Formation

Notre expertise est votre avenir

Unix / Linux - Programmation Shell

Référence Linux : UNLI-SH 1710

SOMMAIRE

Unix - Linux

1. Rappels

Objectifs

- Rappel des commandes et concepts nécessaires pour aborder les chapitres suivants.
- Arborescence, fichiers et répertoires, droits d'accès, redirections et pipes, variables et paramètres, processus et Shell scripts.

Notez que l'objectif de ce chapitre est de rappeler les fondamentaux du shell, ces notions sont supposées acquises lors du stage précédent ou par expérience.
Les différents éléments seront abordés rapidement, ils ne seront pas nécessairement détaillés.

1.1 La ligne de commande

Pour être en mesure de réaliser l'interprétation, le shell (sh, ksh, bash...) exige une structure précise de la ligne de commande.

Une ligne de commande peut être réalisée au moyen d'une commande simple ou en combinant des options et ou des paramètres.

```
commande   [-option(s) | --option(s)]   [par-1]   …   [par-n]
```

Commande avec options et paramètre

```
$ ls -lai /home/user1/tmp
total 20
543628 drwxr-xr-x.  5 user1 gp1 4096  1 déc.  17:54 .
389377 drwx------. 45 user1 gp1 4096  1 déc.  17:53 ..
519385 -rw-r--r--.  1 user1 gp1    0  1 déc.  17:54 fic1
519388 -rw-r--r--.  1 user1 gp1    0  1 déc.  17:54 fic2
519389 -rw-r--r--.  1 user1 gp1    0  1 déc.  17:54 fic3
543629 drwxr-xr-x.  2 user1 gp1 4096  1 déc.  17:54 rep1
543630 drwxr-xr-x.  3 user1 gp1 4096  1 déc.  17:59 rep2
543631 drwxr-xr-x.  2 user1 gp1 4096  1 déc.  17:54 rep3
```

1.2 Les types de fichiers

Tout est fichier pour Unix ou Linux !

1.2.1 Fichier ordinaire

Un fichier contient un ensemble d'informations.

Unix et Linux ne reconnaissent pas de structure particulière à un fichier ordinaire.

Un fichier n'est qu'une simple suite d'octets enregistrés sur disque.

Cependant le fichier peut être organisé selon le type d'informations qu'il contient, lignes de texte ASCII, tables de base de données, trames de vidéo… Ainsi on ne pourra accéder à un fichier qu'en utilisant la commande ou l'outil correspondant au type de données qu'il contient.

1.2.2 Fichier répertoire (directory, dossier)

Un répertoire est un fichier particulier qui contient un ensemble de noms de fichiers. Chaque nom de fichier est associé à un numéro permettant l'accès physique au fichier sur disque.

1.2.3 Fichier spécial

Un fichier spécial est un nom associé à un élément matériel du système tel qu'un périphérique, un mécanisme de communication…

Ce type de fichier sera étudié au cours du stage d'administration.

Exemples de fichiers spéciaux :

- Le clavier est un fichier d'entrée
- Une fenêtre est un fichier de sortie
- L'imprimante est un fichier de sortie

1.2.4 Fichier lien symbolique

Un lien symbolique est un fichier faisant référence à un autre fichier.

1.3 Les répertoires

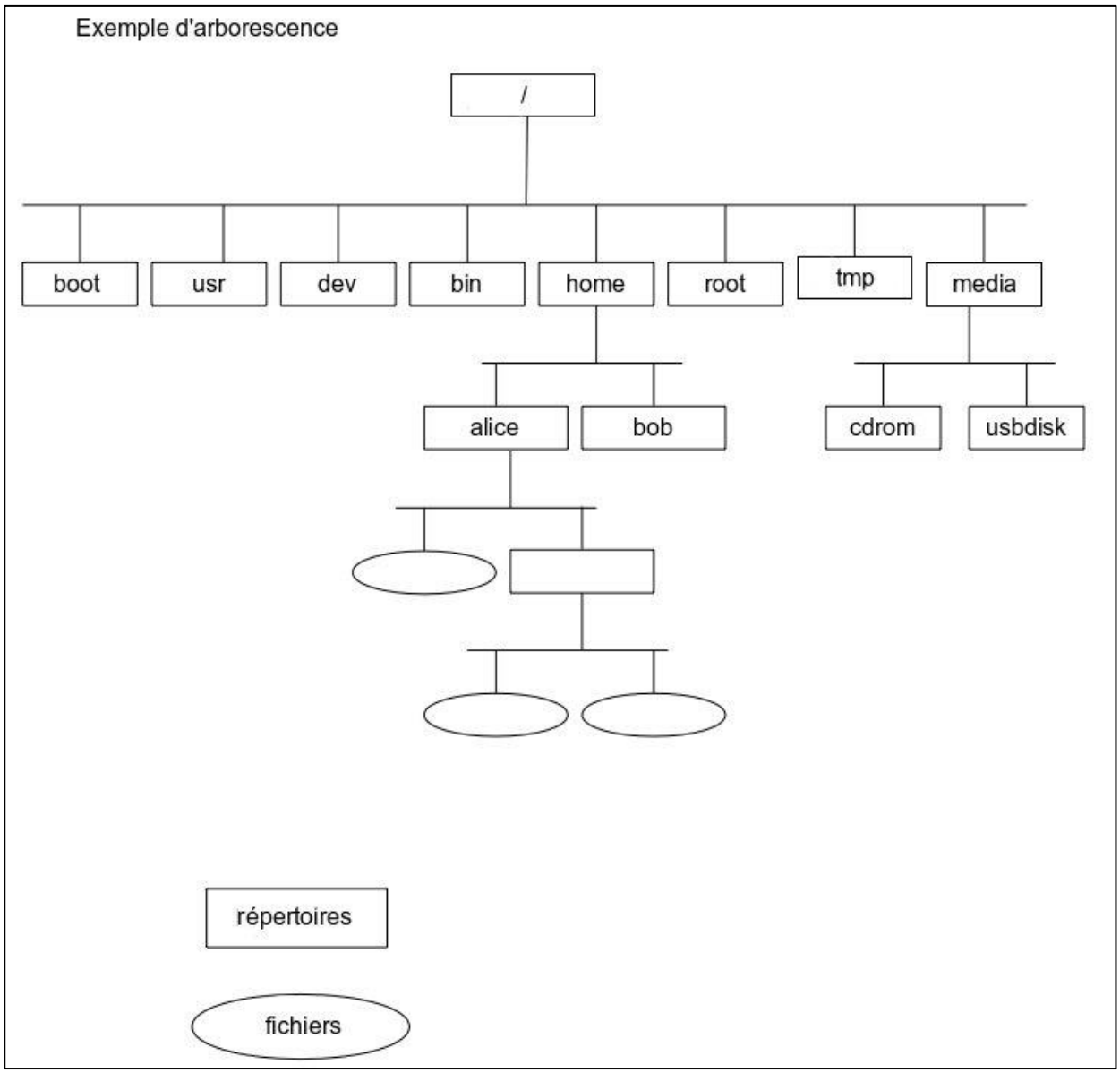
1.3.1 Arborescence

Les fichiers Unix ou Linux sont organisés en une **arborescence** unique.

Le répertoire qui amorce cette arborescence est appelé **répertoire racine** ou **root directory**.

Ce répertoire racine est représenté par **/** (barre oblique droite ou slash).

L'ensemble des répertoires et fichiers est vu par l'utilisateur dans une seule et unique arborescence.



1.3.2 Répertoire de connexion

Chaque utilisateur possède un **répertoire de connexion** encore appelé **répertoire d'accueil** ou **home directory**.

Par défaut ce répertoire porte le nom de l'utilisateur et se trouve dans le répertoire /home.

Lorsqu'un utilisateur se connecte, il est automatiquement positionné dans son répertoire de connexion.

1.3.3 Répertoire de travail

Un utilisateur peut se déplacer dans l'arborescence et se positionner dans le répertoire de son choix. Ce répertoire sera alors son **répertoire de travail** ou **répertoire courant**.

La commande **pwd** (Print Working Directory) permet d'afficher le nom du répertoire de travail dans lequel se trouve positionné l'utilisateur à un instant donné.

```
$ pwd
/home/user1
```

Le **répertoire courant** est représenté par un nom de « fichier » particulier nommé "**.**" (point). (Nous verrons l'utilité de cette syntaxe plus loin dans le cours.)

1.3.4 Changer de répertoire

La commande **cd** (Change Directory) permet à un utilisateur de se déplacer dans l'arborescence et de se positionner dans le répertoire de son choix.

Le **chemin d'accès** du répertoire de destination sera utilisé en paramètre de cette commande.

1.3.5 Le chemin d'accès absolu

```
$ pwd
/home/user1
$ cd /home/user1/programmes/ressources
$ pwd
/home/user1/programmes/ressources
```

1.3.6 Le chemin d'accès relatif

```
$ pwd
/home/user1
$ cd programmes/ressources
$ pwd
/home/user1/programmes/ressources
```

1.3.7 Répertoire père

Il est possible de remonter dans l'arborescence grâce à un « fichier » particulier nommé "**..**" (point point), qui représente le répertoire immédiatement supérieur.

```
$ pwd
/home/user1/program/resource
$ cd ..
$ pwd
/home/user1/program
$ cd ../..
$ pwd
/home
```

1.3.8 Positionnement automatique sur le répertoire de connexion

Pour retourner directement sous son répertoire de connexion à partir de n'importe quel endroit de l'arborescence, il suffit de taper cd.

```
$ pwd
/home/user1/program/resource
$ cd
$ pwd
/home/user1
$
```

1.3.9 Déplacements rapides

On peut aussi utiliser le caractère **~** (tilde) pour indiquer de manière implicite le chemin d'accès au répertoire de connexion.

On peut aussi utiliser le caractère **–** (tiret) pour revenir sous le répertoire précédent

1.3.10 Lister le contenu d'un répertoire

La commande ls (LiSt content of a directory) permet d'afficher la liste des fichiers contenus dans le répertoire.

ls	affichage des fichiers contenus dans le répertoire courant
ls /usr/bin	affichage des fichiers contenus dans le répertoire nommé

Quelques options

ls -a	affiche tous les fichiers y compris les fichiers cachés
ls -i	affiche le numéro du fichier à gauche de son nom
ls -l	affichage au format long
ls -F	ajoute un caractère à chaque nom de fichier pour indiquer son type. (/=répertoire, *=exécutable...)
ls -t	trie les fichiers en fonction de la date et non pas de l'ordre alphabétique
ls --color	utilise les couleurs pour distinguer les types de fichiers

```
$ ls -l
total 116
drwxr-xr-x    3 user1    user1        4096 jan  9 17:01 Desktop
-rw-r--r--    1 root     root        13795 mai  8  2017 LICENSE
-rw-r--r--    1 root     root        12589 mai  8  2017 README
-rw-r-----    1 user1    user1           0 mai 22  2017 aze
-rw-r--r--    1 user1    user1           0 mai 22  2017 azer
-rw-r--r--    1 user1    user1           0 mai 22  2017 azert
drwxr-xr-x    2 user1    user1        4096 mai 22  2017 bin
-rw-------    1 user1    user1         375 sep 26 10:24 lettre
-rw-r--r--    1 user1    user1          16 mai 22  2017 fic
drwxr-xr-x    3 root     root         4096 sep  3 10:54 help
-rwxr--r--    1 user1    user1          18 mai 22  2017 lld
-rw-------    1 user1    user1       19537 sep 27 16:42 mbox
-rwxr--r--    1 user1    user1         109 mai 22  2017 menu
drwxr-xr-x    5 root     root         4096 sep  3 10:56 prog
-rwxr--r--    1 user1    user1          39 mai 22  2017 reponse
-rwxr-----    1 user1    user1          48 mai 22  2017 reponse~
lrwxrwxrwx    1 root     root           25 sep  3 10:55 setup -> prog/setup
drwxr-xr-x   15 root     root         4096 sep  3 10:54 share
drwxr-xr-x    2 user1    user1        4096 mai 22  2017 src
-rwxr--r--    1 user1    user1          19 mai 22  2017 titi
drwx------    2 user1    user1        4096 aoû  7 16:55 tmp
-rwxr--r--    1 user1    user1          19 mai 22  2017 toto
drwxr-xr-x   18 root     root         4096 sep  3 10:55 user
```

1.3.11 Créer un répertoire

La commande **mkdir** permet de créer un répertoire.

```
$ mkdir rep1
$ ls -l
...
drwxr-xr-x    2 user1    user1        4096 jan 30 12:45 rep1/
...
```

1.3.12 Supprimer un répertoire

La commande rmdir permet de supprimer un répertoire vide.

```
$ rmdir rep1
```

Si le répertoire contient des fichiers cette commande est rejetée.

Pour supprimer un répertoire non vide ainsi que tout ce qu'il contient on utilisera la commande **rm**.

```
$ rm -r rep1
```

1.4 Les fichiers

1.4.1 Les types de fichiers ordinaires

La commande **file** permet d'analyser un fichier et affiche la nature des informations contenues.

```
$ file /bin
/bin: directory
$
$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.18, stripped
$
$ file /etc/passwd
/etc/passwd: ASCII text
$
```

1.4.2 Lecture d'un fichier ordinaire ascii

Les commandes **less, more** et **pg** (AIX), permettent d'afficher, page à page, le contenu d'un fichier ascii.

- **more**

Commande d'origine BSD

<Entrée>	avance d'une ligne
<Espace>	avance d'une page
<b>	recule d'une page
<q >	sortie de more

- **less**

Alternative à more (nombreuses améliorations)

-? ou **--help**	informations sur la commande ou <h>
<f>	avance d'une page
<b>	recule d'une page
<e>	avance d'une ligne
<y>	recule d'une ligne
<q>	sortie de less

1.4.3 Création d'un fichier ordinaire

Sous Unix et sous Linux il n'existe pas de commande spécialisée pour créer des fichiers.

Chaque utilitaire (copies, éditeurs de textes…) crée des fichiers si besoin.

Bien que les mécanismes utilisés ci-dessous soient détaillés plus tard dans ce cours, voici des moyens simples de créer un fichier ordinaire.

Création d'un fichier vide :

```
$ >fichier1
$
$ ls -l
...
-rw-r--r--    1 user1    grp1       0  7 mai  13:09   fichier1
...
$
```

Ou bien :

```
$ touch fichier2
$
$ ls -l
...
-rw-r--r--    1 user1    grp1       0  7 mai  13:10   fichier2
...
$
```

Attention

Dans les exemples ci-dessus,

si fichier1 existait déjà il serait écrasé... sans avertissement !!!

si fichier2 existait déjà il conserverait ses données.

1.4.4 Suppression d'un fichier ordinaire

La commande rm (ReMove) permet de supprimer un fichier ordinaire.

Si l'on ne possède pas les autorisations nécessaires, la commande sera rejetée.

```
rm nom_de_fichier
```

Options utiles :

-i	interactif, l'utilisateur doit confirmer ou non la suppression du fichier option souvent par défaut en Linux.
-f	forcé, ne tient pas compte des protections du fichier (dangereux !)
-r	récursif, supprime récursivement le contenu des répertoires.

```
$ rm -i fichier2
rm: détruire 'fichier2'? o
$
```

1.4.5 Caractères génériques

Le shell permet l'emploi de caractères génériques qui sont utilisés pour construire des noms de fichiers à partir des fichiers sous le répertoire courant ou le répertoire cité.

Ces caractères génériques peuvent être utilisés dans toutes les commandes où des noms de fichiers sont cités (ls, less, cp, mv, rm …).

Signification des caractères génériques en fonction des noms des fichiers présents sous le répertoire courant ou le répertoire cité :

*****	Remplace n'importe quelle chaîne de caractères y compris la chaîne vide
?	Remplace un caractère quelconque
[ ]	Remplace un des caractères contenus entre les crochets
[!]	Remplace un des caractères non contenus entre les crochets.

1.5 Les droits d'accès

1.5.1 Généralités

Pour l'accès à un fichier, les utilisateurs sont classés en trois ensembles :

- Le propriétaire du fichier (en général l'utilisateur qui l'a créé),
- Les utilisateurs du groupe propriétaire du fichier,
- Les utilisateurs des autres groupes.

r w x suid	r w x sgid	r w x sticky-bit
propriétaire (user - u)	groupe (group - g)	autres (other - o)

A chaque ensemble est associé trois types d'autorisations dont la signification dépend du type du fichier

Fichiers ordinaires

- **r** : autorisation de lire
- **w** : autorisation d'écrire
- **x** : autorisation d'exécuter

Répertoires

- **r** : autorisation de lister
- **w** : autorisation de créer, renomer ou supprimer des fichiers
- **x** : autorisation de passer dans le répertoire

```
$ ls -l
total 120
drwxr-xr-x    3 user1    user1        4096 jan  9 17:01 Desktop
-rw-r--r--    1 root     root        13795 mai  8  2017 LICENSE
-rw-r--r--    1 root     root        12589 mai  8  2017 README
-rw-r--r--    1 user1    user1           0 mai 22  2017 azer
drwxr-xr-x    2 user1    user1        4096 mai 22  2017 bin
-rw-------    1 user1    user1         375 sep 26 10:24 dead.letter
-rw-r--r--    1 user1    user1          16 mai 22  2017 fic
drwxr-xr-x    3 root     root         4096 sep  3 10:54 help
-rwxr--r--    1 user1    user1          18 mai 22  2017 lld
-rw-------    1 user1    user1       19537 sep 27 16:42 mbox
-rwxr--r--    1 user1    user1         109 mai 22  2017 menu
-rw-r--r--    1 user1    user1           0 jan 30 14:24 mon-fichier1
drwxr-xr-x    5 root     root         4096 sep  3 10:56 program
drwxr-xr-x    2 user1    user1        4096 jan 30 12:45 rep1
-rwxr--r--    1 user1    user1          39 mai 22  2017 reponse
```

1.5.2 Droits d'accès supplémentaires

- **suid** set user-id

Positionné sur un fichier exécutable, le programme s'exécutera en prenant l'identité du propriétaire du fichier.

Exemple de la commande passwd possédant le **suid** permettant à tout utilisateur de modifier son mot de passe dans des fichiers qu'il n'a pas le droit de modifier directement :

```
$ ls -l /usr/bin/passwd
-r-s--x--x    1 root    root     13044 mai 21   2013 /usr/bin/passwd

$ passwd
Changer le mot de passe pour user1
Entrer le mot de passe :
Nouveau mot de passe:
Nouveau mot de passe:
...

$ ls -l /etc/passwd
-rw-r--r--    1 root    root      2677 14 août 15:58  /etc/passwd

$ ls -l /etc/shadow
-r--------  1 root    root      2130 14 août 15:58  /etc/shadow
```

- **sgid** set group-id

Positionné sur un fichier exécutable, le programme s'exécutera en prenant l'identité du groupe propriétaire du fichier.

- **Sticky-bit ou save-text**

Sert uniquement pour les répertoires. Il indique que seuls le propriétaire du répertoire, et le propriétaire d'un fichier qui s'y trouve ont le droit de supprimer ce fichier. Typiquement utilisé pour les répertoires comme */tmp* ayant une autorisation d'écriture générale.

```
$ ls -ld /tmp
drwxrwxrwt  41 root     root  4096 14 août 15:52  /tmp
$
```

1.5.3 Modification des droits d'accès `chmod`

La commande *chmod* permet de modifier les droits d'accès à un fichier. Seuls root ou le propriétaire d'un fichier peuvent en changer les droits.

Deux syntaxes sont possibles en octal ou en symbolique.

- **En octal**

```
chmod valeur_octale fichier
```

Valeurs octales :

r	w	x	r	-	x	r	-	-
1	1	1	1	0	1	1	0	0
	7			5			4	

```
$ chmod 754 prog1
```

- **En symbolique**

```
chmod [qui] opération [permission] fichier
```

qui :	**u**	user
	g	group
	o	others
	a	all
opération :	**+**	ajoute des droits
	-	enlève des droits
	=	positionne des droits, enlève les précédents
permission :	**r**	read, lecture
	w	write, écriture
	x	exécution
	s	positionne suid ou sgid (fonction de [qui])
	t	positionne sticky-bit

```
$ chmod u+x,g-w,o=r  prog1
```

1.5.4 Changement de groupe et de propriétaire

La commande `chgrp` permet de changer le groupe propriétaire d'un fichier.

La commande `chown` permet de changer soit simplement le propriétaire soit à la fois le propriétaire et le groupe d'un fichier.

```
chgrp groupe fichier
chown utilisateur fichier
chown utilisateur:groupe fichier
```

Les commandes chgrp et chown s'appliquent à tous les types de fichiers (fichiers ordinaires, répertoires, fichiers spéciaux…).

Seul root a le droit de changer le propriétaire d'un fichier.

Un utilisateur peut changer le groupe propriétaire d'un fichier à la condition d'appartenir à ce nouveau groupe ou d'en être invité (cf. la commande id).

root peut attribuer tout fichier à n'importe quel utilisateur ou groupe.

```
$ ls -l azer
-rw-r--r--    1 user1    user1           0 mai 22  2017 azer
$ chgrp user2 azer
chgrp: Vous n'êtes pas membre du groupe `user2'.: Opération non
permise
$ id
uid=501(user1) gid=501(user1) groupes=501(user1),503(xgrp)
$ chgrp xgrp azer
$ ls -l azer
-rw-r--r--    1 user1    xgrp            0 mai 22  2017 azer
$
$ su                      # on passe administrateur root
Password:*****
#
# chown user2 azer
# exit                    # on retourne à la connexion user1
$
$ ls -l azer
-rw-r--r--    1 user2    xgrp            0 mai 22  2017 azer
```

1.6 Commandes et programmes

1.6.1 Règle de recherche d'une commande

Pour exécuter une commande le shell recherche la commande dans les répertoires cités dans la variable `PATH`.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:
/home/user1/bin:
```

1.6.2 Shell-script

Un shell script ou procédure, est un fichier contenant des lignes de commandes. Pour qu'il soit utilisable il faut rendre ce fichier exécutable.

```
$ more mon_programme
echo "Bonjour"
$ ls -l mon_programme
-rw-r--r--. 1 pat gp1 15 14 août  14:33 mon_programme
$ chmod u+x mon_programme
$ ls -l mon_programme
-rwxr--r--. 1 pat gp1 15 14 août  14:33 mon_programme
```

1.6.3 Lancement d'un programme utilisateur

Un programme qui n'est pas une commande système ne sera pas trouvé automatiquement.

```
$ mon_programme
bash: mon_programme : commande introuvable
```

Pour exécuter un programme on peut indiquer son chemin d'accès…

```
$ ./mon_programme
Ceci est mon programme
```

…ou bien modifier la variable `PATH`.

