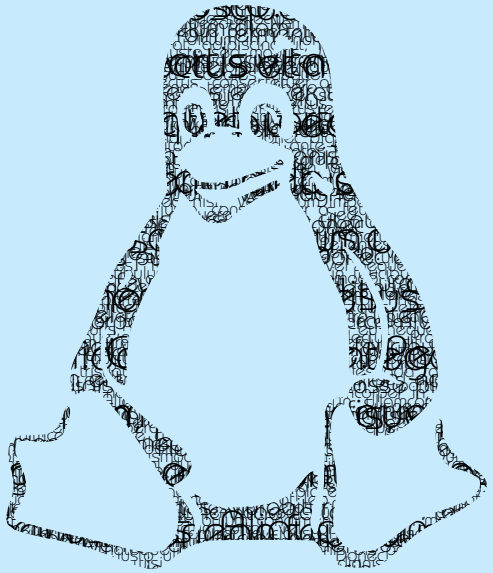
Université Mohammed V
Faculté des Sciences
Rabat

Systèmes d'Exploitation 1

Filière SMI—S3

Par : Pr. Hicham LAANAYA

hicham.laanaya@gmail.com



Année universitaire 2017—2018

Table des matières

1

Systèmes d'exploitation

1 Introduction

Nous présentons dans cette première partie une présentation générale sur les systèmes d'exploitation. La section suivante donne un rappel sur le matériel. La deuxième section de cette partie donne une introduction sur la notion de système d'exploitation. La troisième section donne les principaux systèmes d'exploitations.

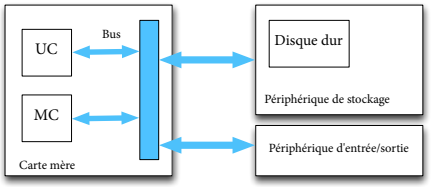
2 Rappels sur le matériel

2.1 Architecture simplifiée d'un ordinateur

Un ordinateur est composé (cf. figure 1.1) :

- d'une unité pour effectuer les traitements, également appelée unité centrale (UC) ou processeur,
- d'une unité pour contenir les programmes à exécuter appelée mémoire centrale (MC),
- des périphériques de stockage permanent pour y enregistrer les travaux effectués en mémoire centrale tel que le disque dur,
- des dispositifs pour entrer et récupérer des données appelés périphériques d'entrée-sortie : un écran, une souris, un clavier, un lecteur de disquettes et un lecteur de CD-ROM ou DVD-ROM

Figure 1.1: Architecture simplifiée d'un ordinateur



2.2 Carte mère

La carte mère est une plaque de résine contenant à l'intérieur et sur les deux faces une fine couche de cuivre sur laquelle est imprimé le circuit imprimé, On y trouve les éléments suivants :

- La mémoire vive RAM (***Random Access Memory***) : La mémoire vive RAM (Random Access Memory) présente le lieu de travail dans un ordinateur à savoir qu'un programme stocké sur le disque dur est chargé en mémoire centrale où ses instructions seront accédées une à une pour être exécutées par le processeur. La RAM est une mémoire volatile c'est-à-dire que son contenu serait perdu en cas de coupure d'électricité
- La mémoire morte ROM (***Read Only memory***) : Elle contient les programmes du BIOS qui gèrent le chargement du système et les entrées-sorties. On distingue plusieurs puces ROM tel que la PROM (***Programmable ROM***) et EPROM (***Erasable Programmable ROM***)
- L'horloge qui permet de cadencer le fonctionnement du processeur, du bus. Sa

fréquence caractérise la carte mère. Elle est généralement très inférieure à celle du processeur.

- Un ensemble de bus : un bus est un ensemble de fils de cuivre incrustés dans la carte mère qui permettent de véhiculer l'information. Le bus se caractérise par le nombre de fils qui le composent. Si le nombre de fils est de 64, on parle alors de bus 64 bits. Il est également caractérisé par sa fréquence de fonctionnement.

- Le "chipset" ou "jeu de composants" soudé sur la carte mère. Le chipset régit tous les échanges au sein du PC en aiguillant les données sur les différents bus de la carte mère.

- Le microprocesseur

L'unité centrale est un circuit intégré qui réalise les traitements et les décisions, elle se compose :

- d'une unité de commande et de contrôle UCC : elle recherche les instructions, les décode et en supervise leur exécution par l'UAL.

- d'une unité arithmétique et logique UAL : elle réalise les traitements qu'ils soient arithmétiques ou logiques.

- de registres : ils sont des zones mémoires internes au processeur destinées à accueillir les données, les instructions et les résultats.

- d'une horloge qui rythme le processeur : à chaque ***top*** d'horloge le processeur effectue une instruction, ainsi plus l'horloge a une fréquence élevée, plus le processeur effectue d'instructions par seconde (MIPS : Millions d'instruction par seconde). Par exemple un ordinateur ayant une fréquence de 1 GHz (1000 MHz) effectue 1000 millions d'instructions par seconde.

- d'un bus interne qui relie ces unités aux registres.

De nos jours d'autres composants sont intégrés au processeur tels que :

- Une unité flottante pour le calcul des opérations sur les nombres réels.

- La mémoire cache : c'est une mémoire de petite taille, à accès plus rapide que la mémoire principale. Elle permet au processeur de se "rappeler" les opérations déjà effectuées auparavant. Ce type de mémoire résidait sur la carte mère, sur les ordinateurs récents ce type de mémoire est directement intégré dans le processeur.

- Les unités de gestion mémoire servent à convertir des adresses logiques en des adresses réelles situées en mémoire.

3 Notions de systèmes d'exploitation

3.1 Introdution

Le système d'exploitation est un gestionnaire de ressources contrôlant l'accès à toutes les ressources de la machine. Il permet l'attribution de ces ressources aux différents utilisateurs, et aussi à la libération de ces ressources lorsqu'elles ne sont plus utilisées. Tous les périphériques comme la mémoire, le disque dur ou les imprimantes sont des ressources, le processeur également est une ressource.

3.2 Les principaux systèmes d'exploitation

MS-DOS est le plus connu des premiers systèmes d'exploitation pour PC. Il est mono-utilisateur et mono-tâche. On a du greffer des couches logicielles pour répondre aux évolutions matérielles et aux demandes des utilisateurs. MS-DOS a été rapidement supplanté par les systèmes Windows.

Mac OS : C'est le système d'exploitation d'Apple. Il a été livré pour le Macintosh en 1984. La version actuelle de ce système est macOS (10.12). Mac OS X se distingue par un noyau Darwin (Unix) qui est un open source. Mac OS est un des principaux rivaux des Windows.

Unix étant distribué gratuitement, il a donné naissance à de nombreuses versions :

- Les versions les plus connues Unix SYSTEM V (évolution de la version initiale d'AT&T et Bell) et Unix BSD

- Les principaux Unix du marché sur Intel sont : Open Server et Unixware de SCO (Santa Cruz Operation), Solaris (Sun Microsystems), BSD (Berkeley), …

- Trois Unix dominent le monde des serveurs : HP/UX, Sun Solaris, IBM AIX

Linux a pris des parts de marché aux Unix, à Novell Netware et à Windows NT-2000 serveur. Il s'est imposé dès la fin du 20ème siècle. Linux est multi-utilisateurs, multi-tâches, stable et gratuit. Parmi ses principales distributions on trouve : RedHat, Debian, Caldera, Ubuntu, …

La famille des Windows :

- Microsoft propose en 1992 Windows 3.10 et Windows pour Workgroups 3.11 qui sont Multi-fenêtres et Multi-tâches coopératif. En 1993, on voit apparaître la première version de Windows NT 3.1 suivie par NT 3.5 en 1994

- L'année 1995, verra la sortie du fort célèbre Windows 95

- En 1996, Windows NT 4 avec deux versions station de travail et Serveur.
- Windows Terminal Server : un système qui simule un environnement multi-utilisateurs et prend en charge la connexion de plusieurs terminaux
- En 1998 Windows 98 et en 2000, Microsoft commercialise Windows 2000 professionnel et serveur, Windows Millenium, suivi de Windows XP familial et serveur
- Windows 2003 (initialement baptisé .NET) sort en 2003 suivi de Windows VISTA, Windows Seven, Windows 8 et Windows 10.

1. Systèmes d'exploitation Notions de systèmes d'exploitation 3.2 Les principaux systèmes d'exploitation

2

Système Unix

1 Introduction au système Unix

1.1 Système Unix

Unix est un système d'exploitation (*Operating System*) :

- Multi-utilisateurs : le système identifie des personnes logiques et permet à ces personnes d'utiliser le système dans certaines limites
- Multi-tâches : le système est étudié pour exécuter plusieurs programmes en même temps, grâce au concept de "temps partagé"
- Multi-plateforme : Unix n'est pas un système dédié à un processeur, mais que c'est une famille de systèmes que l'on retrouve sur une multitude de plates-formes.

