

# UNIX/Linux et Langages de Scripts

1

ENSAM-MEKNES

A.AHMADI

2024/2025

## I- Introduction

2

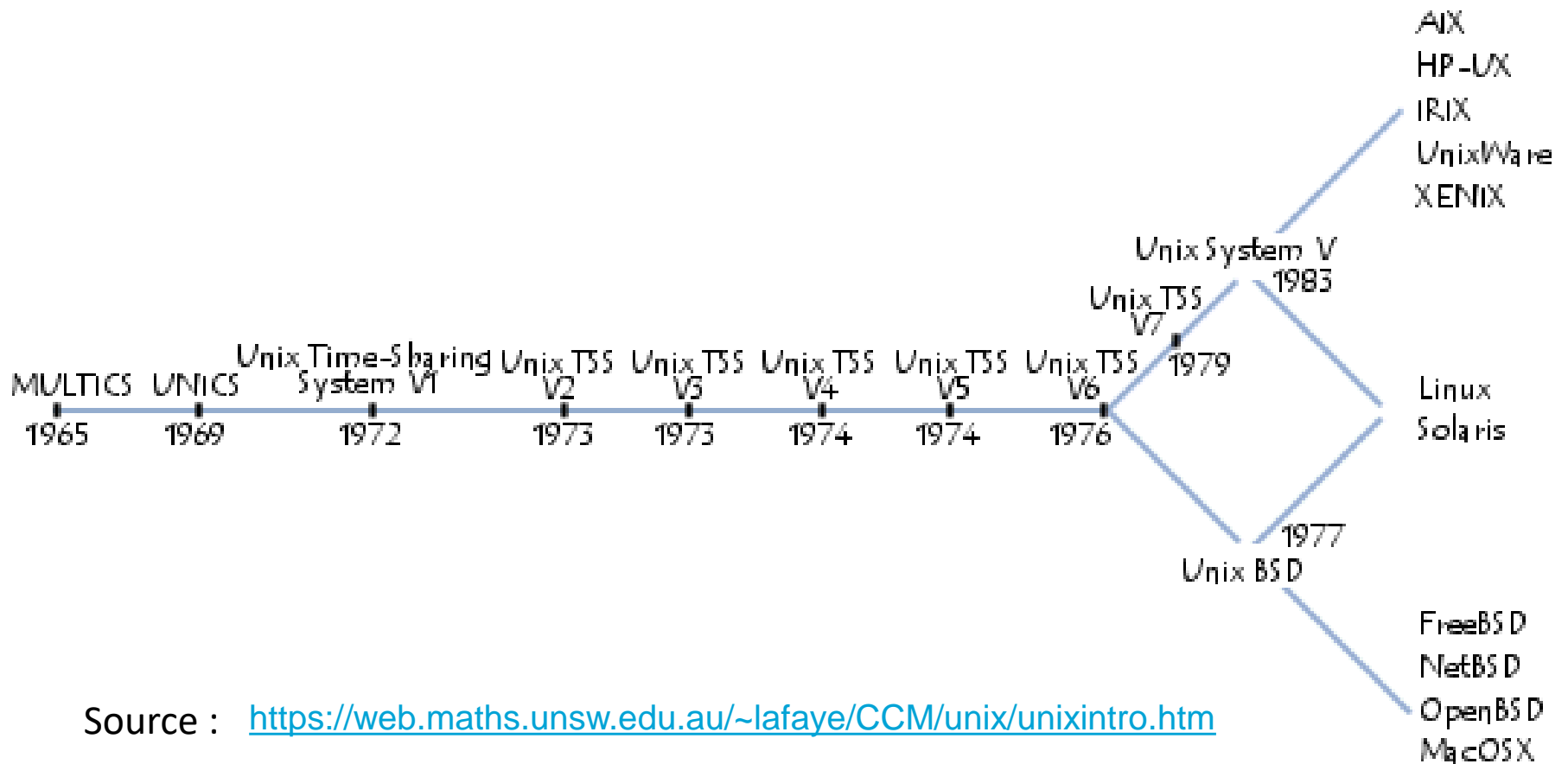
- Le système d'exploitation UNIX est issu du milieu universitaire ;
- Il s'est imposé dans le milieu industriel : tous les constructeurs informatiques le proposaient à leur catalogue, parfois en alternative à leur propre SE ;
- UNIX est le seul SE disponible sur :
  - Les micro-ordinateurs ;
  - Les stations de travail ;
  - Les mini-ordinateurs ;
  - Les mainframes ;
  - Les supercalculateurs. (En 2024, le supercalculateur le plus puissant au monde est **Frontier** avec **8 730 112** cœurs de processeurs : **1 exaFLOPS =  $10^{18}$  FLOPS** )
- Le succès d'UNIX est justifié par :
  - **Portabilité du savoir** : une même connaissance pouvait s'appliquer à de nombreux environnements différents.
  - **Portabilité des programmes** : Un même programme peut être compilé et exécuté sur des machines et environnements différents.

# Chap 1

# Présentation d'UNIX/Linux

## II- Historique

3



- **1969** : naissance d'UNIX, dans les **Bell Laboratories** (AT&T) par **Ken Thompson** et **Dennis Ritchie**.  
Objectif : fournir un environnement de développement aux programmeurs maison.
- **1973** : nouvelle version d'UNIX a été réécrite, en grande partie (90%) en **C** et en **Assembleur** (10%).
- **1974** : Développement de la **version IV** d'UNIX par **l'université Berkeley** (Californie)
  - Début d'une divergence entre les 2 versions d'UNIX : **AT&T** et **BSD**.
  - Grand succès d'UNIX dans les universités américaines : adopté par les départements « Computer Sciences » pour la formation des étudiants en "Informatique Système".
  - Le nombre d'experts UNIX croît à une grande vitesse !
- **1977-1979** : Thompson et Ritchie réécrivent UNIX pour le rendre réellement portable ;
- **1980** : 1<sup>ères</sup> licences de distribution d'UNIX d'AT&T délivrées aux constructeurs ;

- **1984** : Création du groupe **X/Open**. But : Normaliser les différentes versions.
  - Création de la norme **X-Window** (système de fenêtrage graphique) par **MIT**.
- **1987** : Alliance entre **AT&T** et **SUN** visant la convergence entre les 2 systèmes ;
- **1988** : Création des consortiums : **OSF** (Open Software Foundation) et **UNIX International** :
  - **OSF** : DEC, HP, IBM, ... → Normalisation d'un nouvel UNIX : **OSF1** ;
  - **UNIX International** : AT&T, SUN, ... → **UNIX System V** ;
- **1992** : **DEC/OSF1** : 1ère version commerciale d'OSF proposée par **DEC** ;
  - **SystemV+BSD** : 1ère version commerciale proposée par **SUN** ;

Il y a plusieurs systèmes UNIX, avec quelques petites différences. Exemple : **LINUX** ;

- Le grand succès d'UNIX est dû aux facteurs suivants :
  - Son adoption par les universités américaines → formation de plus d'experts.
  - Besoin d'un standard exprimé par les utilisateurs finaux et les développeurs : système ouvert permettant une évolution en douceur.
  - UNIX est le seul OS multiutilisateurs disponible à faible coût.
- Il existe de nombreuses versions dérivées d'UNIX qui peuvent être classées en 2 groupes :
  - Les **UNIX Based** : dérivées des sources AT&T et/ou Berkeley, comme :  
Ultrix (DEC), HP-UX (HP), AIX (IBM), SOLARIS (SUN), LINUX.
  - Les **UNIX Like** : Systèmes UNIX mais contenant un noyau totalement réécrit visant des applications bien précises (temps réel, transactionnelles, etc.).
- Création de groupes d'utilisateurs afin d'exprimer une indépendance vis-à-vis des constructeurs :
  - Le groupe **X/Open** (1984) : Européen au début et international plus tard ;
  - Le groupe **POSIX** : fait partie de IEEE. Ses travaux sont plus déterminants.

## III- Notion de Système d'Exploitation

7

### 1- Définition

- Un système d'exploitation (SE=OS) est un logiciel qui agit comme une interface entre les utilisateurs et le matériel d'un ordinateur. Il gère les ressources matérielles et logicielles de l'ordinateur, en assurant la coordination entre différents programmes et les périphériques physiques comme la mémoire, les processeurs et les unités de stockage([Wikipedia](#)).
- C'est un programme essentiel qui contrôle et coordonne le fonctionnement de l'ordinateur, y compris le traitement des données, l'exécution des programmes et la communication avec d'autres dispositifs. Il fournit l'environnement dans lequel d'autres logiciels peuvent fonctionner efficacement et en toute sécurité([Wikipedia](#)).

