

- `[ n1 -lt n2 ]`

```
vrai si n1 est strictement inférieur à n2 (Less Than),
```

- `[ n1 -le n2 ]`

```
vrai si n1 est inférieur ou égal à n2 (Less or Equal),
```

- `[ n1 -gt n2 ]`

```
vrai si n1 est strictement supérieur à n2 (Greater Than),
```

- `[ n1 -ge n2 ]`

```
vrai si n1 est supérieur ou égal à n2 (Greater or Equal).
```

Tests et logique

Ou comment introduire une alternative logique :

- `[ ! e ]`

```
vrai si e est faux. ! est la négation.
```

- `[ e1 -a e2 ]`

```
vrai si e1 et e2 sont vrais. -a ou le et logique (And).
```

- `[ e1 -o e2 ]`

```
vrai si e1 ou e2 est vrai. -o ou le ou logique (Or).
```

Un exemple complet

```
#!/bin/bash
read -p "Si vous etes d'accord entrez o ou oui : " reponse
if [ ! "$reponse" = "o" -a ! "$reponse" = "oui" ]; then
    echo "Non, je ne suis pas d'accord !"
else
    echo "Oui, je suis d'accord"
fi
```

L'exemple montre la manière dont on utilise des négations avec un *et* logique. En particulier, il ne faut pas utiliser de parenthèses. Le *non* (le point d'exclamation) s'applique à la proposition logique qui vient ensuite (seulement *"$reponse" = "o"*). À noter que *read -p* permet de poser une question et de stocker la réponse de l'utilisateur dans une variable.

Structures conditionnelles

if

Test simple

L'instruction `if` permet d'effectuer des opérations si une condition est réalisée.

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

Dans ce cas `condition` est une commande et l'expression conditionelle est exécutée si son code de retour vaut zéro. Par exemple

```
if pkg-config gtk+-2.0
then
    echo "gtk+ est installé sur votre système"
fi
```

exécute le programme `pkg-config` avec le paramètre `gtk+-2.0` et exécute `echo "gtk+ est installé sur votre système"` si la valeur de retour de `pkg-config` est zéro. Il est bien sur possible d'utiliser des enchainements de commandes, ainsi :

```
if pkg-config gtk+-2.0 && pkg-config gtkglext-1.0
then
    echo "gtk+ et gtkglext sont installés sur votre système"
fi
```

n'affichera quelque chose que si les deux appels à `pkg-config` ont un code de retour à 0.

Comme vu précédemment, `test` et `[` renvoient un code égal à zéro si une condition est vraie et différent de zéro sinon, il est donc possible de les utiliser avec une instrution `if` :

```
if [ "$mavariable" = "unevaleur" ]
then
    echo "ma variable a la bonne valeur"
fi
```

Test avancé

L'instruction `if` peut aussi inclure une instruction else permettant d'exécuter des instructions dans le cas ou la condition n'est pas réalisée.

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

Il est bien sur possible d'imbriquer des `if` dans d'autres `if` et notamment des constructions telles que celle ci sont assez courantes :

```
if condition1
then
    instruction(s)
else
    if condition2
    then
        instruction(s)
    else
        if condition3
            ...
        fi
    fi
fi
```

Pour permettre d'alléger ce type de code, ksh fournit un raccourci d'écriture : `elif`. Le code précédent pourrait être réécrit ainsi :

```
if condition1
then
    instruction(s)
elif condition2
then
    instruction(s)
elif condition3
    ...
fi
```

case

L'instruction `case` permet de comparer une valeur avec une liste d'autres valeurs et d'exécuter un bloc d'instructions lorsque une des valeurs de la liste correspond.

```
case valeur_testee in
valeur1) instruction(s);;
valeur2) instruction(s);;
valeur3) instruction(s);;
...
esac
```

Ce code est plus lisible que son équivalent avec `if` :

```
if [ valeur_teste = valeur1 ]
    then instruction(s)
elif [ valeur_testee = valeur2 ]
    then instruction(s)
elif [ valeur_testee = valeur3 ]
    then instruction(s)
...
fi
```

Les valeurs peuvent contenir des caractères joker, ainsi :

```
case $a in
t?t?) echo "ok";;
*) exit 1;;
esac
```

affichera `ok` si la variable `a` a une valeur correspondant au motif `t?t?`, comme par exemple `toto` ou `tata`.