On trouve des Unix :

- propriétaires :
- et des Unix libres :
  - Linux sur plate-forme Intel, Sparc, Alpha, Mac, ...
  - FreeBSD sur plate-forme Intel, Alpha, PC-98
  - OpenBSD également multi-plate-forme

Table 2.1: Distributions Unix

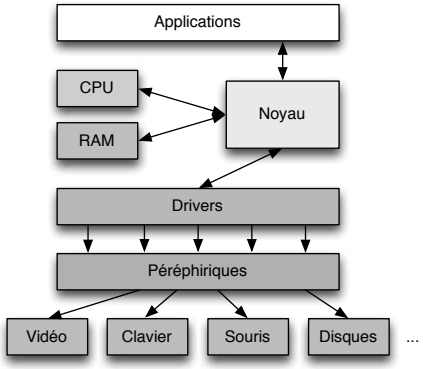
Nom	Propriétaire	Processeur
Solaris	Sun	Sparc & Intel
HPUX	HP	PA
AIX	IBM	Risc & PowerPC
Digital Unix	Digital	Alpha

1.2 Architecture et caractéristiques

On peut décomposer un système Unix en trois grandes entités :

- Le noyau : il assure la gestion de la mémoire et des entrées sorties de bas niveau et l'enchaînement des tâches
- Un ensemble d'utilitaires : dédiés à des tâches diverses :
  - des interpréteurs de commande appelés *Shells* permettant de soumettre des tâches au système, tâches pouvant être concurrentes et/ou communicantes
  - des commandes de manipulation de fichiers (copie, déplacement, effacement, etc.)
- Une base de données système : un ensemble de fichiers contenant :
  - des informations sur la configuration des différents services
  - des scripts de changement d'état du système (démarrage, arrêt, ...)

Figure 2.1: Architecture Unix



1.3 Logiciels propriétaires et logiciels libres

Les logiciels sont vendus et sont régis par une licence restrictive qui interdit aux utilisateurs de copier, distribuer, modifier ou vendre le programme en question.

Les logiciels libres sont les logiciels que l'ont peut librement utiliser, échanger, étudier et redistribuer. Cela implique que l'on ait accès à leur code source (d'où le terme équivalent *OpenSource*)

i — la liberté d'exécution : tout le monde a le droit de lancer le programme, quel qu'en soit le but

ii — la liberté de modification : tout le monde a le droit d'étudier le programme et de le modifier, ce qui implique un accès au code source

iii — la liberté de redistribution : tout le monde a le droit de rediffuser le programme, gratuitement ou non

iv — la liberté d'amélioration : tout le monde a le droit de redistribuer une version modifiée du programme

2 Commandes de base du *Shell*

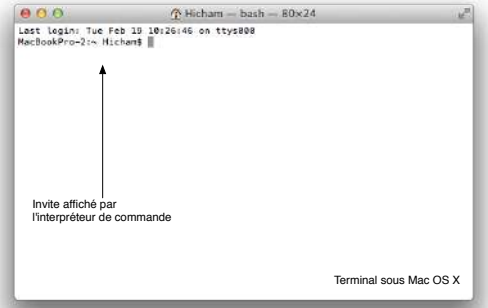
2.1 Introduction

Un *Shell* est un interpréteur de commande en mode texte et peut s'utiliser en mode interactif ou pour exécuter des programmes écrits dans le langage de programmation du *Shell*

(appelés scripts *Shell*)

En mode interactif, le *Shell* affiche une invite en début de ligne (*prompt*) (*cf.* figure 2.2). La commande est interprétée et exécutée après la frappe de la touche "**Entrée**"

Figure 2.2: Interpréteur de commandes sous Mac OS X



2.2 Format des commandes

Le format des commandes suit une convention bien établie :

- `commande [-options] [paramètres]`
- Les `options` et les `paramètres` sont parfois facultatifs.

Exemple : `cp -i /home/profs/prof1/Hello.c /home/etudiants/etudiant1`

- `cp` : commande qui va lancer la fonction de copie
- l'option `-i` : permet de contrôler certains aspects du comportement de la commande
- `/home/profs/prof1/Hello.c` : Il s'agit de la source ou le fichier que vous souhaitez copier
- `/home/etudiants/etudiant1` : Il s'agit de la destination ou l'emplacement de la copie

2.3 Méta-caractères du *Shell*

Ils sont interprétés spécialement par le *Shell* avant de lancer la commande entrée par l'utilisateur ; et permettent de spécifier des ensembles de fichiers, sans avoir à rentrer tous leurs noms en voici les plus utilisés :

- `*` : remplacé par n'importe quelle suite de caractères
- `?` : remplacé par un seul caractère quelconque
- `[ ]` : remplacé par l'un des caractères mentionnés entre les crochets. On peut spécifier un intervalle avec `-` : `[a-z]` spécifie donc l'ensemble des lettres minuscules

3 Système de gestion de fichiers

3.1 Concept de base

Le système de fichiers d'Unix est une vaste arborescence dont les nœuds sont des répertoires et les feuilles des fichiers. Un fichier peut :

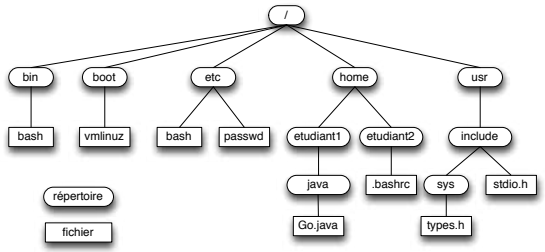
i — contenir des données

ii — être un lien sur un autre fichier

iii — être un moyen d'accès à un périphérique (mémoire, écran, disque dur, ...)

iv — être un canal de communication entre processus

Figure 2.3: Hiérarchie du système de fichiers



3.2 Les différents types de fichiers

On distingue :

- Les fichiers ordinaires (réguliers) sont une suite d'octets sans structure
- Les répertoires contiennent des informations sur les fichiers et les sous-répertoires
- Les liens symboliques sont une catégorie particulière de fichiers (qui contiennent l'emplacement du fichier à prendre en compte)
- Les périphériques sont vus comme des fichiers spéciaux du répertoire `/dev`
- Les tubes nommés sont des fichiers sur disque gérés comme un tube (***pipe***) entre deux processus échangeant des données

3.3 Les i-nœuds

À chaque fichier correspond un i-nœud contenant :

- le type du fichier et les droits d'accès des différents utilisateurs
- l'identification du propriétaire du fichier
- la taille du fichier exprimée en nombre de caractères (pas de sens pour les fichiers spéciaux)
- le nombre de liens physiques sur le fichier
- la date de dernière modification/consultation (écriture/lecture) du fichier
- la date de dernière modification du nœud (modification d'attributs)
- l'identification de la ressource associée (pour les fichiers spéciaux)

3.4 Le nom des fichiers

Le nom d'un fichier doit permettre de l'identifier dans un ensemble de fichiers :

- Le nom est composé de caractères (*cf.* tableau 2.2 pour les caractères acceptables)
- Le nom est souvent composé de deux parties :
  - *i* — la base ; et
  - *ii* — l'extension qui se trouve après le caractère '.'
- L'extension d'un nom de fichier désigne la nature de son contenu (texte, image, son, ...)
- UNIX est un système qui distingue les caractères majuscules et minuscules

Remarque 1

Ne pas utiliser le caractère espace comme nom de fichier ou répertoire !

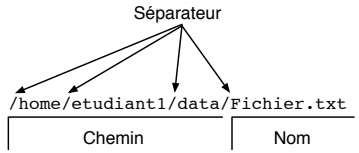
Table 2.2: Caractères acceptables pour les noms de fichiers

Caractères	*Signification*
A—Z	Lettres majuscules
a—z	Lettres minuscules
0—9	Chiffres
_	Caractère souligné
,	Caractère virgule
.	Caractère point

3.5 Les chemins d'accès

Pour identifier un fichier dans l'arborescence on indique le nom complet du fichier. Ce nom est représenté par :

- Le chemin composé de répertoires qui conduit de la racine de l'arborescence du système de fichiers jusqu'au répertoire qui contient le fichier
- Chaque répertoire est distingué des autres par un symbole séparateur "/"
- le nom du fichier



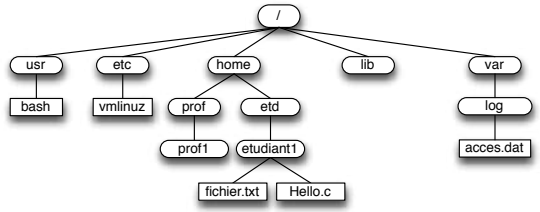
On distingue deux expressions d'un chemin :

- Le chemin d'accès absolu (chemin absolu) : commence par le symbole séparateur, il exprime le chemin complet à partir de la racine de l'arborescence
- Le chemin d'accès relatif (chemin relatif) : commence par un autre caractère que le caractère séparateur.