## III- Notion de Système d'Exploitation

8

### 2- Rôle d'un Système d'Exploitation

- C'est le maître d'orchestre : indispensable pour utiliser les ressources de la machine.
- Ses principales fonctions sont :
  - **Gestion des processus** : Création, planification et terminaison des processus.
  - **Gestion de la mémoire** : Allocation et libération de mémoire pour les processus, gestion de la mémoire virtuelle.
  - **Gestion des fichiers** : Organisation, stockage, récupération et protection des données sur les supports de stockage.
  - **Gestion des périphériques** : Communication avec le matériel, abstraction des détails matériels via des pilotes.
  - **Gestion des utilisateurs** : Authentification, autorisations et gestion des comptes d'utilisateurs.
  - **Interface utilisateur** : Fourniture d'interfaces, que ce soit en ligne de commande ou graphiques, pour interagir avec le système.
  - **Sécurité** : Protection des données et des ressources contre les accès non autorisés.
  - **Réseau** : Gestion des communications entre machines et accès aux ressources réseau.



## IV- Le Système d'Exploitation UNIX

9

### 1- Caractéristiques d'UNIX/Linux

UNIX est un système d'exploitation :

- **Multitâche** : Exécution simultanée de plusieurs processus.
- **Multi-utilisateur** : Plusieurs utilisateurs peuvent accéder au système en même temps, chacun avec ses propres permissions.
- **Portabilité** : Conçu pour fonctionner sur différentes architectures matérielles.
- **Système de fichiers hiérarchique** : Structure organisée en arborescence, facilitant l'organisation et la gestion des fichiers.
- **Sécurité et permissions** : Utilisation de permissions pour contrôler l'accès aux fichiers et aux ressources.
- **Utilisation de la ligne de commande** : Interface en ligne de commande puissante pour l'administration et l'automatisation des tâches.
- **Modularité** : Composants du système (noyau, utilitaires) sont souvent séparés, permettant des mises à jour et des modifications faciles.
- **Scriptabilité** : Support de scripts pour automatiser des tâches répétitives.
- **Open Source (pour Linux)** : La plupart des distributions Linux sont open source, ce qui permet aux utilisateurs de modifier et de distribuer le code source.

## IV- Le Système d'Exploitation UNIX

10

### 2- Composants d'UNIX/Linux

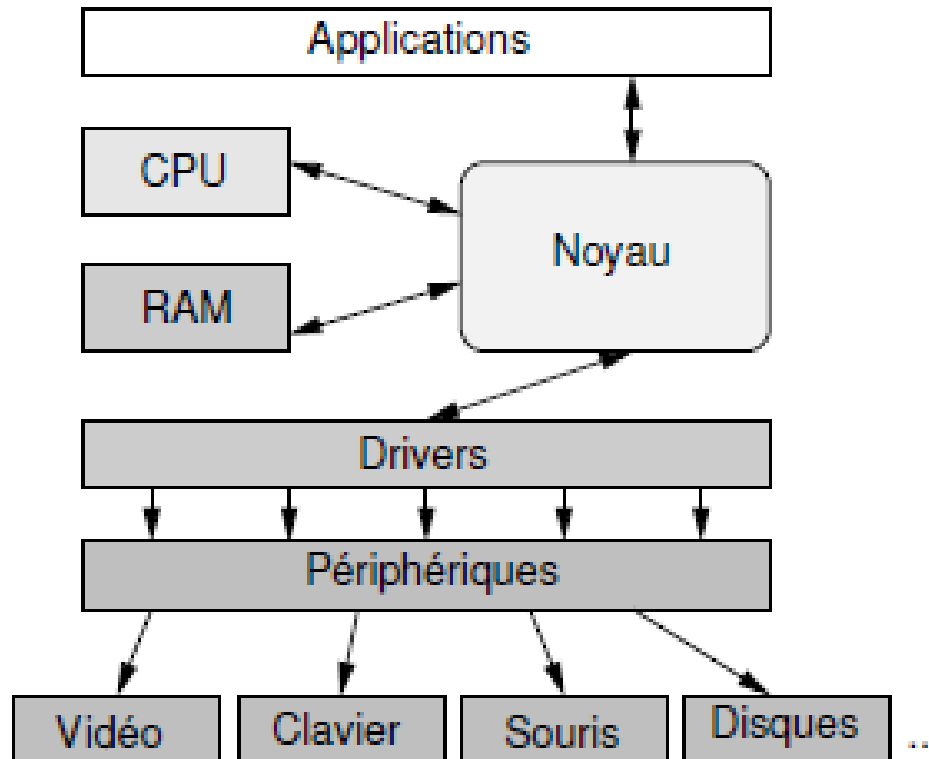
Principaux composants d'Unix/Linux : **Noyau**, **Interpréteur de commandes** et **Utilitaires**.

#### a- Noyau

- Il assure la gestion des ressources physiques (processeur, mémoire, périphériques) et logicielles (processus, fichiers, etc) ;
- Il accepte et traite les instructions du shell ;
- Il gère les permissions d'accès dans le système de fichier ;
- Il est constitué d'un ensemble de procédures et de fonctions écrites en C et en Assembleur (entre **5** et **20 Mo**) ;
- La structure du noyau est monolithique (pas de couches comme les autres SE).

### 2- Composants d'UNIX

#### a- Noyau



Source : [https://linux.developpez.com/tutoriels/apprendre-unix-aller-plus-loin-ligne-commande/?page=unix\\_logiciels\\_libres](https://linux.developpez.com/tutoriels/apprendre-unix-aller-plus-loin-ligne-commande/?page=unix_logiciels_libres)

## IV- Le Système d'Exploitation UNIX

12

### 2- Composants d'UNIX/Linux

#### b- Interpréteur de commandes (Shell)

- L'utilisateur d'Unix communique indirectement avec le noyau via le **Shell**.
- **Interface utilisateur** : Le shell peut être utilisé en ligne de commande (CLI). Il existe également des interfaces graphiques qui peuvent masquer le shell.
- **Types de shell** : chacun ayant ses propres fonctionnalités : **Bash** (Bourne Again Shell) : Le shell par défaut pour de nombreuses distributions Linux. **sh** (Bourne Shell) : Un des premiers shells, souvent utilisé pour des scripts. **csh** (C Shell) : Connu pour sa syntaxe inspirée du langage C. **zsh** : Un shell interactif avancé, avec de nombreuses fonctionnalités améliorées.
- **Exécution de commandes** : Les utilisateurs peuvent exécuter des commandes individuelles, lancer des programmes, et gérer des fichiers directement à partir du shell.
- **Scripting** : Le shell permet d'écrire des scripts, des fichiers texte contenant une série de commandes. Ces scripts peuvent automatiser des tâches répétitives.
- **Gestion des processus** : Le shell offre des commandes pour gérer les processus, comme fg, bg, jobs, et kill.
- **Redirection et pipes** : Les utilisateurs peuvent rediriger l'entrée et la sortie des commandes, et utiliser des pipes pour combiner plusieurs commandes.
- **Personnalisation** : Les shells peuvent être configurés et personnalisés à l'aide de fichiers de configuration (par exemple, .bashrc pour Bash), permettant aux utilisateurs de définir des alias, des fonctions, et des variables d'environnement.

