

# Systemes d'exploitation I

Pr. HICHAM LAANAYA — [hicham.laanaya@gmail.com](mailto:hicham.laanaya@gmail.com)

2015—2016

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES d'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## ARCHITECTURE simplifiée d'un ORDINATEUR

- Une unité pour effectuer les traitements, également appelée **unité centrale (UC)** ou processeur,
- Une unité pour contenir les programmes à exécuter qui est le lieu de travail dans un ordinateur 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

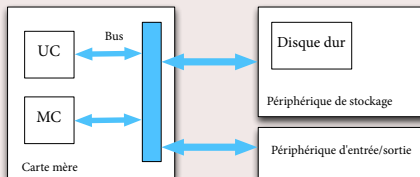


Figure: Architecture simplifiée d'un ordinateur

## LA 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 :

- Le **microprocesseur**
- La **mémoire vive RAM** (*Random Access Memory*) : La mémoire vive RAM (Random Access Memory). Elle représente 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 (de l'ordre de quelques centaines de MHz).
- 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.

## L'UNITÉ CENTRALE OU 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. La taille de la mémoire cache est généralement de l'ordre de quelques centaines de KO. 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.

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## NOTIONS DE SYSTÈMES D'EXPLOITATION

Le système d'exploitation est un gestionnaire de ressources :

- il contrôle l'accès à toutes les ressources de la machine,
- l'attribution de ces ressources aux différents utilisateurs,
- 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.



## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## 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 est la X (prononcer dix). Mac OS X se distingue par un noyau Darwin 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 dominant 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
  - Principales distributions de Linux : RedHat, Debian, Caldera, Ubuntu, ...

## LES PRINCIPAUX SYSTÈMES D'EXPLOITATION (SUITE)

### ■ La famille des Windows :

- Microsoft propose en 1992 Windows 3.10 et Windows pour Workgroups 3.11 dont les mots clés sont Multifenêtres et Multitâches coopératif. En 1993, on voit apparaître la première version de Windows NT 3.1 suivie en 1994 par NT 3.5
- L'année 1995, verra la sortie du fort célèbre Windows 95
- En 1996, Windows NT 4 avec deux versions station de travail et Serveur. Ensuite,
- Windows Terminal Server : un système qui simule un environnement multi-utilisateurs et prend en charge la connexion de plusieurs terminaux
- En 1998 Windows 98
- 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
- VISTA, Windows Seven, Windows 8

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES d'exploitation

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME Unix

Introduction au système Unix

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## 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.

## Distributions Unix

- Des Unix propriétaires :

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

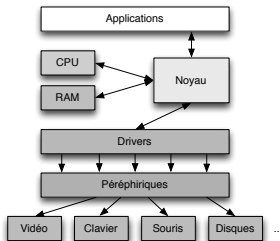
- Des Unix libres

- Linux sur plate-forme Intel, Sparc, Alpha, Mac, ...
- FreeBSD sur plate-forme Intel, Alpha, PC-98
- OpenBSD également multi-plate-forme

## 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, ...)



## Logiciels PROPRIÉTAIRES

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

## Logiciels libres

- Les logiciels libres sont les logiciels que l'on 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

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système Unix

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

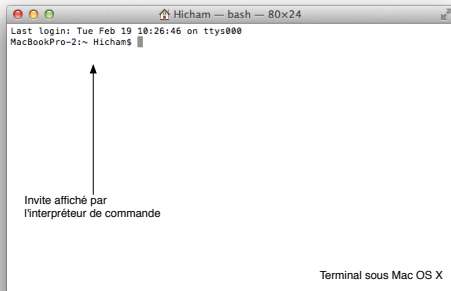


COMMANDES DE BASE du *Shell*Un *Shell*

- est un interpréteur de commande en mode texte
- 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*)
- La commande est interprétée et exécutée après la frappe de la touche "**Entrée**"



## 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

## MÉTA-CARACTÈRES du *Shell*

- sont interprétés spécialement par le *Shell* avant de lancer la commande entrée par l'utilisateur
- permettent de spécifier des ensembles de fichiers, sans avoir à rentrer tous leurs noms