Boucles

Boucle for

Première syntaxe

La boucle `for` permet de parcourir une liste de valeurs, elle effectue donc un nombre de tours de boucle qui est connu à l'avance.

```
for variable in liste_valeurs
    do instruction(s)
done
```

Par exemple, on peut utiliser la boucle `for` pour programmer un clone de la fonction `ls` :

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

Dans cet exemple, l'étoile est remplacée par tous les fichiers du répertoire courant, la boucle `for` va donc donner successivement comme valeur à la variable `i` tous ces noms de fichier. Le corps de la boucle affichant la valeur de la variable `i`, le nom de tous les fichiers du répertoire courant sont affichés successivement.

La boucle for est très souvent utilisée dans un script pour parcourir la liste des arguments fournie au script. Par exemple :

```
for i in "$@"
do
 echo "$i"
done
```

liste tous les arguments transmis au script.

Il est souvent utile de pouvoir effectuer une boucle sur une liste de valeurs, pour cela on utilise la fonction `seq`. Cette fonction prend en arguments deux entiers et renvoie la liste de tous les entiers compris entre ces bornes.

```
$ seq 1 4
1
2
3
4
```

Utilisée conjointement avec la boucle `for`, la commande `seq` permet donc d'avoir un compteur de boucle :

```
$ for i in `seq 1 4`; do echo "tour de boucle $i"; done
tour de boucle 1
tour de boucle 2
tour de boucle 3
tour de boucle 4
```

Seconde syntaxe

La boucle `for` possède une deuxième syntaxe :

```
for ((e1;e2;e3))
    do instruction(s)
done
```

Dans laquelle, `e1`, `e2` et `e3` sont des expressions arithmétiques. Une telle boucle commence par exécuter l'expression `e1`, puis tant que l'expression `e2` est différente de zéro le bloc d'instructions est exécuté et l'expression `e3` évaluée.

```
for ((i=0 ; 10 - $i ; i++))
    do echo $i
done
```

Boucles `until` et `while`

La boucle `while` exécute un bloc d'instructions tant qu'une certaine condition est satisfaite, lorsque cette condition devient fausse la boucle se termine. Cette boucle permet donc de faire un nombre indéterminé de tours de boucle, voire infini si la condition ne devient jamais fausse.

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

```
 1 #! /bin/bash
 2
 3 a_trouver=$((($RANDOM % 100) + 1))
 4
 5 echo "entrez un nombre compris entre 1 et 100"
 6 read i
 7 while [ "$i" -ne "$a_trouver" ]; do
 8         if [ "$i" -lt "$a_trouver" ]; then
 9                 echo "trop petit, entrez un nombre compris entre 1 et 100"
10         else
11                 echo "trop grand, entrez un nombre compris entre 1 et 100"
12         fi
13         read i
14 done
15 echo "bravo, le nombre etait en effet $a_trouver"
```

La syntaxe de la boucle `until` est exactement la même que celle de la boucle `while`, mais sa signification est inversée : la boucle est exécutée tant que la condition est fausse.

Scripts

Définition

Un script est une suite d’instructions, de commandes qui constituent un scénario d'actions. C’est un fichier texte que l’on peut exécuter, c’est à dire, lancer comme une commande.

Au commencement, la ligne shebang

Dans la pratique, on utilise plusieurs invites de commandes pour commander des systèmes ou des applications différentes. Par exemple, il est possible d’utiliser Perl, Python, Bash, Tcsh. Il ne suffit donc pas d'écrire une suite d’instructions pour que le système puisse l'exécuter. Il faut également préciser l’interprète pour lequel ce script est écrit. C’est l’objet de la première ligne d’un script : la ligne ”shebang”. Pour un script en shell bash, elle se présente ainsi :

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

Pour utiliser tcsh, alors il faut écrire :

```
#!/bin/tcsh
```

Exécution d'un script

Avant toute chose, il faut rendre un script exécutable. Sous Unix, il s'agit d'utiliser la commande `chmod`.

```
$ chmod +x script.sh
$ ls -l
-rwxr-xr-x   1 shadouser  shadogroup  26 Mar  5 15:31 script.sh
```

Quatre solutions sont possibles pour exécuter un script.

En utilisant ./ si l'on se trouve dans le répertoire du script :

```
$ ./script.sh
```

ou en spécifiant le chemin absolu :

```
$ /home/jojo/script.sh
```

si script se trouve dans le répertoire `/home/jojo`.