Exemple : Le répertoire courant est : `/var/log`

- Le chemin absolu pour désigner le fichier `fichier.txt` est `/home/etd/etudiant1/fichier.txt`
- Le chemin relatif est : `../../home/etd/etudiant1/fichier.txt`

Figure 2.4: Exemple de chemins



- "`..`" désigne le répertoire parent, "`.`" désigne le répertoire courant

Exercice 1

Dans la hiérarchie présentée dans la figure 2.4, exprimez les chemins suivants :

1. *absolu pour* ***prof1***
2. *absolu pour* ***etc***
3. *absolu pour* ***prof***
4. *relatif à* ***log*** *pour* ***acces.dat***
5. *relatif à* ***prof*** *pour* ***acces.dat***
6. *relatif à* ***etudiant1*** *pour* ***acces.dat***

Solution 1

Dans la hiérarchie précédente, exprimez les chemins suivants :

1. absolu pour **prof1** : `/home/prof/prof1`
2. absolu pour **etc** : `/etc`
3. absolu pour **prof** : `/home/prof`
4. relatif à **log** pour **acces.dat** : `acces.dat`
5. relatif à **prof** pour **acces.dat** : `../../var/log/acces.dat`
6. relatif à **etudiant1** pour **acces.dat** : `../../../var/log/acces.dat`

3.6 Commandes de base de manipulation de fichiers

Les commandes de base d'accès aux fichiers sont :

- `cat` : affiche le contenu du fichier
- `stat` : affiche les caractéristiques du fichier
- `ls` : affiche les caractéristiques d'une liste de fichiers (l'option `-i` affiche les numéros d'i-nœuds des fichiers)
- `rm` : supprime un fichier
- `touch` : modifie les caractéristiques de date d'un fichier (permet également de créer un fichier vide)

Les commandes d'accès aux répertoires sont :

- `ls` : affiche la liste des fichiers contenus dans un répertoire et utilise les options :
  - `-a` liste aussi les fichiers cachés (fichiers où le nom commence par .)
  - `-l` donne des informations détaillées sur chaque fichier
  - `-i` donne le numéro de l'i-nœud du fichier
- `mkdir` : crée un répertoire

- `cd` : change le répertoire de travail (répertoire courant)
- `pwd` : donne le chemin absolu du répertoire courant
- `rmdir` : supprime un répertoire vide

Les commandes utilisées pour la manipulation du système de fichiers sont :

- `cp` : copie de fichier
  - syntaxe `cp <source> <destination>`
  - duplication du contenu du fichier et création d'une entrée dans un répertoire
- `mv` : déplace/renomme un fichier
  - syntaxe `mv <source> <destination>`
  - suppression d'une entrée dans un répertoire et création d'une nouvelle entrée dans un répertoire

Remarque 2

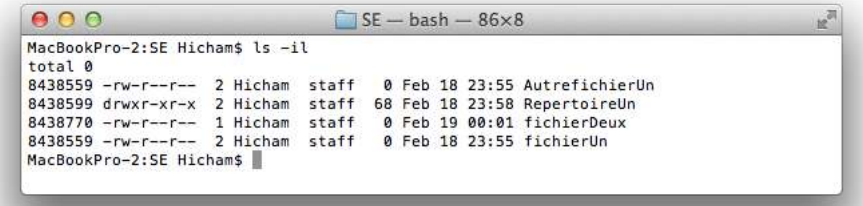
Si on copie (déplace/renomme) un fichier dans un fichier qui existe déjà, ce second fichier est modifié (contenu écrasé et caractéristiques modifiées)

3.7 Notion de liens

Il existe deux types de liens :

1. Lien physique : Un même fichier peut avoir donc plusieurs noms (Il y a plusieurs liens physiques sur le fichier).
2. Lien symbolique : c'est un fichier (de type lien) qui contient le chemin et le nom d'un autre fichier
   - Les accès à un lien sont donc des redirections vers un autre fichier : les commandes qui manipulent des liens manipulent en fait le fichier dont le nom est stocké dans le lien
   - Un lien se comporte comme un raccourci (*alias*) vers un autre fichier
   - Le contenu d'un lien est soit un chemin absolu ou un chemin relatif (qui doit être valide depuis le répertoire dans lequel se trouve le lien !)

Figure 2.5: Liens physiques



Les liens physiques sont plusieurs entrées de répertoires du même i-nœud (ce sont donc des fichiers réguliers). Par contre, les liens symboliques ont chacun leur propre i-nœud ; leur contenu désigne un même fichier régulier (ils sont du type liens)

La commande `ln` permet de créer des liens :

```
ln [options] <destination> <nom_du_lien>
```

- sans option : création de liens physiques
- avec l'option `-s` : création de liens symboliques

Figure 2.6: Exemple de lien physique et de lien symbolique

3.8 Notions d'utilisateur et de groupe

Pour pouvoir accéder au système de fichier, l'utilisateur doit être connecté. Pour se connecter, l'utilisateur doit saisir son `login` et le mot de passe associé. A chaque login le système fait correspondre un numéro d'identification `uid` (*User IDentifier*). Chaque utilisateur appartient à au moins un groupe d'utilisateurs et à chaque groupe d'utilisateur le système fait correspondre un numéro d'identification `gid` (*Group IDentifier*). Ces informations sont stockées dans des fichiers d'administration

Le fichier `/etc/passwd` contient les informations relatives aux utilisateurs (login, mot de passe crypté, `uid, gid`, nom complet, répertoire de travail au login, commande exécutée au login) ; et `/etc/group` contient les informations relatives aux groupes (nom, mot de passe, `gid`, liste des membres du groupe)

Le super-utilisateur (root) : Il est toujours considéré par le système comme propriétaire de tous les fichiers (et des processus). La personne qui gère le système est normalement la seule à connaître son mot de passe. Lui seul peut ajouter de nouveaux utilisateurs au système.

Table 2.3: Notions de sécurité

Concept de base	
sujet	Utilisateur ou Processus qui veut exécuter une opération ;
objet	Fichier sur lequel on veut exécuter une opération ; et enfin
opération	Action que l'on veut exécuter.

- Des règles de sécurité ont pour rôle d'indiquer les opérations (droits) qui seront autorisées pour un **sujet** sur un **objet**
- Le système a pour rôle de vérifier que le **sujet** a le droit d'exécuter l'**opération** sur l'**objet**.

3.9 Sécurité sous Unix

Le système Unix associe des droits à chaque fichier (règles). Ces droits sont fonctions du sujet. Un fichier appartient à un utilisateur et à un groupe. Unix distingue le sujet comme étant :

1. le propriétaire de l'objet (fichier)
2. membre du groupe propriétaire de l'objet (fichier)
3. les autres

Il faut faire la différence entre les opérations pour les fichiers réguliers et celles pour les répertoires. Pour les fichiers réguliers, nous avons les opérations :

- `r` : droit de lecture
- `w` : droit de modification et de suppression

- `x` : droit d'exécution

et les opérations pour les répertoires sont données par :

- `r` : droit de lister le contenu du répertoire
- `w` : droit de modification et de suppression du contenu du répertoire
- `x` : droit d'accès comme répertoire de travail sur le répertoire

Pour chaque fichier, la règle va indiquer les opérations acceptées en fonction de la catégorie de sujet (propriétaire, groupe, autre). La commande **`ls`** permet de visualiser les droits. Elle présente pour chaque catégorie de gauche à droite les droits :

i — pour l'utilisateur propriétaire du fichier

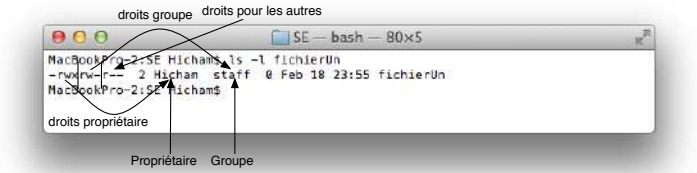
ii — pour l'utilisateur membre du groupe propriétaire du fichier

iii — pour les autres utilisateurs ;

Chaque droit est désigné par une lettre :

- `r` : signifie que le droit en lecture est accordé
- `w` : droit en écriture
- `x` : droit d'exécution
- `-` : le droit correspondant n'est pas accordé

Figure 2.7: Droits d'accès



3.10 Commandes pour modifier les règles

Des commandes permettent de modifier les règles de droits sur les fichiers :

- `chown` : permet de changer le propriétaire (utilisateur et groupe)
- `chgrp` : permet de changer le groupe propriétaire
- `chmod` : permet de changer les droits
- `umask` : permet d'indiquer les droits à la création

La commande `chown` permet de changer le propriétaire d'un fichier et/ou d'un répertoire et récursivement ce qu'il contient. Sa syntaxe est :

`chown [OPTION]…[OWNER][:[GROUP]] FILE…`

La commande `chgrp` permet de changer le groupe d'un fichier et/ou d'un répertoire et récursivement ce qu'il contient. La syntaxe est :

`chgrp [OPTION]…[GROUP]] FILE….` Pour pouvoir exécuter la commande **`chgrp`**, il faut être le propriétaire du fichier et être membre du groupe auquel on veut donner le fichier.

La commande `chmod` permet de changer les droits sur les fichiers. Sa syntaxe est :

`chmod [options] mode fichier`.

L'utilisation de l'option **`-R`** permet de modifier récursivement les autorisations d'un répertoire et de son contenu.