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

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES d'exploitation

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME Unix

Introduction au système Unix

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

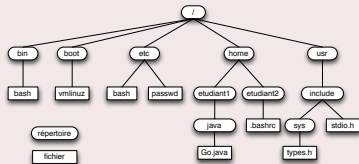
Expressions régulières et commande *egrep*

Filtre programmable *awk*

## 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

## HIÉRARCHIE DU SYSTÈME DE FICHIERS



## LES DIFFÉRENTS TYPES DE FICHIERS

- 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

## LES i-noœuds

À chaque fichier correspond un **i-noœud** 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 noœud** (modification d'attributs)
- l'identification de la **ressource** associée (pour les fichiers spéciaux)

## 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
- 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, ...)

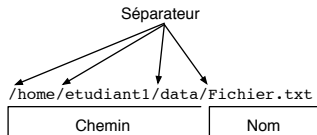
## CARACTÈRES ACCEPTABLES POUR LES NOMS DE FICHIERS

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

- UNIX est un système qui distingue les caractères majuscules et minuscules
- Ne pas utiliser le caractère espace comme nom de fichier ou répertoire !!

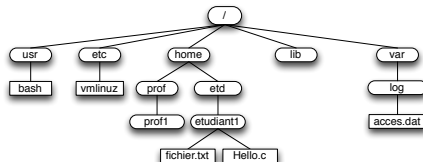
## LES CHEMINS D'ACCÈS

- Pour identifier un fichier dans l'arborescence on indique le nom complet du fichier
- Le nom complet du fichier 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



## LES CHEMINS D'ACCÈS (SUITE)

- On distingue deux expressions d'un chemin :
  - Le chemin d'accès absolu (**chemin absolu**)
  - Le chemin d'accès relatif (**chemin relatif**)
- Le chemin absolu commence par le symbole séparateur, il exprime le chemin complet à partir de la racine de l'arborescence
- Le chemin relatif commence par un autre caractère que le caractère séparateur. Il indique un chemin à partir du répertoire de travail courant
- **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`
- `".."` désigne le répertoire parent, `"."` désigne le répertoire courant

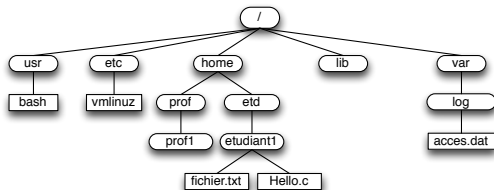




## LES CHEMINS D'ACCÈS : EXERCICE

Dans la hiérarchie précédente, exprimez les chemins suivants :

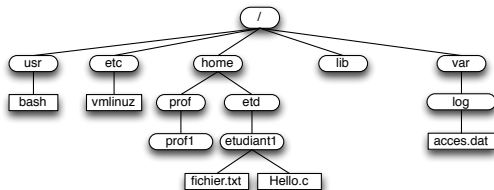
- 1 absolu pour **prof1**
- 2 absolu pour **etc**
- 3 absolu pour **prof**
- 4 relatif à **log** pour **acces.dat**
- 5 relatif à **prof** pour **acces.dat**
- 6 relatif à **etudiant1** pour **acces.dat**



## LES CHEMINS D'ACCÈS : CORRECTION

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`



## Quelques commandes d'accès aux fichiers

Commande	Explications
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)

## Quelques commandes d'accès aux répertoires

Commande	Explications
ls	affiche la liste des fichiers contenus dans un répertoire <b>Options :</b> -a liste aussi les fichiers cachés -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

## QUELQUES COMMANDES DE MANIPULATION DU SYSTÈME DE FICHIERS

Commande	Explications
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

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

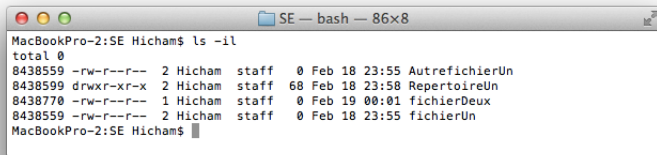
## NOTION DE LIENS PHYSIQUES

- Un même fichier peut avoir donc plusieurs noms
- Il y a plusieurs liens physiques sur le fichier

## NOTION DE LIENS SYMBOLIQUES

Un lien symbolique 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
  - soit un chemin relatif (qui doit être valide depuis le répertoire dans lequel se trouve le lien !)