Une autre solution est de modifier la variable d'environnement PATH et d'y faire figurer le répertoire qui contient le script à exécuter. Dans ce cas, il est possible d'invoquer le script depuis n'importe quel endroit du système de fichiers.

```
$ PATH=$PATH:/home/jojo/
$ script.sh
```

Enfin, une dernière solution est d'appeler directement l'interprète et de lui transmettre le script à exécuter.

```
$ bash script.sh
```

Exemple de script

```
#!/bin/bash

for FILE in $*
do
if [ -e $FILE ]
then
echo "Le fichier $FILE est présent dans le répertoire courant."
fi
done
exit 0
```

Paramètres

Les paramètres fournit par l'utilisateur lors de l'appel de scripts sont accessibles grace aux paramètres positionnels : $0, $1, ..., $9. $0 représente le nom du script tel qu'il a été appelé, $1 est le premier argument du script, $2 le deuxième, etc. Il n'est pas possible d'accéder directement au dixième et suivant arguments de cette manière.

Une autre variable essentielle lors de la gestion des paramètres est $# qui a pour valeur le nombre d'arguments transmis lors de l'appel du script.

```
#!/bin/bash
if [ $# -lt 1 ]; then
    echo "Usage: $0 <votre prenom>"
    exit 1
fi

echo "Bonjour $1"
```

```
#!/bin/bash
#Usage ./script je vais changer -ifs : le:nouveau:IFS
printf "je restitue des arguments 1 à 1.\nje change IFS.(cf man bash IFS/paramètres
spéciaux)\n"
```

```
change=0
#cf man bash: Remplacement des paramètres
val=${@/*-ifs ? /}
args=${@/$val/}

printf "$args\n$val\n"

while test -n "$args"
do
	for v in $args
	do
	 if test $change -eq 1; then change=2; IFS="$v"; printf "<== NEW IFS: \'$IFS\'
==>\n"; fi
	 if test "$v" = "-ifs";then change=1; continue; fi
	 test $change -eq 0 && printf "==> $v\n"
	 test $change -eq 2 && change=0
	done
	args="$val"
	val=""
done
```

Enchaînements et scripts

Scripts

Exécution d'un script

Il y a deux manières d'exécuter un script, soit en rendant le script exécutable, soit en passant le fichier comme argument à la commande `ksh`.

```
chmod +x script
./script
```

```
ksh script
```

Variables spéciales

Plusieurs variables spéciales sont disponibles lors de l'exécution d'un script.

- `$0` a pour valeur le nom du script ;
- `$1` jusqu'à `$9` ont respectivement pour valeur les neuf premiers arguments du script ;
- `$#` a pour valeur le nombre d'arguments passés au script ;
- `$@` contient la liste de tous les arguments du script.
- `$*` contient la liste de tous les arguments du script (décomposée).

Attention !

En ksh, à partir du dixième argument, il faut coder la valeur numérique entre accolades, exemple : `${10}` pour le 10ème[1].

Pour tester si le script contient au moins un paramètre :

```
#!/bin/bash
if [ "x$1" = "x" ]
 then echo "argument vide"
 else echo "argument non vide"
fi
```

Enchaînements

Enchaînements simples

```
com1 & com2 & ... & comN
```

Les commandes `com1` jusqu'à `comN` sont exécutées parallèlement.

```
com1 ; com2 ; ... ; comN
```

Les commandes `com1` jusqu'à `comN` sont exécutées successivement.

Enchaînements conditionnels

```
com1 && com2 && ... && comN
```

Cet exemple va exécuter toutes les commandes `com1` jusqu'à `comN` tant que celles ci se terminent correctement.

```
com1 || com2 || ... || comN
```

Dans cet exemple les commandes `com1` jusqu'à `comN` seront exécutées successivement tant qu'aucune ne se termine correctement.

Références

1. http://www.dartmouth.edu/~rc/classes/ksh/arguments.html

Calculs

Trois méthodes permettent d'effectuer des calculs, la première utilise la syntaxe spéciale `$((operation))`, la seconde utilise la commande `let`. La troisième utilise la commande `bc`, qui accepte aussi les nombres décimaux. Taper `bc` seul sur la ligne de commande permet de passer en mode interactif. Voici comment on peut incrémenter une variable avec chacune des méthodes :

```
$ a=1
$ a=$(($a + 1))
$ echo $a
2
```

```
$ a=1
$ let "a=$a + 1"
$ echo $a
2
```

```
$ a=1
$ a=$(echo "$a+1" |bc )
$ echo $a
2
```

mais il est possible "dans certaines circonstances" (man bash,ÉVALUATION ARITHMÉTIQUE) de réaliser des opérations plus complexes:

```
a=1; let a++; echo $a
```

let *argument [argument]* (cf man bash,COMMANDES INTERNES DE L'INTERPRÉTEUR)