## IV- Le Système d'Exploitation UNIX

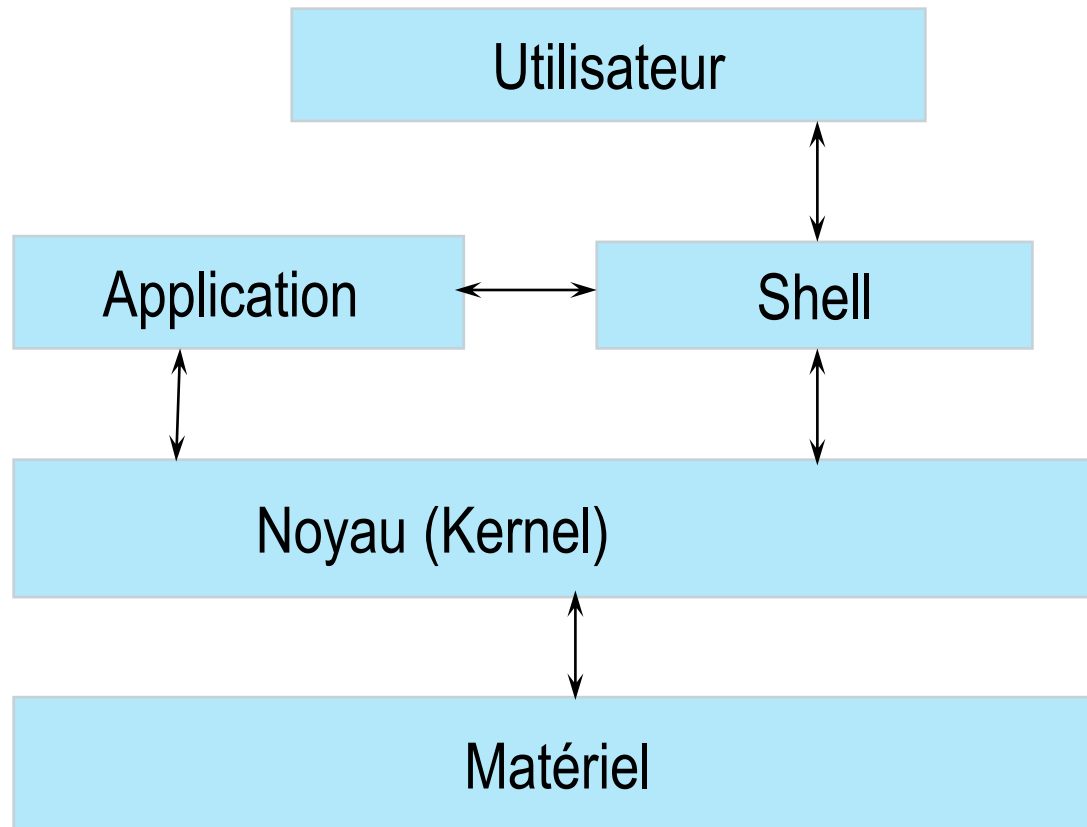
13

### 2- Composants d'UNIX/Linux

#### c- Utilitaires

- **Assembleurs** et **éditeurs de liens** (as, ld, etc.).
- **Compilateurs** pour différents langages pour C, Pascal, Ada, Fortran, etc. (gcc, g++, gfortran, gpc, etc.)
- **Outils de bureautique** :
  - messagerie (mutt, thunderbird) ;
  - traitement de textes (thunderbird, AbiWord, etc.).
- **Editeurs de texte** (ed, vi, vim, emacs, gedit, etc.) ;
- **Outils pour le Web** :
  - Navigateurs (Lynx, FireFox, Chrome) ;
  - Serveurs Web (Apache, Nginx).
- **Etc.**

### 3- Structures d'UNIX



## V- Le concept "Open Source"

15

- **Richard Stallman** (chercheur au MIT, auteur de gcc, Emacs ...) énonce clairement le concept de **logiciel libre** (free software) :



ENSAM-Meknès-2016

- " ... **Un savoir scientifique doit être partagé en le distribuant, ... les codes sources doivent être libres d'accès ...** "
- Il démarre le projet **GNU** (1984). But : recréer un système d'exploitation complet (Unix-like), composé uniquement de logiciels libres.
- Il crée la **FSF** (Free Software Foundation, 1985) pour gérer le projet GNU.

### Remarque :

"**Free**" dans la culture hacker signifie "**libre**", pas nécessairement "**gratuit**" ou "**non commercial**".

## V- Le concept "Open Source"

16

### 1- Principe de base du projet GNU

- L'accès libre au code source accélère le progrès en matière d'informatique car l'innovation dépend de la diffusion du code source.
- Richard Stallman décrit dans le Manifeste GNU les quatre libertés fondamentales que doit respecter un logiciel pour être qualifié de logiciel libre (free software en anglais) :
  - ① liberté d'exécution : tout le monde a le droit de lancer le programme, quel qu'en soit le but ;
  - ② 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 ;
  - ③ liberté de redistribution : tout le monde a le droit de rediffuser le programme, gratuitement ou non ;
  - ④ liberté d'amélioration : tout le monde a le droit de redistribuer une version modifiée du programme.

**Remarque** : La 2ème liberté se concentre sur la modification pour un usage personnel, tandis que la 4ème liberté concerne la redistribution des modifications pour qu'elles puissent profiter à d'autres personnes.



## V- Le concept "Open Source"

17

### 2- La licence GPL (General Public licence)

- Autorise l'utilisateur à copier et à distribuer à volonté le logiciel qu'elle protège, pourvu qu'il n'interdise pas à ses pairs de le faire aussi ;
- Requier aussi que tout dérivé d'un travail placé sous sa protection soit lui aussi protégé par elle ;
- Quand la GPL évoque les logiciels libres, elle traite de liberté et non de gratuité (un logiciel GPL peut être vendu) ;

**Exemple** : **Red Hat Enterprise Linux** est un système d'exploitation libre, mais sa distribution officielle est vendue avec des services de support. Toutefois, les "codes sources" de Red Hat étant libres, d'autres projets comme **CentOS** en redistribuent des versions gratuites.

**Conclusion** : un logiciel libre peut être vendu, mais ce qui compte, c'est que les utilisateurs disposent toujours des 4 libertés fondamentales qui définissent le logiciel libre.

## V- Le concept "Open Source"

18

### 3- Le copyleft de la licence GPL

Créé par Stallman en 1984 :

- Garantit les 4 libertés fondamentales pour tous les utilisateurs (artistes, informaticiens, ou quiconque produisant un travail soumis au droit d'auteur) ;
- Evite de mettre les logiciels GNU dans le domaine public (pas de protection) ;
- Spécifie que quiconque redistribue le logiciel, avec ou sans modifications, doit aussi transmettre la liberté de les copier et de les modifier ;
- Encourage et aide les programmeurs (entreprises, universités) qui veulent ajouter et/ou contribuer à des améliorations des logiciels libres ;
- Un logiciel **copyleft** est d'abord déclaré sous copyright, puis on ajoute les conditions de distribution et les libertés légalement indissociables.

**Conclusion** : Un logiciel sous copyleft est libre, et on peut l'utiliser, le modifier et le redistribuer. Cependant, si on redistribue une version modifiée du logiciel, on doit le faire sous les mêmes conditions de copyleft. Cela signifie que les versions modifiées doivent également être libres et offrir les mêmes libertés que l'original.



- 1991 : **Linus Torvalds** (ingénieur Finlandais) développe un noyau s'inspirant d'Unix : Linux. Il le met très vite sous licence GPL, rejoint par de nombreux développeurs.
- Succès : qualité technique du noyau + nombreuses distributions qui facilitent l'installation du système et des programmes.
- Le **noyau Linux** est un projet **open source**, donc des milliers de développeurs du monde entier contribuent régulièrement en soumettant des correctifs, de nouvelles fonctionnalités, et des améliorations.
- Les contributeurs proviennent de diverses organisations (entreprises technologiques comme **Intel**, **IBM**, **Google**, **Red Hat**, etc.) ainsi que des **individus indépendants**.
- Tous les changements passent par des processus rigoureux de révision de code.

### Distributions Linux :

- Qu'est-ce qu'une Distribution Linux ?
  - un noyau Linux
  - des programmes, en majorité libres (un navigateur Web, un lecteur de Mail, un serveur FTP, etc...)
  - une méthode pour installer et désinstaller facilement ces programmes
  - un programme d'installation du système d'exploitation
- Principales distributions :
  - **Ubuntu** : Orientation utilisateur, très populaire pour les débutants et professionnels.
  - **Linux Mint** : Basée sur Ubuntu, attire également beaucoup d'utilisateurs pour sa facilité d'utilisation
  - **Debian** : Stabilité et sécurité, base de plusieurs autres distributions (comme Ubuntu).
  - **Fedora** : Distribution sponsorisée par Red Hat, avec des logiciels récents.
  - **Arch Linux** : Très personnalisable, pour utilisateurs avancés.
  - **Red Hat Enterprise Linux** (RHEL) : Utilisé dans les entreprises, avec un support payant.
- Environnement de bureau : **GNOME, KDE, Xfce, Cinnamon, Pantheon, ...** selon les préférences utilisateur.

### Résumé :

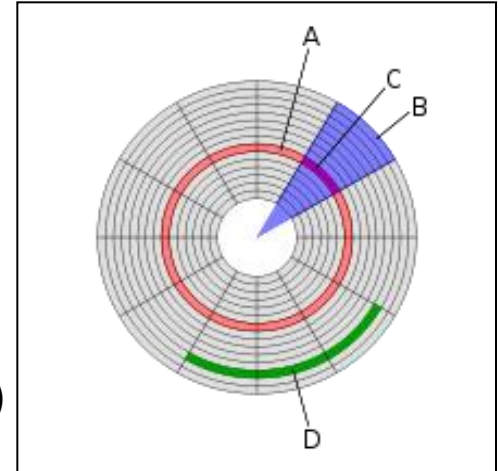
- **Unix** a été pionnier dans la création de systèmes d'exploitation multi-utilisateurs et portables.
- **Linux** a pris le relais avec une grande flexibilité et un modèle **open source**.
- Choix d'une distribution dépend de l'usage (serveur, desktop, entreprise, personnalisation).
- Aujourd'hui, Linux est utilisé dans les serveurs, les ordinateurs personnels, les systèmes embarqués, et plus encore.
- En **2023**, environ **70 à 90 %** des serveurs dans le monde fonctionnent sous Linux :
  - **Serveurs web** (80%).
  - **Supercalculateurs** (100%).
  - **Infrastructure cloud** : grands acteurs du cloud: **Amazon Web Services (AWS)**, **Google Cloud**, et **Microsoft Azure**, utilisent massivement Linux dans leurs infrastructures.