```
$ PATH=$PATH:.
$ mon_programme
Ceci est mon programme
```

1.7 Opérations sur les fichiers

1.7.1 Copie d'un fichier ordinaire

```
cp source destination
```

Nous appelons source *le fichier qui va être copié et* destination *la copie du fichier d'origine.*

```
cp fichier1 fichier2 ... répertoire
cp * répertoire
```

On n'emploiera les caractères génériques exclusivement dans le champ *source* de la commande, sous peine de non-sens fâcheux.

Options utiles

-i	Demande confirmation avant d'écraser des fichiers existants
-R	Copie récursivement les répertoires
-f	Efface les fichiers existant dans le répertoire cible
-p	Conserve propriétaire, groupe, droits et horodatages du fichier original

1.7.2 Déplacer un fichier

```
mv source destination
mv fichier1 fichier2 … répertoire
mv * répertoire
```

Options utiles

-i	demande confirmation avant d'écraser des fichiers existants
-f	efface les fichiers existant dans le répertoire cible

1.7.3 Renommer un fichier

```
mv ancien-nom nouveau-nom
```

1.7.4 Créer un lien sur un fichier

- **Création d'un lien physique**

```
ln source destination
```

Le compteur de liens est une des caractéristiques du fichier, il est contenu dans l'inode.

La commande **`ls -l`** affiche la valeur du compteur de liens d'un fichier.

La commande rm permet de supprimer un lien. Elle décrémente le compteur de liens.

Pour supprimer un fichier il est nécessaire de supprimer tous les liens à ce fichier.

- **Création d'un lien symbolique avec l'option -s**

```
ln -s source lien
```

`source` Le fichier *source* existe, il a son propre inode

`lien` Le fichier *lien* est créé par la commande, son inode est différent de celui du fichier source. Son contenu est le chemin d'accès au fichier source.

1.8 Rechercher un fichier : la commande find

La commande find permet de rechercher des fichiers dans l'arborescence.

```
find [chemins] [-critères] [actions]
```

Le répertoire de recherche par défaut est le répertoire de travail (.)

```
$ pwd
/home/pat/tmp

$ find . -name fic1
./fic1

$ find -name fic1
./fic1

$ find /home/pat/tmp -name fic1
/home/pat/tmp/fic1

$ cd /

$ find . -name core
find: ./mnt/cdrom: Erreur d'entrée/sortie
find: ./mnt/floppy: Erreur d'entrée/sortie
./dev/core
./etc/X11/xdm/core
find: ./etc/skel/tmp: Permission non accordée
find: ./etc/default: Permission non accordée
find: ./etc/gconf/gconf.xml.defaults: Permission non accordée
find: ./etc/Bastille: Permission non accordée
find: ./etc/uucp: Permission non accordée
find: ./etc/webmin: Permission non accordée
find: ./etc/linuxconf/archive: Permission non accordée
<CTL><c>

$ find . -name core 2>/dev/null
./dev/core
./etc/X11/xdm/core
./home/ftp/core
./var/log/httpd/core
./proc/sys/net/core
```

1.9 Les redirections

1.9.1 Redirection des données en sortie

La sortie standard (écran) peut être redirigée à l'aide du symbole >

La sortie des données se fera dans un fichier qui sera créé ou dont le contenu sera écrasé.

```
commande  > fichier
```

```
$
$ ls -l > liste_rep
$
```

Pour ajouter les données en fin d'un fichier existant

```
commande  >> fichier
```

1.9.2 Redirection des erreurs en sortie

La sortie d'erreur (écran) peut être redirigée à l'aide du symbole 2>

La sortie des messages d'erreur se fera dans un fichier qui sera créé ou dont le contenu sera écrasé (comme pour la sortie standard).

```
commande 2> fichier
```

```
$
$ ls bidon  2>fic_err
$
```

Pour ajouter les données en fin d'un fichier existant :

```
commande 2>> fichier
```

/dev/null pseudo-périphérique permettant de se débarrasser des sorties inutiles :

```
$
$ find / -name prog1 2>/dev/null
...
```

1.9.3 Le pipe

Utilisation du symbole **|** (barre verticale : **<Alt Gr> <6>**)

Appelé **tube** de communication ou plus communément **pipe**.

```
commande1 | commande2
```

Avantage : inutile de créer un fichier temporaire (gain de temps et d'espace disque).

Toute sortie produite sur le stdout de commande1 est immédiatement transféré sur le stdin de commande2.

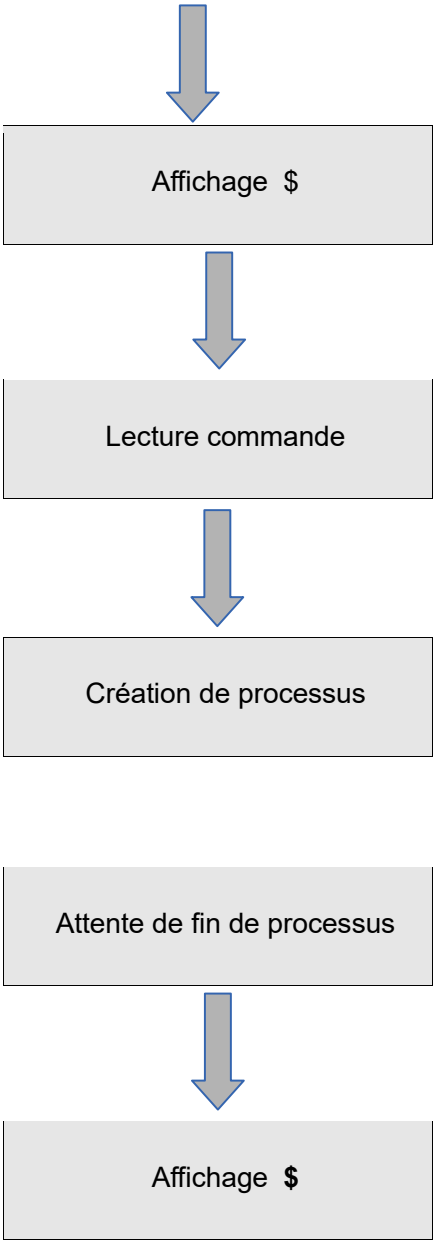
```
$ ls -l /bin | more
total 8080
-rwxr-xr-x. 1 root root  27776 12 mars  12:12 arch
lrwxrwxrwx. 1 root root      4 23 avril  2013 awk -> gawk
-rwxr-xr-x. 1 root root  26264 12 mars  12:12 basename
-rwxr-xr-x. 1 root root 938832 18 juil.  2013 bash
-rwxr-xr-x. 1 root root  48568 12 mars  12:12 cat
...
--Plus--
```

Le nombre de pipes n'est pas limité.

1.10 Gestion des processus

1.10.1 Le rôle du shell

Le shell est un interpréteur de commandes.



1.10.2 Liste des processus

On appelle **processus** un programme en exécution.

Un processus est créé par un autre processus.

Le shell crée un **processus fils** qui se charge de l'exécution.

Les processus sont organisés en arborescence : **père → fils**

La commande ps affiche la liste des processus en cours où :

PID **P**rocess **Id**entifier : numéro du processus

PPID **P**arent **P**rocess **ID** : numéro du processus père

```
$ ps
  PID TTY          TIME CMD
13297 pts/1    00:00:00 bash
13342 pts/1    00:00:00 ps
$
$ ps -f
UID         PID  PPID  C STIME TTY          TIME CMD
user1     13297 13296  0 09:44 pts/1    00:00:00 -bash
user1     13343 13297  0 11:43 pts/1    00:00:00 ps -f
$
$ ps -ef | more
UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 Jun20 ?        00:00:04 init [3]
root          2     1  0 Jun20 ?        00:00:00 [keventd]
...
root       1225     1  0 Jun20 ?        00:00:00 [nfsd]
root       1226     1  0 Jun20 ?        00:00:00 [nfsd]
root       1227     1  0 Jun20 ?        00:00:00 [nfsd]
root       1228  1225  0 Jun20 ?        00:00:00 [lockd]
root       1229  1228  0 Jun20 ?        00:00:00 [rpciod]
root       1230     1  0 Jun20 ?        00:00:00 [nfsd]
...
root       1821     1  0 Jun20 tty1     00:00:00 /sbin/mingetty tty1
root       1822     1  0 Jun20 tty2     00:00:00 /sbin/mingetty tty2
root       1823     1  0 Jun20 tty3     00:00:00 /sbin/mingetty tty3
...
user1     13297 13296  0 09:44 pts/1    00:00:00 -bash
user1     13344 13297  0 11:45 pts/1    00:00:00 ps -f
$
```

1.10.3 Procédure ou shell-script

Le shell (ksh ou bash) peut lire un fichier contenant un ensemble de commandes à exécuter.

On appelle ces fichiers de commandes des procédures ou shell-scripts.

➢ Exécution d'une procédure

```
$
$ chmod u+x procedure1
$
$ procedure1
...
```

➢ La variable PATH

Il est important d'afficher la variable PATH pour vérifier la liste des répertoires dans lesquels le shell va chercher les commandes (y compris les shell-scripts).

```
$
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:
/home/user1/bin:.
$
```

Si le **.** (point) représentant le répertoire courant, n'est pas présent, on devra lancer le script de la manière suivante :

```
$
$ ./procedure1
...
```

➢ Mise au point d'une procédure

La commande set permet de positionner les options du shell pour la mise au point d'une procédure.

set -x	Affichage de chaque commande **après** expansion.
set -v	Affichage de chaque commande **avant** expansion.
set -e	Sortie immédiate sur erreur.
set +x	Invalide l'option (ici x) positionnée auparavant.

➢ Commentaires dans une procédure

Le caractère **#** permet de commenter une procédure.

Si le commentaire est à la suite d'une ligne de commande, il doit être précédé d'au moins un espace.

En typographie le dièse *(♯) est un* caractère *différent du croisillon ou hashtag (#). Ce dernier a les deux barres transversales horizontales, alors que celles du* dièse *sont montantes. (Wikipédia)*

1.10.4 Le shabang

Le shabang ou shebang, représenté par #! est vraisemblablement un mot valise représentant pour sharp (dièse) et bang désignant le point d'exclamation !

Lorsque la première ligne d'une procédure est de la forme :

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

Le système reconnait que ce fichier de texte est un script. L'interpréteur permettant d'exécuter ce script est indiqué à la suite.

1.11 Les variables

1.11.1 Les variables prédéfinies

L'environnement d'un shell contient un certain nombre de variables prédéfinies.

La liste de toutes les variables (prédéfinies ou initialisées pendant la session) peut s'obtenir par la commande **`set`**.

(L'exemple ci-dessous n'est qu'un échantillonnage du résultat.)

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:hostcompl
ete:interactive_comments:progcomp:promptvars:sourcepath
BASH_VERSION='4.1.2(1)-release'
DESKTOP_SESSION=gnome
GDM_KEYBOARD_LAYOUT=$'fr\tlatin9
GDM_LANG=fr_FR.UTF-8
HISTFILE=/home/user1/.bash_history
HISTFILESIZE=1000
HOME=/home/pat
HOSTNAME=localhost.localdomain
HOSTTYPE=x86_64
ID=501
IFS=$' \t\n'
LANG=fr_FR.UTF-8
LOGNAME=user1
LS_COLORS='rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:*.tar=01;3
1:*.tgz=01;31:*.arj=01;31:*.taz=01;31:...
MACHTYPE=x86_64-redhat-linux-gnu
MAIL=/var/spool/mail/user1
OLDPWD=/tmp
OSTYPE=linux-gnu
PATH=/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user1/bin:.
PPID=11001
PROMPT_COMMAND='printf "\033]0;%s@%s:%s\007" "${USER}" "${HOSTNAME%%.*}"
"${PWD/#$HOME/~}"'
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/home/user1
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
TERM=xterm
UID=501
USER=user1
USERNAME=user1
```

1.12 Création d'une variable

```
variable=contenu_de_la_variable
```

```
$ var=bonjour
$ echo $var
bonjour
$
```

var	est le nom de la variable
bonjour	est le contenu de la variable

1.12.1 Exportation des variables

Les variables du shell père ne sont pas connues des processus fils. Pour qu'une variable soit commune à tous les processus fils d'un processus, elle doit être **exportée**.

```
Var=18 ; export var
```

ou

```
export var=18
```

ou

```
typeset -x var=18
```

```
$ more prog
echo $var
$ prog

$ var=18
$ echo $var
18
$ prog

$ export var
$ echo $var
18
$ prog          # Après "export var", var est connue de prog
18
```

Unix - Linux

2. La ligne de commandes

Objectifs

- Syntaxe d'une ligne de commandes, enchaînement de commandes, regroupement de commandes.
- Rôle du shell dans l'analyse d'une ligne de commandes.

Remarque :
Les exemples proposés dans les pages suivantes ont pour but d'illustrer différents mécanismes du shell, l'objectif n'est pas d'expliciter ici ces exemples, certains seront étudiés plus loin dans le cours.

2.1 La ligne de commandes

Pour être en mesure de réaliser l'interprétation, le shell (sh, ksh, bash...) exige une structure précise de la ligne de commande.

Une ligne de commandes peut être réalisée au moyen d'une commande simple ou en combinant des options et/ou des paramètres.

2.1.1 Commande simple

```
commande   [-option(s) | --option(s)]   [par-1]   …   [par-n]
```

Commande avec options et paramètre

```
$ ls -l /etc /home
```

2.1.2 Pipelines

```
commande_1 | commande_2 | commande_3 [   …   | commande_n]
```

La sortie standard de commande_1 est connectée à l'entrée standard de commande_2 ; la sortie standard de commande_2 est connectée à l'entrée standard de commande_3 et ainsi de suite.

Le code retour d'un pipeline est celui de la dernière commande exécutée.

```
$ ls -l /etc | grep '^d' | more
```

2.1.3 Opérateurs logiques

Les opérateurs logiques **&&** et **||** indiquent respectivement une liste de commandes liée par ET, et une liste liée par OU.

La valeur de retour des listes ET, et des listes OU est celle de la dernière commande exécutée.

- **ET logique : &&**

```
commande_1 && commande_2
```

commande_2 sera exécutée si, et seulement si, commande_1 renvoie un code retour nul.

```
$ cd repertoire && rm -r *
```

Si on peut se positionner sur "repertoire", alors on supprimera tous les fichiers qui sont dessous.

- **OU logique : ||**

```
commande_1 || commande_2
```

commande_2 sera exécutée si, et seulement si, commande_1 renvoie un code retour non nul.

```
$ test -e repertoire || mkdir repertoire
```

S'il n'existe pas de fichier nommé `repertoire` alors on crée le répertoire.

2.2 Regroupements de commandes

Le regroupement de commandes forme une commande composée qui alimente le code retour.

2.2.1 Regroupement implicite

Certaines structures de contrôle du shell (les `boucles for, select, while, until` *) réalisent un groupement implicite de commandes.
Cela permet de considérer les différentes sorties de commandes (résultats et/ou erreurs) comme un seul flux de données et de les rediriger en une seule opération.

(*) Ces structures vont être étudiées dans ce cours dans un prochain chapitre.

```
$ more prog_rep1
for fic in *
do
      [ -d $fic ] && ls -l $fic
done >.listerep1

$ more .listerep1
total 0
total 8
-rw-rw-r-- 1 user1           g1    6     janv  11    16:06 f1
total 8
-rw-rw-r-- 1 user1           g1    30    janv  11    16:06 f2
```

Le programme `prog_rep1` va s'exécuter autant de fois qu'il y a de fichier (au sens large) dans le répertoire courant. Si un nom de fichier correspond à un répertoire alors le programme lance la commande `ls -l` pour afficher le contenu de ce répertoire. Mais au lieu d'afficher le résultat à l'écran il sera mis dans le fichier `.listerep1`.
Le regroupement fait par `for...done` permet de ne préciser qu'une seule fois la redirection de la sortie vers `.listerep1` .

2.2.2 Regroupement par accolades { }

Le regroupement par accolades ne provoque pas la création d'un shell fils.

Cela peut, par exemple, clarifier l'utilisation des opérateurs logiques. Il doit commencer par un espace après l'accolade ouvrante et se terminer soit par une fin de ligne soit par un point-virgule avant l'accolade fermante.

```
$ cat prog_rep2
for fic in *
do
            test -d $fic && { echo $fic; ls -l $fic;}
done >.listerep2
$

$ cat .listerep2
Desktop
total 0
rep1
total 8
-rw-rw-r-- 1 user1            g1          6     janv  11    16:06 f1

rep2
total 8
-rw-rw-r-- 1 user1            g1          30    janv  11    16:06 f2
```

Le programme `prog_rep2` va s'exécuter autant de fois qu'il y a de fichiers (au sens large) dans le répertoire courant. Si un nom de fichier correspond à un répertoire alors le programme affiche le nom du répertoire puis lance la commande `ls -l` pour afficher le contenu de ce répertoire. Mais au lieu d'afficher le résultat à l'écran il sera mis dans le fichier `.listerep2`. Les deux commandes `echo` et ls doivent donc être regroupées pour que leurs sorties soient redirigées vers le même fichier `.listerep2`.

2.2.3 Regroupement par parenthèses ( )

Le regroupement par parenthèses provoque la création d'un shell fils.

Cela permet, par exemple, un parcours facile de l'arborescence.

```
$ cat prog_rep3
for fic in *
do
            [ -d $fic ] && (cd $fic; pwd; ls -l)
done >.listerep3

$ cat .listerep3
/home/user1/Desktop
total 0
/home/user1/rep1
total 8
-rw-rw-r-- 1 user1            g1     6      jui 11       16:06 f1
/home/user1/rep2
total 8
-rw-rw-r-- 1 user1            g1           30     jui 11       16:06 f2
```

Rappelons-nous que la commande `pwd` modifie la variable `PWD`. En conséquence, les commandes regroupées doivent s'exécuter dans un shell fils pour ne pas changer le répertoire de travail du shell appelant.

2.3 Lecture de la ligne de commandes

Lors de la lecture de la ligne de commandes, avant de s'intéresser à l'exécution de la commande elle-même, le shell exécute systématiquement les étapes de l'algorithme suivant :

1. Découpage de la ligne en mots et en opérateurs[1] (le quoting est respecté)
2. Traitement des alias
3. Analyse du résultat obtenu en termes de commandes simples ou composées
4. Expansion des accolades
5. Développement du ~
6. Expansion des variables et paramètres
7. Évaluation des commandes
8. Expansion des expressions arithmétiques

9. Découpage en mots de la ligne obtenue[2]
10. Développement des noms de fichiers
11. Mise en place des redirections
12. Suppression des protections[3]
13. Localisation du fichier de la commande
14. Lancement du processus

Les pages suivantes décrivent ces mécanismes.

[1]Recherche des méta-caractères non protégés : | & ; ( ) < > espace, tabulation ainsi que des opérateurs ayant une fonction de contrôle : || & && ; ;; ( ) | passage à la ligne. Une chaîne de caractères entre apostrophes ou guillemets est considérée comme un mot.

[2]Par défaut, le shell considère les caractères espace, tabulation et passage à la ligne comme des délimiteurs de mots et découpe le résultat des transformations précédentes en fonction de ceux-ci. Après le découpage, tous les mots de la ligne de commande seront séparés par un seul espace, sachant qu'une chaîne de caractères entre apostrophes ou guillemets est considérée comme un mot.

[3]Toutes les occurrences non protégées de \ " " ' ' sont supprimées.

2.4 Traitement des alias

2.4.1 Définition

Les alias permettent de substituer une chaîne de caractères à un mot lorsqu'il est utilisé comme premier mot d'une commande simple.

```
$ alias heure='date "+Il est %Hh%M"'    # Définition de l'alias
$ heure
Il est 14h15
```

Lors de la lecture de la ligne de commande, le mot heure, défini comme un alias, est remplacé par la chaîne de caractères qui lui a été attribuée lors de la définition puis la commande date est exécutée.

Le rôle du shell peut être vérifié en forçant l'affichage de la commande après expansion :

```
$ set -x
$ heure
+ date '+Il est %Hh%M'
Il est 14h15
```

2.5 Expansion des accolades

C'est un mécanisme qui permet la création puis l'utilisation de chaînes quelconques. Il autorise l'emploi de caractères génériques.

Mécanisme propre au bash.

```
$ mkdir rep3

$ mkdir rep3/{r1,r2}

$ mkdir rep3/{r3,r4}doc

$ ls -l rep3
total 32
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r1
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r2
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r3doc
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r4doc

$ chmod o= rep3/{r1,r3*}

$ ls -l rep3
total 32
drwxrwx---  2 user1 g1        4096 jui 16    09:03 r1
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r2
drwxrwx---  2 user1 g1        4096 jui 16    09:03 r3doc
drwxrwxr-x  2 user1 g1        4096 jui 16    09:03 r4doc
```

2.6 Développement du tilde

~	Représente le répertoire de connexion de l'utilisateur
~nom	Représente le répertoire d'accueil de l'utilisateur nom
~+	Représente la valeur de PWD
~-	Représente la valeur de OLDPWD

```
$ pwd
/home/user1/rep3

$ mkdir srep3
$ cd srep3

$ echo $HOME
/home/user1

$ echo ~
/home/user1

$ echo $PWD
/home/user1/rep3/srep3

$ echo ~+
/home/user1/rep3/srep3

$ echo $OLDPWD
/home/user1/rep3

$ echo ~-
/home/user1/rep3

$ echo ~root
/root

$ echo ~user2
/home/user2
```

Ici nous sommes en bash sous Linux ! En Unix, le répertoire d'accueil de root est /

2.7 Expansion des paramètres positionnels

2.7.1 Les paramètres ou arguments

Les paramètres positionnels ou arguments sont des variables du shell portant un numéro, à partir de 1.

Ils sont renseignés par le shell au moment de l'invocation d'un shell-script.

```
$ more copier
echo Premier argument : $1
echo Deuxième argument : $2
echo Troisième argument : $3
cp $1 $2/$1.$3
chmod u=r,go= $2/$1.$3

$ copier sel rep1 sv
Premier argument : sel
Deuxième argument : rep1
Troisième argument : sv

$ ls -l rep1/sel.sv
-r--------  1 user1 g1   787 jui 13 10:46 rep1/sel.sv
```

2.7.2 Modification des paramètres

Les paramètres positionnels ne peuvent être modifiés que par :

- La commande `shift`
- La commande `set` suivie de mots

➢ shift

```
$
$ more decalage
echo le premier paramètre est : $1
echo le neuvième paramètre est : $9
echo nombre de paramètres : $#
echo " ATTENTION DECALAGE"
shift
echo le premier paramètre est : $1
echo le neuvième paramètre est : $9
echo nombre de paramètres : $#
$
$ decalage A B C D E F G H I J K
le premier paramètre est : A
le neuvième paramètre est : I
nombre de paramètres : 11
ATTENTION DECALAGE
le premier paramètre est : B
le neuvième paramètre est : J
nombre de paramètres : 10
$
```

➢ set

La commande set permet de passer des paramètres au shell lui-même.

```
$ set Lun Mar Mer Jeu Ven Sam Dim
$ echo $1 $2 $3 $4 $5 $6 $7
Lun Mar Mer Jeu Ven Sam Dim
```

2.8 Expansion des paramètres spéciaux

Les paramètres spéciaux peuvent uniquement être consultés, ils ne sont pas modifiables.

Ils sont accessibles en tant que variables. En conséquence, ces caractères doivent être précédés de $ lors de leur utilisation.

`$0`	Contient le nom de la commande
`$*`	Contient la liste des paramètres
`$@`	Contient la liste des paramètres
`$#`	Contient le nombre de paramètres
`$?`	Contient le code retour de la dernière commande . 0 : la commande a répondu vrai différent de 0 : la commande a répondu faux
`$-`	Contient les options du shell
`$$`	Contient le pid du shell, y compris dans un shell fils créé par ( )
`$!`	Contient le pid de la dernière commande lancée en arrière-plan

2.9 Expansion des variables

Le shell substitue l'écriture « `$variable` » par la valeur de la variable.