Calcul avancé avec `bc`

En natif, Bash ne propose que des fonctionnalités de calcul limitées (additions simples ...). `bc` permet des calculs plus complexes, avec gestion des décimales (ne pas oublier l'option `-l`, exemple :

```
$ echo "1/3" |bc -l
.33333333333333333333
```

Calcul d'une racine carrée :

```
$ echo "sqrt(2)" |bc -l
1.41421356237309504880
```

Interactions avec l'utilisateur

Lire la saisie d'un utilisateur

Les commandes suivantes permettent de gérer l'interaction avec l'utilisateur :

- la commande echo affiche des données soit sur la sortie standard, soit sur la sortie d'erreur,
- la commande read lit les valeurs entrées au clavier et les stocke dans une variable.

`read var` permet de lire une valeur entrée au clavier par l'utilisateur et de stocker cette valeur dans la variable `var`.

Exemple :

```
$ read a
toto
$ echo $a
toto
```

Dans cet exemple, `read` lit une valeur que l'utilisateur saisit au clavier en l'occurence : toto. Cette valeur est stockée dans la variable `a`. Le contenu de cette variable `a` est affiché grâce à `echo` et son contenu est effectivement `toto`.

Si aucun nom de variable n'est fourni lors de l'appel de `read`, la valeur entrée par l'utilisateur est stockée dans la variable REPLY.

Exemple

```
$ read
sidonie
$ echo $REPLY
sidonie
```

Interaction et case

Souvent, dans les scripts, on trouve la structure suivante :

```
read
case $REPLY in
valeur1) instruction(s);;
valeur2) instruction(s);;
valeur3) instruction(s);;
...
esac
```

Instruction `select`

```
echo "Etes vous un homme ou une femme ?"
select i in homme femme; do
        if [ "$i" = "homme" ]; then
                echo "Bonjour monsieur"
                break
        elif [ "$i" = "femme" ]; then
                echo "Bonjour madame"
                break
        else
                echo "mauvaise reponse"
        fi
done
```

Regex

Expressions rationnelles courantes

Caractère	Type	Explication
.	Point	n'importe quel caractère
[...]	crochets	classe de caractères : tous les caractères énumérés dans la classe
[^...]	crochets et circonflexe	classe complémentée : tous les caractères sauf ceux énumérés
^	circonflexe	marque le début de la chaine, la ligne...
$	dollar	marque la fin d'une chaine, ligne...
|	barre verticale	alternative - ou reconnaît l'un ou l'autre
(...)	parenthèses	groupe de capture : utilisée pour limiter la portée d'un masque ou de l'alternative
*	astérisque	0, 1 ou plusieurs occurrences
+	le plus	1 ou plusieurs occurrences
?	interrogation	0 ou 1 occurrence

Classes de caractères POSIX[1]

Classe	Signification
[[:alpha:]]	n'importe quelle lettre
[[:digit:]]	n'importe quel chiffre
[[:xdigit:]]	caractères hexadécimaux
[[:alnum:]]	n'importe quelle lettre ou chiffre
[[:space:]]	n'importe quel espace blanc
[[:punct:]]	n'importe quel signe de ponctuation
[[:lower:]]	n'importe quelle lettre en minuscule
[[:upper:]]	n'importe quelle lettre capitale
[[:blank:]]	espace ou tabulation
[[:graph:]]	caractères affichables et imprimables
[[:cntrl:]]	caractères d'échappement
[[:print:]]	caractères imprimables exceptés ceux de contrôle

Expressions rationnelles Unicode[2]

Expression	Signification
\A	Début de chaine
\b	Caractère de début ou fin de mot
\d	Chiffre
\D	Non chiffre
\s	Caractères espace
\S	Non caractères espace
\w	Lettre, chiffre ou underscore
\W	Caractère qui n'est pas lettre, chiffre ou underscore
\X	Caractère Unicode
\z	Fin de chaine

En Bash, le regex peut être employé depuis la version 3 après l'opérateur de comparaison =~[3].

Remarque : les utilitaires Linux suivants acceptent également du regex.