## VII- Installation de Linux

22

### 1- Partitionnement

- Disquette de **1,41 Mo** : **18** secteurs de **512** octets sur chacune des **160** pistes (**80** sur chaque face) du périphérique. Cela offre **1 474 560** ( $2 \times 80 \times 18 \times 512$ ) octets de stockage.
- Disque dur **SATA** de **4 To** (teraoctets), utilisé fréquemment pour le stockage dans les ordinateurs et les serveurs récents :
  - Capacité totale : **4 To** ( $\approx 4\,000$  Go, soit environ 4 000 milliards d'octets)
  - Taille de **secteur** : **4096 octets** (ou 4 Ko, secteurs avancés utilisés dans les disques modernes).
  - Taille d'un **cluster** : **4 Ko** (peut varier selon le système de fichiers, mais 4 Ko est courant pour NTFS).
  - Nombre de plateaux : **4** (par exemple, les disques de grande capacité utilisent plusieurs plateaux).
  - Nombre de surfaces (faces) : **8** (chaque plateau a **2** faces).
  - Nombre de **pistes** par surface : **100 000** (un disque moderne peut avoir beaucoup plus).
  - Nombre de secteurs par piste : Variable en fonction du format du disque, mais on peut utiliser une estimation de **2000** secteurs par piste dans les pistes extérieures.



Structure d'un disque magnétique:  
A : **piste**  
B : **secteur**  
C : **secteur d'une piste**  
D : **cluster** de secteurs

## VII- Installation de Linux

23

### 1- Partitionnement

- La plupart des systèmes d'exploitation (fixes) « correctement » installés utilisent un disque à plusieurs partitions :
  - partition **Système** (fichiers systèmes, fichiers de configuration ...)
  - partition **Utilisateurs** (données des utilisateurs)
  - etc.
- Le partitionnement permet une exploitation plus sécurisée ;
- On peut formater une partition indépendamment des autres ;
- On peut utiliser une partition en lecture seule ;
- Partitionnement **statique** => planifier le partitionnement ;
- On ne peut pas modifier simplement un partitionnement statique ;
- Partitionner est une opération "**low level**" risquée !!
- Pour bénéficier des avantages du partitionnement **dynamique** il faut passer à des solutions de type RAID (Redundant Array of Independent Disks) ou LVM (Logical Volume Manager). Pour Windows, on peut utiliser la "**Gestion des disques**".

## VII- Installation de Linux

24

### 2- Schémas de partitionnement (MBR/GPT)

- Schéma de partitionnement : permet d'organiser les données sur un disque dur, un SSD ou tout autre périphérique de stockage.
- Il définit comment les fichiers sont structurés et où se trouvent les partitions (zones du disque qui contiennent les données).
- Il définit également le processus de démarrage en indiquant au système d'exploitation où trouver les données nécessaires pour initialiser le matériel.
- Les 2 principaux types de schémas utilisés aujourd'hui sont MBR et GPT :
  - MBR (Master Boot Record) est un ancien schéma datant des années 1980.
  - GPT (GUID Partition Table) est une technologie plus moderne, introduite avec l'UEFI. GUID (Globally Unique Identifier).
- **MBR** (Master Boot Record) : utilisé historiquement sur les disques durs depuis l'époque de MS-DOS. Il se trouve dans le premier secteur du disque dur (512 octets). Sa structure est :
  - **Code d'amorçage** (bootloader) : Petit programme qui démarre le système d'exploitation.
  - **Table de partitions** : Informations sur les partitions présentes sur le disque (jusqu'à 4 partitions principales).
  - **Signature du disque** : Identifiant unique du disque.

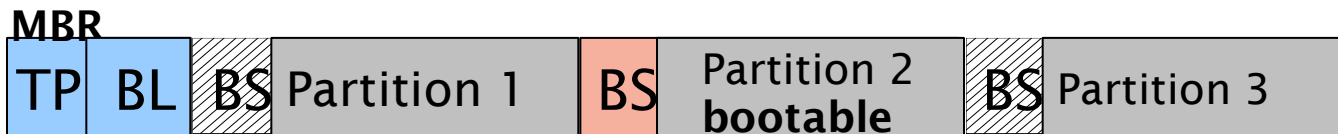
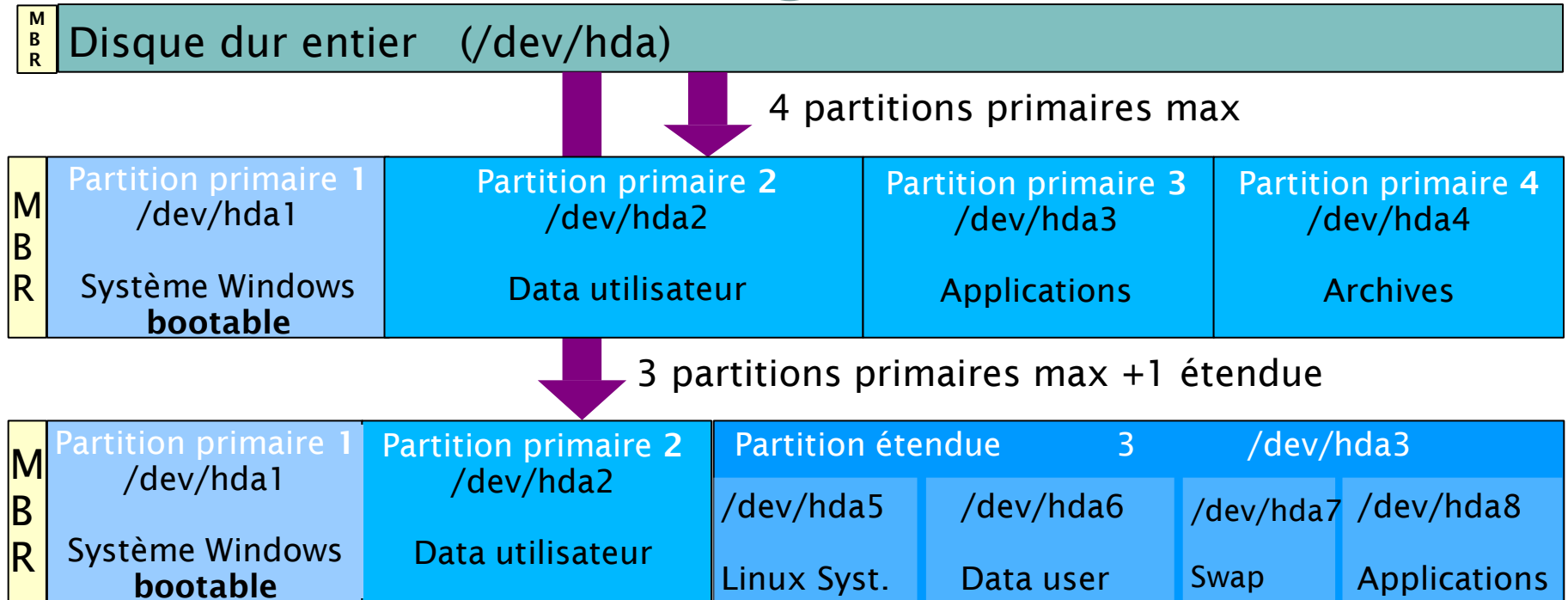


### 2- Schémas de partitionnement (MBR/GPT)

- MBR utilise un adressage sur **32** bits, limitant la taille maximale des disques à **2,2 To (2<sup>32</sup> secteurs)**
- MBR ne prend en charge que **4 partitions principales**. Pour en avoir plus, une partition **étendue** doit être utilisée, à l'intérieur de laquelle des partitions logiques peuvent être créées.
- Si le MBR est corrompu (en raison d'un logiciel malveillant ou d'un dysfonctionnement matériel), --> tout le disque devient non amorçable. Il n'y a pas de redondance ni de mécanisme de sauvegarde dans MBR, ce qui le rend moins fiable.
- MBR est bien adapté pour les systèmes plus anciens, mais il est insuffisant pour les nouveaux systèmes nécessitant des disques plus grands.

## VII- Installation de Linux

26



MBR : Master Boot Record  
 TP : Table Partition  
 BL : Boot Loader  
 BS : Boot Sector

**Remaque :** Windows et Linux :

C: <=> /dev/hda1  
 D: <=> /dev/hda2  
 ...