Le nom de la variable peut être encadré par des accolades. Celles-ci sont obligatoires dans le cas d'une variable positionnelle dont le nom est composé de deux chiffres.

2.9.1 Expansion des paramètres

```
$ annee="Douze mois"

$ set janvier février mars avril mai juin juillet août septembre
octobre novembre décembre

$ set -x

$ echo $annee ${annee}
+ echo Douze mois Douze mois
Douze mois Douze mois

$ echo $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11} ${12}
+ echo janvier $'f\303\251vrier' mars avril mai juin juillet
$'ao\303\273t' septembre octobre novembre $'d\303\251cembre'
janvier février mars avril mai juin juillet août septembre
octobre novembre décembre

$ set +x
```

RAPPEL : une variable est l'élément numéro 0 du tableau de même nom et de même type. Pour le typage des variables, voir la commande typeset (ksh et bash) ou declare (bash).

2.9.2 Règles de substitution de variables

Lors de la lecture de la ligne de commande, le shell substitue les écritures suivantes en utilisant l'algorithme décrit.

2.9.3 Valeurs par défaut d'une variable

${variable:-valeur}	**Utilisation d'une valeur par défaut**
	Le shell substitue l'écriture par :
	1. La valeur si la variable n'est pas définie ou si elle est nulle,
	2. Le contenu de la variable si celle-ci est définie.
	
	`$ echo ${var1:-bonjour}`
	`bonjour`
	`$ var1=bonsoir`
	`$ echo ${var1:-bonjour}`
	`bonsoir`

${variable:=valeur}	**Attribution d'une valeur par défaut**
	Si la variable n'est pas définie ou si elle est nulle, le shell affecte la valeur à la variable et substitue l'écriture par la valeur. Sinon, l'écriture est substituée par le contenu de la variable.
	
	`$ echo ${var2:=bonjour}`
	`bonjour`
	`$ echo $var2 ${var2:=bonsoir}`
	`bonjour bonjour`

${variable:?message}	**Affichage d'un message d'erreur**
	Le message peut être en clair ou contenu dans une variable autre que celle désignée avant le « : ».
	Si la variable n'est pas définie ou si elle est nulle :
	1. Le shell affiche le message sur la sortie standard,
	2. Dans un contexte non interactif, le processus se termine. Si la variable est définie, c'est son contenu qui est utilisé.
	
	`$ cat prog`
	`: ${1:?"USAGE: prog arg"}`
	`echo Suite du programme`
	`$ prog`
	`./prog:  line 1:  1: USAGE: prog arg`
	`$ prog fic`
	`...suite du programme...`

Sans le caractère « : », ces algorithmes s'appliquent seulement lorsque la variable n'est pas définie.

2.9.4 Troncature d'une variable

<table>
<tr>
<td>${variable#valeur}
${variable##valeur}</td>
<td>
Troncature du début d'une variable
La valeur peut être construite avec les caractères génériques du shell.
Si le début de la variable correspond à la valeur, le shell substitue l'écriture par le contenu de la variable après suppression de la plus petite longueur de la valeur (cas de #), ou de la plus grande longueur de la valeur (cas de ##).
Sinon, le shell substitue l'écriture par le contenu de la variable.

$ REP=/etc
$ echo ${REP#$HOME/}
/etc
$ REP=/home/user1
$ echo ${REP#$HOME/}
/home/user1
$ REP=/home/user1/rep
$ echo ${REP#$HOME/}
rep

S'applique aussi à * et @, dans ce cas l'opération de suppression s'applique à chaque paramètre positionnel.
$ set jour1 jour2 jour3
$ echo ${*#jour} ${@#jour}
1 2 3 1 2 3
</td>
</tr>
<tr>
<td>${variable%valeur}
<b>${variable%%valeur}</td>
<td>
Troncature de la fin d'une variable
L'opérateur % utilise le même algorithme que # mais il s'applique à la fin de la variable.
Même remarque pour les variables * et @.

$ PROG=calcul.c
$ echo ${PROG%.c}
calcul
</td>
</tr>
</table>

2.9.5 Autres substitutions

${#variable}	Substitué par la longueur, en nombre de caractères, de la variable.
	
	`$ mot=anticonstitutionnellement`
	`$ echo ${#mot}`
	`25`
	
	Avec * ou @, la substitution retourne le nombre de paramètres.
	
	`$ set lun mar mer jeu ven sam dim`
	`$ echo "${#*}  ${#@}  $#"`
	`7  7  7`

${#tab[*]}
${#tab[@]}
	Substitué par le **nombre d'éléments** renseignés du tableau tab.
	
	`$ tab[0]=0  tab[1]=1  tab[2]=2`
	`$ echo "${#tab[*]}  ${#tab[@]}"`
	`3  3`

${tab[*]}
${tab[@]}
	Substitué par l'**ensemble des éléments** renseignés du tableau tab.
	
	`$ echo "${tab[*]}  ${tab[@]}"`
	`0 1 2  0 1 2`

2.9.6 Recherche d'un motif dans une chaîne de caractères et substition (bash)

${variable/motif/chaîne}** **${variable//motif/chaîne}	Le motif peut être construit avec les caractères génériques du shell. Le shell substitue l'écriture par une chaîne de caractères construite comme suit. Recherche dans la variable de la plus grande portion du motif et remplacement par la chaîne. / seule la première occurrence est remplacée ; // toutes les occurrences sont remplacées. Si le motif n'est pas trouvé, le shell ne fait aucun remplacement. $ **ch="le gel le parasite la chambre"** $ **echo ${ch/l[ae] /anti}** antigel le parasite la chambre $ **echo ${ch//l[ae] /anti}** antigel antiparasite antichambre
${variable/motif}** **${variable//motif}	Même algorithme que ci-dessus avec suppression du (des) motif(s) construit(s). $ **echo ${ch/l[ae] }** gel le parasite la chambre $ **echo ${ch//l[ae] }** gel parasite chambre
${variable/#motif/chaîne}	Même algorithme que ci-dessus ; le motif doit correspondre au début de la variable. $ **echo ${ch/#le parasite/antiparasite}** le gel le parasite la chambre $ **echo ${ch/#le gel/antigel}** antigel le parasite la chambre
${variable/%motif/chaîne}	Même algorithme que ci-dessus ; le motif doit correspondre à la fin de la variable. $ **echo ${ch/%le parasite/antiparasite}** le gel le parasite la chambre $ **echo ${ch/%la chambre/antichambre}** le gel le parasite antichambre

2.9.7 Extraction de sous-chaîne (bash uniquement)

${variable:position}
${variable:position:longueur}
	A partir du contenu de la variable, le shell fournit la sous-chaîne commençant à « position » (l'indexation commence à 0) sur « longueur » (si « longueur » omise : depuis « position » jusqu'à la fin de la variable). S'applique aussi à * ou @, dans ce cas l'indexation commence à 1.
	
	`$ set lun mar mer jeu ven sam dim`
	`$ echo -e "$*\n$@"`
	`lun mar mer jeu ven sam dim`
	`lun mar mer jeu ven sam dim`
	`$ echo ${*:2:3} ${@:2:3}`
	`mar mer jeu mar mer jeu`

2.10 Évaluation des commandes

$(commande) ou **`commande`**

Représentent une évaluation de commande permet de remplacer la commande par son résultat.

Le shell exécute la commande, la remplace par son résultat et réalise une séparation en mot en utilisant le premier caractère de la variable IFS.

```
$ set -x

$ echo date du jour $(date) sur `uname -n`
++ date
++ uname -n
+ echo date du jour jeu sep 20 10:21:25 CEST 2007 sur localhost.localdomain
date du jour jeu sep 20 10:21:25 CEST 2007 sur localhost.localdomain
```

La séparation en mot ne sera pas réalisée si l'évaluation de commande est protégée par des guillemets "..."

```
$ echo "$(ls -l)"
++ ls --color=tty -l
+ echo 'total 136
-rw-rw-r--  1 user1    g1  20480   sep 14      14:04 archi.tar
drwxrwxr-x  4 user1    g1    4096  sep 11      00:16 cours
drwxr-xr-x  3 user1    g1    4096  sep  8      02:55 Desktop
-r--------  1 user1    g1    199   sep 11      13:54 dupliquer.sv
-rw-rw-r--  1 user1    g1    4     sep 20      08:47 ficcase
-rw-rw-r--  1 user1    g1    6493  sep  9      21:59 ficps
-rw-rw-r--  1 user1    g1    500   sep 20      10:21 histo
-rwxrw-r--  1 user1    g1    61    sep 19      10:10 prog
drwxrwxr-x  2 user1    g1    4096  sep 11      01:35 restaure
-rw-rw-r--  1 user1    g1    89    sep  9      23:00 resul
-r--------  1 user1    g1    301   sep 11      13:58 tri.sv'
total 136
-rw-rw-r--  1 user1    g1  20480 sep 14  14:04 archi.tar
drwxrwxr-x  4 user1    g1    4096  sep 11      00:16 cours
drwxr-xr-x  3 user1    g1    4096  sep  8      02:55 Desktop
-r--------  1 user1    g1    199   sep 11      13:54 dupliquer.sv
-rw-rw-r--  1 user1    g1    4     sep 20      08:47 ficcase
-rw-rw-r--  1 user1    g1    6493  sep  9      21:59 ficps
-rw-rw-r--  1 user1    g1    500   sep 20      10:21 histo
-rwxrw-r--  1 user1    g1    61    sep 19      10:10 prog
drwxrwxr-x  2 user1    g1    4096  sep 11      01:35 restaure
-rw-rw-r--  1 user1    g1    89    sep  9      23:00 resul
-r--------  1 user1    g1    301   sep 11      13:58 tri.sv
```

2.11 Évaluation des expressions arithmétiques

L'évaluation arithmétique est l'opération par laquelle le shell remplace l'écriture :

`$((expression_arithmétique))`

Par le résultat du calcul arithmétique.

Il est possible d'utiliser ou non l'écriture "`$variable`".

La mise entre parenthèses modifie les priorités de calcul.

```
$ typeset -i v1=10 v2=20 v3=30 v4

$ echo $(( ($v1 + 5 ) / 3 ))
5
$ echo $(( (v1 + 5 ) / 3 ))
5
$ echo $(( v1 + v2 + v3 ))
60
$ echo $(( ( (v1 + 10) + ( v2 * 3 ) ) / 2 ))
40

Il est possible de traiter la substitution et de conserver le résultat du
calcul dans une variable.
$ echo $(( v4 = v1 + v2 + v3 ))
60
$ echo $v4
60

L'évaluation de commande est traitée.
$ date
ven sep 21 09:10:55 CEST 2007
$ echo $(( $(date +%H) + 8 ))
17
```

2.12 Développement des noms de fichiers

2.12.1 Cas général

Le shell recherche des caractères génériques (*, ?, [ ]) dans chaque mot de la ligne de commande.

Les caractères génériques permettent de construire des noms de fichiers à partir des fichiers, sous le répertoire courant ou le répertoire cité.

Les noms de fichiers construits sont listés dans l'ordre alphabétique.

*	N'importe quelle chaîne y compris la chaîne vide.
	`$ ls`
	`f f3 f35 f5 fa fic FIC ft fx toto`
	`$ echo *`
	`f f3 f35 f5 fa fic FIC ft fx toto`
	`$ echo f*`
	`f f3 f35 f5 fa fic ft fx`

?	Un caractère quelconque.
	`$ echo f?`
	`f3 f5 fa ft fx`
	`$ echo ?`
	`f`

[ ]	Un des caractères entre crochets.
	`$ echo f[35t]`
	`f3 f5 ft`
	`$ echo f[1-8]`
	`f3 f5`

[!]
[^]
	Un caractère non pris parmi les caractères entre crochets.
	`$ echo f[^35t]  # bash uniquement`
	`fa fx`
	`$ echo f[!35t]`
	`fa fx`

Si on cite un répertoire, l'expression générique porte sur le contenu du répertoire cité.

```
$ echo /etc/f*
/etc/fb.modes /etc/fdprm /etc/filesystems /etc/firmware /etc/fonts
/etc/foomatic /etc/frysk /etc/fstab
```

2.12.2 Cas particuliers : les options du bash

Certaines options du shell, positionnées par `set` ou `shopt`, imposent des traitements particuliers des caractères génériques.

*ATTENTION : la commande **shopt** ainsi que la variable `GLOBIGNORE` sont spécifiques du bash.*

set -f
set -o noglob
	Désactivation du traitement des caractères génériques.
	`$ set -f  # ou set -o noglob`
	`$ ls`
	`f  f3  f35  f5  fa  fic  FIC  ft  toto`
	`$ echo *`
	`*`
	`$ set +f`
	`$ echo *`
	`f f3 f35 f5 fa fic FIC ft toto`

shopt -s nullglob	Si le motif ne correspond à aucun nom de fichier il se développe en une chaîne nulle.
	`$ ls t*p`
	`ls: t*p: Aucun fichier ou répertoire de ce type`
	`$ shopt -s nullglob  # Active l'option`
	`$ ls t*p`
	`f  f3  f35  f5  fa  fic  FIC  ft  toto`
	`$ shopt -u nullglob  # Désactive l'option`
	`$ ls t*p`
	`ls: t*p: Aucun fichier ou répertoire de ce type`

shopt -s nocaseglob	Ignore la casse lors du développement des noms de fichiers.
	`$ ls`
	`f  f3  f35  f5  fa  fic  FIC  ft  toto`
	`$ shopt -s nocaseglob`
	`$ ls f*c`
	`fic FIC`
	`$ shopt -u nocaseglob`
	`$ ls f*c`
	`fic`

shopt -s dotglob	Les fichiers cachés sont pris en compte lors du développement des noms de fichiers.
	`$ >.fichier`
	`$ shopt -s dotglob`
	`$ echo *`
	`f f3 f35 f5 fa fic FIC .fichier ft toto`
	`$ shopt -u dotglob`
	`$ echo *`
	`f f3 f35 f5 fa fic FIC ft toto`

2.12.3 Cas particuliers : la variable GLOBIGNORE du bash

La variable `GLOBIGNORE`, si elle est définie, contient une liste de motifs séparés par des " : ". Chacun de ces motifs représente les noms de fichiers à ignorer lors du développement des noms de fichiers.

Si `GLOBIGNORE` est renseignée, l'option dotglob est automatiquement activée.

```
$ shopt -s                    # dotglob n'est pas activée
checkwinsize      on
cmdhist           on
expand_aliases    on
extquote          on
force_fignore     on
hostcomplete      on
interactive_comments    on
progcomp          on
promptvars        on
sourcepath        on

$ echo *
f f3 f35 f5 fa fic FIC ft toto

$ GLOBIGNORE='t*o'                  # Active dotglob

$ shopt -s
checkwinsize      on
cmdhist           on
dotglob           on          <---
expand_aliases    on
extquote          on
force_fignore     on
hostcomplete      on
interactive_comments    on
progcomp          on
promptvars        on
sourcepath        on

$ echo *
f f3 f35 f5 fa fic FIC .fichier ft

$ GLOBIGNORE='t*o:.*'   # ".*" dans GLOBIGNORE permet de contourner
                        # l'activation de dotglob
$ echo *
f f3 f35 f5 fa fic FIC ft
```

2.13 Mise en place des redirections

Avant son exécution, toute commande se voit attribuer un certain nombre de descripteurs de fichiers. Les descripteurs numérotés 0, 1 et 2 sont des descripteurs standard qu'il est possible de rediriger comme suit :

[0]<nom_fichier	Redirection de l'entrée standard (stdin)
[1]>nom_fichier	Redirection de la sortie standard des résultats (stdout)
2>nom_fichier	Redirection de la sortie standard des erreurs (stderr)

nom_fichier est soumis à l'expansion des accolades, du tilde, des paramètres, à la substitution des commandes, à l'évaluation arithmétique, au développement des noms de fichiers, à la suppression des protections.

Si plusieurs noms résultent de ces transformations :

- ksh ne fera pas de substitution et créera un fichier dont le nom comportera les caractères spéciaux.
- bash détectera une erreur.

Les redirections peuvent apparaître n'importe où sur la ligne de commande.

Les redirections sont mises en place au fur et à mesure de la lecture.

➢ Mise en place de la redirection du stderr avant celle du stdout

Le stderr est redirigé à l'écran et le stdout dans redir1

```
$ 2>&1 grep "[^ef]grep " /etc/* >redir1
grep: /etc/aliases.db: Permission non accordée
grep: /etc/amd.conf: Permission non accordée
...
$ cat redir1
/etc/a2ps.cfg:  if echo '$f' | grep '^/' >/dev/null 2>&1; then \
/etc/gpm-root.conf: "grep '^From ' /var/spool/mail/$USER | tail"
...
```

➢ Mise en place de la redirection du stderr après celle du stdout

Le stderr est redirigé au même endroit que le stdout

```
$ grep "[^ef]grep " /etc/* >redir2 2>&1
$ cat redir2
/etc/a2ps.cfg:  if echo '$f' | grep '^/' >/dev/null 2>&1; then \
grep: /etc/aliases.db: Permission non accordée
grep: /etc/amd.conf: Permission non accordée
...
```

2.14 Suppression des protections

2.14.1 Les différents caractères de protection

' '	Les apostrophes (ou simples quottes) réalisent une protection absolue, le shell ne fait aucune interprétation.
	
	`$ echo '$HOME'`
	`$HOME`

" "	Les guillemets (ou doubles quottes) réalisent une protection mais permettent l'interprétation de « $ », « \ », « ` ` », ou « $( ) »
	
	`$ echo -e "$LOGNAME\t$(whoami)"`
	`user1 c700541`
	`$ echo -e $LOGNAME\t$(whoami)`
	`user1tuser1`

\	Réalise une protection du caractère qui suit.
	`$ echo \*`
	`*`

2.14.2 Le rôle des protections

Au cours des différentes étapes de la lecture de la ligne de commande, le shell réalise les substitutions et expansions que nous venons de décrire à condition que les écritures à l'origine de ces transformations ne soient pas protégées.

Les exemples qui suivent illustrent les possibilités des différentes protections.

➢ Exemple 1

```
$ echo -e "\ndate        :      $(date)         # -e si bash
  utilisateur :      $LOGNAME
  machine     :      `uname -n`\n"

date        :      dim déc 16 06:22:45 CET 2014
utilisateur :      user1
machine     :      localhost.localdomain
$
```

Les guillemets permettent :

- L'expansion de `$LOGNAME` (étape 6)
- L'évaluation des commandes date et uname -n (étape7)
- La conservation, lors de l'étape 9, des espaces multiples consécutifs et des passages à la ligne. Cette conservation serait également réalisée par une protection par apostrophes mais en pareil cas, l'expansion de `$LOGNAME` et l'évaluation de `date` et `uname -n` ne serait pas faite.

Dans cet exemple, la protection est obligatoire pour conserver les espaces multiples consécutifs.

Cette protection ne peut se faire que par des guillemets car ils permettent l'expansion de variables et l'évaluation de commandes.

➢ Exemple 2

Par `grep`, on veut rechercher, dans l'ensemble des fichiers du répertoire courant, les lignes qui contiennent une lettre majuscule.

```
$ ls
$ echo BON >f1
$ echo SOIR >f2
$ ls
f1 f2
$ set -x
$ grep [A-Z] *          # (1)
+ grep '[A-Z]' f1 f2
f1:BON
f2:SOIR
$ echo JOUR >T
+ echo JOUR
$ ls
f1 f2 T
$ grep [A-Z] *          # (2)
+ grep T f1 f2 T
$ grep '[A-Z]' *        # (3)
+ grep '[A-Z]' f1 f2 T
f1:BON
f2:SOIR
T:JOUR
$ grep "[A-Z]" *        # (4)
+ grep '[A-Z]' f1 f2 T
f1:BON
f2:SOIR
T:JOUR
```

Etude des lignes de commande `grep` :

Cas (1) et (2) : à l'étape « développement des noms de fichiers », le shell substitue :

- Dans tous les cas, « * » par l'ensemble des fichiers sous le répertoire courant.
- Dans le cas (2) seulement, « [A-Z] » par T. La substitution a lieu à partir du moment où il y a sous le répertoire courant un (ou plusieurs) nom(s) de fichier(s) qui correspondent à l'expression « une lettre majuscule ». Dans le cas (2), après la transformation de la ligne de commande par le shell, `grep` recherche « T » dans l'ensemble des fichiers du répertoire courant et ne trouve rien !

Cas (3) et (4) : à l'étape « développement des noms de fichiers », le shell substitue « * » par l'ensemble des fichiers sous le répertoire courant mais n'examine pas « [A-Z] » puisque cette chaîne est protégée. Les protections seront enlevées en fin d'examen de la ligne de commande quand sera passé l'étape « développement des noms de fichiers »

L'expression régulière de `grep` doit être quottée pour éviter que le shell ne risque de l'analyser comme une chaîne générique.

➢ Exemple 3

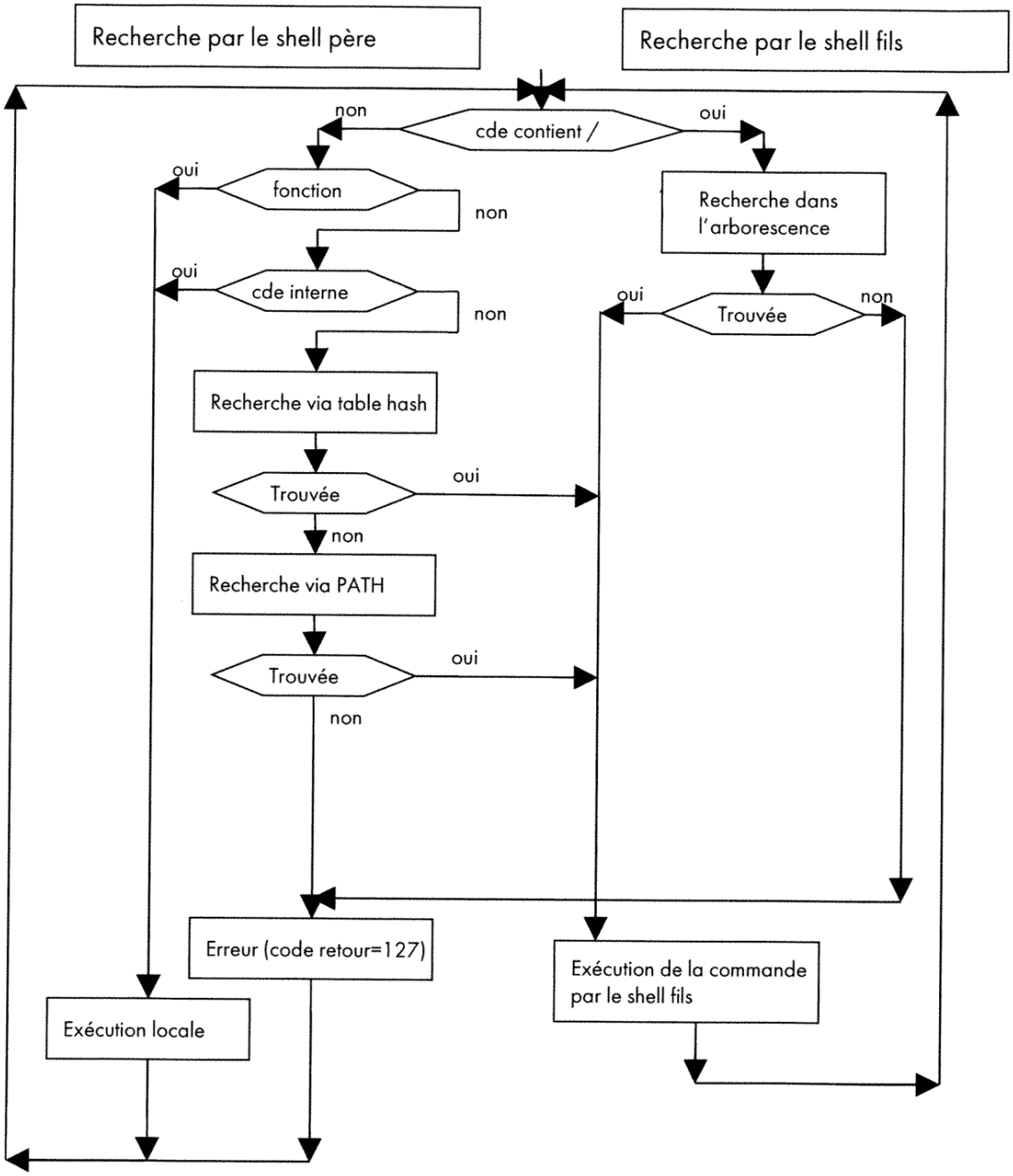
On veut avoir dans l'invite du shell (variable `PS1`) le chemin d'accès au répertoire courant. Ce chemin est contenu dans la variable d'environnement `PWD`.

```
$ pwd
/home/user1/newrep
$ PS1="$PWD $ "
/home/user1/newrep $ cd /etc
/home/user1/newrep $ pwd
/etc
/home/c70051/newrep $ echo $PS1                # (1)
/home/user1/newrep $
/home/user1/newrep $ PS1='$PWD $ '
/etc $ cd
/home/user1 $ echo $PS1                        # (2)
$PWD $
/home/user1 $
```

Lors de l'affectation de `PS1`, la protection par guillemets permet l'interprétation de la variable `PWD` et donc provoque l'enregistrement « en dur » dans `PS1` du chemin d'accès au répertoire sous lequel on se trouve au moment de l'affectation (comme le montre le résultat de la commande (1)). Lorsque, par la suite, on change de répertoire la valeur de `PS1` ne change pas… Ce qui est source d'erreurs.