Le mode permet de spécifier les droits :

- de manière symbolique (en utilisant les lettres **`r,w,x`** et les symboles **`+,-,=`**)
- de manière numérique (en octal — base 8)

Le mode est spécifié par : **`personne action droits`**

Personne	Action	Droit
`u` : l'utilisateur propriétaire	`+` : ajouter	`r` : lecture
`g` : le groupe propriétaire	`-` : supprimer	`w` : écriture
`o` : les autres	`=` : initialiser	`x` : exécution
`a` : tous les utilisateurs		

Exemple :

- `u+rwx` : ajouter tous les droits au propriétaire
- `og-w` : enlever le droit d'écriture aux autres
- `a=rx` : donner le droit de lecture et exécution à tous (propriétaire, groupe et autres)
- `g=rwx` : accorder tous les droits au groupe

Le mode peut être spécifié par un nombre en octal (base 8 *cf.* tableau 2.4), dont les chiffres représentent (dans l'ordre de gauche à droite) :

- les droits pour l'utilisateur propriétaire du fichier

Table 2.4: Mode octal

Droits	Binaire	Octal
`---`	000	0
`--x`	001	1
`-w-`	010	2
`-wx`	011	3
`r--`	100	4
`r-x`	101	5
`rw-`	110	6
`rwx`	111	7

- les droits pour le groupe propriétaire du fichier
- les droits pour tous les autres

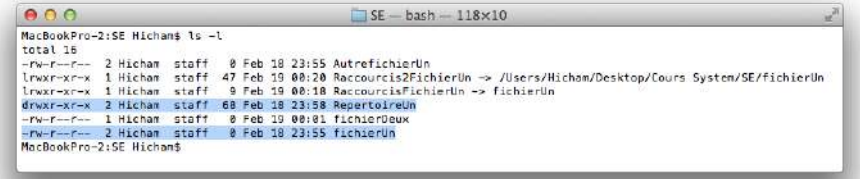
Exemples :

- 700 : représente les droits `rwx------`
- 751 : représente les droits `rwxr-x--x`
- 640 : représente les droits `rw-r-----`

Exercice 2

1. Interdire la lecture et l'accès au répertoire `RepertoireUn` aux utilisateurs ne faisant pas partie du groupe `staff`
2. Donner les droits d'écriture au groupe sur le fichier `fichierUn`
3. Donner le droit d'exécution sur le fichier `fichierUn` au propriétaire
4. Prévoir les droits affichés par la commande `ls -l` après l'exécution de ces commandes
5. Réécrire les commandes avec l'utilisation numérique de la commande `chmod`

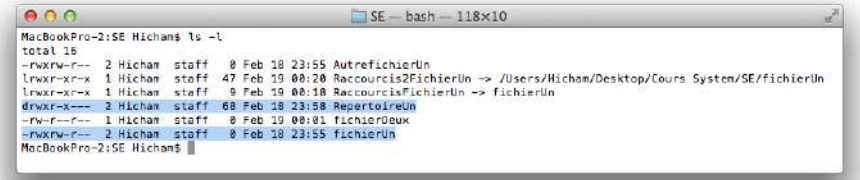
Figure 2.8: Données de l'exercice sur chmod



Solution 2

- Mode symbolique
  1. `chmod o-rx RepertoireUn`
  2. `chmod g+w fichierUn`
  3. `chmod u+x fichierUn`
- Mode numérique
  1. `chmod 750 RepertoireUn`
  2. `chmod 664 fichierUn`
  3. `chmod 764 fichierUn`

Figure 2.9: Solution de l'exercice sur `chmod`



3.11 La commande `umask`

Cette commande permet de spécifier des droits par défaut lors de la création des fichiers. Pour cela, elle utilise des masques sous forme numérique octale. En exécutant cette commande :

- sans paramètre : elle indique le masque courant
- avec le masque en paramètre : elle modifie le masque courant

les droits obtenus sont le complémentaire (à `777` pour les répertoires et à `666` pour les fichiers) de ceux indiqués par le masque.

Table 2.5: Exemple du calcul pour un répertoire

Droits	`777`	`rwxrwxrwx`	`777`	`rwxrwxrwx`
Masque	`022`	`----w--w-`	`227`	`-w--w-rwx`
Droits obtenus	`755`	`rwxr-xr-x`	`550`	`r-xr-x---`

2. Système Unix
Système de gestion de fichiers
3.11 La commande umask

3

Programmation *Shell*

1 Introduction à *bash*

1.1 Les différents Shells et leur initialisation

Il existe plusieurs Shells UNIX :

- C-Shell (`csh` ou `tcsh`)
- Bourne Shell (`sh` ou `bash`)
- Korn Shell (`ksh`), ...

L'interprétation des commandes simples est semblable pour tous. Par contre, l'utilisation pour écrire des scripts diffère beaucoup (définition des variables, structures de contrôle, etc)

L'initialisation du Shell se fait au démarrage du système : Les *Shell* exécutent des fichiers de configuration, qui peuvent contenir des commandes quelconques et sont généralement utilisés pour définir des variables d'environnement et des alias

- `csh` exécute le fichier `~/.cshrc` (le "`rc`" signifie *run command*)
- `tcsh` exécute `~/.cshrc`
- `sh` exécute `~/.profile`
- `bash` exécute `~/.bash_profile` ou à défaut `~/.profile`

Ces fichiers d'initialisation sont "invisibles".

1.2 Variables d'environnement

Elles sont instanciées lorsqu'un Shell est exécuté par le système. Ce sont des variables dynamiques utilisées par les différents processus d'un système d'exploitation. Elles donnent des informations sur le système, la machine et l'utilisateur, entre autres.

La commande `env` permet d'afficher à l'écran toutes les variables d'environnement pour le Shell (cf. figure 3.1).

Pour définir ses propres variables d'environnement, on utilise la commande `export` :

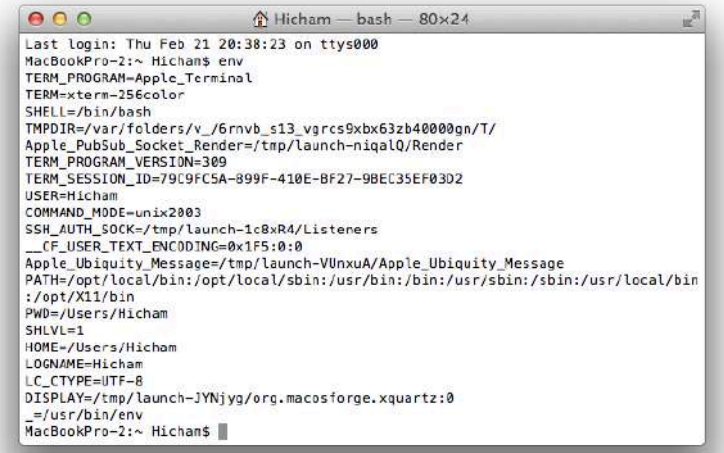
- `VARIABLE=VALEUR` : donne une valeur à une variable
- `export VARIABLE` : définit `VARIABLE` comme une variable d'environnement
- `echo $VARIABLE` : affiche la valeur de la variable

1.3 Entrée, sortie et erreur standards

L'entrée standard est attachée au clavier et a un numéro de descripteur égale à 0. La sortie standard est attachée à l'écran dont le numéro de descripteur est 1. Enfin, l'erreur standard, elle aussi, est attachée à l'écran et a un numéro de descripteur égale à 2. Il est possible de rediriger ces trois flux :

- Redirection de la sortie standard : `$ echo bonjour > test.txt`
- Concaténation pour préserver le contenu du fichier de sortie :
  `$ cat < toto.txt >> FichierSortie`
- Redirection de l'erreur standard : `$ ls abdnkjf 2> erreur.txt`

Figure 3.1: Exemple d'exécution de la commande `env`



- Redirection de l'entrée standard : `$ bc < calcul.dat`
- Redirection de l'entrée et de la sortie standard :
  `$ bc < calcul.dat > resultat.txt`

Exemple : `$ ls abdnkjf > sortie.txt 2>&1` : La sortie de `ls` est enregistrée dans le fichier `sortie.txt`. L'erreur standard est redirigée à la sortie standard. Donc, l'erreur standard est également redirigée au fichier `sortie.txt`

Le Pipe, "|", permet de brancher la sortie standard d'une commande à l'entrée standard d'une autre commande.

Exemple : `$ ls -l | sort -r` (affiche le contenu du répertoire courant trié à l'envers).

1.4 Regroupement des commandes

Pour lancer l'exécution séquentielle de plusieurs commandes sur la même ligne de commande, il suffit de les séparer par un caractère ;

Exemple : `$ cd /tmp ; pwd; echo bonjour; cd ; pwd`

Nous pouvons utiliser l'exécution séquentielle d'une ligne de commandes par regroupement :

- `(cmd1 ; cmd2) ; cmd3`
- `(cmd1 ; cmd2) & cmd3`

- Les commandes regroupés sont exécutées dans un interpréteur enfant (***subshell***)
- *Exemple* :
  `pwd ; (cd ..; pwd; cp user.txt test.txt; ls -l test.txt); pwd`

1.5 Contrôle de tâches

Le Shell attend la fin de l'exécution d'une commande avant d'afficher le prompt suivant. L'exécution en arrière-plan permet à un utilisateur de lancer une commande et de récupérer immédiatement la main pour lancer "en parallèle" la commande suivante (parallélisme logique). Pour cela, on utilise le caractère `&` après la commande qu'on souhaite lancée en arrière-plan.

Exemple : `$ sleep 100 &`

Les commandes utilisées pour le contrôle des tâches sont :

- La commande `jobs` affiche les commandes lancées en arrière plan.
- La combinaison de touches *Ctrl+Z* permet de suspendre la tâche courante, que nous pouvons relancer en arrière-plan en exécutant la commande `bg`
- La commande `fg` permet de récupérer la dernière tâche lancée en arrière plan. Pour récupérer la tâche numéro n, on utilise `fg %`n
- La commande `kill %`n permet de tuer la tâche numéro n lancée en arrière-plan.

2 Les scripts *Shell*

2.1 Définition

Les scripts sont des programmes écrits dans un langage interprété, par exemple le langage du ***Shell***. Un script peut être une simple liste de commandes. La première ligne du script doit préciser l'interpréteur utilisé, elle commence par les deux caractères #!, suivis du chemin de l'interpréteur.

Exemple d'un script shell

```
#!/bin/bash
# liste
echo "Contenu du répertoire courant"
ls -l
echo "----------------------------"
```

Dans cet exemple, la deuxième ligne est un commentaire (commence par le caractère #). Le fichier contenant ces commandes doit être rendu exécutable :

```
$ chmod u+x liste
```

2.2 Variables et substitution

Les variables du ***Shell*** sont des symboles auxquels on affecte des valeurs. Une variable est identifiée par son nom et son contenu est identifié par le symbole $ placé devant son nom.