### 2- Schémas de partitionnement (MBR/GPT)

- **GPT** : Développé dans les années 2000 avec la norme UEFI. Il a une structure qui :
  - Utilise une table de partitions plus avancée basée sur GUID (Globally Unique Identifier).
  - Contient une table de partitions principale et une sauvegarde (redondance) pour plus de sécurité.
- GPT peut gérer des disques jusqu'à **9,4 zettaoctets** (1 **Zo** =1 073 741 824 **To**).
- GPT peut créer jusqu'à **128 partitions** sans besoin de partition étendue.
- GPT enregistre plusieurs copies de la table de partition (au début et à la fin du disque), ce qui permet de récupérer plus facilement des erreurs ou des corruptions.
- GPT utilise des checksums CRC32 pour s'assurer que la table de partition n'est pas corrompue. Cela renforce la sécurité et l'intégrité des données sur le disque.
- GPT est conçu pour fonctionner avec les systèmes modernes utilisant le firmware UEFI, remplaçant ainsi le BIOS traditionnel.

### 3- Système de fichiers

Fonctions d'un système de fichiers :

- Indique la manière de stocker les données et de les organiser dans des fichiers sur les mémoires secondaires (Disque dur, CD-ROM, clé USB, etc.).
- Il offre à l'utilisateur une vue abstraite sur ses données et lui permet de les localiser en utilisant un chemin d'accès.
- Permet de créer/détruire des fichiers, d'y insérer, modifier ou supprimer des données.
- partage des données entre plusieurs programmes.
- Empêche des accès non autorisés aux fichiers.
- Peut offrir la possibilité de chiffrer (crypter) et de compresser les données.
- Conserve les fichiers en cas de panne matérielle ou logicielle.

## VII- Installation de Linux

29

### 3- Système de fichiers

Système de fichiers	Systèmes d'exploitation	Caractéristiques principales
NTFS	Windows	Journalisation, permissions avancées
FAT32	Windows, macOS, Linux	Compatibilité élevée, limite de 4 Go par fichier
exFAT	Windows, macOS, Linux	Pas de limite de 4 Go, compatible avec les dispositifs amovibles
APFS	macOS, iOS	Optimisé pour SSD, gestion de l'espace
HFS+	macOS	Journalisation, utilisé avant APFS
EXT4	Linux	Journalisation, support de fichiers très volumineux
Btrfs	Linux	Instantanés, compression, déduplication
ZFS	FreeBSD, Linux, Solaris	Intégrité des données, gestion avancée des volumes
XFS	Linux	Optimisé pour les gros fichiers, redimensionnement en ligne

## VII- Installation de Linux

30

### 4- Installation de Linux en Dual Boot

- a. Télécharger une distribution Linux :
  - Visiter le site officiel de la distribution choisie (par exemple, Ubuntu, Debian).
  - Télécharger l'image **ISO** (fichier d'installation).
- b. Créer une clé **USB bootable** :
  - Utiliser des outils comme **Rufus** (Windows) ou **Etcher** (multi-plateforme) pour créer une clé USB bootable.
  - Choisir l'image ISO téléchargée, puis sélectionner la clé USB.
- c. Sauvegarder les données :
  - Faire une sauvegarde des fichiers importants sous Windows au cas où quelque chose tournerait mal.
- d. Libérer de l'espace disque :
  - Sous Windows, ouvrez le "**Gestionnaire de disques**" et réduisez la taille de la partition Windows pour libérer de l'espace pour Linux.

## VII- Installation de Linux

31

### 4- Installation de Linux en Dual Boot

- e. Redémarrer l'ordinateur et accéder au **BIOS** (généralement en appuyant sur une touche comme **F2**, **F12** ou **DEL** au démarrage. Cela dépend de la machine).
  - Configurez l'ordinateur pour démarrer à partir de la clé USB.
- f. Une fois démarré sur la clé USB, un menu d'installation Linux s'affichera. Choisir "Installer Linux".
- g. Choisir l'option d'installation :
  - L'installateur proposera d'"Installer Linux à côté de Windows" (Dual Boot). Cette option configurera automatiquement le partitionnement nécessaire.
- h. Configurer les partitions :
  - L'installation peut se faire automatiquement, ou choisir le partitionnement manuel pour personnaliser l'espace alloué à Linux.
  - Au minimum, on aura besoin de 2 partitions :
    - Partition **racine** (/) : pour le système Linux (ext4 recommandé).
    - Partition **swap** : pour l'espace de swap (optionnel mais recommandé, généralement 1 ou 2 x la taille de la RAM).

### 4- Installation de Linux en Dual Boot

- i. Configurer les paramètres :
  - Choisir le fuseau horaire, la disposition de clavier, et créer un utilisateur et mot de passe.
- j. Installation du chargeur de démarrage (GRUB) :
  - Le chargeur de démarrage GRUB sera installé automatiquement. Il permettra de choisir entre Windows et Linux au démarrage de la machine.
- k. Une fois l'installation terminée, retirer la clé USB et redémarrer la machine.
- l. Au démarrage, GRUB s'affichera, permettant de choisir entre Linux ou Windows.



## VII- Installation de Linux

33

### 5- Installation de Linux dans une machine virtuelle

- a. Télécharger et Installer un outil de virtualisation comme **VirtualBox** (gratuit) ou **VMware Workstation Player**.
- b. Créer une nouvelle machine virtuelle :
  - Dans VirtualBox, cliquer sur "Nouvelle machine virtuelle" et donner un nom à la VM (par exemple, "**Ubuntu**").
  - Type de système : Choisir "**Linux**" et la version correspondante (par exemple, **Ubuntu 64 bits**).
- c. Configurer la machine virtuelle :
  - Mémoire RAM : Allouer au moins **2 Go** de RAM (**4 Go** ou plus recommandé si possible).
  - Disque dur virtuel : Créez un disque dur virtuel d'au moins **20 Go** pour Linux.
- d. Monter l'image ISO :
  - Sélectionnez l'image ISO de la distribution Linux déjà téléchargée et la monter dans la machine virtuelle pour démarrer l'installation.

## VII- Installation de Linux

34

### 5- Installation de Linux dans une machine virtuelle

- e. Démarrer la machine virtuelle :
  - Lancer la machine virtuelle avec l'ISO monté, et elle démarrera comme si l'on utilisait une clé USB.
  - On verra l'écran de démarrage de Linux, choisir "Installer Linux".
- f. Suivre les étapes d'installation :
  - Comme pour le dual boot, choisir la langue, le fuseau horaire, créer un utilisateur, etc.
- g. L'installateur proposera d'utiliser tout l'espace disque disponible dans la machine virtuelle. Accepter cette option, car il s'agit d'un disque virtuel.
- h. Une fois l'installation terminée, redémarrer la machine virtuelle. On a maintenant un système Linux fonctionnant dans une machine virtuelle.

#### Avantages de l'installation en machine virtuelle :

- On peut utiliser Windows et Linux en même temps sans redémarrer.
- On peut expérimenter et tester Linux sans affecter le système principal.

**Lien utile :** <https://www.youtube.com/watch?v=DhVjgl57Ino>

# Chap 2 Premier contact avec Linux

35

- Se connecter en tant que root (Administrateur) : login : **sudo -i**, puis le **mot de passe** du root (Sur Ubuntu, l'utilisateur courant doit appartenir au groupe "**sudo**" dans le fichier **/etc/group**).  
ou : **sudo commande** suivie du **mot de passe** du root
- Quitter la session du root : **<Ctrl-d>** ou **exit**
- Plusieurs terminaux peuvent être lancés simultanément : **graphiquement** ou **<Ctrl-Alt-T>**,  
Pour quitter un terminal : **exit**
- Pour rebooter l'ordinateur : **sudo shutdown -r now** (ou **sudo reboot**) (r : reboot)
- Pour arrêter l'ordinateur : **sudo shutdown -h now** (ou **sudo poweroff**) (h : halt)
- Pour arrêter la machine dans 20 minutes : **sudo shutdown +20**
- Pour annuler l'arrêt programmé : **sudo shutdown -c**
- Création d'un compte utilisateur :

## Méthode 1

- Se connecter en tant que root et taper **useradd ali** pour créer le compte "**ali**" (avec les informations par défaut).
- Taper **passwd ali** pour affecter un nouveau mot de passe au compte **ali**, sinon l'utilisateur pourra remplacer son mot de passe vide par la commande **passwd** une fois connecté.

**Remarque** : sous certains shells, la commande **adduser** permet de créer un compte en demandant ses informations de manière interactive.

# Chap 2 Premier contact avec Linux

36

## Méthode 2

- Se connecter en tant que **root**.
  - Editer le fichier **/etc/passwd** : **vi /etc/passwd** (:q! pour quitter vi) ou **nano /etc/passwd** ou **gedit /etc/passwd** (à installer : **sudo apt install gedit**)
  - Copier une ligne d'un utilisateur, la modifier et sauvegrader le fichier **/etc/passwd** ;
  - Taper **passwd ali** pour affecter un nouveau mot de passe au **compte ali**, sinon l'utilisateur pourra remplacer son mot de passe vide par la commande **passwd** une fois connecté.
- Personne (même le root), ne saurait retrouver le mot de passe en clair d'un utilisateur donné. Mais le root peut accéder et modifier le contenu du repertoire personnel de n'importe quel utilisateur.