Une protection par apostrophes interdira l'évaluation de `$PWD` au moment de l'affectation. La chaîne `$PWD` sera donc enregistrée « en dur » dans `PS1` (voir résultat de commande (2)) et interprétée à chaque appui sur la touche `<Entrée>`.

2.15 Localisation et exécution de la commande



Unix - Linux

3. Gestion des fichiers

Objectifs

- Approfondir la maîtrise des fichiers sous Unix et Linux
- Notions de systèmes de fichiers, définitions, montages, liens, liens symboliques
- Noms de fichiers, basename, dirname
- La commande find

3.1 Les systèmes de fichiers

3.1.1 Généralités

Un **système de fichiers** (File System) est une structure logicielle permettant de stocker des fichiers organisés en arborescence.

Contenu dans une partition ou un volume logique, un système de fichiers est composé de tables et de blocs de données.

Chaque système de fichiers est composé d'un répertoire racine (répertoire de plus haut niveau), lequel contient des fichiers ordinaires, spéciaux, répertoires...

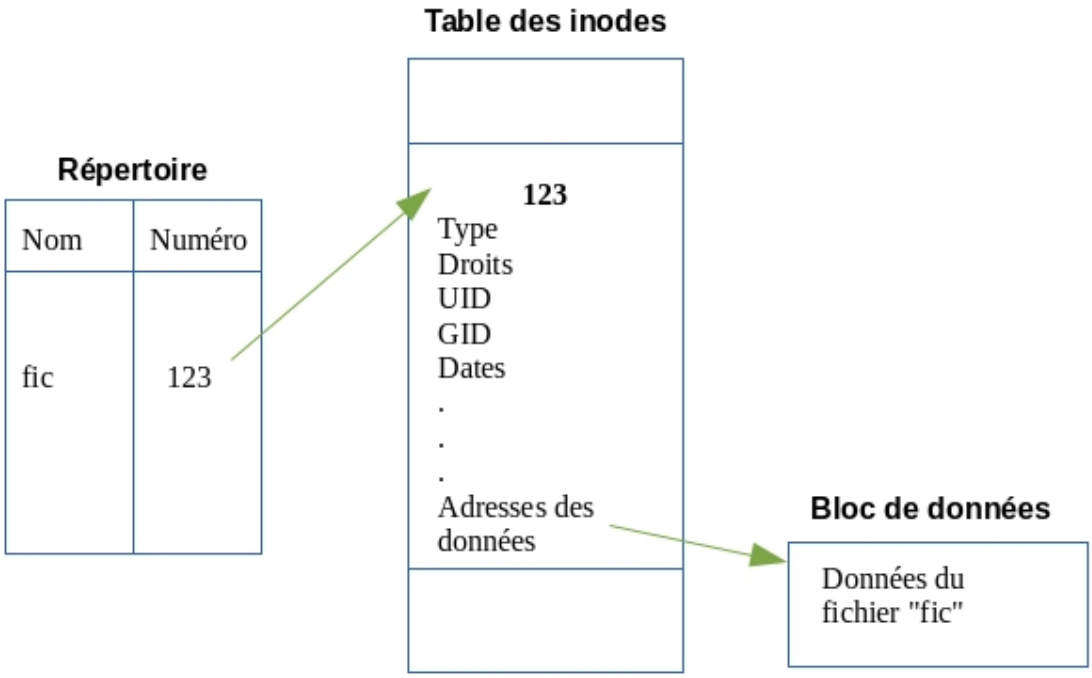
Les répertoires permettent de classer les autres fichiers en fonction de leurs utilisations. Chaque répertoire correspond à un nœud dans une arborescence.

Tout répertoire contient le nom de chaque fichier référencé dans ce répertoire auquel est associé un numéro.

Le numéro référençant chaque fichier désigne une entrée vers une table appelée i_list. La table i_list est composée d'i_node.

Un numéro d'i_node est un i_num.

L'i_node contient toutes les caractéristiques (type de fichier, droits d'accès, dates, adresses des données, …) du fichier auquel il correspond.



3.1.2 Principe de montage

Pour qu'un fichier soit accessible il faut que le système de fichiers auquel il appartient soit monté, c'est-à-dire que le répertoire racine du système de fichiers soit lié à un répertoire d'un système de fichiers déjà monté.

Exemple :

Pour que l'utilisateur user1 accède au fichier fic, il faut que le système de fichiers `/dev/hd4` (ex. Unix), `/dev/sda1` (ex. Linux) soit monté sur le répertoire `/home`.

Le répertoire racine du système de fichiers situé sur le volume logique `/dev/hd4` (Unix) ou la partition `/dev/sda1` (Linux) est lié logiciellement au répertoire /home qui seul reste visible après l'opération de montage.

3.1.3 Gestion des systèmes de fichiers montés

`Mount` Affiche les systèmes de fichiers montés avec leurs caractéristiques.

`df` Affiche pour les systèmes de fichiers montés ; les espaces disponible et utilisé, les inodes libres et utilisés, ainsi que les taux d'occupation.

```
$ mount                  # Réalisé sous Unix
noeud monté           monté sur     vfs    date         options
----- -------------  ------------  ------ ------------ ---------------
    /dev/hd4          /             jfs2   07 mar 22:36 rw,log=/dev/hd8
    /dev/hd2          /usr          jfs2   07 mar 22:36 rw,log=/dev/hd8
    /dev/hd9var       /var          jfs2   07 mar 22:36 rw,log=/dev/hd8
    /dev/hd3          /tmp          jfs2   07 mar 22:36 rw,log=/dev/hd8
    /dev/hd1          /home         jfs2   07 mar 22:36 rw,log=/dev/hd8
    /proc             /proc         procfs 07 mar 22:36 rw
    /dev/hd10opt      /opt          jfs2   07 mar 22:36 rw,log=/dev/hd8
                  :
                  :

$ mount                  # Réalisé sous Linux
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sda2 on /home type ext3 (rw)
/dev/hda1 on /mnt/windows type vfat (rw)
/dev/hdb1 on /mnt/disk type ext3 (rw)


$ df        # Réalisé sous Unix
Système de fichiers  Blocs 512 octets  Libre %Util    Iutil %Iutil Monté sur
/dev/hd4           1048576     527944   50%    13374    18% /
/dev/hd2           4980736     446616   92%    46658    38% /usr
/dev/hd9var         786432     513512   35%    19583    26% /var
/dev/hd3           1048576     773632   27%       77     1% /tmp
/dev/hd1           2883584    2275576   22%     1074     1% /home
/proc                    -          -    -        -     -  /proc
/dev/hd10opt        524288     178840   66%     3281    13% /opt
                  :
                  :

$ df        # Réalisé sous Linux
SysFichier           1K-blocs    Utilisé Dispo.   Util% Monté sur
/dev/sda1              7716496   2719876   4604636  38% /
tmpfs                    63192         0     63192   0% /dev/shm
/dev/sda2              1486108     35104   1374296   3% /home
/dev/hda1              1661916   1534036    127880  93% /mnt/windows
/dev/hdb1               403086      8239    374034   3% /mnt/disk
```

3.2 Les liens

Les systèmes de fichiers Linux supportent deux types de liens : les liens physiques et les liens symboliques. Ils sont créés tous les deux avec la commande `ln`.

3.2.1 Les liens physiques

➢ Définition

Un **lien physique** est l'association **<nom de fichier><i_num>**. Il permet, à partir d'un nom de fichier d'atteindre l'i_node et donc les caractéristiques du fichier (type, droits d'accès, UID, GID, …).

En plus des caractéristiques du fichier, l'i_node contient les numéros de blocs disques où se trouvent les données. Ainsi, atteignant l'i_node, on atteint les données d'un fichier.

Lorsque plusieurs noms de fichier pointent vers le même i_node, on dit qu'il s'agit de liens physiques vers un seul et même fichier.

Le compteur de liens est une des caractéristiques d'un fichier. Il est contenu dans l'i_node.

Exemple :

```
$ ls -li /bin/{gzip,zcat,gunzip}          # Syntaxe bash
685251 -rwxr-xr-x 3 root root       62132 mar  29 13:27 /bin/gunzip
685251 -rwxr-xr-x 3 root root       62132 mar  29 13:27 /bin/gzip
685251 -rwxr-xr-x 3 root root       62132 mar  29 13:27 /bin/zcat
```

➢ Création et suppression

La création d'un lien physique est réalisée lors de la création d'un fichier puis au travers de la commande ln.

$ echo "je suis le fichier fic1" >fic1

A la création du fichier fic1 le système :

- Recherche un i_node libre pour fic1 et déclare cet i_node occupé
- Met à jour le répertoire courant en associant le nom fic1 à l'i_num réservé
- Renseigne l'i_node avec les caractéristiques de fic1, notamment le compteur de lien qui vaut 1

```
$ cat fic1
je suis le fichier fic1

$ ls -li fic1
1237470 -rw-r--r-- 1 c700541 avance       24 sept  5 16:38 fic1
```

Lors de la création, le système a :

- Attribué à fic1 l'i_node 1237470
- Initialisé le compteur de liens à 1

```
$ ln fic1 fic2
```

fic1 et fic2 sont des liens physiques

```
$ ls -li fic1 fic2
1237470 -rw-r--r-- 2 c700541 avance       24 sept  5 16:38 fic1
1237470 -rw-r--r-- 2 c700541 avance       24 sept  5 16:38 fic2
```

On a :

- Créé un nouveau lien vers l'i_node 1237470
- Incrémenté le compteur de liens de cet i_node

Cette opération n'est possible que sur un même système de fichiers.

```
$ cat fic2
je suis le fichier fic1

fic1 et fic2 sont un seul et même fichier
```

```
$ rm fic1
$ ls -li fic2
1237470 -rw-r--r-- 1 c700541 avance       24 sept  5 16:38 fic2
```

rm a décrémenté le compteur de liens.

```
$ cat fic2
je suis le fichier fic1
```

Les données restent accessibles par le lien fic2.
Pour que la suppression des données soit effective le compteur de lien doit passer à 0.

3.2.2 Les liens symboliques

- **Définition**

Un lien symbolique est un fichier dont les données sont le chemin d'accès absolu ou relatif à un autre fichier, appelé cible.

Le fichier cible peut :

- Etre de n'importe quel type de fichiers.
- Se trouver dans n'importe quel système de fichiers.

Exemple :

```
$ ls -l /bin|grep '^l'
lrwxrwxrwx 1 root root       4 sep  5 13:49 awk -> gawk
lrwxrwxrwx 1 root root       4 sep  5 13:53 csh -> tcsh
lrwxrwxrwx 1 root root       8 sep  5 13:52 dnsdomainname -> hostname
lrwxrwxrwx 1 root root       8 sep  5 13:52 domainname -> hostname
lrwxrwxrwx 1 root root       4 sep  5 13:49 egrep -> grep
lrwxrwxrwx 1 root root       2 sep  5 13:52 ex -> vi
lrwxrwxrwx 1 root root       4 sep  5 13:49 fgrep -> grep
lrwxrwxrwx 1 root root       3 sep  5 13:52 gtar -> tar
lrwxrwxrwx 1 root root       4 sep  5 13:54 mailx -> mail
lrwxrwxrwx 1 root root       8 sep  5 13:52 nisdomainname -> hostname
lrwxrwxrwx 1 root root       2 sep  5 13:49 red -> ed
lrwxrwxrwx 1 root root       2 sep  5 13:52 rvi -> vi
lrwxrwxrwx 1 root root       2 sep  5 13:52 rview -> vi
lrwxrwxrwx 1 root root       4 sep  5 13:49 sh -> bash
lrwxrwxrwx 1 root root      10 sep  5 13:50 tcptraceroute -> traceroute
lrwxrwxrwx 1 root root      10 sep  5 13:50 traceroute6 -> traceroute
lrwxrwxrwx 1 root root      10 sep  5 13:50 tracert -> traceroute
lrwxrwxrwx 1 root root       2 sep  5 13:52 view -> vi
```

- **Création et suppression**

La création d'un lien symbolique est réalisée avec l'option `-s` de la commande `ln`.

Elle permet de créer un fichier dont tous les droits sont positionnés et qui contient le nom du fichier cible tel qu'il est écrit dans la commande `ln -s`.

La suppression du lien symbolique se fait avec la commande `rm`.

- **Lien symbolique vers un fichier ordinaire**

```
$ cd
$ mkdir /tmp/bidon
$ echo Je suis le fichier fic3 >/tmp/bidon/fic3

$ cat /tmp/bidon/fic3
Je suis le fichier fic3

$ ln -s /tmp/bidon/fic3 fic3.lien
```

Des liens symboliques peuvent être réalisés entre fichiers appartenant à des systèmes de fichiers différents.

```
$ ls -li /tmp/bidon/fic3 fic3.lien
31976  -rw-r--r-- 1 c700541 avance 24 sep  5 17:36 /tmp/bidon/fic3
716180 lrwxrwxrwx 1 c700541 avance 15 sep  5 17:36 fic3.lien ->
/tmp/bidon/fic3

Il s'agit de deux fichiers différents
```

Les données de *fic3·lien* sont : */tmp/bidon/fic3* et sa taille est de 15 caractères.

```
$ cat fic3.lien
Je suis le fichier fic3

$ cd /tmp/bidon
$ ln -s fic3 /home/c700541/lien.symbolique
$ cd
$ ll -i lien.symbolique
716181 lrwxrwxrwx 1 c700541 avance 4 sep  5 17:36 lien.symbolique -> fic3

$ cat lien.symbolique
cat: lien.symbolique: Aucun fichier ou répertoire de ce type
```

Un fichier lien symbolique contenant un chemin d'accès relatif n'est pas utilisable depuis n'importe quel point de l'arborescence !!!

```
$ rm /tmp/bidon/fic3
$ cat fic3.lien
cat: fic3.lien: Aucun fichier ou répertoire de ce type

Si le fichier cible est supprimé,
```

Le lien symbolique n'est pas affecté par cette suppression mais il n'est plus utilisable !!!

- **Lien symbolique sur un répertoire**

```
$ ln -s /tmp/bidon

$ ll bidon
lrwxrwxrwx  1 c700541 avance 10 sep  5 18:19 bidon -> /tmp/bidon

$ ll bidon/
total 8
-rw-r--r--  1 c700541 avance 24 sep  5 18:14 fic3

$ rm bidon/
rm: ne peut détruire le répertoire `bidon/': est un répertoire

$ unlink bidon/
rm: ne peut détruire le répertoire `bidon/': est un répertoire

$ unlink bidon

$ ll bidon
ls: bidon: Aucun fichier ou répertoire de ce type
```

`bidon/` est obtenu en appuyant sur la touche `<tab>` pour utiliser la complétion de la commande.

- **Suivi d'un lien symbolique**

L'option P des commandes `cd` et `pwd` permet de suivre le lien symbolique.

```
$ pwd
/home/c700541

$ ln -s /tmp tmp

$ cd tmp

$ pwd
/home/c700541/tmp

$ pwd -P
/tmp

$ cd

$ pwd
/home/c700541

$ cd -P tmp

$ pwd
/tmp

$ pwd -P
/tmp
```

Remarque : si on travaille sur une arborescence comportant de nombreux liens symboliques sur des répertoires, il peut être utile de créer, dans les shells scripts de démarrage de la session shell, des alias qui imposent l'option P pour cd et pwd.

```
Dans $HOME/.kshrc (Unix)
alias -x cd='cd -P'
alias -x pwd='pwd -P'

Dans ~/.bashrc (Linux)
alias cd='cd -P'
alias pwd='pwd -P'
```

3.3 Manipulation des noms de fichiers

3.3.1 La commande `basename`

Examine le chemin d'accès à un fichier et en retourne le dernier élément.

```
$ basename /tmp/bidon/fic3
fic3
```

3.3.2 La commande `dirname`

Examine le chemin d'accès à un fichier et retourne tous les éléments sauf le dernier.

```
$ dirname /tmp/bidon/fic3
/tmp/bidon
```

3.4 La commande find

`find` réalise une descente récursive de l'arborescence, à partir du(des) répertoires(s) cité(s). Si aucun répertoire n'est cité, le répertoire courant est pris par défaut.

`find` recherche tous les fichiers correspondant à la (aux) sélection(s) éventuellement indiquée(s). Pour chaque fichier répondant vrai à la (aux) sélection(s) (toujours vrai si pas de sélection), `find` exécute la (les) action(s) éventuellement précisée(s).

```
find répertoire(s) [sélection(s)] [action(s)]
```

3.4.1 Les primitives de sélections

-name fichier	Sélectionne sur le nom du fichier. Les caractères génériques *, ?, [ ] sont traités
-iname fichier	Identique à –name sans différencier les minuscules et les majuscules
-perm [+-]onum	onum : notation des droits en octal Sélectionne les fichiers dont les droits d'accès : onum : sont exactement égaux à onum - onum : sont au moins égaux à onum + onum : sont au moins en partie cités par onum - perm +111 signifie : le droit d'exécution positionné pour le propriétaire et/ou le groupe et/ou les autres utilisateurs.
-type [b|c|d|f]	Sélectionner les fichiers suivant un type donné : b = bloc ; c = caractère ; d = répertoire ; f = ordinaire.
-links [-|+]n	Sélectionner les fichiers de n liens physiques. (*)
-user nom	Sélectionne les fichiers sur le nom de propriétaire.
-group nom	Sélectionne les fichiers sur le nom de groupe.
-inum n	Sélectionne les fichiers sur le numéro d'inode.
-size [-|+]n[c]	Sélectionne les fichiers de n blocs. Si c : n caractères. (*)
-atime [-|+]n	Fichiers dont le dernier accès date de n*24h. (*)
-amin [-|+]n	Idem -atime, n est donnée en minutes.
-ctime [-|+]n	Fichiers dont la dernière modification des attributs date de n*24h. (*)
-cmin [-|+]n	Idem -ctime, n est donnée en minutes.
-mtime [-|+]n	Fichiers dont la date de dernière modification des données date de n*24h. (*)
-mmin [-|+]n	Idem -mtime, n est donné en minutes.
-newer fic	Fichiers dont la dernière modification des données est plus récente que celle de fic.

(*) n : n exactement ; +n supérieur à n ; -n : inférieur à n.

3.4.2 Les primitives d'actions

`-print`	Affichage du chemin d'accès au fichier sélectionné (défaut).
`-exec cmd {} \;`	Exécution de la commande cmd. {} correspond au nom du fichier en cours de traitement. cmd renvoie un code retour égal à 0.
`-ok cmd {} \;`	Idem exec mais il est demandé confirmation à l'utilisateur. Si la réponse commence par y, Y, o, O la commande est exécutée.

3.4.3 Les primitives de combinaison

Elles permettent de combiner des actions entre elles, des sélections entre elles ainsi que des sélections/actions entre elles.

`-a / -and`	ET : les deux objets combinés doivent répondre vrai. Implicite si deux expressions non pas de séparateur logique.
`-o / -or`	OU : au moins un des deux objets combinés doit répondre vrai.
`!`	NOT : négation.

Exemples :

```
$ find / -name toto -a -type f -ok rm {} \; 2>/dev/null
```

Cette commande ne fera aucun affichage écran !

En effet, les erreurs mais aussi le dialogue avec l'utilisateur transitent par le descripteur 2.

```
$ find / -name toto -type f -ok rm {} \;
```

Les erreurs, mais également le dialogue avec l'utilisateur, seront affichés à l'écran, ce qui permettra de répondre.

Ici, les actions -name et -type ne sont pas reliées par -a

```
$ find / \( -name toto -a -type f -ok rm {} \; \) -o \( -name
toto -a -type d -ok rm -r {} \; \)
```

Les parenthèses permettent de modifier les priorités.

Elles doivent être protégées.

Unix - Linux

4. Fonctions avancées de l'éditeur vi

Objectifs

- Rappel des fonctionnalités de base de l'éditeur de texte vi.
- Fonctions avancées de vi / vim : utilisation des tampons, déplacement de texte, substitution de chaînes de caractères.

4.1 Les éditeurs de texte

4.1.1 Les éditeurs les plus souvent rencontrés

Editeurs graphiques

gedit	éditeur graphique par défaut de Gnome
kwrite	éditeur graphique élémentaire de l'environnement de bureau KDE
kate	éditeurs graphiques avancé de l'environnement de bureau KDE
mousepad	éditeur de texte par défaut de l'environnement de bureau xfce
leafpad	éditeur de texte par défaut de l'environnement de bureau lxde
Gvim, Xvim	vim avec une interface graphique

Les éditeurs graphiques étant en grande partie d'une utilisation intuitive nous laisserons le soin à chacun de les découvrir selon ses besoins.

Editeurs ascii

nano	très simple, adapté à une utilisation basique occasionnelle
emacs	éditeur de texte très complet sert d'environnement de développement pour beaucoup de langages
vi	éditeur de texte présent par défaut sur la majorité des systèmes Unix.
vim	« VI iMproved», un clone de vi amélioré et massivement utilisé, présent par défaut sur un très grand nombre de systèmes Linux, sert d'environnement de développement pour beaucoup de langages

L'éditeur "universel" présent sous la majorité des Unix et GNU/Linux est **vi** et son évolution **vim**.