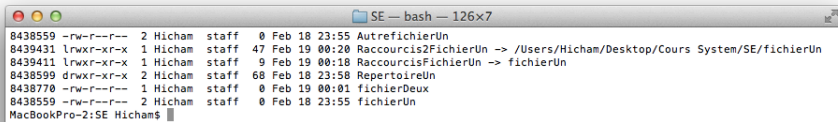
```
SE — bash — 86x8
MacBookPro-2:SE Hicham$ ls -il
total 0
8438559 -rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 AutrefichierUn
8438599 drwxr-xr-x  2 Hicham  staff  68 Feb 18 23:58 RepertoireUn
8438770 -rw-r--r--  1 Hicham  staff   0 Feb 19 00:01 fichierDeux
8438559 -rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 fichierUn
MacBookPro-2:SE Hicham$
```

## NOTION DE LIENS : RÉSUMÉ

- Les liens physiques sont plusieurs entrées de répertoires du même i-nœud (ce sont donc des fichiers réguliers)
- 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



```
SE — bash — 126x7
8438559 -rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 AutrefichierUn
8439431 lrwxr-xr-x  1 Hicham  staff  47 Feb 19 00:20 Raccourcis2FichierUn -> /Users/Hicham/Desktop/Cours System/SE/fichierUn
8439411 lrwxr-xr-x  1 Hicham  staff   9 Feb 19 00:18 RaccourcisFichierUn -> fichierUn
8438599 drwxr-xr-x  2 Hicham  staff  68 Feb 18 23:58 RepertoireUn
8438770 -rw-r--r--  1 Hicham  staff   0 Feb 19 00:01 fichierDeux
8438559 -rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 fichierUn
MacBookPro-2:SE Hicham$
```

## NOTIONS D'UTILISATEUR ET DE GROUPE

- Pour pouvoir accéder au système de fichier, l'utilisateur doit être connecté
- Pour ce 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
- A chaque groupe d'utilisateur le système fait correspondre un numéro d'identification `gid` (*Group IDentifier*)
- Ces informations sont stockées dans des fichiers d'administration
  - `/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)
- Un utilisateur peut appartenir à plusieurs groupes.

## 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.

## NOTIONS DE SÉCURITÉ

*Concept de base*

<b>sujet</b>	Utilisateur ou Processus qui veut exécuter une opération ;
<b>objet</b>	Fichier sur lequel on veut exécuter une opération ; et enfin
<b>opération</b>	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**.

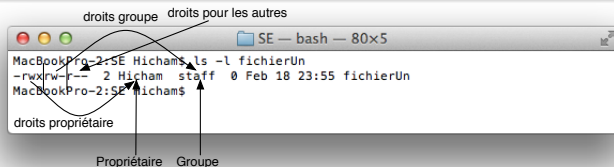


## SÉCURITÉ sous UNIX

- 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
- Les opérations pour les fichiers réguliers :
  - *r* : droit de lecture
  - *w* : droit de modification et de suppression
  - *x* : droit d'exécution
- Les opérations pour les répertoires :
  - *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

## SÉCURITÉ sous UNIX (suite)

- 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



## 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`

- La commande `chown` permet de changer le propriétaire d'un fichier et/ou d'un répertoire et récursivement ce qu'il contient
- La syntaxe : `chown [OPTION]...[OWNER][:[GROUP]] FILE...`

### LA COMMANDE `chgrp`

- La commande `chgrp` permet de changer le groupe d'un fichier et/ou d'un répertoire et récursivement ce qu'il contient
- La syntaxe : `chgrp [OPTION]...[GROUP] FILE...`
- Conditions : être le propriétaire du fichier et être membre du groupe auquel on veut donner le fichier