- grep, egrep
- more
- sed
- vi

Exemple

```
if [[ "Wikibooks" =~ o+ ]]; then echo "ok"; else echo "ko"; fi
ok
if [[ "Wikibooks" =~ a+ ]]; then echo "ok"; else echo "ko"; fi
ko
```

Références

1. https://www.regular-expressions.info/posixbrackets.html
2. http://www.regular-expressions.info/unicode.html
3. http://www.linuxjournal.com/content/bash-regular-expressions

Fonctions

Comme pour tout autre langage, l'utilisation de fonctions facilite le développement de scripts et la structuration de ceux-ci.

Déclaration

Pour déclarer une fonction, on utilise la syntaxe suivante:

```
maFonction(param_1, param_2,...)
{
  instructions
}
```

La déclaration d'une fonction doit toujours se situer avant son appel.

Appel et paramètres

Appel

Pour appeler une fonction, on utilise la syntaxe suivante:

```
maFonction param_1 param_2 ... param_n
```

Paramètres passés à la fonction

Ces paramètres sont bien sûrs optionnels. À l'intérieur de la fonction, ils sont représentés, respectivement, par les variables $1, $2,... , $n. $0 représente toujours le nom du **script** (et non de la fonction) qui s'exécute.

Le nombre de paramètres passés à une fonction est représenté par la variable $#

Exemple:

```
#!/bin/bash

# déclaration dune fonction
maFonction()
{ local varlocal="je suis la fonction"
  echo "$varlocal"
  echo "Nombres de paramètres : $#"
  echo $1
  echo $2
}

# appel de ma fonction
maFonction "Hello" "World!"
```

Ce qui donne le résultat suivant:

```
je suis la fonction
Nombres de paramètres : 2
Hello
World!
```

Fin

Une fonction termine son exécution lorsqu'elle n'a plus d'instructions à exécuter ou lorsqu'elle rencontre l'instruction `return` ou `exit`. Ces instructions peuvent être suivies d'un entier positif, qui correspond à la valeur de retour de la fonction. Si aucune valeur n'est spécifiée, c'est la valeur 0 qui est renvoyée.

La valeur de retour de la dernière fonction appelée est stockée dans la variable `$?`

Variables

Outre les paramètres, une fonction peut utiliser plusieurs variables:

- **Réutiliser toutes les variables globales du script**. Par défaut dans un script shell, les variables sont déclarées comme étant **globales**.
- **En déclarer de nouvelles**.
- **Déclarer des variables locales**. Pour déclarer une variable localement, il faut la faire précéder du mot clé `local`. Une telle variable ne sera utilisable que dans la fonction qui la déclare durant l'exécution de celle-ci mais sera considérée comme globale (donc connue) par les fonctions appelées à partir de cette fonction (après la déclaration locale de la variable).

Pour éviter d'accéder en lecture à une variable n'ayant pas d'existence, on peut inscrire la commande set -u en début de script.

Commandes shell

Une commande, dans le sens plus général, est un fichier exécutable ou un shell *builtin*. Par exemple, `cd, ls, echo et firefox` sont des commandes.

Les commandes *builtins* sont intégrées dans le bash. Pour le reste, les commandes disponibles sont celles installées sur le système. Pour cette raison, le détail des commandes varient d'une version à commandes disponibles et varient d'une distribution à l'autre.

A

- `alias` - Vous autorise à créer un raccourci ou des noms de commande familiers ou très utilisées
- `at` - Exécute une ligne de commande à un moment spécifié dans le futur
- `apropos` - Donne des informations sur la commande
- awk - Écrit uniquement le *n-ième* mot d'une ligne de commande en entrée et plus
- `aspell` - Vérificateur d'orthographe interactif
- `autoexpect` - Log les touches appuyées - Attention

B

- `bash` - Le *Bourne Again SHell*, un des shells
- `bunzip2` - Décompresse les fichiers compressés avec `bzip2`
- `bzip2` - Un outil de compression

C

- `cat` - Réception de chaîne de caractère depuis *stdin* ou un fichier et sortie de celui-ci par *stdout* ou par un fichier
- `chgrp` - Change le groupe du propriétaire d'un fichier
- `chmod` - Change le mode de permission d'un fichier
- `chown` - Change le propriétaire d'un fichier
- `cp` - Copie un fichier
- `cpio` - Crée des fichiers d'archives dans différents formats
- `cron` - Service planifiant des taches à exécuter à des dates spécifiques
- `crontab` - Contrôle le service `cron`
- `chsh` - Change l'interpreteur de commande
- `cut` - Affichage des colonnes d'un fichier délimité par un caractère
- `cvs` - Un système de gestion de version