Affichage en utilisant la commande echo

```
$ VARIABLE=VALEUR
$ echo $VARIABLE
VALEUR
```

Bash réalisera la substitution du contenu d'une variable lorsqu'il rencontre $ suivi d'un nom de variable. Deux comportements sont possibles :

1. Substitution vide : La variable n'est pas définie ou la variable est définie mais son contenu est vide.

2. Substitution du contenu : La variable est définie et son contenu est non nul.

Substitution de variable : Exemple 1

```
$ MSG1="Jean est un "
$ MSG2="chien fort réputé"
$ echo "$MSG1 $METIER $MSG2"
Jean est un chien fort réputé
```

Substitution de variable : Exemple 2

```
$ MSG1="Jean est un "
$ MSG2="chien fort réputé"
$ METIER="dresseur de"
$ echo "$MSG1 $METIER $MSG2"
Jean est un dresseur de chien fort réputé
```

2.3 Substitution de commandes

Bash est en mesure de substituer le résultat d'une ligne de commandes UNIX. Le symbole impliqué dans ce genre de substitution est l'accent grave (`).

Exemple de substitution de commande

```
$ echo pwd
pwd
$ echo `pwd`
/home/compte1/Desktop
$ echo "Mon répertoire de travail est: `pwd`"
Mon répertoire de travail est: /home/compte1/Desktop
```

Il est possible d'assigner le résultat d'une ligne de commande UNIX à une variable.

Affectation du résultat d'une commande à une variable : Exemple

```
$ rep=`pwd`
$ moi=`who am i`
$ machine=`hostname`
$ echo -e "Utilisateur: $moi\n Répertoire de travail:$rep\nMachine:
    $machine"
Utilisateur: compte1   ttys000  Feb 24 22:46
Répertoire de travail: /home/compte1/Desktop
Machine: MacBookPro.local
```

2.4 Neutralisation des caractères

Certains caractères ont des significations particulières pour l'interpréteur de commandes. Par exemple : `&, (, ), *, !, {, }`. Sans un mécanisme d'échappement, ces caractères spéciaux seront interprétés par ***Bash***. Les commandes et programmes qui utilisent ces caractères spéciaux ne pourront pas s'exécuter correctement. D'où la nécessité de neutraliser la signification particulière de ces caractères spéciaux pour ***Bash***.

Exemple : Nous désirons afficher la chaîne de caractères "TOTO & TATA"

```
$ echo TOTO & TATA
[1] 2527
TOTO
-bash: TATA: command not found
$ echo TOTO \& TATA
TOTO & TATA
```

Donc, le symbole `\` permet la neutralisation du caractère qui le suit. Nous pouvons neutraliser la signification spéciale du caractère Espace par les symboles `" "` et `' '` :

- Le guillemet : élimine la signification spéciale du caractère Espace mais permet la substitution des variables et commandes.

- L'apostrophe : élimine la signification spéciale du caractère Espace et empêche la substitution des variables et commandes.

2.5 Paramètres de Bash

Les script shell ont la possibilité d'utiliser des paramètres de position. Par exemple, l'exécution de la commande `cmd $ cmd par1 par2 par3 par4` utilise 4 paramètres `par1 par2 par3 par4`. Dans un programme ***Bash***, le contenu de ces paramètres de position est représenté par : `$1`, `$2`, `$3` jusqu'à `$9`. Le nom du fichier (`cmd`) est représenté par `$0`.

Remarque 3

Pour accéder à un paramètre de position dont le numéro est strictement supérieur à 9, il faut entourer ce numéro par `{}`. (`${10}` *désigne le contenu du dixième paramètre de position. Par contre* `$10`, *sera substitué par le contenu de* `$1` *suivi du caractère 0).*

Exemple de paramètres de position à l'aide d'un programme Bash

```
#!/bin/bash
# Nom du fichier param
# param : montrer l'utilisation des parametres Bourne shell
# Lancer le fichier de commande : param -A -B -C
echo "Numero PID de l'interpreteur de commande: $$"
echo "Nom du fichier de commande: $0"
echo "Nombre de parametres: $#"
echo "Parametre 1: $1"
echo "Parametre 2: $2"
echo "Parametre 3: $3"
echo "Parametre 4: $4"
echo "Toute la ligne de commande: $@"
```

2.6 Lecture et affichage

La commande `read` réalise la lecture à partir de l'entrée standard :
`$ read var1 var2 var3`. Cette commande permet de lire de l'entrée standard et placer les données dans les variables `var1, var2` et `var3`. La séparation des données d'entrée en champs est réalisée par ***Bash*** à l'aide de la variable IFS (***Internal Field Separator***).

Exemple : voici une-ligne de données (Il existe 4 champs)
Dans l'exemple ci-dessous :

- La lecture est réalisée à partir de l'entrée standard
- Les données lues sont placées dans trois variables (`repertoire1`, `repertoire2` et `repertoire3`)
- Le programme termine son exécution par l'affichage des données lues

Exemple d'utilisation de read

```
#!/bin/bash
# nom du fichier: lecture
# lecture : montrer comment lire des données à partir de l'entrée
# standard
echo -e "Les repertoires de l'installation? \c"
read repertoire1 repertoire2 repertoire3
echo "Merci !"
echo -e "L'entree lue : $repertoire1\n $repertoire2\n $repertoire3"
```

2.7 Décalage de paramètres : `shift`

La commande `shift` agit sur les paramètres de position du ***Bash***. A chaque emploi de `shift`

- le paramètre `$1` précédent est perdu
- `$1` est supprimé de `$*` et `$@`
- `$#` est décrémenté de `1`

L'emploi de `shift` nécessite que le ***Shell*** script ait au moins un paramètre (Le code de retour dans le cas où il n'y a pas de paramètres est 1).

Exemple d'utilisation de shift

```
#!/bin/bash
echo "$# : arg1 = $1, arg2 = $2; total : $@"
shift; echo "$# : arg1 = $1, arg2 = $2; total : $@"
shift; echo "$# : arg1 = $1, arg2 = $2; total : $@"
shift; echo "$# : arg1 = $1, arg2 = $2; total : $@"
shift; exit 0
```

2.8 Commandes de test : `test`, `[ ]`

Cette commande permet d'évaluer une expression :

- Si vrai, renvoie 0 (***true***), sinon, renvoie 1 (***false***)
- S'il n'y a pas d'expression, renvoie 1

La commande `test expression` est équivalente à `[ expression ]`. Le tableau 3.1 donne quelques exemples d'expressions pour le test sur les fichiers, les répertoires, sur les chaînes de caractères et pour le test sur des variables numériques.

2.9 Branchement conditionnel : `if-then-elif-else-fi`

Syntaxe de la commande if

```
if liste-commandes-1
then liste-commandes-2
elif liste-commandes-3  # autant de fois que nécessaire
else liste-commandes-4 # si nécessaire
fi
```

Table 3.1: Test sur fichiers, répertoires et chaînes

Expression	vrai si :
`-e fic`	`fic` existe
`-d fic`	`fic` existe et est un répertoire
`-f fic`	`fic` existe et est un fichier « ordinaire»
`-h fic`	`fic` existe et est un lien symbolique
`-s fic`	`fic` existe et est non vide
`-r fic`	`fic` existe et est autorisé en lecture
`-w fic`	`fic` existe et est autorisé en écriture
`-x fic`	`fic` existe et est autorisé en exécution
`ch1 = ch2`	les deux chaînes sont identiques
`ch1 != ch2`	les deux chaînes sont différentes
`n1 -eq n2`	`n1` = `n2`
`n1 -ne n2`	`n1` ≠ `n2`
`n1 -le n2`	`n1` ≤ `n2`
`n1 -ge n2`	`n1` ≥ `n2`
`n1 -lt n2`	`n1` < `n2`
`n1 -gt n2`	`n1` > `n2`
`! exp1`	`exp1` est fausse
`exp1 -a exp2`	`exp1` et `exp2` vraies
`exp1 -o exp2`	`exp1` ou `exp2` est vraie

La condition (booléenne) est en général le code de retour d'une commande UNIX. Le code de retour de la commande détermine le test « `if` » :

- Code de retour valant zéro : Le test « `if` » est vrai.
- Code de retour non nul : Le test « `if` » est faux.