## LA COMMANDE CHMOD

- La commande chmod permet de changer les droits sur les fichiers
- Syntaxe : `chmod [options] mode fichier`
- options : -R : modifier récursivement les autorisations d'un répertoire et 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+rw` : ajouter tous les droits au propriétaire
- `og-w` : enlever le droit d'écriture aux autres
- `a=r` : donner le droit de lecture et exécution à tous (propriétaire, groupe et autres)
- `g=rwx` : accorder tous les droits au groupe

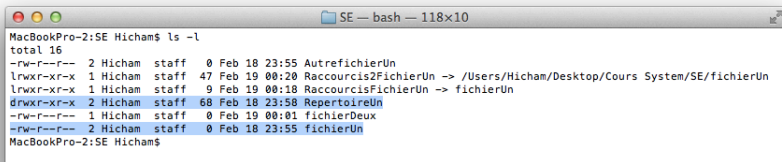
## LA COMMANDE CHMOD (SUITE)

- Le mode est spécifié par un nombre en octal (base 8), dont les chiffres représentent (dans l'ordre de gauche à droite) :
  - les droits pour l'utilisateur propriétaire du fichier
  - les droits pour le groupe propriétaire du fichier
  - les droits pour tous les autres

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

## Exemples

700 : représente les droits `rwX-----`751 : représente les droits `rwXr-x--x`640 : représente les droits `rw-r-----`



```
MacBookPro-2:SE Hicham$ ls -l
total 16
-rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 AutrefichierUn
lrwxr-xr-x  1 Hicham  staff  47 Feb 19 00:20 Raccourcis2FichierUn -> /Users/Hicham/Desktop/Cours System/SE/fichierUn
lrwxr-xr-x  1 Hicham  staff   9 Feb 19 00:18 RaccourcisFichierUn -> fichierUn
drwxr-xr-x  2 Hicham  staff  68 Feb 18 23:58 RepertoireUn
-rw-r--r--  1 Hicham  staff   0 Feb 19 00:01 fichierDeux
-rw-r--r--  2 Hicham  staff   0 Feb 18 23:55 fichierUn
MacBookPro-2:SE Hicham$
```

## LA COMMANDE CHMOD : EXERCICES

- 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 à l'utilisateur propriétaire
- 4 Prévoir les droits affichés par la commande `ls -l` après exécution de ces commandes
- 5 Réécrire les commandes avec l'utilisation numérique de la commande `chmod`

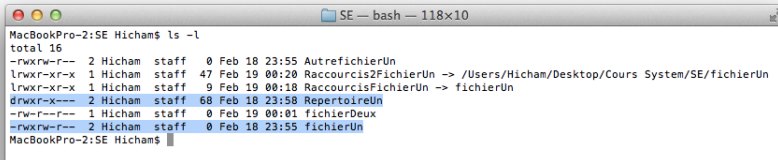
## LA COMMANDE CHMOD : CORRECTION

## ■ 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`



```
MacBookPro-2:SE Hicham$ ls -l
total 16
-rwxrw-r-- 2 Hicham  staff   0 Feb 18 23:55 AutrefichierUn
lrwxr-xr-x 1 Hicham  staff  47 Feb 19 00:20 Raccourcis2FichierUn -> /Users/Hicham/Desktop/Cours System/SE/fichierUn
lrwxr-xr-x 1 Hicham  staff   9 Feb 19 00:18 RaccourcisFichierUn -> fichierUn
drwxr-x--- 2 Hicham  staff  68 Feb 18 23:58 RepertoireUn
-rw-r--r-- 1 Hicham  staff   0 Feb 19 00:01 fichierDeux
-rwxrw-r-- 2 Hicham  staff   0 Feb 18 23:55 fichierUn
MacBookPro-2:SE Hicham$
```

## LA COMMANDE UMASK

### La commande umask

- permet de spécifier des droits par défaut lors de la création des fichiers
- utilise des masques sous forme numérique octale
- sans paramètre : indique le masque courant
- avec le masque en paramètre : 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

### Exemple d'un répertoire :

Droits	777	rw-rw-rw-	777	rw-rw-rw-
Masque	022	---w--w-	227	-w--w-rw-
Droits obtenus	755	rw-r-xr-x	550	r-xr-x---



## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## LES DIFFÉRENTS SHELLS

- 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
- L'utilisation pour écrire des scripts diffère beaucoup (définition des variables, structures de contrôle, etc)