Après une découverte des fonctionnalités de bases dans le stage précédent, nous allons découvrir un certain nombre de fonctionnalités plus élaborées.

4.2 Fonctions de base de vi

4.2.1 Entrée dans vi

Appel de l'éditeur de texte **vi**

```
vi fichier
```

```
view fichier_en_lecture_seule
```

La fenêtre est au début du fichier.

Si le fichier n'existe pas, il sera créé par des directives de sorties.

La dernière ligne au bas de l'écran sert pour :

- Frapper des directives en mode « éditeur ligne »
  précédées de « **:** » (deux points) et validées par <Entrée>.
- Afficher des statistiques.
- Afficher des messages d'erreurs.

```
$ vi
```

```
~
~
~                          VIM - Vi Amélioré
~
~                           version 7.2.411
~                     par Bram Moolenaar et al.
~                  Modifié par <bugzilla@redhat.com>
~                     Vim est un logiciel libre
~
~             Devenez un utilisateur de Vim enregistré !
~       tapez  :help register<Entrée>  pour plus d'informations
~
~ tapez  :q<Entrée>                 pour sortir du programme
~ tapez  :help<Entrée>  ou  <F1>    pour accéder à l'aide en ligne
~ tapez  :help version7<Entrée>     pour lire les notes de mise à jour
~
~
~
~
~                                                      0,0-1         Tout
```

4.2.2 Sorties de vi

- **Avec sauvegarde du texte**

ZZ

ou

:x <Entrée>

ou

:w <Entrée> suivi de **:q <Entrée>**

ou

:wq <Entrée>

- **Sans sauvegarde**

:q <Entrée> s'il n'y a pas eu de modifications

ou

:q!<Entrée> s'il y a eu des modifications à ignorer

- **Accès au shell**

Durant une session de vi, nous pouvons accéder momentanément au shell sans sortir de l'éditeur.

:! commande

ou

:sh

$ commande

$ commande

$ <CTL><d>

4.2.3 Déplacer le curseur

Le curseur est le "centre du monde" pour vi.

- **Déplacement du curseur dans le texte**

Vers la gauche	<**h**> *ou* <**backspace**>	*ou* <**flèche gauche**>
Vers le bas	<**j**> *ou* <**Entrée**>	*ou* <**flèche basse**>
Vers la droite	<**l**> *ou* <**Espace**>	*ou* <**flèche droite**>
Vers le haut	<**k**> *ou* <**-**>	*ou* <**flèche haute**>

- **Déplacement du curseur sur une ligne**

Début de ligne	**<^>**
Fin de ligne	**<$>**

4.2.4 Coloration syntaxique

Vim utilisera la coloration syntaxique si le fichier est identifié comme un shell script.

Un fichier de texte sera considéré comme shell script s'il est exécutable ou si la première ligne est le shabang **#!/bin/sh** , **#!/bin/ksh** , **#!/bin/bash** ...

4.2.5 Copie de secours

Conservez une copie de secours du fichier original.

Sauvegardez le fichier (**:w**) régulièrement pour éviter de tout perdre en cas d'erreur de manipulation.

4.3 Les modes de vi

Mode « insertion »	Création ou insertion de texte.
Mode « commandes »	Suppression, copie, remplacement.
Mode « éditeur ligne »	Sauvegarde, recherche, substitution, numérotation lignes...

4.3.1 Mode insertion

Commandes d'entrée en mode insertion (entrée de texte) :

`a`	(append) Le texte sera **ajouté après** le curseur.
`i`	(insert) Le texte sera **inséré avant** le curseur.
`o`	(open minuscule) Nouvelle ligne **sous** la ligne courante.
`O`	(Open majuscule) Nouvelle ligne **au-dessus** de la ligne courante.

Ligne courante : ligne sur laquelle se trouve le curseur.

Tous les caractères frappés après une de ces commandes seront insérés dans le fichier à partir de la position du curseur.

Passage d'une ligne à la suivante : **<Entrée>**

Sortie du mode insertion : **<Echap>**

4.3.2 Utilisation basique de vi

Pour créer un nouveau fichier, il faut donc :

1. appeler vi avec le nom du fichier à créer
2. a ou i
3. saisir le texte
4. <Echap>
5. sortir en sauvegardant le texte par :wq

Ceci est le minimum vital à retenir !

4.3.3 Mode commandes

C'est ce mode qui est présent à l'entrée dans « vi ».

Principales commandes du "mode commandes" :

`u`	(undo) Annule la dernière commande frappée.
`U`	Annule toutes modifications sur une ligne.
`ZZ`	Sortie de vi avec réécriture du fichier si modification (donc sauvegarde des modifications effectuées).
`[n]x`	(extract) Suppression du caractère sous le curseur* ou bien utiliser la touche <Suppr> du clavier.
`[n]r`	(replace) Remplacement du caractère sous le curseur par le caractère frappé ensuite *.
`[n]cw`	(change word) Remplacement du mot sous le curseur par le mot frappé suivi de <Echap> *.
`[n]yw`	(yank word) Mise en mémoire du mot sous le curseur *.
`[n]dw`	(delete word) Suppression et mémorisation du mot sous le curseur *.
`[n]dd`	Suppression de la ligne et mise en mémoire de celle-ci *.
`[n]yy`	(yank) Mise en mémoire de la ligne*.
`p`	(put minuscule) Insertion au-dessous de la ligne courante de la ligne en mémoire.
`P`	(PUT majuscule) Insertion au-dessus de la ligne courante de la ligne en mémoire.
`J`	(Join) Jonction de la ligne courante avec la suivante.
`/mot`	Positionne le curseur sur la ligne suivante contenant "mot".
`5dd`	Suppression de 5 lignes à partir de la ligne où se trouve le curseur.

(*) n n = nombre de répétitions de la commande si indiqué.

4.3.4 Mode éditeur ligne

Les commandes de l'éditeur ligne, appelées aussi directives, commencent par **" : "** (deux points) et doivent être validées par <Entrée>.

" : " s'affiche sur la dernière ligne en bas de la fenêtre de shell.

Les plus utiles sont :

`:set nu`	Numérote les lignes.
`:set nonu`	Suppression de la numérotation des lignes.
`:set showmode`	Le message "-- INSERTION –" s'affiche sur la dernière ligne (validé par défaut avec vim).
`:15`	Positionne le curseur en ligne 15 ("$" désigne la dernière ligne).
`:w [fichier]`	(write) Sauvegarde du fichier modifié sous le nom donné (ou le nom initial si le nom du fichier est omis).
`:q!`	(quit) Sortie de vi sans sauvegarde des modifications.
`:f`	Donne le nom du fichier en cours.
`:1,$s/fic/fichier/`	De la première à la dernière ligne, substitution de la 1ère chaîne de caractère "fic" par "fichier".
`:set showmatch`	Permet de retrouver les parenthèses, crochets ou accolades fermantes correspondant au caractère ouvrant.
`:set all`	Affiche toutes les options de vi.

4.4 Fonctions avancées

4.4.1 Configuration par défaut

Afin de conserver la configuration de *vi*, nous pouvons mettre des directives du mode éditeur-ligne dans les fichiers suivant :

```
~/.vimrc ou /etc/vimrc   vim sous Linux
ou
~/.exrc ou /etc/.exrc          vi sous Unix
```

Autre méthode, initialiser la variable d'environnement `EXINIT` dans le fichier `~/.profile` ou `~/.bashrc` (ou équivalent).

L'exemple suivant permettra d'afficher systématiquement les numéros de lignes à chaque ouverture de `vi`.

```
$ more ~/.bashrc
...
export EXINIT="set nu"
...
```

Il sera parfois plus aisé de mettre toute une suite de directives dans un fichier.

```
$ more ~/.vimrc
set showmode
set nu
set ll=20000000
```

`showmode`	Affichera "INSERTION" quand on sera en mode de saisie.
`nu`	Affichera les numéros de lignes.
`ll=xxxxxx`	Taille du fichier temporaire de vi, permettant de travailler sur de très gros fichiers comprenant un très grand nombre de lignes.

4.4.2 Copie et déplacement de texte

La directive **:r** permet l'insertion d'un autre fichier dans le tampon de manœuvre.

```
$ vi fic1
```

… positionner le curseur à l'endroit de l'insertion…

```
:r fic2
```

4.4.3 Insérer le résultat d'une commande dans le fichier

```
:r! commande_bash
```

```
:r! date
```
…la date est insérée à partir de la position courante du curseur.

4.4.4 Copie d'une partie du tampon de manœuvre dans un autre fichier

```
:adresse1,adresse2w fic
```

Avec adresse correspondant à un numéro de ligne ou une chaîne de caractères.

```
:3,12w  fic.save
```

Copie les lignes 3 à 12 du tampon dans le fichier fic.save.

```
:/chaîne1/,/chaîne2/w fic
```

Copie les lignes de la ligne contenant chaîne1 à la ligne contenant chaîne2 dans le fichier fic.

```
:w>>fic
```

Copie tout le tampon en fin du fichier fic.

4.4.5 Copie et déplacement de texte

Les directives y (yank), d (delete) et p (put) permettent de copier ou de déplacer du texte dans le tampon de manœuvre banalisé (sans nom).

4.4.6 Les tampons nommés

Il est possible d'utiliser des tampons dits « tampons nommés » en plus du tampon sans nom.

Il suffit d'utiliser la directive « suivie d'un nom de tampon » choisi par l'utilisateur. Ce nom est une seule lettre minuscule. Il y a donc 26 tampons nommés possibles.

`"a5yy`	Met 5 lignes dans le tampon a.
`"f3yy`	Met 3 lignes dans le tampon f.
`"ap`	Copie le contenu du tampon a après le curseur.
`"Add`	Détruit la ligne courante et l'ajoute au tampon a. a minuscule : remplace le tampon. A majuscule : ajoute au tampon.

4.4.7 Échanges de tampons nommés entre fichiers

Il est possible de modifier un autre fichier sans sortir de vi grâce aux tampons nommés (le tampon sans nom n'est pas mémorisé d'un fichier à l'autre).

`:w`	Ecriture du tampon de manœuvre dans le fichier courant
`"u5yy`	Mémorisation de 5 lignes dans le tampon nommé u.
`:e fic.new`	Edition du fichier fic.new dans le tampon de manœuvre.
`"up`	Après positionnement du curseur, insertion des 5 lignes dans la copie de fic.new.
`:w`	Ecriture de fic.new.
`:e#`	Rappel du fichier courant dans le tampon de manœuvre.

Attention

`:q!`	Force la sortie de vi sans modification.
`:w!`	Force l'écriture du tampon de manœuvre.
`:e! fic.new`	Force l'écriture de fic.new dans le tampon de manœuvre.

4.4.8 Substitution de chaînes de caractères

Vi permet de faire des substitutions multiples sur tout ou partie d'un fichier grâce à l'utilisation de la directive **:s**

Substitution de la première occurrence de "chaine1" par "chaine2" dans les lignes d'adresse1 à adresse2 :

```
:adresse1,adresse2s/chaine1/chaine2/
```

Substitution de la première occurrence de "chaine1" par "chaine2" dans la prochaine ligne contenant "chaine1" :

```
:/chaine1/s//chaine2/
```

Substitution de toutes les occurrences de "chaine1" par "chaine2" dans la prochaine ligne contenant "chaîne1" :

```
:/chaine1/s//chaine2/g
```

Substitution de toutes les occurrences de "chaine1" par "chaine2" dans tout le tampon :

```
:g/chaine1/s//chaine2/g
```

Ajout d'une chaine à une autre :

```
:g/chaine1/s// chaine1 et chaine2/g
```

ou

```
:g/chaine1/s//& et chaine2/g
```

Substitution de "chaine2" par "chaine3" sur toutes les lignes ne contenant pas "chaine1" :

```
:v/chaine1/s/chaine2/chaine3/g
```

Unix - Linux

5. Les conditions

Objectifs

- La structure si... alors... sinon, appelée aussi "structure conditionnelle", en abrégé "conditions".
- Les tests dans le shell

5.1 Code de retour

5.1.1 Code de retour d'une commande

A la fin de l'exécution d'une commande, un **code de retour** est toujours positionné.

Le code de retour d'une commande peut être vérifié en affichant sa valeur contenue dans la variable **?** .

```
echo $?
```

Exemples :

Admettons que le fichier fic1 existe dans notre répertoire de travail, et que le fichier fic3 n'existe pas.

```
$
$ cat fic1
Ceci est le fichier fic1
$ echo $?
0
$ cat fic3
fic3 : Aucun fichier ou dossier de ce type
$ echo $?
1
$
```

5.1.2 Valeurs du code de retour

Lorsqu'une commande se déroule "normalement" son code de retour est **égal à 0** (zéro), on dit alors que le code de retour est "**vrai**".

Lorsqu'une commande sort sur un message d'erreur son code de retour est **différent de 0**, on dit alors que le code de retour est "**faux**".

Attention il n'y a pas de jugement de valeur lorsqu'on dit "vrai" ou "faux".

5.1.3 Forcer le code de retour

Les commandes usuelles sont programmées pour donner des codes de retour particuliers si elles ne peuvent se dérouler normalement.

```
$
$ ls azertyu
ls : impossible d'accéder à azertyu : aucun fichier ou dossier
de ce type
$ echo $?
2
$ ls -z
ls : option invalide – 'z'
Saisissez " ls –help " pour plus d'informations.
$ echo $?
2
$ azertyu
bash : azertyu : commande introuvable
$ echo $?
127
$
```

On peut faire de même dans l'écriture d'un shell script.
La commande exit n provoque la sortie du shell script en positionnant le code de retour à une valeur donnée.

```
$ cat prog3
exit 3
$
$ prog3
$ echo $?
3
$
```

5.2 if... then... else

Un shell script n'est autre qu'une suite de lignes de commandes. Mais bien souvent une portion de programme ne devra s'exécuter que dans des conditions bien précises.

Les "structures conditionnelles", en abrégé les "conditions", constituent un moyen de dire dans notre script :

"SI cette commande marche, ALORS fais ceci, SINON fais cela".

Il s'agit d'une fonction classique présente dans tout langage de programmation.

5.2.1 La structure if simple

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

```
if  commande1
then commande2

fi
```

Si le résultat de la commande1 donne le code de retour "**vrai**" **alors** on exécute la commande2.
Si le résultat de la commande1 donne le code de retour "**faux**", on passe directement **après le fi**.

Fi n'est que le "verlan" de if, cela indique la fin de la structure.

Exemple :

```
nom=toto
if test "$nom" = toto
then echo " Bonjour $nom et bien venu chez vous"
fi
```

Plus utile :

```
if mkdir rep3 2>/dev/null
then cd rep3 ; pwd
fi
```

5.2.2 La structure if avancée

L'instruction *if* permet d'effectuer une opération si une condition est réalisée, sinon on effectue une autre opération.

```
if  commande1
  then commande2
  else commande3
fi
```

Si le résultat de la commande1 donne le code de retour "**vrai**" **alors** on exécute la commande2, la commande3 est ignorée.
Si le résultat de la commande1 donne le code de retour "**faux**", la commande2 est ignorée, on exécute la commande3.

Exemple :

```
$
$ more bonjour
if test "$USER" = toto
  then echo " Salut mon pote !"
  else echo " Bonjour $USER"
fi
$
$
$ echo $USER
Maurice
$
$ bonjour
 Bonjour Maurice
$
```

Remarque

Il n'est pas obligatoire, en shell, de décaler les ligne, indentation, mais dans le cas où il y a de nombreuses lignes de commandes après le then et le else, cela facilite grandement la lecture.

Plus utile :

Ici le programme teste si le répertoire donné en paramètre existe déjà.
S'il existe, un message est affiché et on affiche son contenu.
S'il n'existe pas, il est créé et on affiche ses caractéristiques.

```
$
$ cat creer_rep
if test -d $1
  then echo "Le répertoire existe déjà..."
       ls -l $1
  else echo "Création du répertoire $1"
       mkdir $1
       ls -ld $1
fi
$ creer_rep rep22
Le répertoire existe déjà...
total 12
-rw-r--r--. 1 pat gp1    0  5 sept. 12:31 grosfichier
-rw-r--r--. 1 pat gp1    0  5 sept. 12:31 petitfichier
-rw-r--r--. 1 pat gp1 1251 17 avril  2014 texte
-rw-r--r--. 1 pat gp1 1301 17 avril  2014 texte1
-rw-r--r--. 1 pat gp1 1251 17 avril  2014 texte_original
$
$
$
$ creer_rep rep33
Création du répertoire
drwxr-xr-x. 2 pat gp1 4096  9 janv. 12:29 rep33
$
```

Notez la facilité de lecture apportée par l'indentation.

Nous avons utilisé dans cet exemple la commande *test*. Bien qu'après un *if* on puisse utiliser n'importe quelle commande, il faut bien admettre que c'est cette commande *test* qui est le plus souvent rencontré.

5.3 Les tests dans le shell

5.3.1 Trois types de test

La commande `test` permet de faire trois sortes de tests :

- Tests sur les caractéristiques des fichiers
- Tests de comparaisons de valeurs numériques
- Tests de comparaisons de chaines de caractères

5.3.2 Syntaxe de la commande `test`

La commande *test* peut se faire avec deux syntaxes :

```
test expression
```

ou alors

```
[ expression ]
```

Notez bien :

Les espaces sont obligatoires après le crochet ouvrant et avant le crochet fermant.

Dans les exemples qui suivent les deux syntaxes seront indifféremment utilisées.

5.3.3 Test sur les fichiers

`test -e fichier`	Vrai si fichier existe
`test -f fichier`	Vrai si fichier est un fichier ordinaire (file)
`[ -d fichier ]`	Vrai si fichier est un répertoire (directory)
`[ -r fichier ]`	Vrai si fichier est autorisé en lecture (read)
`test -w fichier`	Vrai si fichier est autorisé en écriture (write)
`test -x fichier`	Vrai si fichier est exécutable
`test -s fichier`	Vrai si fichier n'est pas vide (size)
`[ fichier1 -ef fichier2 ]`	Vrai si ce sont des liens physiques (equal files)
`test -L fichier`	Vrai si fichier est un lien symbolique (link)
`[ fichier1 -nt fichier2 ]`	Vrai si fichier1 plus récent que fichier2 (newer than)
`[ fichier1 -ot fichier2 ]`	Vrai si fichier1 plus ancien que fichier2 (older than)

Ce que nous avons noté ici `fichier` se présente sous forme de variables dans l'immense majorité des cas.

5.3.4 Test numériques

-eq	Egal à (equal to)
-ne	Différent de (not equal to)
-gt	Plus grand que (greater than)
-ge	Plus grand ou égal (greater than or equal to)
-lt	Plus petit que (less than)
-le	Plus petit ou égal (less than or equal to)

```
$ test "$a" -eq 36
```

Vrai si le contenu de la variable a est égal à 36.

```
$ [ "$a" -lt "$b" ]
```

Vrai si le contenu de la variable a est plus petit que celui de la variable b.

5.3.5 Test sur les chaînes de caractères

["chaîne1" = "chaîne2"]	Vrai si chaîne1 est égale à chaîne2
test "chaîne1" != "chaîne2"	Vrai si chaîne1 différente de chaîne2
test -z "$a"	Vrai si variable a est vide
test -n "$a"	Vrai si variable a n'est pas vide

```
$
$ test "$LOGNAME" = "user1"
$
```

Notez bien :

Les espaces sont obligatoires de part et d'autre du signe **=**

5.3.6 Les conditions composées

Les opérateurs peuvent être combinés avec :

-a	et
-o	ou
!	négation
\(… \)	groupement

```
$
$ test \( -r fichier1 -o -r fichier2 \) -a -w fichier3
$
```

Vrai si `fichier1` ou `fichier2` accessible en lecture et `fichier3` accessible en écriture.

5.4 Les opérateurs du shell

Les opérateurs && et || permettent de programmer les mêmes structures que le if dans des cas plus simples.

5.4.1 L'opérateur &&

```
commande1 && commande2
```

Exécution de commande1, si commande1 fourni un code de retour égal à 0 alors exécution de commande2.

5.4.2 L'opérateur ||

```
commande1 || commande2
```

Exécution de commande1, si commande1 fourni un code de retour différent de 0 alors exécution de commande2.

```
test -d demo && echo "demo est un répertoire"

test -d demo || echo "demo n'est pas un répertoire"

test -d demo &&ls -l demo ||echo "demo n'est pas un répertoire "
```

5.5 Branchement à choix multiple

- **case in esac**

La construction `case in esac` permet de rapprocher une chaîne de caractères d'une liste d'expressions génériques auxquelles sont associées une série de commandes et d'exécuter les commandes correspondantes si le rapprochement est réussi.

```
case chaine in
      expr1) cdes ;;
      expr2) cdes ;;
      expr3 | expr4) cdes ;;
esac
```

Si "chaine" concorde avec :	expr1	Exécuter les commandes correspondant à expr1. Passer après esac.
sinon :	expr2	Exécuter les commandes correspondant à expr2. Passer après esac.
sinon : ou :	expr3 expr4	Exécuter les commandes correspondant à expr3 ou expr4. Passer après esac.

Rappel :

Les caractères génériques * ? [ ] peuvent être utilisés pour composer les expressions génériques.

| (barre verticale) est ici l'opérateur logique OU.

Exemple :

```
echo "POUR DIALOGUER EN FRANÇAIS TAPEZ : f"
echo "FOR AN ENGLISH DIALOG TYPE : e"
echo
echo -e "VOTRE REPONSE / YOUR CHOICE : \c"
read reponse
echo
case "$reponse" in
     [fF]* )        LANG=fr_FR ;;       #français
     [aAeE]* )      LANG=en_US ;;       #english
     * )  echo -e "\n ERREUR: LANGUE NON PRISE EN COMPTE"
     echo -e "ERROR : WRONG LANGUAGE \n" ;;
esac
```

Unix - Linux

6. Programmation d'une boucle

Objectifs

- Programmer des boucles for
- Programmer des boucles while
- Programmer des boucles until
- Faire des calculs arithmétiques
- Programmer des boucles select

6.1 La boucle for

6.1.1 for in do done

Cette forme de la boucle *for* permet d'effectuer la même séquence de commandes sur chaque valeur d'une liste.

Il y a en général autant de boucles que de valeurs dans la liste.

```
for variable in val1 val2 val3 …
do
        commande1
        commande2
        commande3
             ...
done
```

Pour la variable prenant successivement les différentes valeurs val1 val2 val3 … exécuter les commandes du bloc do … done.