Branchement conditionnel : if-then-elif-else-fi : Exemple 1

```
#!/bin/bash
if [ -d toto ] ; then
    echo "toto est un répertoire"
elif [ -h toto ] ; then
    echo "toto est un lien symbolique"
else
    echo "autre que répertoire ou lien"
fi
```

Branchement conditionnel : if-then-elif-else-fi : Exemple 2

```
#!/bin/bash
if ls toto > /dev/null 2>&1; then
    echo "le fichier toto existe"
else
    echo "le fichier toto n'existe pas"
fi
```

Branchement conditionnel : if-then-elif-else-fi : Exemple 3

```
#!/bin/bash
# Mot secret
# Ce programme demande à l'utilisateur de deviner un mot.
SECRET_WORD="SMI"
echo "Votre nom ?"
read NAME
echo "Bonjour $NAME. Devinez un mot."
echo -e "Vous avez le choix entre : SMA, SMI et SMP : \c"
read GUESS
if [ $GUESS=$SECRET_WORD ]
then
   echo "Congratulations !"
fi
```

2.10 Branchement conditionnel : case-esac

Syntaxe de la commande case-esac

```
case expression in
    motif) liste-commandes-1 ;; # autant de fois
    ...
    *) liste-commandes-2 ;;
esac
```

Exécute la liste-commandes suivant le motif (***pattern*** en anglais) reconnu. Le motif à reconnaitre peut s'exprimer sous forme d'expression rationnelle (ou régulière) utilisant les méta-caractères : `* ? [ ] -`

La commande case-esac : Exemple 1

```
#!/bin/bash
case $1 in
    [Yy][eE][sS] | [oO][uU][iI]) echo "affirmatif" ;;
        [Nn][oO] | [Nn][Oo][Nn]) echo "négatif" ;;
                          yesno) echo "décide-toi" ;;
                              *) echo "quelle réponse!" ;;
esac
```

La commande case-esac : Exemple 2

```
#!/bin/bash
# traiter les options d'une commande ;
# utiliser case - esac pour traiter les options
if [ $# = 0 ]; then
  echo "Usage : casesac -t -q -l NomFich"
  exit 1
fi
for option; do
  case "$option" in
    -t) echo "option -t recu" ;;
    -q) echo "option -q recu" ;;
    -l) echo "option -l recu" ;;
 [!-]*) if [ -f $option ]
        then
          echo "fichier $option trouve"
        else
          echo "fichier $option introuvable"
        fi
        ;;
     *) echo "option inconnue $option recontree"
  esac
done
```

2.11 Boucle `for-do-done`

Syntaxe de la commande for-do-done

```
for variable in liste-de-mots
do
    liste-commandes
done
```

La variable prend successivement les valeurs de la liste de mots, et pour chaque valeur, `liste-commandes` est exécutée.

La commande for-do-done : Exemple 1

```
#!/bin/bash
for i in un deux trois quatre cinq six
do
    echo "Semestre $i"
done
```

La commande for-do-done : Exemple 2

```
#!/bin/bash
for i in ~/Desktop/*.pdf
do
    echo $i
done
```

La commande for-do-done : Exemple 3

```
#!/bin/bash
for fichier in $@; do
    if [ -d $fichier ]; then # Test fichier type
        echo "$fichier est un répertoire"
    elif [ -f $fichier ]; then
        echo "$fichier est régulier"
    elif [ -h $fichier ]; then
        echo "$fichier est un lien symbolique"
    else
        echo "$fichier n'existe pas ou autre type"
    fi
    if [ -o $fichier ]; then # vérifie le propriétaire
        echo "Propriétaire de $fichier"
    else
        echo "Pas propriétaire de $fichier"
    fi
    if [ -r $fichier ]; then # Droit de lecture
        echo "Droit de lecture sur $fichier"
    fi
    if [ -w $fichier]; then # Droit de modification
        echo "Droit d'écriture sur $fichier"
    fi
    if [ -x $fichier ]; then # Droit d'exécution
        echo "Droit d'exécution sur $fichier"
    fi
done
```


2.12 L'instruction `select-do-done`

Syntaxe de la commande select-do-done

```
select variable in liste-de-mots
do
    liste-commandes
done
```

L'instruction `select` génère un menu à partir de `liste-de-mots` et demande à l'utilisateur de faire un choix. Cette commande permet à l'utilisateur de sélectionner une variable parmi une liste de mots. `liste-commandes` est exécutée.

La commande select-do-done : Exemple 1

```
#!/bin/bash
echo "Quel est votre OS préféré ?"
select var in "Linux" "Mac OS X" "Autre"; do
   echo "Vous avez sélectionné $var"
   if [ "$var" = "Autre" ]; then
      break
   fi
done
```

Dans ce premier exemple, l'utilisateur fait entrée un choix parmi les trois mots `"Linux" "Mac OS X" "Autre"`. Le script affichera la phrase `"Vous avez sélectionné"` suivi du choix de l'utilisateur. Si le choix est `Autre`, le script quitte la boucle `select` et termine son exécution.

La commande select-do-done : Exemple 2

```
#!/bin/bash
PS3="Que voulez vous ? "
select ch in "1er" "2eme" "Abandon" ; do
  case $REPLY in
    1) echo "C'est du $ch choix" ;;
    2) echo "Que du $ch choix" ;;
    3) echo "On abandonne ..." ; break ;;
    *) echo "Choix invalide" ;;
  esac
done
```

Dans cet exemple, On remarque l'utilisation de la variable d'environnement REPLY. Cette variable stocke le nombre entrée par l'utilisateur et non pas le mot correspondant au choix. Ceci permettra un meilleur traitement du choix de l'utilisateur avec un simple `case–esac` sur la variable `REPLY`.

2.13 Boucle `while-do-done`

Syntaxe de la commande while-do-done

```
while liste-commandes-1
do
    liste-commandes-2
done
```

La valeur testée par la commande `while` est l'état de sortie de la dernière commande de `liste-commandes-1`. Si l'état de sortie est `0`, alors le *Shell* exécute `liste-commandes-2` puis recommence la boucle.

La commande while-do-done : Exemple 1

```
#!/bin/bash
echo -e "Devinez le mot secret : SMI, SMA, SMP : \c"
read GUESS
while [ $GUESS != "SMI" ]; do
    echo -e "Ce n'est pas $GUESS, devinez : \c"
    read GUESS
done
echo "Bravo"
```

La commande while-do-done : Exemple 2

```
#!/bin/bash
compteur=5
while [ $compt -ge 0 ]; do
    echo $compt
    compt=`expr $compt - 1`
done
```

2.14 Boucle `until-do-done`

Syntaxe de la commande until-do-done

```
until liste-commandes-1
do
    liste-commandes-2
done
```

Le *Shell* teste l'état de sortie de `liste-commandes-1`. Si l'état de sortie est 1, alors, `liste-commandes-2` est exécutée puis la boucle est recommencée.

La commande until-do-done : Exemple 1

```
#!/bin/bash
echo -e "Devinez le mot secret : SMI, SMA, SMP : \c"
read GUESS
until [ $GUESS = "SMI" ]; do
    echo -e "Ce n'est pas $GUESS, devinez encore : \c"
    read GUESS
done
echo "Bravo."
```

La commande until-do-done : Exemple 2

```
#!/bin/bash
compt=5
until [ $compt -lt 0 ]; do
    echo $compt
    compt=`expr $compt - 1`
done
```


2.15 Fonctions Bourne Shell

Nous pouvons rendre la programmation plus structurée en utilisant des fonctions :

Syntaxe de fonction

```
NomDeFonction ( ){
    commandes
}
```

La définition des fonctions Bourne shell doit être faite au début du fichier de commande et il faut prendre préséance sur les commandes systèmes de même nom. Ces fonctions peuvent avoir une valeur de retour : `exit n` (ou `return n`) où `n` est une valeur numérique (`=0 : OK`, `≠0` (et `<256`) : Erreur)

Fonction Bourne Shell : Exemple 1

```
#!/bin/bash
mafonction() {
    echo "Celle-ci est mafonction"
}
# Code principal commence ici
echo "Appel de mafonction..."
mafonction
echo "Done."
```

Fonction Bourne Shell : Exemple 2

```
#!/bin/sh
repertoire () {
    echo "Entrer un nom de repertoire: \c"
    read repertoire
    if [ -d "$repertoire" ]; then
        return 0  # repertoire existe
    else
        return 101  # repertoire inexistant
    fi
}
gestion_erreur () {
    case $ERRNO in
       0) ;;   # pas d'erreur
     101) echo "Répertoire inexistant" ;;
       *) echo "Code d'erreur inconnu"
          exit 1 ;;
    esac
}
# Programme principal
ERRNO=123
while [ $ERRNO -ne 0 ]; do
  # statut de sortie de repertoire () assigné à ERRNO
    repertoire; ERRNO=$?
  # invoquer le gestionnaire d'erreur
    gestion_erreur
done
```


2.16 Différence entre "$@" et "$*"

Lorsque utilisé entre guillemet, la variable `$@` et la variable `$*` n'ont pas la même signification. Pour `"$@"` l'interpréteur de commandes substitue les paramètres de position en leur entourant par des guillemets. Ce n'est pas le cas pour `"$*"`.

Exemple : Différence entre "$@" et "$"*

```
#!/bin/bash
# Nom fichier : exemple_arobase.sh
echo "Utilisation de \$*"
for OPTION in "$*"; do
     echo "Itération : $OPTION"
done
echo "Utilisation de \$@"
for OPTION in "$@"; do
    echo "Itération : $OPTION"
done
```

Exemple : Résultat d'exécution

```
$ ./exemple_arobase.sh un deux troix quatre
Utilisation de $*
Itération : un deux troix quatre
Utilisation de $@
Itération : un
Itération : deux
Itération : troix
Itération : quatre
```

On remarque que le résultat n'est pas le même. C'est pour cela qu'il faut penser à

utiliser `"$@"` dans vos programmes *Bash*.