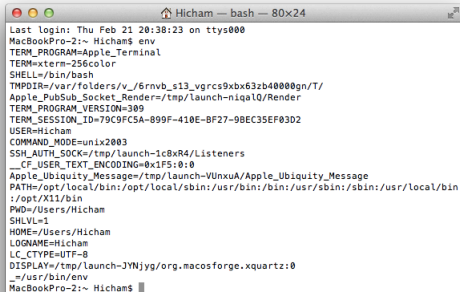
## INITIALISATION d'UN SHELL

Lors de leur démarrage,

- 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
- les fichiers d'initialisation sont "invisibles"

## 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 `set` affiche à l'écran toutes les variables d'environnement pour le Shell
- `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

A terminal window titled 'Hicham — bash — 80x24' showing the output of the 'env' command. The output lists various environment variables such as 'Last login', 'TERM\_PROGRAM', 'SHELL', 'TMPDIR', 'Apple\_PubSub\_Socket\_Render', 'TERM\_PROGRAM\_VERSION', 'TERM\_SESSION\_ID', 'USER', 'COMMAND\_MODE', 'SSH\_AUTH\_SOCK', '\_\_CF\_USER\_TEXT\_ENCODING', 'Apple\_Ubiquity\_Message', 'PATH', 'PWD', 'SHLVL', 'HOME', 'LOGNAME', 'LC\_CTYPE', 'DISPLAY', and '.\_'.

```
MacBookPro-2:~ Hicham$ env
TERM_PROGRAM=Apple_Terminal
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/v_/6rnvb_s13_vgrcs9xbx63zb40000gn/T/
Apple_PubSub_Socket_Render=/tmp/launch-niqlQ/Render
TERM_PROGRAM_VERSION=309
TERM_SESSION_ID=79C9FCSA-899F-410E-BF27-9BEC35EF03D2
USER=Hicham
COMMAND_MODE=unix2003
SSH_AUTH_SOCK=/tmp/launch-1c8xR4/Listeners
__CF_USER_TEXT_ENCODING=0x1F5:0:0
Apple_Ubiquity_Message=/tmp/launch-VUxua/AppUbiqUity_Message
PATH=/opt/local/bin:/opt/local/sbin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
:/opt/X11/bin
PWD=/Users/Hicham
SHLVL=1
HOME=/Users/Hicham
LOGNAME=Hicham
LC_CTYPE=UTF-8
DISPLAY=/tmp/launch-JYNjyg/org.macosforge.xquartz:0
._=/usr/bin/env
MacBookPro-2:~ Hicham$
```

## ENTRÉE STANDARD

- attachée au clavier
- numéro de descripteur 0

## SORTIE STANDARD

- attachée à l'écran
- numéro de descripteur 1

## ERREUR STANDARD

- attachée à l'écran
- numéro de descripteur 2

## REDIRECTIONS DES ENTRÉES/SORTIES

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`
- 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`

## Pipe "|"

- 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)

## 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`
- Exécution séquentielle d'une ligne de commandes par regroupement
  - `(cmd1 ; cmd2) ; cmd3`
  - `(cmd1 ; cmd2) & cmd3`
  - Les commandes regroupées sont exécutées dans un interpréteur enfant (*subshell*)
  - **Exemple** : `pwd ; (cd .. ; pwd ; cp user.txt test.txt ; ls -l test.txt) ; pwd`

## CONTRÔLE DES 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)
- On utilise le caractère & pour lancer une commande en arrière-plan
- **Exemple** : `$ sleep 100 &`
- La commande `jobs` affiche les commandes lancées en arrière plan.