La procédure suivante donne le droit d'exécution aux fichiers proced1 proced2 et proced3 du répertoire courant.

```
$
$ more chg_droits
for fic in proced[123]
do chmod u+x $fic
done
$
```

6.1.2 for do done

Cette deuxième forme de la boucle for permet d'effectuer la même séquence de commandes sur tous les paramètres ($1 $2 $3…) passés à l'appel de la procédure.

```
for variable
do
        commande1
        commande2
        commande3
        ...
done
```

Pour la variable prenant successivement les différentes valeurs passées en paramètres positionnels ; exécuter les commandes du bloc do … done.

```
$ cat copier_fic
for fic
do
     if [ -f $fic ]
     then cp $fic /home/user2/tmp
          chgrp user2 /home/user2/tmp/$fic
          chown base /home/user2/tmp/$fic
     else echo "$fic n'est pas un fichier ordinaire"
     fi
done
$ cd ; ls -l
-rw-r--r--    1 user1    user1           0 fév 13 11:07 banane
drwxr-xr-x    2 user1    user1         516 fév 13 09:14 bin
-rw-r--r--    1 user1    user1           0 fév 13 11:07 figue
-rw-r--r--    1 user1    user1           0 fév 13 11:07 noix
drwxr-xr-x    2 user1    user1         516 fév 13 10:18 tmp

$ copier_fic figue banane noix gateau
gateau n'est pas un fichier ordinaire

$ ls -l tmp
total 3
-rw-r--r--    1 user2    base            0 fév 13 11:08 banane
-rw-r--r--    1 user2    base            0 fév 13 11:08 figue
-rw-r--r--    1 user2    user2           0 fév 13 11:08 noix
```

6.1.3 for ((… ; … ;…)) do … done

Cette forme de for n'existe qu'en bash.

Elle est très pratique pour visualiser rapidement le compteur de progression de la boucle.

Un exemple de cette programmation est donné, plus loin dans ce chapitre, après la présentation des outils arithmétiques.

```
for ((variable=valeur_initiale; test d'entrée de boucle; progression))
do
        commande1
        commande2
        commande3
        …
done
```

6.2 Les boucles `while` et `until`

6.2.1 *while do done*

La boucle `while` exécute une séquence de commandes tant qu'une condition est vraie.

```
while commande1
do
        commande2
        commande3
        commande4
                ...
done
```

Tant que le code de retour de `commande1` est vrai, exécuter les commandes du bloc do...done.

6.2.2 *La commande `true`*

Fournit toujours le code de retour vrai et ne génère aucune action particulière.

6.2.3 La commande `sleep`

La commande `sleep` permet d'attendre pendant une durée déterminée.

`sleep n [smhd]`

L'unité par défaut est la seconde. Les unités reconnues sont :

s secondes

m minutes

h heures

d jours

La procédure `param1` affiche ligne par ligne tous les paramètres passés à l'appel.

```
$ cat param1
while test $# -ne 0
do   echo $1
     shift
done

$ param1 figue banane noix
figue
banane
noix
$
```

La procédure `temps1` affiche la date à l'écran toutes les 5 secondes.

```
$ cat temps1
while true
do   date
     sleep 5
Done

$ temps1
mer fév 13 11:27:47 EST 2002
mer fév 13 11:27:52 EST 2002
mer fév 13 11:27:57 EST 2002
mer fév 13 11:28:02 EST 2002
mer fév 13 11:28:08 EST 2002
<Ctl><c>
$
```

6.2.4 until do done

La boucle `until` fonctionne à l'inverse de la boucle `while`.

```
until commande1
do
      commande2
      commande3
      commande4
      ...
done
```

Exécute les commandes du bloc do...done jusqu'à ce que le code retour de commande1 soit vrai.

La procédure `param2` écrite avec `until`.

```
$ more param2
until test $# -eq 0
do    echo $1
      shift
Done

$ param2 bonjour les enfants
bonjour
les
enfants
```

6.2.5 La commande `false`

Fournit toujours le code de retour faux et ne génère rien, aucune action particulière.

```
$ more temps2
until false
do   date
     sleep 5
done

$ temps2
mer fév 13 12:27:47 EST 2002
mer fév 13 12:27:52 EST 2002
mer fév 13 12:27:57 EST 2002
mer fév 13 12:28:02 EST 2002
mer fév 13 12:28:08 EST 2002
<Ctl><c>
```

6.3 Arithmétique entière

6.3.1 Les opérateurs arithmétiques

Les commandes let et expr réalisent des opérations arithmétiques :

+ **-** ***** **/** **%** *(reste de la division ou modulo)*

Les différentes syntaxes possibles sont utilisées dans l'exemple ci-dessous :

```
$ typeset -i i=10 j=2 k l

$ let "k=i+j"              # commande let

$ ((k=i+j))                # commande let

$ k=`expr $i + $j`         # commande expr

$ echo $k
12
```

6.3.2 Les opérateurs de comparaison de let

< **<=** **>** **>=** **==** **!=**

Les opérateurs de comparaison permettent de comparer soit des contenus de deux variables soit le contenu d'une variable et d'une valeur concrète.

```
$
$ let "i<j"; echo $?
1

$ ((i<j)); echo $?
1
```

6.3.3 Les opérateurs logiques

`!`	opérateur de négation (not)
`&&`	opérateur et (and)
`||`	opérateur ou (or)

```
$
$ ((l=2*k))
$ ((k==i+j  && l==k*2))
$ echo $?
0
$
```

ATTENTION :

=	*affectation de la variable*
==	*comparaison arithmétique*

6.3.4 Exemple de compteur dans une boucle

Cas classique d'utilisation des opérateurs arithmétique dans une procédure : la programmation d'un compteur de passage dans une boucle.

La procédure creat_fic crée les fichiers vides fic1 fic2 fic3 … fic10

```
$ cat creat_fic2
typeset -i compteur=1
while let "compteur<=10"
do
     >fic$compteur
     ((compteur=compteur+1))
done
$
```

6.3.5 Exemple de programmation de la suite de Fibonacci

La suite de Fibonacci est une suite de nombres entiers dont les deux premiers sont 1 et 1.

Un nombre de rang n dans la suite de Fibonacci est la somme du nombre de rang n-1 et du nombre de rang n-2.

Le shell-script fibo, calcule la suite de Fibonacci jusqu'au rang 11.

```
$ cat fibo
declare -i i tab
tab[0]=1
tab[1]=1
echo ${tab[0]}
echo ${tab[1]}
for ((i=2; ((i<=10)); i++))
do
     ((tab[i]=tab[i-1]+tab[i-2]))
     echo ${tab[i]}
done

$ fibo
1
1
2
3
5
8
13
21
34
55
89
$
```

6.4 Sorties de boucle

6.4.1 La commande `break`

La commande break permet de sortir d'une boucle for while ou until.

La procédure `stocke1` mémorise, jusqu'à la saisie du mot FIN, les lignes entrées au clavier, dans le fichier fic_lignes.

```
$ cat stocke1
while true
do
     echo 'Entrez une ligne :'
     read reponse
     if test "$reponse" = FIN
     then
          break                    # Sortie de la boucle
     else
          echo "$reponse" >>fic-lignes
     fi
done
```

6.4.2 Sortie de boucles imbriquées

La commande `break n` permet de sauter n niveaux de boucles imbriquées.

Le script `stocke2` reprend la procédure stocke1 en ajoutant une boucle qui insère un message à chaque fin de page (toutes les 66 lignes).

```
$ cat stocke2
typeset -i ligne
while true
do
     ligne=1
     while ((ligne<=65))
     do
          echo "Entrez une ligne : "
          read reponse
          if test "$reponse" = FIN
          then
               break 2        # Sortie des 2 boucles
          else
               echo "$reponse" >>fic-lignes
          fi
          ((ligne=ligne+1))
     done
     echo "##### Fin de page #####" >>fic-lignes
done
$
```

6.4.3 La commande `continue`

La commande `continue` permet de reprendre l'exécution à l'itération suivante d'une boucle for, while ou until, sans exécuter toutes les commandes du bloc do … done.

Exemple :

La procédure `supprime` efface tous les fichiers donnés en paramètres sauf save et source.

```
$ cat supprime
set -x
for i
do
     if test "$i" = source -o "$i" = save
     then continue
     fi
     echo $i
     rm $i
done
$
$ cd $HOME/tmp ; ls
banane figue noix save source
$
$ supprime *
+ test banane = source -o banane = save
+ echo banane
banane
+ rm banane
+ test figue = source -o figue = save
+ echo figue
figue
+ rm figue
+ test noix = source -o noix = save
+ echo noix
noix
+ rm noix
+ test save = source -o save = save
+ continue
+ test source = source -o source = save
+ continue
$ ls
save source
$
```

6.5 La boucle `select`

```
select nom [ in mots ]
do
      commandes
done
```

Après `in, mots` constitue une suite d'éléments qui seront utilisés pour construire une liste numérotée (menu), affichée sur la sortie d'erreur standard.

Si `in mots` est omis, les paramètres positionnels sont utilisés.

A la suite du menu, `select` affiche l'invite PS3 (« #? » par défaut) et lit une ligne sur l'entrée standard.

Si la ligne lue est l'un des chiffres de la liste, la variable « nom » est affectée du « mot » correspondant, les « commandes » sont exécutées et l'invite PS3 est réaffichée.

Pour toutes les autres valeurs, la variable « nom » est vidée, les « commandes » sont exécutées et l'invite PS3 est réaffichée.

Si une fin de fichier (EOF) est saisie, la commande se termine.

Un appui sur « Entrée » provoque le réaffichage du menu.

La ligne lue est stockée dans la variable standard `REPLY`.

```
$ more sel1
select cde in date pwd ls break
do
     case "$cde" in
          date)     echo -n "Date du jour : "; date;;
          pwd) echo -n "Répertoire de travail : "; pwd;;
          ls)  echo Contenu du répertoire courant; ls -l;;
          break)    echo Fin programmée du processus; break;;
          *)   echo Choisir un nombre entre 1 et 4;;
     esac
done
$ sel1
1) date
2) pwd
3) ls
4) break
#? 2
Répertoire de travail : /home/user1
#? 1
Date du jour : sam déc 29 16:30:43 CET 2007
#? 3
Contenu du répertoire courant
-rw-rw-r--  1 user1  g1   166 déc 14 15:03 ag
                      :
                      :
-rwxrw-r--  1 user1  g1    61 déc 14 15:13 vrai
#?                 # Un appui sur <Entrée> réaffichage le menu
1) date
2) pwd
3) ls
4) break
#? t
Choisir un nombre entre 1 et 4
#? 4
Fin programmée du processus
```

```
$ cat sel2
select cde in date pwd ls break
do
     case "$REPLY" in
          1) echo -n "Date du jour : "; date;;
          2) echo -n "Répertoire de travail : "; pwd;;
          3) echo Contenu du répertoire courant; ls -l;;
          4) echo Fin programmée du processus; break;;
          *) echo Choisir un nombre entre 1 et 4;;
     esac
done

$ sel2
1) date
2) pwd
3) ls
4) break
#? 2
Répertoire de travail : /home/user1
#? 4
Fin programmée du processus
$
```

Unix - Linux

7. Environnement utilisateur

Objectifs

- Identifier les principaux fichiers de configuration de l'environnement utilisateur
- Personnaliser le contenu de ces fichiers
- Alias, fonctions, droits prédéfinis, prompts

7.1 Session utilisateur Korn shell

L'acceptation d'une session utilisateur Korn shell (nom et éventuellement, mot de passe correct) déclenche sous certaines conditions, l'exécution par ksh d'un certain nombre de shells scripts.

Ces shells scripts ont pour but de construire un environnement de travail à la session utilisateur.

7.1.1 Connexion à partir d'un terminal ASCII

Avant d'établir la connexion ksh va exécuter au moyen de la commande**.** (point - pas de création de fils), les shells scripts suivants :

- `/etc/profile`
- `$HOME/.profile` S'il existe
- `$HOME/.kshrc` S'il existe et si `$HOME/.profile` contient les commandes :

```
ENV=$HOME/.kshrc
export ENV
```

7.1.2 Première connexion à partir d'un terminal graphique (exemple du CDE d'AIX)

Le CDE (Common Desktop Environment) construit l'environnement graphique standard (arborescence `$HOME/.dt` et fichier `$HOME/.dtprofile`) lors de la première connexion.

Les fichiers utilisateurs `$HOME/.profile` et `$HOME/.kshrc`, même s'ils existent ne sont pas exécutés. Pour obtenir leur exécution lors des connexions suivantes, il faut supprimer le commentaire de la dernière ligne du fichier `$HOME/.dtprofile`.

```
# DTSOURCEPROFILE=true
```

Devient :

DTSOURCEPROFILE=true

7.1.3 Connexions suivantes à partir d'un terminal graphique

ksh exécute :

```
. /etc/profile
```

```
. $HOME/.profile          S'il existe
```

`Si $HOME/.profile` existe et contient les commandes :

```
ENV=$HOME/.kshrc
```

export ENV

Chaque ouverture d'une fenêtre terminal provoque le lancement d'un processus `/usr/bin/ksh` qui exécute `. $HOME/.kshrc`

7.1.4 Rôle de la variable ENV et du fichier $HOME/.kshrc

Lors de son lancement, un processus ksh possédant dans son environnement une variable `ENV`, correctement renseigné et exporté, exécute, sans création de fils, le shell script dont le chemin d'accès est enregistré dans la variable `ENV` (en général : `$HOME/.kshrc.)`.

`$HOME/.kshrc` a pour objet d'implanter dans l'environnement :

Les objets non transmissibles par appel de ksh (alias, fonctions),

Les options car elles ne sont pas exportables,

Les variables que l'on ne veut pas exporter (exemple : `TMOUT`).

7.1.5 Terminaison d'une session utilisateur

L'ordre de fin de session est donné par :

`exit` ou `<Ctrl> <d>`

Si l'option `ignoreeof` est positionnée (`set -o ignoreeof)` la sortie de session se fait obligatoirement par `exit`.

La fin de session déclenche l'exécution, si elles existent, de toutes les commandes attachées au pseudo-signal 0 et positionnées par l'intermédiaire de la commande `trap` :

```
trap  'cmd1; cmd2; cmd3; ....'  0
```

7.1.6 Environnement d'une session

L'environnement d'une session (variables exportées) est donné par la commande `env`.

```
$ env
```

7.2 Session utilisateur bash

7.2.1 A la connexion de l'utilisateur

Le lancement automatique du bash se fait dans la mesure où l'administrateur a défini comme tâche à exécuter au moment de la connexion de l'utilisateur la commande `/bin/bash`. Ceci étant configuré dans le fichier `/etc/passwd`.

Le bash de connexion possède comme premier caractère de son paramètre zéro ($0) un "-" (tiret).

Est aussi considéré comme bash de connexion, le shell qui a été invoqué :

Avec l'option --login ou -l.

Par la commande `su` avec l'option "-" ou "-l" ou "--login"

```
login as: user1
user1@192.168.18.129's password: xxxxxxxx
Last login: Wed Oct  3 23:31:40 2007 from 192.168.18.1

$ echo $0
-bash
```

Lorsque le bash est dit de connexion, il exécute :

Si elles existent

Si elles sont en lecture

Sans création d'un nouvel environnement (exécution par .)

Et dans cet ordre là

Les procédures suivantes :

. `/etc/profile`

. `/etc/profile.d/*.sh`

. `~/.bash_profile` ou

. `~/.bash_login` ou

. `~/.profile` la première rencontrée

. `~/.bashrc` lancée par la précédente

. `/etc/bashrc` lancée par la précédente

Qui vont mettre en place les différents éléments (variables, alias…) de l'environnement utilisateur.

Ces procédures sont exécutées au moment de la connexion d'un utilisateur que ce soit à partir d'un terminal texte ou graphique.

Exemple :

Après modification de ces procédures, afin d'afficher un message, cela donne :

```
login as: user1
user1@192.168.18.129's password: xxxxxxx
Last login: Thu Oct  4 13:48:16 2007 from 192.168.18.1
Exécution de /etc/profile
Exécution des procédures de /etc/profile.d/*.sh
Exécution de .bash_profile
Exécution de .bashrc
c'est un Shell de login
Exécution de /etc/bashrc

$ more .bashrc
# .bashrc
echo "Exécution de .bashrc"
shopt -q login_shell && echo "c'est un Shell de login" || echo
"Ce Shell n'est pas de Login"
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
```

7.2.2 En dehors d'un contexte de connexion

➢ Le bash interactif

Lors de son lancement, un processus bash interactif exécute automatiquement, dans son propre environnement (pas de création de fils), les procédures :

`. ~/.bashrc`	
`. /etc/bashrc`	lancée par la précédente
`. /etc/profile.d/*.sh`	lancées par la précédente, car nous sommes hors contexte de connexion

➢ Le bash non interactif

Au lancement d'une procédure, sous l'une des formes indiquées ci-dessous, si la variable BASH_ENV existe, elle est développée et le shell script associé est auto exécuté.

```
bash procédure                ou
procédure                     ou
./procédure             ou
exec procédure
```

Exemple :

```
$ export BASH_ENV=~/.bashrc

$ cat .bashrc
#!/bin/bash
echo -e "Exécution du processus : $0 par $USER\n"

$ cat proc
#!/bin/bash
echo "Exécution de la procédure : $0"
echo -e "************Procedure de test************"

$ bash proc
Exécution du processus : bash par user1
Exécution de la procédure : proc
************Procedure de test************

$ proc
Exécution du processus : bash par user1
Exécution de la procédure : ./proc
************Procedure de test************

$ exec proc
Exécution du processus : bash par user

Exécution de la procédure : /home/user1/proc
************Procedure de test************
```

7.3 Symboles d'accueil d'une session shell

7.3.1 Variables concernées

PS1

Symbole d'accueil principal d'un shell interactif.

PS1 est affiché dès que le shell est prêt à lire une commande.

La valeur de PS1 est développée avant d'être utilisée.

PS2

Symbole d'accueil secondaire.

PS2 est affichée quand le shell a besoin de données supplémentaires pour exécuter une commande.

Valeur par défaut : "> ".

PS3

Symbole utilisé par la commande select.

PS3 est affichée quand select est en attente d'une réponse utilisateur.

Valeur par défaut : "#? ".

PS4

Symbole affiché sur la ligne de mise au point lors d'un suivi d'exécution.

Valeur par défaut : "+ ".

La valeur des variables PS1, PS2 ou PS4 (mais pas celle de PS3) est développée avant d'être utilisée. Cela signifie que ces variables peuvent contenir les séquences d'échappement décrites à la page suivante qui seront interprétées avant affichage.

7.3.2 Séquences d'échappement possibles

\a	Caractère d'alarme ASCII 07
\d	"Nom du jour" "Mois" "Numéro du jour dans le mois"
\e	Caractère d'échappement ASCII 033
\h	Nom d'hôte de la machine jusqu'au point (.)
\H	Nom d'hôte complet de la machine
\n	Saut de ligne
\r	Retour chariot
\s	Nom du shell
\t	Heure sur 24 heures au format HH:MM:SS
\T	Heure sur 12 heures au format HH:MM:SS
\@	Heure sur 12 heures au format HH:MM am/pm
\u	Nom de l'utilisateur
\v	Version du bash (exemple : 2.04)
\V	(exemple : 2.04.0)
\w	Répertoire de travail
\W	Nom de base du répertoire de travail
\!	Numéro d'historique de la commande
\#	Numéro de la commande depuis le début de la session
\$	# si l'UID effectif est 0 (zéro), $ sinon
\[	Début d'une série de caractères non imprimables permettant d'inclure des séquences de contrôle de terminal
\]	Fin de série de caractères non imprimables

Après que la chaîne d'invite contenue dans les variables PS1, PS2 ou PS4 ait été décodée, elle est soumise à l'expansion des paramètres, la substitution de commandes, l'évaluation arithmétique et le découpage en mots.

En bash, dans un environnement graphique, pour être correctement interprétées, la modification de ces variables doit être placée dans `~/.bashrc`.

7.4 Changement de session utilisateur

su nom_utilisateur

Permet de changer d'identité (uid, gid) tout en conservant l'environnement de la session suspendue. Retour à la session suspendue par exit ou <Ctrl> <d>

su - nom_utilisateur

Permet de changer d'identité et d'environnement grâce à l'exécution, comme lors d'une connexion de premier niveau, des shell scripts de l'utilisateur dont on veut prendre l'identité. Retour à la session suspendue par exit ou <Ctrl> <d>

su [-] nom_utilisateur –c "commande(s) [-options] [arguments]"

Exécute la (ou les) commande(s) citée(s) avec l'identité, les droits et éventuellement l'environnement de l'utilisateur désigné. Attention, la (ou les) commande(s) doivent être protégées de la session en cours. Il y a retour automatique à la session suspendue.

Conseil

Lors d'une session su vérifier son environnement et son identité :

```
$ id
uid=501(user1) gid=500(g1) groupes=500(g1)
$ pwd
/home/user1
```

Comparaison `su` et `su -`

```
$ su user2
$ pwd
/home/user1
$ whoami
user2
$ who am i
user1  pts/1 Apr 2 15:41 (:0.0)
$ echo $LOGNAME
user1
$ logname
user1
$ id
uid=502(user2) gid=500(g1)
groupes=500(g1)
$ exit
```

```
$ su - user2
$ pwd
/home/user2
$ whoami
user2
$ who am i
user1  pts/1 Apr 2 15:41 (:0.0)
$ echo $LOGNAME
user2
$ logname
user1
$ id
uid=502(user2) gid=500(g1)
groupes=500(g1)
$ exit
```

7.5 Traitement des alias

7.5.1 La commande alias

Affiche les alias actuellement définis.

```
alias
```

Affiche la définition des alias cités.

```
alias nom1 nom2 … nomn
```

Définit des alias.

```
alias nom1=ch1 ...nomn=chn
```

Un espace final dans la chaîne de définition forcera la recherche d'un alias pour le mot suivant sur la ligne de commande :

```
$ alias c='chmod' u='u+x'            # Pas d'espace après chmod
                                     # c u …   ne fonctionne pas

$ >f1

$ c u f1
chmod: chaîne de mode invalide: `u'

$ unalias c

$ alias c='chmod '                   # Espace après chmod
                                     # c u … fonctionne

$ c u f1

$ ls -l f1
-rwxrw-r--  1 user1 g1   0  5 févr. 10:48  f1
```

7.6 La commande unalias

Suppression de l'alias de la liste des alias définis.

```
unalias nom
```

Suppression de tous les alias définis.

```
unalias -a
```

L'option expand_aliases

En bash uniquement, les alias ne sont développés que si l'option expand_aliases est activée. Cette option est automatiquement présente dans les shells interactifs.

```
$ alias heure='date "+Il est : %Hh%M"'

$ heure
Il est : 14h46

$ cat f1
alias heure='date "+Il est : %Hh%M"'
heure

$ f1
./f1: line 2: heure: command not found

$ vi f1

$ cat f1
shopt -s expand_aliases
alias heure='date "+Il est : %Hh%M"'
heure

$ f1
Il est : 14h47
```

7.7 Fonctions

7.7.1 Définition d'une fonction

Ensemble de commandes regroupé sous un nom quelconque, exécutable dans le shell courant, par référence à ce nom.

Les fonctions, comme les procédures, peuvent prendre des paramètres.

7.7.2 Déclaration d'une fonction

```
function nom
{
liste de commandes
}
```

ou

```
Nom ()
{
liste de commandes
}
```

Les fonctions ne sont pas transmissibles par appel de ksh ou bash .

Pour en disposer dans toutes les sessions utilisateur il faut impérativement les déclarer dans le fichier référencé par la variable `ENV ($HOME/.kshrc)` ou `BASH_ENV (~/.bashrc)`. Fichier qui est exécuté à chaque appel de shell.

7.7.3 Commandes propres à une fonction

`typeset` Dans une fonction, permet de définir des variables locales.

`return [n]` Permet de sortir d'une fonction sans quitter le shell.
Code retour par défaut : celui de la dernière commande exécutée, sinon code retour vaut n.