**Remarque** : Si le mot de passe est oublié : 2 cas

- Pour un utilisateur ordinaire : 2 solutions
  - Le root peut effacer le password crypté du fichier **/etc/shadow**, puis en affecter un nouveau ;
  - Le root peut modifier le password (**passwd ali**) et le communiquer à **ali**.
- Pour le root : 2 Solutions
  - Réinitialisation via le mode de récupération (Recovery Mode).
  - Réinitialisation via la modification des paramètres GRUB.

# Chap 2 Premier contact avec Linux

37

- Suppression d'un utilisateur : seul le root pour supprimer un compte : **userdel ali**
- Configuration clavier AZERTY (sur Ubuntu) : Paramètres → Clavier → Source de saisie → + → choisir France(Azerty) et ajouter → la définir comme disposition principale en le déplaçant en haut de la liste. (On peut le faire aussi à l'aide d'une commande)
- Avoir de l'aide sur une commande : **man commande** (exple : **man ls**). (il y a aussi **help** et **info**)
- Affichage du nom de la machine, Unix, version, ... : **uname -a**
- Afficher un message : **echo message** (Exemple 1 : **echo salut tout le monde**)

Exemple 2 : **echo -n il ; echo pleut** : -n élimine le retour à ligne.

- Pour remplacer le shell dans une session : **exec nom\_shell** (Expels: **exec sh, exec csh, exec ksh, exec bash**,...). Pour créer un processus fils ayant un autre shell, on enlève **exec**. Dans ce cas, pour revenir au shell precedent parent : **<ctrl-d>** ou **exit**.
- Pour changer le shell par défaut (dans **/etc/passwd**), taper: **chsh**. Il demandera le password root puis la saisie du shell (exple: **/bin/csh**). Le nouveau shell sera appliqué à la nouvelle session ;
- Le root peut aussi éditer le fichier **/etc/passwd**, puis modifier le shell par défaut d'un user.

# Chap 2 Premier contact avec Linux

38

- Affichage du shell par défaut : **echo \$SHELL**
- Affichage du nom du terminal utilisé : **tty**
- Affichage de la liste des infos (liste de commandes de contrôle) : **stty -a**
- Modification d'une commande : tty commande combinaison (exple : **stty erase ^H**)  
→ effacement du dernier caractère par **<ctrl-H>**
- Affichage du nom de login d'une session : **logname**
- Affichage du nom d'utilistateur actuel du terminal : **whoami**  
(login : toto .... su ali ...., **logname** → toto et **whoami** → ali )
- Effacement de l'écran : **clear**
- Affichage des jours du mois actuel (calendrier) : **cal** (si la commande n'est pas disponible : **sudo apt update** puis **sudo apt install util-linux**)
- Affichage du calendrier complet d'une année donnée : **cal annee** (exple : **cal 2024**)
- Affichage des jours du mois d'une année : **cal mois annee** (exple : **cal 3 2015**)
- Affichage la date d'aujourd'hui et de l'heure : **date**

# Chap 2 Premier contact avec Linux

39

- Agrémenter le message de retour de la date :

`date '+Le %d/%m/%y à %H:%M:%S'` → 19/10/09 a 10:15:47

`date '+Le %d/%m/%y%à %H heures %M minutes %S secondes'` →

Le 19/10/09

à 17 heures 45 minutes 17 secondes

- Pour éviter de taper à chaque fois cette longue commande, on peut définir des alias :

Exemple: `alias madate="date '+Le %d/%m/%y%à %H heures %M minutes %S secondes' "`

Si l'on tape la commande `madate` Linux affichera par exemple :

Le 30/09/24 a 10heures 45minutes 17secondes

- Réveil : `leave hhmm` ou `leave +hhmm` ou `leave +mm` (installer `leave`)

`leave 1432` → réveil émet des appels sonores à 14h32).

`leave +0215` → réveil dans 2h et 15 mn.

`leave + 13` → réveil dans 13 mn.

Un triple signal sonore : **5mn** avant l'instant prévu, **1mn** avant et à **l'instant prévu**.

## I- Système de fichiers

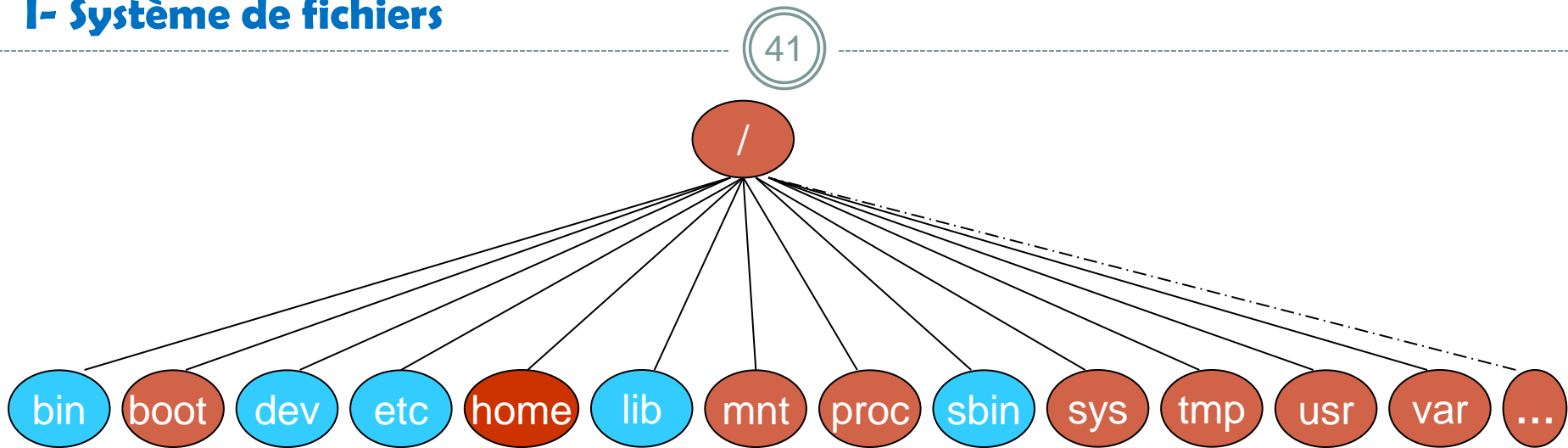
40

- Système de fichiers : c'est une méthode d'organisation et de stockage des données sur un périphérique de stockage (disque dur, SSD, etc.).
- Rôles :
  - Stockage et organisation des fichiers et des répertoires.
  - Gestion des accès aux fichiers (lecture, écriture, exécution).
  - Maintien de l'intégrité des données.
- Comme la majorité des systèmes d'exploitation, les données sous UNIX/Linux sont stockées dans des fichiers. Ces fichiers sont organisés hiérarchiquement en répertoires ;
- On peut imaginer le système de fichiers UNIX comme un **arbre** (renversé). Les répertoires sont représentés par les **branches**, et les fichiers par les **feuilles**;
- Le sommet de l'arbre s'appelle le répertoire racine (root) ( "/" )
- Chaque fichier (ou répertoire) dans l'arbre est nommé en listant toutes les branches qui ramènent vers la racine, et en les séparant par des "/", comme suit:

***`/home/elevel/documents/rapports/tp.txt`***



## I- Système de fichiers



- **Root directory ( / )** : C'est le haut de l'arborescence. Il n'y a qu'une et une seule entrée sur le *file system*. Le root directory est le seul répertoire sans père. Tous les fichiers et chemins d'accès absolus ont le root directory dans le chemin d'accès.
- **/bin** et **/sbin** : contiennent les commandes de base Unix/Linux (ls, cp, rm, mv, ln, etc.) utilisées entre autres lors du démarrage du système. Les fichiers contenus dans ces répertoires ne sont que des exécutables.
- **/boot** : Contient les fichiers nécessaires au démarrage du système, comme le noyau Linux (vmlinuz), les fichiers d'initialisation du chargeur de démarrage (ex : GRUB).

## I- Système de fichiers

42

- **/dev** : contient les fichiers spéciaux permettant de communiquer avec tous les périphériques comme les disques, les terminaux, les dérouleurs de bandes, les imprimantes, etc.
  - **/dev/sda, /dev/sdb** : Représentent les disques durs et les périphériques de stockage.
  - **/dev/tty** : Représente les terminaux.
  - **/dev/null** : Périphérique spécial qui "avale" toute donnée qu'on lui envoie (utilisé pour rediriger des sorties inutiles).
- **/etc** : contient tous les fichiers de configuration et d'administration et un certain nombre de commandes système.
- **/home** : contient les répertoires personnels des utilisateurs du système.
- **/lib** : Contient les bibliothèques partagées (librairies) essentielles utilisées par les programmes du système. Ce répertoire est lié aux fichiers exécutables de /bin et /sbin.
- **/mnt** et **/media** : **/mnt** : Utilisé pour monter temporairement des systèmes de fichiers, comme des disques externes ou des partitions. **/media** : Points de montage pour les périphériques amovibles comme les clés USB, les disques durs externes, et les CD/DVD.
- **/root** : c'est le répertoire personnel de l'administrateur.