2.17 Décodage des paramètres

Il existe une commande simple pour le décodage systématique des paramètres de position. Il s'agit de la commande `getopts`. La syntaxe de cette commande :
`getopts optstring name [arg …]` où :

- `optstring` représente les options à reconnaître par `getopts`
- `name` les options reconnues par `getopts` sont placées dans cette variable
- `arg` s'il existe, `getopts` va tenter d'extraire les options à partir de cet argument

Exemple 1 (getopts)

```
#!/bin/sh
while getopts lq OPT; do
  case "$OPT" in
    l) echo "OPTION $OPT reçue" ;;
    q) echo "OPTION $OPT reçue" ;;
    ?) echo "Usage: install [-lq]" ; exit 1 ;;
  esac
  echo "Indice de la prochaine option à traiter : $OPTIND"
done
```

Exemple 1 (getopts) : Résultat d'exécution

```
$ install -l
OPTION l reçue
Indice de la prochaine option à traiter : 2
$ install -l -q
OPTION l reçue
Indice de la prochaine option à traiter : 2
OPTION q reçue
Indice de la prochaine option à traiter : 3
$ install -x
./install : option incorrecte -- x
Usage: install [-lq]
```

Exemple 2 (getopts)

```
#!/bin/sh
while getopts l:q OPT; do
  case "$OPT" in
    l) OPTION="$OPT"
       LOGARG="$OPTARG" ;
       echo "OPTION $OPT reçue; son argument est $LOGARG" ;;
    q) OPTION="$OPT" ;
    ^^Iecho "OPTION $OPT reçue" ;;
    ?) echo "Option invalide détectée"
       echo "Usage: install [-l logfile -q] [nom_repertoire]"
       exit 1 ;;
  esac
  echo "Indice de la prochaine option à traiter : $OPTIND"
done
# Chercher le paramètre nom_repertoire
shift `expr $OPTIND - 1`
if [ "$1" ]
then
  repertoire="$1"
  echo "Répertoire d'installation: $repertoire"
fi
```

2.18 Débogage de Script Shell

Pour simplifier la recherche des erreurs dans un programme ***Bourne Shell***, ce dernier met à notre disposition la commande `set` pour activer les modes de débogage. Les options disponibles sont :

- `-n` : Lire les commandes mais ne pas les exécuter (Vérifie les erreurs de syntaxe sans exécuter le script)
- `-v` : Affiche les lignes lues du programme lors de son exécution
- `-x` : Afficher les commandes et les substitutions lors de leur exécution

Exemple d'utilisation de la commande (set)

```
#!/bin/sh
# settest : montrer l'utilisation des options de set
# pour le débogage
# Utiliser l'option -v (affiche les commandes et leur argument)
set -x

pwd
ls -l
echo `who`
```

4

Filtre programmable *awk*

1 Introduction

Les filtres sont des programmes qui lisent une entrée, effectuent une transformation et écrivent un résultat (sur la sortie standard). Parmi ces filtres on trouve :

- `grep, egrep, fgrep` : recherche d'une expression dans des fichiers
- `diff, cmp, uniq, tr, dd …` : outils de comparaison, conversion de fichiers
- `sed` : éditeur de flots
- `awk` : outil programmable de transformation de texte

Ces outils utilisent les "expressions régulières". Ces expressions sont :

- un moyen algébrique pour représenter un langage régulier
- et permettent de décrire une famille de chaînes de caractères au moyen de métacaractères

2 Expressions régulières et commande `egrep`

- un "caractère simple" ***"matche"*** avec lui-même :
  - a matche avec a
  - 6 matche avec 6
- un métacaractère génère ou précise un ensemble de possibilités
  - . matche avec n'importe quel caractère
  - ^ indique un début de chaîne, etc …
- les métacaractères sont neutralisés par le caractère \

Table 4.1: Expressions régulières définies par des méta-caractères

.	n'importe quel caractère
?	0 ou une occurrence du caractère
*	répétition du caractère précédent
+	Une ou une infinité d'occurrence
[<liste>]	Un choix parmi un ensemble
[^<liste>]	Tout sauf un certain caractère
^ a	a en début de chaîne
$a$$	a en fin de chaîne
$a\{n\}$	n répétitions du caractère a
$a\{n,\}$	au moins n répétitions de a
$a\{n,p\}$	entre n et p répétitions de a
\(...\)	sous-expression "repérée"
\k	k-ème sous-expression repérée

Exemples 1

- .* : zéro ou une infinité de caractères quelconques
- ^.$: chaîne d'un seul caractère

- a+b* : au moins un 'a' suivi de 0 ou une infinité de 'b'
- [ab]+ : au moins un 'a' ou 'b' ou une infinité
- `^\(.*\)\1$` : ligne constituée de 2 occurrences d'une même chaîne de caractères

Remarque 4

*Attention aux confusions avec les méta-caractères du Shell : Sens différents pour méta-caractères : **

`. ?`

Exemples 2

- `v.+` : Les chaînes contenant un `'v'` suivi de n'importe quelle suite de caractères (vandalisme, vestiaire, lavage, ...)
- `[vs].+` : Les chaînes contenant un `'v'` ou un `'s'` suivi de n'importe quelle suite de caractères (vandalisme, voiture, descendre, sandales, ...)
- `a.*a` : Les chaînes contenant deux `'a'` (palais, sandales, pascale, cascade, ...)
- `[ps].*a.*a` : Les chaînes contenant un `'p'` ou un `'s'` suivi d'une sous chaîne contenant deux `'a'` (sandales, pascale, apprentissage automatique, ...)

2.1 Commande `egrep`

Cette commande permet de rechercher dans des fichiers d'une chaîne ou d'une sous chaîne de caractères ou simplement d'un mot ou d'une chaîne formalisée par une expression régulière. La syntaxe de la commande est :
`egrep [options] <chaîne recherchée> <fichier>` Le résultat de cette commande est les lignes du fichier contenant ce qui est recherché ou autre résultat, suivant les options utilisées. Les options les plus utilisées sont :

- `egrep <chaîne> fichier` : recherche de `<chaîne>` dans `fichier`
- `egrep -v <chaîne> fichier` : recherche inversée
- `egrep -w <chaîne> fichier` : recherche d'un mot exact
- `egrep -<nombre de lignes> <chaîne> fichier` : ligne de contexte
- `egrep -n <chaîne> fichier` : numéros de lignes

- `egrep -n<nombre de lignes> <chaîne> fichier` : combinaison des deux options précédentes
- `egrep -i <chaîne> fichier` : respect de la casse
- `egrep -c <chaîne> fichier` : nombre d'occurrences

Exemple
Considérons le fichier de notes suivant :

Table 4.2: Fichier de notes

```
crepetna:Crepet, Nathalie:CREN1807750:92:87:88:54:70
yosnheat:Yos Nhean, Trakal:YOST19087603:84:73:70:50:73
benelaur:Benel, Aurelien:BENA80207700:84:73:89:45:100
soucypas:Soucy, Pascal:SOUP14067502:95:90:89:87:99
```

On peut extraire les lignes qui contiennent une note comprise entre 90 et 99 : `$ egrep :9 notes.txt`. Maintenant, comment faire pour extraire les lignes où la dernière note est comprise entre 90 et 99 ?. Une solution est d'utiliser une expression régulière :
`$ egrep :.*:.*:.*:.*:.*:.*:9 notes.txt` où `.*:.*:.*:.*:.*:.*:9` représente une chaîne avec `7 ':'` entre lesquels on peut avoir n'importe quoi et dont le dernier ':' est suivi d'un 9. Une autre solution serait :
`$ egrep '\(:.*\)\{6\}:9' notes.txt` ou `$ egrep '9[0-9]$' notes.txt`

3 Filtre programmable *awk*

3.1 Introduction

Le langage *awk* a été developpé par *Alfred Aho, Peter Weinberger & Brian Kernighan*. Il s'agit d'un programme UNIX capable d'interpréter un programme utilisateur. Ce programme doit être écrit en utilisant les instructions légales et selon le format de *awk*. (voir la page de manuel : `man awk`). Le concept de programmation est appelé "piloté par données" (*data-driven*). On peut utiliser *awk* pour :

- récuperer de l'information ;
- générer des rapports ;
- transformer des données...

Table 4.3: Options de la commande *awk*

Option et paramètre	Signification
`-Fc`	Caractère `c` est le séparateur de champ.
`-f prog`	`prog` est le nom du fichier contenant le programme *awk*
`'programme'`	`programme` *awk* donné directement entre apostrophes
`-v var=valeur…`	Initialisation de variables avant l'exécution du programme
`fichier1…`	Fichier contenant les données à traiter

Le synopsis de *awk* :
`awk [-Fc] [-f prog | 'prog' ] [-v var=valeur…] [fich1 fich2 …]` Chaque ligne d'entrée du fichier est séparée en champs `$1, $2, $3, …`.

Remarque 5

Ces champs n'ont rien à voir avec les `$1, $2, …` *du Bourne Shell.*

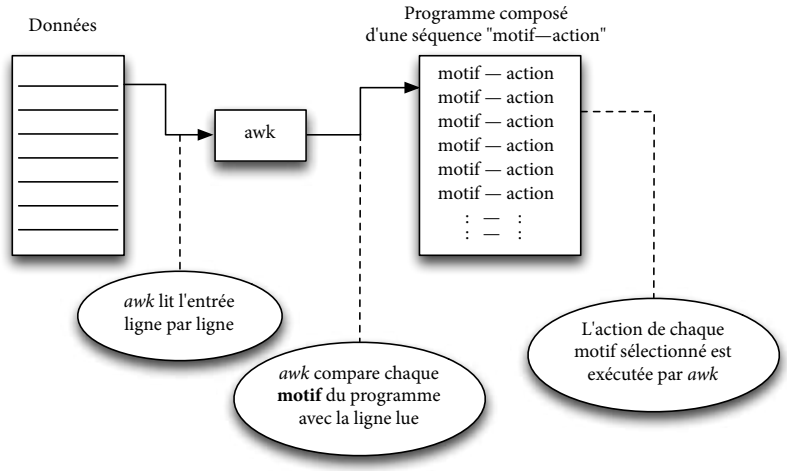
On peut spécifier un programme *awk* dans un fichier par l'option `-f` ou l'écrire directement entre apostrophes. Les données à traiter sont contenues dans les fichiers `fich1, fich2, …` ou acheminées via l'entrée standard. Le corps d'un programme *awk* est une séquence de "motif—action" (*pattern—action*). On peut passer des paramètres à un programme *awk* par l'option `-v`