D

- `date` - Affiche ou configure l'heure et la date

- dd - Transfert du contenu d'un disque de / vers un fichier et plus
- `df` - Affiche la taille libre du disque
- `diff` - Affiche la différence entre deux fichiers et plus
- dpkg - Un gestionnaire de paquets pour Debian (Ubuntu...), de bas niveau sur lequel d'autres gestionnaires plus élaborés comme `apt` et `aptitude` reposent.
- du - Affiche combien d'espace disque est utilisé par un répertoire
- `disown` - Retirer l'appartenance d'une tâche au processus courant (supprime le `pid` d'un travail). Même quand le service tourne, celui-ci ne s'arrête pas.

E

- echo - Affiche une chaîne de caractères vers la sortie standard (*stdout*) par défaut.
- `eject` - Ouvre le lecteur de cd (à noter que `eject -t` le referme)
- env - Affiche les variables d'environnement
- `exit` - Sort de la plupart des shells
- `export` - Crée et valorise une variable d'Environnement en `bash` ou `zsh`
- `expect` - Est un langage de script. Peut être lié avec Python pour des tâches automatisées. Essayez la commande `autoexpect`

F

- `fdisk` - Partitionne un disque
- `fg` - Fait passer un processus de l'arrière-plan (tâche de fond) à l'avant-plan
- `file` (commande) - Détermine le type d'un fichier
- `find` - Trouve des fichiers selon leur nom, taille, date de dernière modification ou autres
- `finger` - Cherche si quelqu'un est connecté
- `ftp` - Utiliser le protocole de transfert de fichiers (FTP) en mode texte

G

- g++ - Compile un fichier source écrit dans le langage C++
- gcc - Compile un fichier source écrit dans le langage C
- `gftp` - Logiciel graphique utilisant le protocole de transfert de fichiers FTP
- grep - Recherche un texte ou un motif textuel dans un ou plusieurs fichiers texte
- `groups` - Montre à quels groupes l'utilisateur appartient
- `gvimdiff` - `diff` graphique (montre les différences entre deux fichiers texte)
- `gunzip` - Décompresse un fichier compressé par `gzip`
- `gzip` - Compresse un fichier

H

- `halt` - Arrête l'ordinateur (*root*)
- head - Affiche seulement les *n* premières lignes d'un fichier
- hexdump - Affiche le contenu d'un fichier sous forme hexadécimale
- `history` (commande) - Affiche l'historique des commandes utilisées dans

l'interprétateur de commande (shell)
- `hostname` - Affiche le nom de l'ordinateur

I

- `id` - Affiche les numéros d'identification de l'utilisateur et des groupes auxquels il appartient
- `ifconfig` - Affiche entre autres l'adresse IP de l'utilisateur
- `info` - Affiche les informations à propos d'une commande
- `init` - Redémarre ou change le niveau d'exécution du système
- `iptables` - Montre la configuration de votre pare-feu
- `iptraf` - [1]suivi des adresses IP dynamiques sur un LAN (*Limited Area Network*)

J

- `jobs` - Donne une liste des travaux courants en arrière plan (processus)

K

- `kill` - Tue un processus
- `killall` - Tue tous les processus d'un nom donné

L

- `ldd` - Affiche les bibliothèques dynamiques dont dépend un exécutable
- `less` - Affiche une sortie dans laquelle vous pouvez vous déplacer et effectuer des recherches. C'est un "pageur".
- `ln` - Établit un lien vers un fichier
- `ls` - Liste le contenu d'un fichier répertoire
- `lsmod` - Établit la liste des modules chargés par le noyau
- `lsof` - Établit la liste des fichiers ouverts et des *sockets* à l'écoute
- `look` - Vérification rapide de l'orthographe

M

- `make` - Permet de compiler des logiciels et plus
- `man` - Fournit une aide à propos des questions que vous n'avez jamais voulu poser
- `md5sum` - Calcule la somme de contrôle d'un fichier (permet par exemple de vérifier l'intégrité d'une copie en comparant sa `md5sum` à celle de la source)
- `mkdir` - Crée un fichier répertoire
- `mkfs` - Formate un périphérique de stockage (crée un système de fichiers)
- `minicom` - Permet de communiquer suivant le protocole RS232 (port série)
- `more` - Comme `less`, il s'agit d'un "pageur"
- `mount` - Prépare un périphérique de stockage à accepter la lecture et l'écriture
- `mv` - Déplace un fichier, permet aussi de le renommer (on le déplace au même endroit

en changeant son nom)