## I- Système de fichiers

43

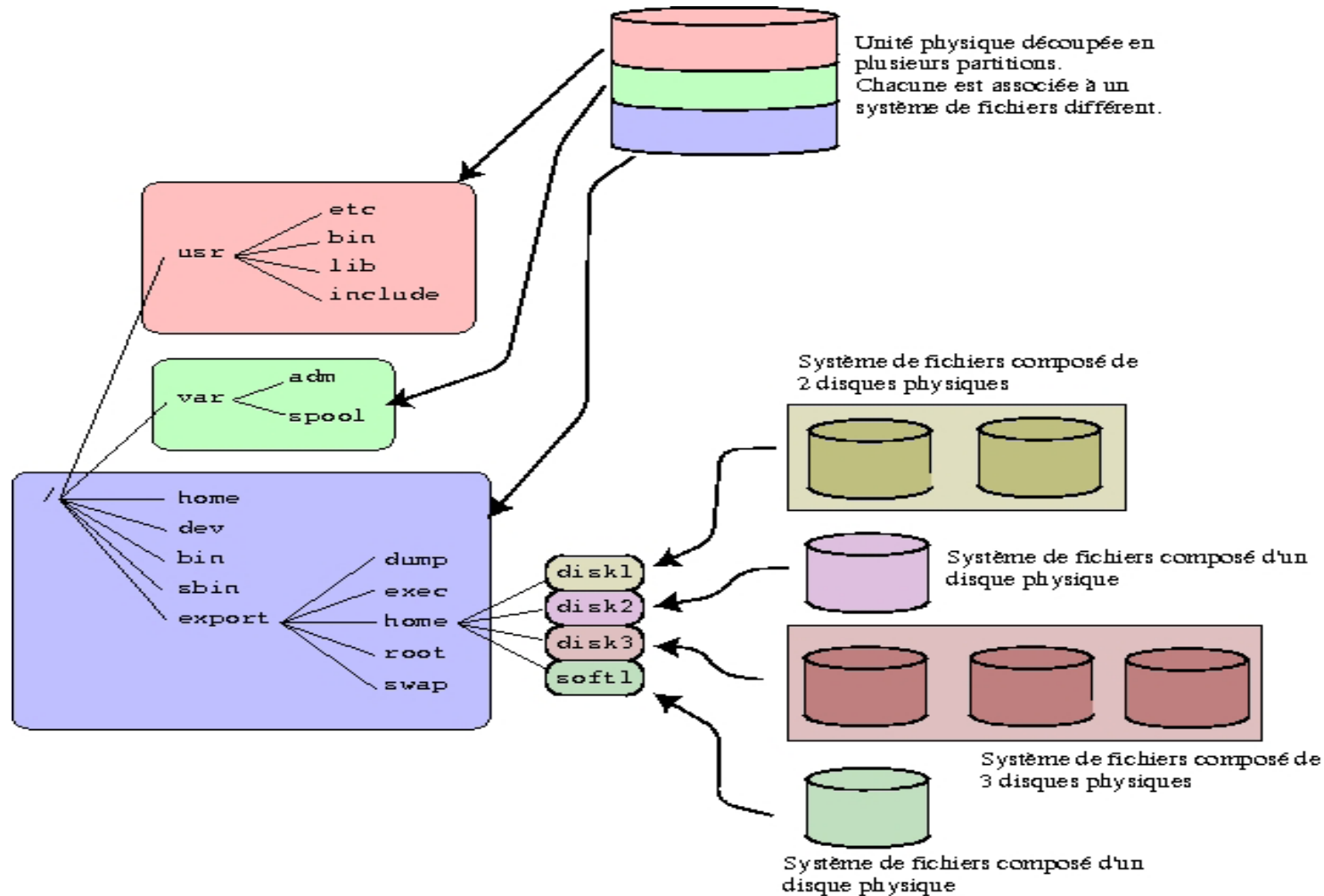
- **/usr** : (Unix System Resources) contient la plupart des programmes, utilitaires et bibliothèques utilisées par les utilisateurs.
  - **/usr/bin** : Contient des binaires supplémentaires pour les utilisateurs, par exemple des programmes non essentiels mais utiles, comme vim, nano, ou gcc.
  - **/usr/lib** : Contient les bibliothèques partagées utilisées par les programmes dans /usr/bin.
  - **/usr/share** : Contient les fichiers partagés entre les programmes, comme les fichiers de documentation ou les icônes.
- **/var** : contient les fichiers variables (susceptibles d'être modifiés fréquemment) : journaux (logs), e-mails, bases de données, archives, . . .
- **/tmp** : utilisé pour stocker des fichiers temporaires créés par les applications ou les utilisateurs. Son contenu est souvent vidé à chaque redémarrage du système.
- **/run** : Répertoire où sont stockées des informations temporaires sur le système et les processus en cours depuis le démarrage. Contrairement à /tmp, il est utilisé pour des informations plus spécifiques, comme les PID des processus ou les fichiers de verrouillage.