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## Scripts

- 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 :

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

- La deuxième ligne est un commentaire
- Le fichier contenant ces commandes doit être rendu exécutable : `$ chmod u+x liste`

## VARIABLES

- Les variables du *Shell* sont des symboles auxquels on affecte des valeurs
- Une variable est identifiée par son nom
- Le contenu de la variable est identifié par le symbole `$` placé devant le nom

**Exemple :** Affichage en utilisant la commande `echo`

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

## Substitution: VARIABLES

- *Bash* réalisera la substitution du contenu d'une variable lorsqu'il rencontre `$` suivi d'un nom de variable
- Deux comportements possibles :
  - Substitution vide : La variable n'est pas définie ou la variable est définie mais son contenu est vide
  - Substitution du contenu : La variable est définie et son contenu est non nul

**Exemple 1 :**

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

**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
```



## Substitution: 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 :

```
$ echo pwd
pwd
$ echo `pwd`
/Users/Hicham/Desktop
$ echo "Mon répertoire de travail est: `pwd`"
Mon répertoire de travail est: /Users/Hicham/Desktop
```

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

## Exemple :

```
$ REPERTOIRE=`pwd`
$ JESUIS=`who am i`
$ MACHINE=`hostname`
$ echo -e "Utilisateur: $JESUIS\n Répertoire de travail: $REPERTOIRE\n\
> Machine: $MACHINE"
Utilisateur: Hicham ttys000 Feb 24 22:46
Répertoire de travail: /Users/Hicham/Desktop
Machine: MacBookPro.local
```

## NEUTRALISATION DES CARACTÈRES

- Certains caractères ont des significations particulières pour l'interpréteur de commandes  
Par exemple : `&`, `(`, `)`, `*`, `!`, `{`, `}`, etc.
- 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

## PARAMÈTRES de BASH

- Paramètres de position : `$ cmd 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`

## 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: $#"
```

## LECTURE ET AFFICHAGE

- La commande `read` réalise la lecture à partir de l'entrée standard : `$ read var1 var2 var3`
- Lecture 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)
- La lecture est réalisée à partir de l'entrée standard
- Les données lues sont placées dans trois variables (`REPertoire1`, `REPertoire2` et `REPertoire3`)
- Le programme terminé 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"
```

## 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

## 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
```

## COMMANDES DE TEST : test, [ ]

- Permet d'évaluer une expression.
  - Si vrai, renvoie 0 (*true*), sinon, renvoie 1 (*false*)
  - S'il n'y a pas d'expression, renvoie 1
- `test expression` est équivalent à `[ expression ]`

## 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

## TEST SUR NOMBRES ET OPÉRATIONS LOGIQUES SUR LES EXPRESSIONS

Expression	vrai si :
n1 -eq n2	$n1 = n2$
n1 -ne n2	$n1 \neq n2$
n1 -le n2	$n1 \leq n2$
n1 -ge n2	$n1 \geq n2$
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

## BRANCHEMENT CONDITIONNEL : if-then-elif-else-fi

## Syntaxe

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

- 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

## 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
```

## 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
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
```



## BRANCHEMENT CONDITIONNEL : case-esac

## Syntaxe

```
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 reconnaître peut s'exprimer sous forme d'expression rationnelle (ou régulière) utilisant les méta-caractères : \* ? [ ] -

## 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
```

## 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
```

## Boucle for-do-done

## Syntaxe

```
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

## Exemple 1

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

```
for i in un deux trois quatre cinq six
do
    echo "Semestre $i"
done
```

## Exemple 2

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

```
for i in ~/Desktop/*.pdf
do
    echo $i
done
```

## BOUCLE for-do-done : EXEMPLE 3

```
#!/bin/bash
# Checkfile. Informations sur des fichiers
for file in $@
do
    if [ -d $file ]
    then # Test file type
        echo "$file est un répertoire"
    elif [ -f $file ]
    then
        echo "$file est régulier"
        if [ -s $file ]
        then
            echo "$file n'est pas vide"
        else
            echo "$file est vide"
        fi
    else
        echo "$file non trouvé"
    fi
fi
```

## EXEMPLE 3 (SUITE)

```
if [ -o $file ] # check ownership
then
    echo "Propriétaire de $file"
else
    echo "Pas propriétaire de $file"
fi
if [ -r $file ] # check permissions
then
    echo "Droit de lecture sur $file"
fi
if [ -w $file ]
then
    echo "Droit d'écriture sur $file"
fi
if [ -x $file ]
then
    echo "Droit d'exécution sur $file"
fi
done
```