- Cette option est utile lorsque *awk* est utilisée à l'intérieur d'un fichier de commandes *Bourne Shell*
- Par exemple, on peut passer la valeur des variables d'un programme *Bourne Shell* à des variables d'un programme *awk*

Structure d'un programme awk

```
BEGIN    {action0}
motif1   {action1}
motif2   {action2}
...
END      {actionF}
```


Figure 4.1: Principe de fonctionnement



Principe (*cf.* figure 4.1):

- Initialisation (`BEGIN`) : effectuer `action0`
- Corps : pour chaque ligne du texte entré
  - si `motif1` est vérifiée effectuer `action1`
  - si `motif2` est vérifiée effectuer `action2`
  - etc …
- Terminaison (`END`) : effectuer `actionF`

Remarque 6

Si le `motif` *est omis alors l'action associée est toujours effectuée et si l'*`action` *est omise on affiche toute la ligne.*

3.2 Variables et structure d'une ligne

Chaque ligne ("*Record*") est automatiquement séparée en champs ("*Fields*") en utilisant un séparateur (par défaut : blancs et/ou tabulations) ou précisé avec l'option `-F` d'*awk*. Les variables utilisées par *awk* sont :

- `NR, NF` numéro de ligne (*Record*), nombre de champs (*Fields*)
- `$0` contenu de la ligne courante
- `$1, $2 … $NF` contenu du ième …dernier champ
- `RS, FS` séparateur de lignes (défaut = `\n`), de champs (défaut = blanc et tab)
- `ORS, OFS` séparateurs en sortie (pour modifier l'impression)

Remarque 7

La variable `FS` peut aussi être initialisée lors de l'appel de awk via l'option : `-Fc` : le séparateur de champs prend la valeur du caractère `c`.

Un premier exemple

```
$ awk 'BEGIN {print "Premier programme awk"} {print $0}\
> END {print "Fin du programme awk"}' data.txt
```

Dans ce premier exemple, le programme ***awk*** est spécifié directement entre apostrophes.

3.3 Motifs et actions

Un motif est une expression régulière qui va être comparée à un champs (`$1, $2, …, $NF`). Si une correspondance est trouvée entre l'expression régulière et l'enregistrement, le motif devient vrai et l'action correspondante est exécutée. La syntaxe des motifs peut s'exprimer de trois façons :

- motif en fonction d'une expression régulière
- motif en fonction d'expressions logiques
- motif en utilisant les deux formats

Table 4.4: Les expressions logiques pour les motifs

Opérateur	Description
<	Inférieur à
>	Supérieur à
==	Egalité
!=	Différent
&&	ET logique
||	OU logique
~	Permet de comparer l'expression régulière à un champ précis

Exemple de motifs

```
$1 == $2
(($2 > 100) || ($2 == $3*50)) && ($4 > 10)
($1 ~ /[a-z]/) && ($2 ~/[0-9]/)
($1 ~ /[a-z]/) && ($2 ~/[0-9]/) && ($3 < 10)
```

Les actions : décrivent les opérations à effectuer lorsque le motif décrit en tête de requête est vérifié et ont une syntaxe similaire à celle du langage C. On trouvera aussi un ensemble de fonctions spécifiques présentées dans le tableau 4.5.

3.4 Fonctions utilisateur

L'utilisateur peut définir ses propres fonctions. Ces fonctions peuvent se trouver n'importe où dans le corps du programme ***awk***. La déclaration d'une fonction se fait de la façon suivante :

Premier exemple awk

```
function nom_fonction (arguments) {
    instructions
}
```

La fonction peut être appelée dans n'importe quel bloc action d'une requête ***awk***. Ci-

Table 4.5: Exemple de fonctions prédéfinies

Fonction	Description
`sqrt(arg)`	renvoie la racine carré de l'argument
`log(arg)`	renvoie le logarithme népérien de l'argument
`exp(arg)`	renvoie l'exponentiel de l'argument
`int(arg)`	renvoie la partie entière de l'argument
`length`	renvoie la longueur de l'enregistrement courant
`length(arg)`	renvoie la longueur de la chaine passée en argument
`print [arg1[,arg2],…] [> dest]`	affiche les arguments `"arg1"`, `"arg2"`, …sur la sortie standard sans les formater. Avec l'option `"> dest"`, l'affichage est redirigé sur le fichier `"dest"` au lieu de la sortie standard
`printf(format,arg1,arg2,…) [> dest]`	affiche les arguments `arg1, arg2, …` sur la sortie standard après les avoir formatés à l'aide de la chaîne de contrôle `"format"`. Avec l'option `"> dest"`, l'affichage est redirigé sur le fichier `"dest"` au lieu de la sortie standard

dessous un exemple de deux fonctions en *awk*. La première calcul le minimum de deux nombres et la deuxième donne le factoriel d'un entier.

Exemple de fonctions

```
function minimum (n,m) {
    return (m < n ? m : n)
}
function factoriel (num) {
    (num == 0) ? return 1 : return (num * factoriel(num - 1))
}
$1 ~ /^Factoriel$/ { print factoriel($2) }
$1 ~ /^Minimum$/ { print minimum($2, $3) }
```

3.5 Les structures de contrôle

L'ensemble des structures de contrôle de *awk* fonctionnent comme celles du langage C. Le terme instruction désigne un ensemble d'instructions *"awk"* séparées par le caractère `";"` ou `"return"` et encadrées par des `"{","}"`

- Structure de contrôle `if, else` :

Structure if

```
if (condition)
    instruction
else
    instruction
```

- Structure de contrôle `while`

Structure while

```
while (condition)
    instruction
```

- Structure de contrôle `for`

Boucle for

```
for (init;condition;itération)  (ou for (var in tableau))
    instruction
```

- Instruction `"break"` : provoque la sortie du niveau courant d'une boucle `"while"` ou `"for"`.
- Instruction `"continue"` : provoque l'itération suivante au niveau courant d'une boucle `"while"` ou `"for"`.
- Instruction `"next"` : force *"awk"* à passer à la ligne suivante du fichier en entrée.
- Instruction `"exit"` : force *"awk"* à interrompre la lecture du fichier d'entrée comme si la fin avait été atteinte.

Exemples de structures de contrôle

```
if ($3 == foo*5) {
    a = $6 % 2;
    print $5, $6, "total", a;
    b = 0;
}
else {
    next
}
while ( i <= NF) {
    print $i;
    i ++;
}
for (i=1; ( i<= NF) && ( i <= 10); i++) {
    if ( i < 0 ) break;
    if ( i == 5) continue;
        print $i;
}
```

Table 4.6: Fichier de données "population"

Russie	8649	275	Asie
Canada	3852	25	Amérique
Chine	3705	1032	Asie
France	211	55	Europe

Exemples

Le premier exemple calcul la superficie totale et la population totale de tous les pays. Le second exemple cherche le pays avec la plus grande population.

Utilisation de awk : Exemple 1

```
BEGIN {printf("%10s %6s %5s %s\n", "Pays","Superf","Pop","Cont")}
{ printf ("%10s %6s %5s %s\n", $1, $2, $3, $4)
  superf = superf + $2
  pop = pop + $3
}
END { printf("\n %10s %6s %5s\n", "TOTAL", superf, pop)}
```

Utilisation de awk : Exemple 2

```
{  if (maxpop < $3) {
       maxpop = $3
       pays = $1
   }
}
END { print "Pays : " pays "/Max-Pop: " maxpop }
```

Exemple : Fréquence des mots dans un texte

Dans le domaine de l'analyse textuelle, la fréquence des mots est un outil très utilisée dans l'authentification des documents. Nous allons créer un petit programme capable de donner la fréquence d'apparition des mots dans un texte. Les étapes à suivre sont :

1. Isolation des mots par `gsub()` revient à éliminer les caractères de ponctuation :

Suppression des caractères de ponctuation

```
gsub(/[.,:;!?()[]]/, "")
```

2. Confondre les mots majuscules et les mots minuscules (Cette conversion est réalisée en dehors du programme *awk*) :

Convertir les minuscules en majuscules

```
$ cat texte.txt | tr 'a-z' 'A-Z' > lignes.tmp
```

3. Compter les mots revient à stocker les mots dans un tableau **associatif**. Les indices du tableau sont les mots eux-mêmes et la valeur d'un élément du tableau est le nombre d'apparitions d'un mot :

Compter les occurrences

```
for (i=1; i<=NF; i++)
    compte[$i]++
```

Exemple : Fréquence des mots dans un texte

```
#!/bin/sh
# Programme awk pour compter le nombre d'occurence de mots d'un texte

# Convertir le texte en majuscule et le mettre dans "lignes.tmp"
cat texte.txt | tr 'a-z' 'A-Z' > lignes.tmp

awk '
# A la fin du programme afficher le résultat en ordre décroissant
# numerique
END     {
           for (mot in compte) {
              print compte[mot], mot
              total += compte[mot]
        }
        print "Nombre total des mots : " total
        }
        {
            gsub(/[.,:;!?(){}]/,"") # elimine la ponctuation
            for (i=1; i<=NF; i++)   # placer les mots trouves dans
               compte[$i]++         # un tableau associatif
        }
' lignes.tmp | sort -nr
```

3.5 Les structures de contrôle
Filtre programmable AWK
4. Filtre programmable AWK