7.7.4 Commandes de gestion des fonctions

`declare -f` (bash uniquement)

`functions`

`typeset -f` Liste toutes les fonctions de l'environnement.

`export nom`

`typeset -fx nom` Exporte la fonction nom, elle est connue des shell scripts.

`unset -f nom` Annulation d'une fonction.

7.8 Droits prédéfinis sur les fichiers

7.8.1 La commande `umask`

La commande `umask` permet de prédéfinir des droits sur les fichiers.

```
umask nnn
```

`umask` positionne dans l'environnement de l'utilisateur un masque octal utilisé pour définir des droits par défaut lors de la création de fichiers.

Le masque octal est utilisé en complément binaire à 2.

Il en est tenu compte dans sa totalité lors de la création de fichiers répertoires.

Lors de la création de fichiers ordinaires, les droits de lecture et d'écriture sont fonction du masque, les droits d'exécution sont systématiquement enlevés (seules les chaînes de production de programmes créent directement des fichiers exécutables).

```
$ umask 022

$ mkdir rep          # droits sur rep :  drwxr-xr-x
                     # Il est tenu compte du masque dans sa
                     # totalité.

$ >fic               # droits sur fic :       -rw-r--r--
                     # Les droits d'exécution
                     # sont systématiquement enlevés.
```

7.8.2 La commande chmod

Permet à l'utilisateur de modifier les droits prédéfinis sur un ou plusieurs fichier(s).

```
chmod droits_en_octal fichier(s)

chmod droits_symboliques fichier(s)
```

Seul le propriétaire des fichiers ou root ont le droit de changer les droits.

7.9 Contrôle du déroulement d'une session

7.9.1 Commandes utiles

who	Identité de connexion des utilisateurs actuellement présents sur la machine
whoami	Identité actuelle de l'utilisateur de la session en cours
who am i	Identité de connexion sur la ligne, au premier niveau
who -u	Affiche les évènements de connexion : heure, numéro de processus shell
who -T	Affiche les autorisations d'écriture sur les terminaux
ps [-options] **-e** **-f** **-t term** **-p pid** **-u util**	Liste les processus associés au terminal Tous les processus actifs Processus associés au terminal avec leur dépendance Processus associés au terminal sans leur dépendance Processus associés au processus cité Processus associés à l'utilisateur cité
uname [-options] **-s** **-v** **-r** **-m** **-a**	Sans option, affiche le nom du système d'exploitation Nom du système d'exploitation (option par défaut) Version du système d'exploitation Etat de la version Numéro d'identification Affiche toutes ces options réunies

7.9.2 Positionnement des options du shell : la commande `set`

On peut positionner plusieurs options, soit à l'appel du shell soit par la commande set.

```
set [-aefhmnsuvx] [-o option ...] [arg1 ...]
```

`-a	-o allexport`	Toutes les variables sont exportées.
`-e	-o errexit`	Si erreur sur une commande et si pseudo-signal 0 positionné, exécution des commandes de trap et exit.
`-f	-o noglob`	Pas de génération de noms de fichiers par * ? []
`-h	-o trackall`	En ksk : Toute commande devient un alias pisté. Automatique dans les shell script. En bash : mémorise l'emplacement des commandes lors de leur exécution. Activée par défaut.
`-m	-o monitor`	Affiche un message à la fin du job.
`-n	-o noexec`	Analyse syntaxique sans exécution.
`-s`	Trie les paramètres positionnels (ksh uniquement)
`-u	-o nounset`	Si les paramètres substitués n'ont pas de valeur, affichage d'une erreur.
`-v	-o verbose`	Affichage de la ligne lue.
**`-x	-o xtrace`**	Affichage après substitutions de la ligne lue.
`-`	Suppression de -x et -v.
`--`	Les arguments de set peuvent commencer par un -

Les arguments de `set` garnissent les paramètres positionnels.

En l'absence d'argument, `set --` détruit les paramètres positionnels.

`-o bgnice`	ksh uniquement. Travaux en arrière plan lancés avec une faible priorité. Automatiquement positionnée en ksh.

`-o emacs`
`-o gmacs`
`-o vi`
	Éditeurs pour l'historique des commandes.

`-o ignoreeof`	Impossibilité de quitter ksh par <Ctrl> <d>, exit obligatoire.
`-o markdirs`	ksh uniquement. Les noms de répertoire obtenus par * ? [] se terminent par /.
`-o noclobber`	Impossibilité d'écraser un fichier par redirection. Utiliser >/fichier.

En utilisant + au lieu de -, on inverse la fonctionnalité.

Autres utilisations de set

`set -o`	Affiche le positionnement des options.
`set`	Liste le nom des variables et leur contenu.

`[ -o noclobber ]`	Répond vrai si l'option est positionnée faux dans le cas contraire

Unix – Linux

8. Expressions régulières GREP

Objectifs

- Découvrir les mécanismes d'expressions régulières
- Utiliser la commande *grep*

8.1 Les expressions régulières

8.1.1 Définition

Une expression régulière (ou rationnelle) est un motif qui permet de construire un ensemble de chaînes de caractères.

8.1.2 Expressions régulières simples

`$`	Chaîne vide en fin de ligne.
`^`	Chaîne vide en début de ligne.
`.`	Un caractère quelconque.
`*`	Un nombre quelconque de fois le caractère qui précède.
`.*`	N'importe quelle chaîne de caractères y compris la chaîne vide.
`[liste]`	Un des caractères de la liste. Possibilité de définir des intervalles : [0-9], [a-z], [A-Z], [a-zA-Z] ou [A-Za-z]
`[^liste]`	Un caractère non inclus dans la liste.

8.1.3 Quelques exemples d'expressions régulières

Expressions régulières	Interprétation
`a.`	aa ou ab ou ac …
`a.c`	aac ou abc ou acc …
`a\.c`	a.c
`[abcd] ou [a-d]`	a, b, c ou d
`[a-zA-Z]`	une lettre minuscule ou majuscule
`[1-37]`	1, 2, 3 ou 7
`[^abcd] ou [^a-d]`	un caractère différent de a, b, c ou d
`^3`	ligne commençant par le chiffre 3
`A$`	ligne se terminant par le caractère A
`xy*`	x ou xy ou xyy ou xyyy…
`^[0-9].*[0-9]$`	ligne commençant et se terminant par un chiffre
`:g/ *$/s///`	Dans vi, supprime tous les espaces en fin de ligne dans tout le tampon de manœuvre.

8.1.4 Expressions régulières étendues

`{n}`	Le caractère qui précède doit être répété n fois exactement.
`{n,}`	Le caractère qui précède doit être répété n fois ou plus.
`{,m}`	Le caractère qui précède doit être répété de 0 à m fois au plus.
`{n,m}`	Le caractère qui précède doit être répété au moins n fois et au plus m fois.
`\b`	Chaîne vide à l'extrémité d'un mot. A placer à l'extrémité qui doit être vide.
`\B`	Chaîne vide ne se trouvant pas à l'extrémité d'un mot.
`e1|e2`	Toute chaîne de caractères engendrée soit par e1, soit par e2.
`?`	Zéro ou une fois le caractère qui précède.
`+`	Une à n fois le caractère qui précède.

➢ Classes de caractères prédéfinies

[:alnum:]	Ensemble des caractères alphanumériques Equivalent à [0-9A-Za-z]
[:alpha:]	Ensemble des caractères alphabétiques
[:digit:]	Ensemble des chiffres en base 10
[:xdigit:]	Ensemble des chiffres en base 16
[:lower:]	Ensemble des lettres minuscules
[:upper:]	Ensemble des lettres majuscules
[:print:]	Ensemble des caractères affichables
[:space:]	Ensemble des caractères d'espacement
[:punct:]	Ensemble des caractères de ponctuation
\w	Identique à `[[:alnum:]]`
\W	Identique à `[^[:alnum:]]`
\<	Chaîne vide en début de mot. A placer au début de l'expression
\>	Chaîne vide en fin de mot. A placer à la fin de l'expression
e1|e2	Toute chaîne de caractères de la forme `ch1ch2` où `ch1` est engendrée par `e1` et `ch2` par `e2`

8.2 La commande grep

8.2.1 Syntaxe de la commande

```
{[f]|[e]}grep [-options] {[-e] motif | -f fichier} [fichiers]
```

Les commandes `fgrep, grep` et `egrep` recherchent dans des fichiers d'entrée des lignes contenant un motif donné.

Si trouvées, ces lignes sont, par défaut, affichées.

Si aucun fichier n'est cité ou si le nom « - » est mentionné, la lecture est faite sur l'entrée standard.

Par défaut, si plusieurs fichiers d'entrée sont cités, lors de l'affichage, la commande fera précéder la ligne, du nom du fichier dans lequel elle a été trouvée.

8.2.2 Les différentes commandes

fgrep	Interprètent le motif comme une chaîne de caractères.
grep -F	
grep	Interprètent le motif comme une expression régulière simple.
grep -G	
egrep	Interprètent le motif comme une expression régulière étendue.
grep -E	

Sous Linux, selon les versions, grep est en mesure d'utiliser des expressions rationnelles simples ou étendues. La version GNU ne fait, en principe, pas de différences entre les fonctionnalités disponibles. Cependant, des tests de contrôle peuvent s'avérer nécessaires. Dans tous les cas, avant d'adresser un rapport de bogue, il faut connaître la version de grep utilisée, elle est donnée par l'option -V.

$ **grep -V**
GNU grep 2.6.3

Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

8.2.3 Les options

-c	Affiche, pour chaque fichier d'entrée, le nombre de lignes correspondant au motif
-e	Indique que ce qui suit est le motif. Permet de commencer un motif par « - »
-f	Le mot suivant est un fichier qui contient le motif recherché Utile pour construire des motifs compliqués Permet également le « - » en début de motif
-h	Invalide l'affichage du nom de fichier en cas de recherche dans plusieurs fichiers
-i	Ignore minuscules et majuscules tant dans le motif que dans les fichiers d'entrée
-L	Affiche le nom des fichiers dont aucune ligne ne contient le motif
-l	Affiche le noms des fichiers dont au moins une ligne contient le motif
-n	Fait précéder chaque ligne trouvée de son numéro de ligne dans le fichier
-q	Supprime l'affichage des résultats(*)
-s	Supprime les messages d'erreur concernant les fichiers inexistants ou illisibles(*)
-v	Affiche les lignes ne contenant pas le motif
-w	Le motif formera une correspondance s'il décrit un mot complet
-x	Affiche uniquement les lignes pour lesquelles le motif décrit toute la ligne

(*) Utile dans une structure `if` ou pour déterminer l'entrée dans un corps de boucle `while` ou `until`.

```
$ grep '^user[1-7]:' /etc/passwd
user1:x:501:501::/home/user1:/bin/bash
user2:x:502:502::/home/user2:/bin/bash
user3:x:503:503::/home/user3:/bin/bash
user4:x:504:504::/home/user4:/bin/bash
user5:x:505:505::/home/user5:/bin/bash
user6:x:506:506::/home/user6:/bin/bash
user7:x:507:507::/home/user7:/bin/bash


$ grep '^[AG].*[6-8]$' edition
Aboaf Maurice 244748
Allo Jean-Pierre 255398
Gross Pascal 245367
Grosbois Anne 122456


$ grep -s '[^fe]grep ' /etc/*
...
```

8.3 Exemples

```
$ more modele1
Aboaf Maurice 244 748
Adda Jen 539 234
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Blanfin Adèle  159 268
Chasserat Paul 245 178
Cousin Pascal 222 222
Dupont Jean 111 112
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456
```

COMMANDES	COMMENTAIRES
`grep '6[78]$' modele1`	Lignes se terminant par 67 ou 68 Le fichier est supposé ne pas avoir d'espace à la fin des lignes.
`grep '6[78] *$' modele1`	Lignes se terminant par 67 ou 68. Les espaces de fin de lignes sont pris en compte.
`grep '^[Aa].*r' modele1`	Lignes commençant par « A » ou « a » et contenant la lettre « r ».
`grep '[Aa].*r' modele1`	Lignes contenant une chaîne qui commence par la lettre « a » en minuscule ou en majuscule et qui se termine la lettre « r ».
`grep '[w-z]' modele1`	Lignes contenant les lettres « w », ou « x », ou « y », ou « z » en minuscule.
`grep -i '[w-z]' modele1`	Lignes contenant les lettres « w », ou « x », ou « y », ou « z » en minuscule ou en majuscule.
`grep '^[^A-D]' modele1`	Lignes dont le premier caractère n'est ni « A », ni « B », ni « C », ni « D ».
`egrep \` `'Allo|Dupont +Jean|Paul' modele1`	Lignes contenant « Allo » et lignes contenant « Dupont Jean » (avec au moins un espace entre Dupont et Jean) et lignes contenant « Paul ».

`egrep \`
`'(Allo|Dupont) +(Jean|Paul)' \`
`modele1`
	Toutes les lignes qui contiennent :
	« Allo Jean »
	« Allo Paul »
	« Dupont Jean »
	« Dupont Paul »
	Au moins un espace entre nom et prénom

```
$ more ensemble

Première ligne : espace, tabulation, passage à la ligne
123098
456
62ab
Ab12
56cf9
45D3
28G
la vie est belle
LA VIE EST BELLE
```

COMMANDES	COMMENTAIRES

`grep '[[:alnum:]]' ensemble`
`grep '[a-zA-Z0-9]' ensemble`
`grep '\w' ensemble`
	Toutes les lignes qui contiennent un caractère alphanumérique.

`grep '[^[:alnum:]]' ensemble` `grep '\W' ensemble`	Toutes les lignes qui contiennent un caractère non alphanumérique (espaces ou ponctuation par exemple).
`egrep '^\b[[:digit:]]*\b$' ensemble` `grep -E '^\b[[:digit:]]*\b$' ensemble`	Lignes contenant un nombre en base 10 et rien d'autre.
`egrep -w '\b[[:digit:]]*\b' ensemble` `grep -Ew '\b[[:digit:]]*\b' ensemble`	Lignes contenant un nombre en base 10.
`egrep '^\b[[:xdigit:]]*\b$' ensemble` `grep -E '^\b[[:xdigit:]]*\b$' ensemble`	Lignes contenant un nombre en base 16 et rien d'autre.
`egrep -w '\b[[:xdigit:]]*\b' ensemble` `grep -Ew '\b[[:xdigit:]]*\b' ensemble`	Lignes contenant un nombre en base 16.
`grep '[[:lower:]]' ensemble`	Lignes contenant une lettre minuscule.
`grep '^[[:lower:] ]*$' ensemble`	Lignes ne contenant que des lettres en minuscule et des espaces.
`grep '^[[:upper:] ]*$' ensemble`	Lignes ne contenant que des lettres en majuscule et des espaces.
`grep '^[[:space:]]*$' ensemble`	Lignes ne contenant que des caractères d'espacement (espace, tabulation, passage à la ligne).
`grep '^[[:print:]]*$' ensemble`	Lignes ne contenant que des caractères affichables (espace et passage à la ligne compris).

```
$ more ponctuation
Point .
Virgule ,
Point-virgule ;
Point d'interrogation ?
Point d'exclamation !
Points de suspenion ...
Deux points :
Parenthèses ()
Accolades {}
Guillemets "
Apostrophe '
Pas de caractère de ponctuation
```

```
$ more repetitions
Don Juan
Donc
Donjon
Donner
Dormir
tante
G
GG
GGG
GGGG
GGGGG
```

COMMANDES	COMMENTAIRES
grep '[[:punct:]]' ponctuation	Lignes contenant un caractère de ponctuation.
egrep 'Don?' repetitions	Lignes contenant une chaîne commençant par « Do » suivi de zéro ou une fois le caractère « n ».
egrep 'Don?[^n]' repetitions	Lignes contenant une chaîne commençant par « Do » suivi de zéro ou une fois le caractère « n », suivi d'un caractère qui n'est pas « n » (« Donner » est éliminé).
egrep 'Don+' repetitions	Lignes contenant une chaîne commençant par « Do » suivi d'une à n fois le caractère « n ».
egrep '\bG{2}\b' repetitions **egrep –w 'G{2}' repetitions**	Lignes où la chaîne « GG » forme un mot.
egrep 'G{2}' repetitions	Lignes où le caractère « G » est répété deux fois consécutivement.
egrep 'G{0,2}' repetitions	Lignes contenant une chaîne où le caractère « G » est répété consécutivement 0 fois minimum, 2 fois maximum
egrep 'G{2,}' repetitions	Lignes contenant une chaîne où le caractère « G » est répété consécutivement deux fois ou plus
egrep 'G{3,4}' repetitions	Lignes contenant une chaîne où le caractère « G » est répété consécutivement 3 fois au moins et 4 fois au plus.

8.4 Exécution des exemples

L'exercice consiste à recopier les fichiers modèles des pages suivantes, et à lancer les différentes commandes étudiées pour s'approprier les syntaxes et les effets.

Ces pages reproduisent les résultats des commandes étudiées ci-dessus. Il revient à chacun d'expérimenter les différentes syntaxes.

Exemples de résultats

```
$ grep '6[78]$' modele1
Bernard Jean-Paul 235 567

$ grep '6[78] *$' modele1
Bernard Jean-Paul 235 567
Blanfin Adèle 159 268

$ grep '^[Aa].*r' modele1
Aboaf Maurice 244 748
Allo Jean-Pierre 255 98

$ grep '[Aa].*r' modele1
Aboaf Maurice 244 748
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Chasserat Paul 245 178
Zingaro Bartabas 123 456

$ grep '[w-z]' modele1
Froideceaux Michel 252 423

$ grep -i '[w-z]' modele1
Froideceaux Michel 252 423
Zingaro Bartabas 123 456

$ grep '^[^A-D]' modele1
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456

$ egrep 'Allo|Dupont +Jean|Paul' modele1
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Chasserat Paul 245 178
Dupont Jean 111 112
```

```
$ egrep '(Allo|Dupont) +(Jean|Paul)' modele1
Allo Jean-Pierre 255 98
Dupont Jean 111 112

$ grep '[[:alnum:]]' ensemble
Première ligne : espace, tabulation, passage à la ligne
123098
456
62ab
Ab12
56cf9
45D3
28G
la vie est belle
LA VIE EST BELLE

$ grep '[a-zA-Z0-9]' ensemble       # Idem ligne précédente
Première ligne : espace, tabulation, passage à la ligne
123098
456
62ab
Ab12
56cf9
45D3
28G
la vie est belle
LA VIE EST BELLE

$ grep '\w' ensemble                # Idem ligne précédente
Première ligne : espace, tabulation, passage à la ligne
123098
456
62ab
Ab12
56cf9
45D3
28G
la vie est belle
LA VIE EST BELLE

$ grep '[^[:alnum:]]' ensemble

Première ligne : espace, tabulation, passage à la ligne
la vie est belle
LA VIE EST BELLE
```

```
$ grep '\W' ensemble                # Idem ligne précédente

Première ligne : espace, tabulation, passage à la ligne
la vie est belle
LA VIE EST BELLE

$ grep -E '^\b[[:digit:]]*\b$' ensemble
123098
456

$ egrep '^\b[[:digit:]]*\b$' ensemble     # Idem ligne précédente
123098
456

$ egrep -w '\b[[:digit:]]*\b' ensemble
123098
456

$ grep -Ew '\b[[:digit:]]*\b' ensemble      # Idem ligne précédente
123098
456

$ grep -E '^\b[[:xdigit:]]*\b$' ensemble
123098
456
62ab
Ab12
56cf9
45D3

$ egrep '^\b[[:xdigit:]]*\b$' ensemble      # Idem ligne précédente
123098
456
62ab
Ab12
56cf9
45D3

$ grep -Ew '\b[[:xdigit:]]*\b' ensemble
123098
456
62ab
Ab12
56cf9
45D3
```

```
$ egrep -w '\b[[:xdigit:]]*\b' ensemble  # Idem ligne précédente
123098
456
62ab
Ab12
56cf9
45D3

$ grep '[[:lower:]]' ensemble
Première ligne : espace, tabulation, passage à la ligne
62ab
Ab12
56cf9
la vie est belle

$ grep '^[[:lower:] ]*$' ensemble
la vie est belle

$ grep '^[[:upper:] ]*$' ensemble
LA VIE EST BELLE

$ grep '^[[:space:]]*$' ensemble
ligne composée d'espace et de tabulation


$ grep '^[[:print:]]*$' ensemble
Première ligne : espace, tabulation, passage à la ligne
123098
456
62ab
Ab12
56cf9
45D3
28G
la vie est belle
LA VIE EST BELLE

$ grep '[[:punct:]]' ponctuation
Point .
Virgule ,
Point-virgule ;
Point d'interrogation ?
Point d'exclamation !
Points de suspenion ...
Deux points :
Parenthèses ()
Accolades {}
Guillemets "
Apostrophe '
```

```
$ egrep 'Don?' repetitions
Don Juan
Donc
Donjon
Donner
Dormir

$ egrep 'Don?[^n]' repetitions
Don Juan
Donc
Donjon
Dormir

$ egrep 'Don+' repetitions
Don Juan
Donc
Donjon
Donner

$ egrep '\bG{2}\b' repetitions
GG

$ egrep -w 'G{2}' repetitions
GG

$ egrep 'G{2}' repetitions
GG
GGG
GGGG
GGGGG

$ egrep 'G{0,2}' repetitions
Don Juan
Donc
Donjon
Donner
Dormir
tante
G
GG
GGG
GGGG
GGGGG
```

```
$ egrep 'G{2,}' repetitions
GG
GGG
GGGG
GGGGG

$ egrep 'G{3,4}' repetitions
GGG
GGGG
GGGGG
```

Unix – Linux

9. Éditeur de texte sed

Objectifs

- Comprendre les principes de fonctionnement de *sed*
- Utiliser la commande *sed*

9.1 Généralités

- `sed` est un éditeur de texte non interactif.
- `sed` utilise les expressions régulières développées dans le chapitre précédent.
- `sed` traite les données en entrée en une seule passe ce qui rend cette commande très utile dans les shell-scripts.
- `sed` est un filtre ; elle lit les données en entrée sur le stdin et affiche les résultats sur le stdout.

9.1.1 Appel de sed

```
sed [-options] ['programme' | -f prog | --file prog] [fichier(s)]
```

9.1.2 Les options de sed

`-n`
`--quiet`
`--silent`
	Supprime automatiquement l'affichage écran sauf si celui-ci est concerné par la fonction p.

`-e cde` `--expression=cde`	Permet d'ajouter une ou plusieurs commandes au programme sed désigné par l'option –f ou –file. Selon qu'elle est placée avant ou après l'option –f ou –file cde sera ajoutée au début ou à la fin du programme.
`-f programme` `--file programme`	Le programme de sed se trouve dans le fichier cité après l'option et non pas en ligne avec la commande.
`-i[suffixe]` `--in-place[=suffixe]`	Il n'y a pas d'affichage sur la sortie standard et le fichier traité par sed est affecté par les modifications. Si un suffixe est donné, le fichier d'origine est sauvegardé, et a comme nom celui d'origine suivi du suffixe.
`-r` `--regexp-extended`	Autorise l'utilisation des expressions régulières étendues.

9.2 Algorithme de la commande `sed`

9.2.1 Principe de fonctionnement

La commande `sed` lit le programme (en ligne ou dans un fichier programme).