## L'instruction select-do-done

## Syntaxe

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

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

## Exemple 1

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

## Exemple 2

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

## Boucle while-do-done

## Syntaxe

```
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

## 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"
```

## Exemple 2

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

## Boucle until-do-done

## Syntaxe

```
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

## Exemple 1

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

## Exemple 2

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

## FONCTIONS BOURNE Shell

- Nous pouvons rendre la programmation plus structurée en utilisant des fonctions
- La syntaxe est :

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

- Définition des fonctions Bourne shell : au début du fichier de commande
- Prend préséance sur les commandes systèmes de même nom
- Peut avoir une valeur de retour : `exit n` où `n` est une valeur numérique (`=0` : OK, `≠0` (et `<256`) : Erreur)

## Exemple 1 : FONCTIONS BOURNE Shell

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



## EXEMPLE 2 : FONCTIONS BOURNE SHELL

```
#!/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
}
```

## EXEMPLE 2 (suite)

```
# 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
```

## 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
```

## Résultat de l'exécution

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

- Le résultat n'est pas le même !
- Pensez à utiliser `"$@"` dans vos programmes *Bash*

## 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 ...]`
  - `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 : $OPTARG"
done
```

### RÉSULTAT DE L'EXÉCUTION

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

## Exemple 2 (getopts)

```
#!/bin/sh
while getopts l:q OPT
do
  case "$OPT" in
    l) OPTION="$OPTARG"
       LOGARG="$OPTARG" ; echo "OPTION $OPT reçue; son argument est $LOGARG" ;;
    q) OPTION="$OPTARG" ; echo "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
```

## DÉBOGAGE DE SCRIPT SHELL

- Pour simplifier la recherche des erreurs dans un programme *Bourne Shell* :
- Utiliser 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