N

- `netcat` - Envoie des bits sur le réseau
- `netstat` - Obtient des informations sur les *sockets* à l'écoute et sur les ports ouverts
- `nice` - Fixe la priorité d'exécution d'un processus
- `nm` - Établit la liste des noms de fonctions d'un fichier objet

O

- `objdump` - Affiche les informations relatives à un fichier objet
- `openssl` - Permet d'utiliser les fonctions de cryptographie qui suivent les protocoles réseaux SSL et TLS

P

- `passwd` - Change votre mot de passe ou celui d'un autre utilisateur
- `ping` - Indique si un ordinateur en particulier est fonctionnel sur un réseau
- `ps` - Affiche la liste des processus lancés à l'état actuel
- `pwd` - Affiche le dossier actuel où se trouve l'utilisateur
- `paste` - Fusionne des lignes en un fichier en les combinant horizontalement

Q

- `quota` - Gère la quantité de ressources qu'un utilisateur est autorisé à utiliser

R

- `rar` - Fichiers ou dossiers compressés, de type `.rar`
- `read` - Lit une ligne depuis votre clavier
- `reboot` - Redémarre l'ordinateur
- `rename` - Renomme des fichiers (pour un fichier, il est plus simple de passer par `mv`)
- `rm` - Efface un fichier
- `route` - Gère la table de routage de votre réseau
- `rpm` - Gère les paquets sous les distributions Redhat et Fedora
- `rsync` - Permet la synchronisation de vos fichiers à travers un réseau

S

- `scp` - Effectue une copie sécurisée à travers un réseau - données chiffrées
- `screen` - Permet de créer plusieurs terminaux à partir d'un seul
- `sed` - Effectue des modifications sur des chaînes de caractères
- `setenv` - Modifie la valeur d'une variable d'environnement d'un shell C

- `shutdown` - Éteint ou redémarre le système
- `sleep` - Retarde d'une certaine quantité de temps à déterminer
- `ssh` - Permet de se connecter de façon sécurisée à un hôte distant
- `su` - Change l'identité de l'utilisateur
- `sudo` - Exécute une commande sous l'identité d'un autre utilisateur (le plus souvent "root" - `/etc/sudoers`)

T

- `tail` - Ne montre que les *n* dernières lignes d'un fichier
- `tar` - Archive des fichiers selon un certain format
- `tcpdump` - Purge le trafic sur le réseau TCP
- `tee` - Duplique la sortie standard vers un fichier
- `time` - Indique le temps nécessaire à une commande pour s'achever
- `top` - Montre les processus utilisant le plus de ressources du processeur
- `touch` - Crée un fichier ou modifie son étiquette temporelle
- `traceroute` - Montre la route empruntée par un paquet sur un réseau
- `tac` - Imprime un fichier en inversant l'ordre des lignes, à l'inverse de `cat` (`cat` X `tac`)

U

- `ulimit` - Lit ou écrit certaines limitations pour le processus en cours
- `umount` - Démonte un périphérique (nécessite souvent les droits du groupe *sudoers*)
- `uname` - Affiche la version du noyau en fonction en plus d'autres détails
- `uniq` - Supprime une ligne en doublon dans un fichier trié
- `unzip` - Décompresse des fichiers
- `unrar` - Décompresse des fichiers d'archivage `rar`
- `uptime` - Affiche la date et l'heure de la dernière mise en route de l'ordinateur
- `useradd` - Ajoute un utilisateur
- `userdel` - Supprime un utilisateur
- `usermod` - Modifie un utilisateur

V

- `vim` - Éditeur de texte pur, fonctionnement modal. À distinguer d'un traitement de texte.
- `Vgcreate` - Créer des groupes de volumes LVM
- `Vgdisplay` - Groupes d'affichage de volumes LVM
- `Vgs` - Afficher des informations sur les groupes de volumes LVM
- `Vgscan` - Rechercher des groupes de volumes LVM
- `vmstat` - Affiche des informations sur l'activité des processus, de la mémoire, des signaux, du processeur, des disques durs, des entrées et sorties, etc.