Chaque ligne de données est envoyée dans le tampon de travail de `sed`. Il lui sera appliqué l'ensemble des instructions du programme.

Après lecture et éventuellement exécution de la dernière instruction du programme, la ligne de données, modifiée ou non, est envoyée en sortie standard par défaut.

9.2.2 Syntaxe d'une commande de mise à jour

```
[adr1[,adr2]]fonction[arguments]
```

9.3 Adressage des lignes

9.3.1 L'adressage se fait

- Par numéro de ligne,
- Par la désignation d'une chaîne de caractères contenue dans la ligne,
- Par l'utilisation d'expressions régulières.

L'adressage relatif est interdit.

9.3.2 Syntaxes d'adressage

Zone adresse	
`10`	Le traitement qui suit portera sur la ligne 10.
`/c700541/`	Le traitement qui suit portera sur toutes les lignes comportant la chaîne `c700541`
`/c700541 /`	Le traitement qui suit portera sur toutes les lignes comportant la chaîne `c700541` suivie d'un espace.
`/expression régulière/`	Le traitement qui suit sera exécuté pour toutes les lignes qui contiennent une chaîne de caractères répondant à l'expression régulière.
`adresse1,adresse2`	Le traitement qui suit est appliqué depuis la première ligne répondant vrai à `adresse1` jusqu'à la première ligne répondant vrai à `adresse2`.

9.4 Les fonctions de mise à jour

9.4.1 Fonctions portant sur la totalité de la ligne

Ces fonctions doivent logiquement être précédées d'un adressage.

Fonction	
d	Supprime les lignes adressées
a **<texte>**	Fait suivre la ligne adressée de la (des) lignes décrites dans <texte>
i **<texte>**	Fait précéder la ligne adressée de la (des) lignes décrites dans <texte>
c **<texte>**	Substitue la ou les lignes adressées par <texte>

NB : Si <texte> est multilignes, tous les passages à la ligne sauf le dernier doivent être protégés par un « \ ».

9.4.2 Exemples de fonctions portant sur la totalité de la ligne

Texte initial, notez que la numérotation des lignes fait partie intégrante du texte.

```
$ cat texte
1    Le système LINUX comprend le noyau, le shell les programmes
2    utilitaires et les langages
3    - le noyau planifie les tâches ; gère la mémoire, la
4    structure et les accès aux fichiers.
5    - le shell est un langage de commandes. Il interprète
6    les commandes et les exécute.
7    - les programmes utilitaires, encore appelés outils,
8    représentent le logiciel utilisateur de base
9    N'est-ce pas tout un programme ?...
```

```
Programme sed.
$ cat prog_lignes
1i\
Bonjour\
les\
amis
2c\
Bonsoir
4a\
Au revoir
6d
/outils/c\
La ligne 6 a été supprimée\
La ligne 7 qui contenait le mot "outils"\
a été substituée par cette ligne et les deux précédentes.\
Les lignes 8 et 9 ne seront pas touchées.
```

Lancement de la commande `sed` et résultat de l'affichage résultant.

```
$ sed -f prog_lignes texte
Bonjour
les
amis
1   Le système LINUX comprend le noyau, le shell les programmes
Bonsoir
3   - le noyau planifie les tâches ; gère la mémoire, la
4   structure et les accès aux fichiers.
Au revoir
5   - le shell est un langage de commandes. Il interprète
La ligne 6 a été supprimée
La ligne 7 qui contenait le mot "outils"
a été substituée par cette ligne et les deux précédentes.
Les lignes 8 et 9 ne seront pas touchées.
8   représentent le logiciel utilisateur de base.
9   N'est-ce pas tout un programme ?...
```

9.4.3 Remplacements dans les chaînes de caractères : la fonction y

[adr1,[adr2]]y<liste1><liste2>

y Remplace toutes les occurrences des caractères de <liste1> par les caractères correspondants de <liste2>.

<liste1> et <liste2> doivent avoir le même nombre de caractères.

Exemple

```
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:
/home/c700541/bin:.

$ echo $PATH |sed 'y;/:; \n;'
 usr kerberos bin
 usr local bin
 usr bin
 bin
 usr X11R6 bin
 home c700541 bin
 .
```

9.4.4 Substitutions, modifications de chaînes de caractères

[adr1[,adr2]]s<expression><remplacement>[fonctions]

La fonction de substitution *s* exige un séparateur qui encadre `<expression>` et `<remplacement>`.

Si `adr1` et/ou `adr2` sont présentes et composées de chaînes de caractères, elles sont encadrées du même séparateur que `<expression>` et `<remplacement>`.

Est considéré comme séparateur, le caractère se trouvant immédiatement après *s* .

Le caractère séparateur ne peut être ni "`espace`", ni "`passage à la ligne`", ni un caractère se trouvant dans `<expression>` ou `<remplacement>` ou `adr1` ou `adr2`.

<expression>	Une chaîne de caractères ou une expression régulière.
<remplacement>	Une chaîne de caractères qui n'utilise pas la syntaxe des expressions régulières. On peut, en revanche, trouver dans cette zone les conventions \n et & & Chaîne qui coïncide avec la description de <expression>. \n Nième expression entre parenthèses.
fonctions	Les fonctions suivantes sont reconnues : g Traitement de toutes les occurrences sur la ligne. n Seule la nième occurrence de la ligne est prise en compte. p Affichage du résultat de la substitution.. w fic Ecriture du résultat de la substitution dans le fichier désigné.

9.4.5 Substitution en utilisant les expressions entre parenthèses

Les expressions entre parenthèses décrivent, des chaînes de caractères que l'on pourra appeler des champs positionnels. Ces champs sont désignés par `\<numéro de champ>`.

```
s/.....\(exp1\).....\(expn\)/\n.....\1/
```

`exp1` représente la première expression entre parenthèses et `expn` la nième.

`\1` et `\n` représentent respectivement les expressions entre parenthèses numéro 1 et n

Exemples

```
$ sed 's/\(.*\) \(.*\) .* .*/\2 \1/' modele1
Maurice Aboaf
Jen Adda
Jean-Pierre Allo
Jean-Paul Bernard
Adèle Blanfin
Paul Chasserat
Pascal Cousin
Jean Dupont
Michel Froideceaux
Lucien Gros
Bartabas Zingaro


$ echo $TERM
xterm

$ echo $TERM | sed 's/^\(.\).*/\1/'
x
```

9.4.6 Substitution utilisant la convention &

```
$ sed 's/[.,;!?:]/*P&P*/g' texte

1       Le système LINUX comprend le noyau*P,P* le shell les programmes
2       utilitaires et les langages
3       - le noyau planifie les tâches *P;P* gère la mémoire*P,P* la
4       structure et les accès aux fichiers*P.P*
5       - le shell est un langage de commandes*P.P* Il interprète
6       les commandes et les exécute*P.P*
7       - les programmes utilitaires*P,P* encore appelés outils*P,P*
8       représentent le logiciel utilisateur de base*P.P*
9       N'est-ce pas tout un programme *P?P**P.P**P.P**P.P*
```

9.4.7 Substitution utilisant la fonction p avec et sans l'option –n

-n ou –quiet ou --silent (en bash)

```
$ cat modele1
Aboaf Maurice 244 748
Adda Jen 539 234
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Blanfin Adèle 159 268
Chasserat Paul 245 178
Cousin Pascal 222 222
Dupont Jean 111 112
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456

$ sed '/Dupont/s//Durand/p' modele1
Aboaf Maurice 244 748
Adda Jen 539 234
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Blanfin Adèle 159 268
Chasserat Paul 245 178
Cousin Pascal 222 222
Durand Jean 111 112
Durand Jean 111 112
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456

$ sed -n '/Dupont/s//Durand/p' modele1
Durand Jean 111 112
```

9.4.8 Utilisation de l'option –e

```
$ cat modele1
Aboaf Maurice 244 748
Adda Jen 539 234
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Blanfin Adèle 159 268
Chasserat Paul 245 178
Cousin Pascal 222 222
Dupont Jean 111 112           <-- 8ème ligne
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456

$ cat prog_e
/Dupont/s//Durand/p


$ sed -n -f prog_e modele1
Durand Jean 111 112
```

Ajout de commande avant le programme

```
$ sed -n -e '8d' -f prog_e modele1
```

rien ne s'affiche !

Ajout de commande après le programme

```
$ sed -n -f prog_e -e '8d' modele1
Durand Jean 111 112
```

9.4.9 Utilisation de l'option –i

- **Sauvegarde du fichier d'origine avec un suffixe cité conjointement à l'option**

```
$ cat annuaire
Aboaf Maurice 244 748
Blanfin Adèle 159 268
Dupont Jean 111 112

$ sed -i .save1 's/1/T/g' annuaire

$ ls -l annuaire*
-rw-rw-r--  1 c700541 avance 67 déc 15 05:32 annuaire
-rw-rw-r--  1 c700541 avance 67 déc 15 05:32 annuaire.save1

$ cat annuaire
Aboaf Maurice 244 748
Blanfin Adèle T59 268
Dupont Jean TTT TT2

$ cat annuaire.save1
Aboaf Maurice 244 748
Blanfin Adèle 159 268
Dupont Jean 111 112
```

- **Écrasement du fichier d'origine**

```
$ ls -l annuaire*
-rw-rw-r--  1 c700541 avance 67 déc 15 05:33 annuaire

$ sed -i 's/1/T/g' annuaire

$ ls -l annuaire*
-rw-rw-r--  1 c700541 avance 67 déc 15 05:33 annuaire

$ cat annuaire
Aboaf Maurice 244 748
Blanfin Adèle T59 268
Dupont Jean TTT TT2
```

Unix - Linux

10. Présentation de awk

Objectifs

- Comprendre les principes de fonctionnement de *awk*
- Utiliser la commande *awk*

10.1 Généralités

`awk`, sous Unix, `gawk`, sous Linux, sont un langage d'examen et de traitement de chaînes de caractères.

`gawk` est l'implémentation dans le projet GNU du langage de programmation `awk` sous Unix.

Les options de `awk` sont des options POSIX composées d'une seule lettre (précédée d'un « - »).

Celles de `gawk` peuvent être soit POSIX, soit des options longues du type GNU (précédées de deux « -- »).

`awk` est un lien sur `gawk` dans certaines distributions

Dans les exemples qui suivent nous emploierons indifféremment l'une ou l'autre des deux commandes.

10.1.1 Appel de `awk`

```
awk [-options] ['prog_en_ligne' | -f prog] [fichier(s)]
```

10.1.2 Les options de `awk`

`-F sc`	Redéfinit le séparateur de champs, qui devient sc Séparateur par défaut : espace, tabulation, passage à la ligne.
`-v var=valeur`	Affecte une variable avant l'exécution du programme et la rend accessible dès le pavé BEGIN (voir structure d'un programme).
`-f programme`	Le programme de *awk* se trouve dans le fichier cité après l'option et non pas en ligne avec la commande.

10.1.3 Structure d'un programme `awk`

```
'BEGIN {instructions initiales}
sélection
{actions}
sélection {actions}
sélection1,sélection2
sélection1,sélection2 {actions}
END   {instructions finales}'
```

- Si le programme est en ligne avec la commande, il est protégé.
- S'il existe des instructions initiales, elles sont dans la paire d'accolades qui suit le mot clé `BEGIN`.
- S'il existe des instructions finales, elles sont dans la paire d'accolades qui suit le mot clé `END`.
- Le corps du programme est composé des sélections et/ou d'actions. Les actions sont entre accolades, les sélections ne sont pas entre accolades.

10.1.4 Algorithme de la commande `awk`

- La commande `awk` lit le programme (en ligne ou dans un fichier programme).

- S'il existe un pavé `BEGIN`, les instructions initiales sont exécutées avant le traitement de la première donnée (on appelle données l'ensemble des lignes du ou des fichiers que `awk` va devoir traiter).

- Chaque ligne de données est envoyée dans le tampon de travail de `awk` nommé $0. Il lui sera appliqué l'ensemble des instructions du corps du programme (cf. tableau page suivante).

- Après le traitement de la dernière donnée, s'il existe un pavé `END`, les instructions finales sont exécutées.

REMARQUE : `awk` est un filtre, par défaut, les lignes à traiter sont lues sur l'entrée standard, le résultat du traitement est écrit sur la sortie standard.

LIGNE DU PROGRAMME `awk`	EXECUTION REALISEE
`sélection`	L'action par défaut (affichage de la ligne courante sur le stdout) est exécutée.
`{action}`	L'action est systématiquement exécutée.
`sélection {action}`	L'action est exécutée si la sélection répond vrai.
`sélection1, sélection2`	L'action par défaut est exécutée depuis la première ligne répondant vrai à sélection1 jusqu'à la dernière ligne répondant vrai à sélection2
`sélection1, sélection2 {action}`	L'action sera exécutée depuis la première ligne répondant vrai à sélection1 jusqu'à la première ligne répondant vrai à sélection2

10.2 Les variables awk

Les variables d'un programme awk sont dynamiques. Elles sont définies lors de la première utilisation en fonction du contexte.

Dans une expression arithmétique, une variable est définie comme numérique, initialisée à 0 par défaut.

$0, le tampon d'entrée est une variable. L'enregistrement contenu dans $0 est lui-même découpé en champs. Chaque champ est une variable.

awk gère les variables de la ligne d'entrée, des variables utilisées pour la sortie, des variables prédéfinies, des variables définies dans le programme.

10.2.1 Structure des données en entrée

L'enregistrement en entrée est lu par awk dans $0 depuis le fichier traité.

awk lit jusqu'à la rencontre du caractère de fin d'enregistrement (passage à la ligne par défaut) contenu dans la variable standard RS (Record Separator).

awk découpe la ligne en champs en fonction du contenu de la variable FS (Field Separator) FS vaut espace par défaut.

Les champs de $0 se nomment $1, $2, $3, ….

10.2.2 Structure des données en sortie

Les données en sortie sont présentées suivant une structure similaire, les variables `OFS` (Output Field Separator) et `ORS` (Output Record Separator) jouant un rôle semblable à celui des variables d'entrée.

Les exemples de ce chapitre utilisent le fichier `modèle` suivant :

```
$ more modele
Aboaf Maurice 244 748
Adda Jen 539 234
Allo Jean-Pierre 255 98
Bernard Jean-Paul 235 567
Blanfin Adèle  159 268
Chasserat Paul 245 178
Cousin Pascal 222 222
Dupont Jean 111 112
Froideceaux Michel 252 423
Gros Lucien 212 121
Zingaro Bartabas 123 456
$ awk '{print $2 , $1}' modele
Maurice Aboaf
Jen Adda
Jean-Pierre Allo
Jean-Paul Bernard
Adèle Blanfin
Paul Chasserat
Pascal Cousin
Jean Dupont
Michel Froideceaux
Lucien Gros
Bartabas Zingaro
```

Dans l'action `print` la virgule impose l'affichage de `OFS` (ici un espace entre le prénom et le nom).

10.2.3 Les variables prédéfinies

ENVIRON	Tableau indicé par les noms des variables d'environnement de awk
FILENAME	Nom du fichier courant
FNR	Numéro d'enregistrement dans fichier courant
NF	Nombre de champs
NR	Numéro courant d'enregistrement
FS	Séparateur de champs en entrée
OFS	Séparateur de champs en sortie
OFMT	Format de sortie numérique (%.6g par défaut)
RS	Séparateur d'enregistrements en entrée
ORS	Séparateur d'enregistrements en sortie

```
$ gawk '{print NR , NF , $0}' modele1
1 4 Aboaf Maurice 244 748
2 4 Adda Jen 539 234
3 4 Allo Jean-Pierre 255 98
4 4 Bernard Jean-Paul 235 567
5 4 Blanfin Adèle 159 268
6 4 Chasserat Paul 245 178
7 4 Cousin Pascal 222 222
8 4 Dupont Jean 111 112
9 4 Froideceaux Michel 252 423
10 4 Gros Lucien 212 121
11 4 Zingaro Bartabas 123 456

$ echo | gawk '
{print "Utilisateur : " ENVIRON["LOGNAME"] "\tRépertoire : "
ENVIRON["HOME"]}'

Utilisateur : c700541     Répertoire : /home/c700541
```

10.3 Les sélections awk

Les sélections sont formées à partir d'expressions relationnelles simples ou composées. Elles mettent en œuvre des opérateurs relationnels et des :

- Variables,
- Expressions régulières (décrites avec la commande grep),
- Expressions arithmétiques (décrites ci-dessous).

10.3.1 Les opérateurs arithmétiques

```
+        -        *        /        %        ^

++       --   +=       -=       *=       /=       ^=
```

10.3.2 Les opérateurs relationnels

➢ Comparaison de deux valeurs

```
<        <=       >        >=       ==       !=
```

➢ Rapprochement d'une valeur d'une expression régulière

`~`	(tilde) contient / peut être généré par
`!~`	ne contient pas / ne peut pas être généré par

Exemples

Opérateurs relationnels : comparaison entre deux valeurs.

```
$ awk '
  $0==61549*2 {print "Ligne no : "NR, "Contenu : " $0}' Ch_Le
Ligne no : 3 Contenu : 123098
```

Opérateurs relationnels : rapprochement d'une valeur et d'une expression régulière.

```
$ gawk '$0~/^[[:digit:]]*$/' Ch_Le
123098
456

$ ls -l / |gawk '$8!~/:/ {print $NF, "a plus de 6 mois"}'
178 a plus de 6 mois
initrd a plus de 6 mois
misc a plus de 6 mois
mnt a plus de 6 mois
opt a plus de 6 mois
srv a plus de 6 mois

$ cat lignes_paires
cd /home/stages_shell/avance
echo -e "\t\tAvec la commande awk"
awk 'NR % 2 == 0
      END {print "\n" ENVIRON["PWD"] "/" FILENAME, NR,
"lignes"}' modele1
echo -e "\n\t\tRésultats identiques avec la commande gawk"
gawk 'NR % 2 == 0
      END {print "\n" ENVIRON["PWD"] "/" FILENAME, NR,
"lignes"}' modele1

$ lignes_paires
          Avec la commande awk
Adda Jen 539 234
Bernard Jean-Paul 235 567
Chasserat Paul 245 178
Dupont Jean 111 112
Gros Lucien 212 121

/home/stages_shell/avance/modele1 11 lignes
```

10.3.3 Les sélections multicritères

Les opérateurs…

&&	ET
//	OU (double pipe)
!	NON (point d'exclamation)

…permettent de combiner des expressions entre elles, avec possibilité de les grouper entre parenthèses.

```
$ awk 'BEGIN {FS=":"} $3<200 && $6=="/" {print $1}' /etc/passwd
nobody
dbus
haldaemon
nscd
rpc
```

```
$ awk -F : '$3<500 && $6=="/" {print $1}' /etc/passwd
nobody
dbus
haldaemon
nscd
rpc
```

10.4 Les actions awk

Les actions sont réalisées à partir de primitives d'actions ou d'instructions, exécutées en cas de sélection(s) vraie(s) ou systématiquement si pas de sélection.

Les actions sont placées entre accolades.

Il peut y avoir plusieurs actions dans une même paire d'accolades. Elles sont séparées par un « ; » si elles sont sur la même ligne, sinon le passage à la ligne réalise la séparation.

10.4.1 L'instruction d'affichage print

```
print [liste]
```

`print` affiche la liste sur le stdout. Par défaut, `print` affiche $0.

La sortie de `print` peut être redirigée dans un fichier ou comme stdin d'une commande (dans ce dernier cas, `print` est placée devant un |).

```
$ gawk '{print $1 >"Noms"}' modele1

$ cat Noms
Aboaf
Adda
Allo
Bernard
Blanfin
Chasserat
Cousin
Dupont
Froideceaux
Gros
Zingaro
```

10.4.2 L'instruction d'affichage formaté printf

```
printf "format1format2texteformat3...",var1,var2,var3...
```

`printf` réalise des affichages formatés sur le stdout.

Le premier argument de `printf` est une chaîne de caractères entre "..." qui représente un masque d'affichage.

La sortie de `printf` peut être redirigée dans un fichier ou comme stdin d'une commande (dans ce dernier cas, `printf` est placée devant un |).

format	Défini par %[cadrage][longueur]caractère (Voir ci-dessous la signification du caractère)
texte	Texte affiché tel quel
variable	Affichée en fonction du format correspondant Format et variable se correspondent positionnellement

10.4.3 Format des caractères

Caractère	Format de l'expression
c	Caractères ASCII
d ou i	Nombre décimal entier
e	[-]***d.ddddd***E[+-]***dd***
f	[-]***ddd.ddddd***
g	Comme e ou f , celui qui est le plus court, les zéros non significatifs sont supprimés
o	Nombre octal non signé
s	Chaîne de caractères
x	Nombre héxadécimal non signé
%	Caractères %

Ci-dessous, quelques exemples d'expressions de printf et de la sortie générée.

{printf "format" , val}	résultat
printf "%c" , 99	c (ascii : 99 décimal)
printf "%d", 99/2	49
printf "%e", 99/2	4.950000e+01
printf "%f", 99/2	49.500000
printf "%6.2f", 99/2	49.50
printf "%g", 99/2	49.5
printf "%o", 99	143
printf "%06o", 99	000143
printf "%x", 99	63
printf "|%s|", "January"	|January|
printf "|%10s|", "January"	| January|
printf "|%-10s|", "January"	|January |
printf "|%.3s|", "January"	|Jan|
printf "|%10.3s|", "January"	| Jan|
printf "|%-10.3s|", "January"	|Jan |
printf "%%"	%

Le format de sortie standard pour les nombres est `%.6g` cela peut être changé dans `OFMT`.

```
$ more prog_printf
BEGIN {print; system("date");print}
{printf "Nom = %-15s\tNuméro =%4s%4s\n",$1,$3,$4}

$ awk -f prog_printf modele1
sam déc  8 05:17:07 CET 2007

Nom = Aboaf             Numéro = 244 748
Nom = Adda              Numéro = 539 234
Nom = Allo              Numéro = 255  98
Nom = Bernard           Numéro = 235 567
Nom = Blanfin           Numéro = 159 268
Nom = Chasserat         Numéro = 245 178
Nom = Cousin            Numéro = 222 222
Nom = Dupont            Numéro = 111 112
Nom = Froideceaux       Numéro = 252 423
Nom = Gros              Numéro = 212 121
Nom = Zingaro           Numéro = 123 456
```

Analyse du programme

Le pavé `BEGIN` :

Le tampon $0 est vide, `print` affiche donc une ligne blanche.
La fonction `system` est appelée pour demander l'exécution de la commande date.

Le corps du programme :

Composé de la commande `printf` qui affiche les variables $1, $3, $4 en appliquant les règles suivantes :

`%-15s`	Se rapporte à l'affichage de $1 L'emplacement réservé pour l'affichage est de 15 colonnes cadrées à gauche (-)
`%4s`	Se rapporte à l'affichage de $3 4 caractères, cadrés à droite par défaut, sont réservés
`%4s`	Se rapporte à l'affichage de $4 4 caractères, cadrés à droite par défaut, sont réservés

Formation

Notre expertise est votre avenir

Découvrez également l'ensemble des stages à votre disposition sur notre site

http://www.m2iformation.fr

N°Azur 0 810 007 689

PRIX D'UN APPEL LOCAL DEPUIS UN POSTE FIXE