```
#!/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`
```

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système Unix

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## LES FILTRES

- Programmes qui :
  - lisent une entrée
  - effectuent une transformation
  - écrivent un résultat (sur la sortie standard)
- `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"

## EXPRESSIONS RÉGULIÈRES

- Moyen algébrique pour représenter un langage régulier
- Les expressions régulières (regexp)
  - permettent de décrire une famille de chaînes de caractères
  - au moyen de métacaractères

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*



## EXPRESSION RÉGULIÈRES

- 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 *\*

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

## Exemple :

- *.\** : 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

Attention aux confusions avec les méta-caractères du Shell !

Sens différents pour méta-caractères : *\* . ?*

## Exemples

- `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
  - pascalle
  - cascade
  
- `[ps].*a.*a` : Les chaînes contenant un `'p'` ou un `'s'` suivi d'une sous chaîne contenant deux `'a'`
  - sandales
  - pascalle
  - apprentissage automatique

COMMANDE *egrep*

- Recherche dans des fichiers
  - D'une chaîne ou d'une sous chaîne de caractères
  - Simplement d'un mot
  - D'une chaîne formalisée par une expression régulière
- Utilisation : `egrep [options] <chaîne recherchée> <fichier>`
- Résultat
  - Lignes du fichier contenant ce qui est recherché
  - Ou autre résultat, suivant les options utilisées
- Options les plus utilisées
  - `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

COMMANDE *egrep* : EXEMPLE

- Imaginons le fichier de notes suivant :

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
soucpas:Soucy, Pascal:SOUN14067502:95:90:89:87:99

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

## ❶ INTRODUCTION GÉNÉRALE SUR LES SYSTÈMES D'EXPLOITATION

Rappels sur le matériel

Notions de systèmes d'exploitation

Les principaux systèmes d'exploitation

## ❷ SYSTÈME UNIX

Introduction au système UNIX

Commandes de base du *Shell*

Système de gestion de fichiers

## ❸ PROGRAMMATION *Shell*

Introduction à *bash*

Les scripts *Shell*

## ❹ FILTRE PROGRAMMABLE *awk*

Introduction

Expressions régulières et commande *egrep*

Filtre programmable *awk*

## LANGAGE AWK

- *awk* : développé par *Alfred Aho*, *Peter Weinberger* & *Brian Kernighan*
- Il s'agit d'un programme UNIX capable d'interpréter un programme utilisateur
- Le 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 "pilote par données" (*data-driven*)
- On peut utiliser le *awk* pour :
  - récupérer de l'information ;
  - générer des rapports ;
  - transformer des données...

## INVOCATION D'AWK

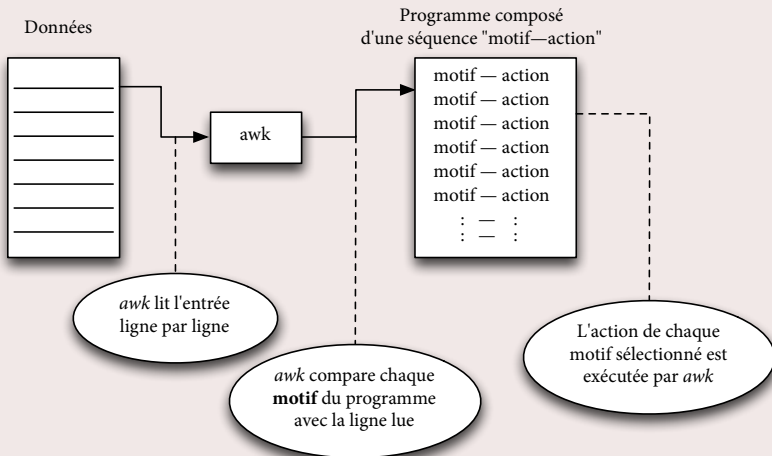
- Le synopsis de *awk* : `awk [-Fc] [-f prog | 'prog' ] [-v var=valeur...] [fich1 fich2 ...]`

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

INVOCATION d'*awk* (suite)

- Chaque ligne d'entrée est séparée en champs `$0`, `$1`, `$2`, `$3`, ...
- 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 programme *awk* proprement dit 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*

## Principe de fonctionnement





## STRUCTURE d'UN PROGRAMME AWK

### Structure :

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

### Principe :

- 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*

### Remarques :

- *motif* omis : action toujours effectuée
- *action* omise : affichage de la ligne

## VARIABLES ET STRUCTURE d'UNE ligne

- Chaque ligne ("*Record*") est automatiquement séparée en champs ("*Fields*")
- Séparateur par défaut : blancs et/ou tabulations
- **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 :** 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**

## Exemple 1

Voici un premier exemple. Le programme *awk* est spécifié directement entre apostrophes

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

## RÈGLES SUR LES MOTIFS

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.

## 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

## SYNTAXE DES MOTIFS

La syntaxe 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

## EXEMPLES 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

- Elles décrivent les opérations à effectuer lorsque le motif décrit en tête de requête est vérifié
- Syntaxe similaire à celle du langage C
- On trouvera aussi un ensemble de fonctions spécifiques

## EXEMPLE DE FONCTIONS PRÉDÉFINIES

Fonction	Description
<code>sqrt(arg)</code>	renvoie la racine carrée de l'argument
<code>log(arg)</code>	renvoie le logarithme népérien de l'argument
<code>exp(arg)</code>	renvoie l'exponentiel de l'argument
<code>int(arg)</code>	renvoie la partie entière de l'argument
<code>length</code>	renvoie la longueur de l'enregistrement courant
<code>length(arg)</code>	renvoie la longueur de la chaîne passée en argument
<code>print [arg1[,arg2],...] [&gt; dest]</code>	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
<code>printf(format,arg1,arg2,...) [&gt; dest]</code>	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

## 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 :

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

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

## Exemples

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

## 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** :

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

- Structure de contrôle **while**

```
while (condition)
    instruction
```

- Structure de contrôle **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;  
}
```

## Fichier de données "population"

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

## Exemples

## 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)}
```

## 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
- Mots majuscules = Mots minuscules
- Isolation des mots par `gsub()` revient à éliminer les caractères de ponctuation :

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

- Confondre les mots majuscules et les mots minuscules :

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

- Cette conversion est réalisée en dehors du programme *awk*
- 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 :

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

## Exemple : FRÉQUENCE DES MOTS DANS UN TEXTE (SUITE)

```
#!/bin/sh

# Programme utilisant awk pour compter le nombre d'apparitions des mots dans un texte

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

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