W

- `who` - Affiche qui est connecté sur le système
- `which` - Affiche le chemin d'un fichier exécutable
- `whoami` - Affiche votre véritable nom d'utilisateur
- `wc` - *Word count* ; permet de compter des bits, des caractères, des lignes, des mots dans un fichier
- `write` - Envoie un message à un autre utilisateur connecté

X

- `xargs` - Exécute des commandes en utilisant pour arguments les données issues de l'entrée standard (pratique pour utiliser des commandes comme des filtres alors que ce n'en sont pas)
- `xev` - Affiche tous les événements relatifs à une fenêtre
- `xkill` - “Tue” une fenêtre de processus en le déconnectant du serveur X
- `xosview` - Affiche l'activité du processeur, de la mémoire, des disques-durs et plus encore...

Y

- `yacc` - Générateur de programmes d'analyse syntaxique écrits en C
- `yes` - Affiche continuellement une chaîne de caractères
- `yum` - Un gestionnaire de paquets utilisé par les distributions Redhat et Fedora
- `yast` - Un gestionnaire de paquets utilisé par la distribution SUSE

Z

- `zip` - Compresse un fichier

Commandes ksh

Liste des commandes Bash compatibles ksh.

Commandes d'aide

- `man` : *Manual* - Obtenir le manuel d'une commande.

Écran

- `clear` : efface le contenu affiché à l'écran.
- `more` : Affiche le contenu d'un fichier texte, page par page (la page correspond à la taille du terminal).

Système de fichiers

- `cd` : *Change Directory* - permet de se déplacer dans le système de fichiers.
- `cp` : *CoPy* - Copie un fichier ou copie une liste de fichiers dans un autre répertoire en conservant leur nom.
- `ls` : *LiSt* - Affiche la liste des fichiers dans le dossier courant ou d'un autre dossier.
- `mkdir` : *MaKe DIRectory* - Crée un ou plusieurs répertoires.
- `mv` : *MoVe* - Déplace (ou renomme) un fichier, y compris si c'est un répertoire ou déplace une liste de fichiers dans un autre répertoire en conservant leur nom.
- `pwd` : *Print Working Directory* - permet d'afficher l'endroit où l'on se trouve actuellement dans le système de fichiers.
- `rm` : *ReMove* - Supprime un/des fichier(s) ou des répertoires.
- `rmdir` : *ReMove DIRectory* - Supprime un ou plusieurs répertoires s'ils sont vides.
- `touch`: modifie le *timestamp* d'un fichier existant. S'il n'existe pas un fichier vide est créé.

Recherche

- `egrep` : même commande que *grep* mais plus riche en possibilités.
- `find` : Recherche récursive, à partir d'un répertoire, de fichiers ayant des caractéristiques données.
- `grep` : Affiche les lignes qui contiennent une expression régulière donnée.

Gestion de texte

- `cat` : *CATenate* - Concatène des fichiers texte. Peut aussi servir à simplement afficher ou lire un fichier.
- `cut` : Supprime une partie des lignes d'un fichier selon un critère.
- `echo` : Affiche une ligne de texte donnée en paramètre.
- `expr` : Évalue une expression (mathématique ou sur une chaîne de caractères)

- `head` : Affiche les premières lignes d'un fichier (Voir *tail*).
- `join` : Fusionne les lignes de deux fichiers contenant un ou plusieurs champs identiques.
- `read` : Lit une chaîne de caractères à partir de l'entrée standard.
- `sed` : *Stream EDitor* - Effectue des transformations sur un flux de texte.
- `sort` : Trie les lignes d'un texte selon l'ordre alphabétique ou numérique.
- `tail` : Affiche les dernières lignes d'un fichier (Voir *head*).

Permissions

- `chmod` : *CHange MODes* - Change les permissions en lecture, écriture et/ou exécution d'un fichier.
- `chown` : *CHange OWNer* - Change le propriétaire, et éventuellement le groupe propriétaire d'un fichier.

Processus

- `ps` : *Process Status* - Affiche les processus en cours d'exécution.
- `kill` : Envoyer un message à un processus donné, généralement pour y mettre fin.

Gestion de disques

- `mount` : permet de monter un système de fichier.
- `umount` : permet de démonter un système de fichier.

GFDL

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page de couverture.

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Programmation_Bash/Version_imprimable&oldid=492333 »

La dernière modification de cette page a été faite le 18 novembre 2015 à 22:05.

Les textes sont disponibles sous licence Creative Commons attribution partage à l’identique ; d’autres termes peuvent s’appliquer.
Voyez les termes d’utilisation pour plus de détails.