# Chap 3

# Système de fichiers

## I- Système de fichiers

44

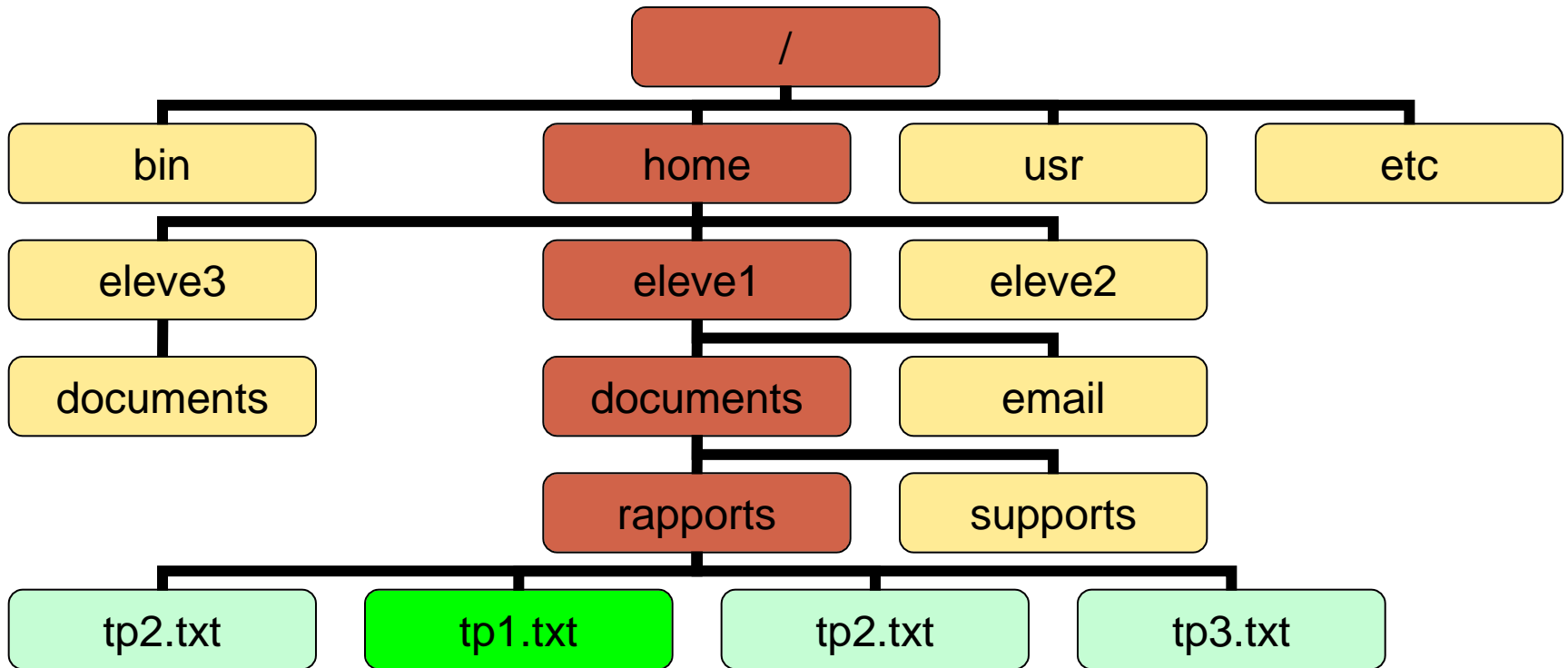


# Chap 3

# Système de fichiers

## I- Système de fichiers

45



## I- Système de fichiers

46

- Organisation interne du système de fichiers : autour de concepts tels que les **inodes**, les **blocs** et les **superblocs**.
- **Inode** : c'est une structure de données qui contient les métadonnées d'un fichier ou d'un répertoire, telles que :
  - **Numéro d'inode** : Identifiant unique pour chaque inode dans un système de fichiers.
  - **Type de fichier** : Indique si l'inode représente un fichier régulier, un répertoire, un lien symbolique, un fichier spécial, etc.
  - **Droits d'accès** : Permissions associées au fichier (lecture, écriture, exécution) pour le propriétaire, le groupe et les autres utilisateurs.
  - **Propriétaire** : Identifiant de l'utilisateur (UID) qui possède le fichier.
  - **Groupe** : Identifiant du groupe (GID) auquel le fichier appartient.
  - **Taille du fichier** : Taille du fichier en octets.
  - **Horodatages** : ctime : Date et heure de la dernière modification des métadonnées de l'inode. mtime : Date et heure de la dernière modification du contenu du fichier.atime : Date et heure de la dernière lecture du fichier.
  - **Nombre de liens** : Nombre de noms de fichiers (liens) qui pointent vers cet inode.Pointeurs de bloc :
  - **Adresses des blocs de données** sur le disque où le contenu du fichier est stocké.

## I- Système de fichiers

47

- Chaque fichier ou répertoire a un inode unique. Toutefois, l'inode ne contient pas le nom du fichier ni le chemin d'accès, seulement les informations sur son emplacement physique sur le disque.
- **Blocs** : Les fichiers eux-mêmes sont stockés dans des blocs sur le disque. Un bloc est une unité de stockage physique généralement comprise entre **1 Ko** et **4 Ko**.
  - Lorsque l'on crée un fichier, le système de fichiers attribue un inode au fichier, et ses données sont stockées dans un ou plusieurs blocs.
  - Les pointeurs dans l'inode indiquent où se trouvent les blocs physiques sur le disque qui contiennent les données du fichier.
- **Superbloc** : c'est une structure spéciale du système de fichiers qui contient des informations globales, comme la taille du système de fichiers, le nombre d'inodes et de blocs, et les statistiques sur l'utilisation.
- Pour afficher le numéro de l'inode d'un fichier (**`ls -li <nom_fich>`**) et l'inode d'un rép (**`ls -li <non-rep>`**) et les inodes des éléments d'un rép (**`ls -li <nom_rep>`**).
- Pour afficher le contenu de l'inode d'un fichier/rép : **`stat <nom_fich>`**.
- Pour afficher les infos d'un superbloc : **`df -h`** pour identifier le nom du système de fichiers, puis **`sudo dumpe2fs <nom_syst_fichiers> | less`**

- Une commande Unix est composée d'un code mnémonique (son nom), suivi parfois d'options et/ou paramètres. L'espace est le caractère séparateur.
- Une commande n'est interprétée que lorsque l'utilisateur a tapé **<Entrée>**.

Exemple : **ls -l /home/toto**

- Il est possible, à tout moment (avant <Entrée>) de modifier ou d'effacer une commande saisie au clavier :
  - **<Backspace>** efface le dernier caractère (ou **<Ctrl-h>**).
  - **<Ctrl-u>** permet d'annuler toute la ligne.

### Remarques :

- **Bash, C-shell et Korn-shell** permettent de modifier des commandes déjà exécutées et de les relancer.
- L'exécution d'une commande peut être interrompue par **<Ctrl-C>** :  
exple : **sleep 60** (son exécution dure 60 secondes. Pour l'interrompre : **<Ctrl-C>**)
- Il est formellement déconseillé de redémarrer le serveur Unix/Linux.
- Arrêt brutal → système de fichiers incohérent.



## III- Commandes de manipulation de fichiers

49

- La commande **ls** permet d'afficher le contenu des répertoires et les détails concernant les fichiers.

Les options les plus utiles pour **ls** :

- **-l** : affiche le type de fichier, les protections, le nombre de liens avec le fichier, le propriétaire, le groupe, la taille en octets, la date de dernière modification et le nom du fichier ;
  - **-F** : "/" après les noms des répertoires. "\*" : après les noms des fichiers exécutables (programmes ou scripts avec les permissions d'exécution). "@" : après les liens symboliques. "|" : après les fifos ou pipes. "=" : après les sockets.
  - **-a** : liste tous les fichiers y compris les fichiers cachés.
  - **-R** : liste les fichiers et les répertoires de façon récursive.
- Les arguments de **ls** (s'ils existent) sont interprétés comme des noms de fichiers ou de répertoires.

## III- Commandes de manipulation de fichiers

50

- Pour dénommer un fichier on peut utiliser la dénomination absolue, c'est à dire en partant de la racine (/) : */users/Etudiants/ali/mail/imp.txt* (nommage **absolu**)
- Mais si le répertoire de travail est */users/Etudiants/ali*, il suffira de nommer ce fichier:  
*mail/imp.txt* (nommage **relatif**).
- Et si le répertoire de travail est */users/Etudiants/ali/mail*: *imp.txt* (nommage relatif)
- *pwd* (Print Working Directory) : Affiche le chemin d'accès absolu du répertoire courant.
- *cd rep* (Change Directory) : Change le répertoire courant et déplace l'utilisateur vers **rep** ;  
La commande *cd* sans arguments déplace l'utilisateur vers **son répertoire personnel**.
- *more* (MORE) : Permet d'afficher le contenu d'un fichier page par page :
  - **Espace** : affichage page par page
  - **<Entrée>** : affichage ligne par ligne
  - On peut visualiser le % de lecture en bas. On quitte en tapant la touche "q".
- *less* : Comme *more*, mais elle permet la navigation dans les 2 sens du contenu de fichier en utilisant les touches "**flèche bas**" et "**flèche haut**". Elle est rapide et plus interactive que *more*.

## III- Commandes de manipulation de fichiers

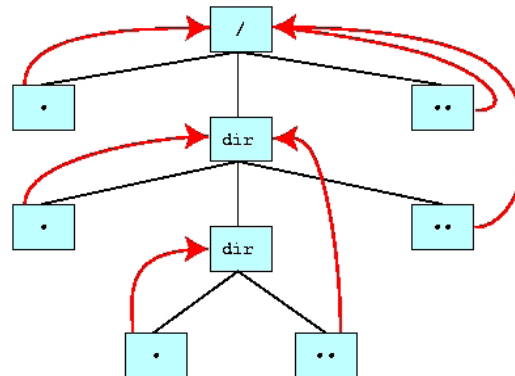
51

- Quand un répertoire est créé, le système génère automatiquement 2 sous-répertoires représentant des références vers le répertoire créé et le répertoire père :
  - le répertoire "." = répertoire courant.
  - le répertoire ".." = répertoire père.
- Le répertoire ".." est très utile pour référencer ce qui se trouve au dessus du répertoire courant dans l'arborescence du système de fichier.

Ainsi il suffira d'utiliser ".." dans un chemin d'accès relatif pour référencer le répertoire père.

### Exemples :

- **cd ..** : remonte d'un cran dans l'arborescence.
- **more ../../fich** : liste le contenu d'un fichier deux niveaux au dessus dans l'arborescence.



## III- Commandes de manipulation de fichiers

52

- **mkdir** (make directory) : permet de créer des répertoires. Syntaxe : **mkdir [-p] rep1 ...**  
Exemple : **mkdir /users/Etudiants/ali/temporaire**  
**mkdir -p** : permet de créer tous les répertoires parents qui n'existent pas encore.  
Exemple : **mkdir -p pgmes/java/essais**
- **rmdir** (remove directory) : permet de supprimer des répertoires. Les répertoires à supprimer doivent impérativement être vides. Syntaxe : **rmdir rep1 ...**
- **rm -r rep** : permet de supprimer manière récursive tout le contenu du dossier **rep** (supprimer le sous-arbre en dessous de rep, y compris **rep**).
- il est impossible de supprimer des répertoires se trouvant entre la racine et le répertoire courant.

**Remarque** : Dans les 2 commandes, on peut utiliser des chemins d'accès relatifs ou absolus.

Exemples : **mkdir monrep**

**mkdir monrep/sd1 monrep/sd2 monrep/sd1/sd11**

**mkdir -p monrep/sd3/sd31**

**rmdir monrep/sd2**

**rmdir monrep/sd3/sd31 monrep/sd3**

**rmdir monrep/sd1/sd11 monrep/sd1 monrep**

- **du** (Disk Usage) : donne l'occupation disque en blocs (généralement 1 bloc = **4 Ko=4096 Octets**) des sous-répertoires du rep. spécifié ou du rep. courant (si aucun rep. n'est spécifié).

## III- Commandes de manipulation de fichiers

53

### Attributs d'un fichier

Par définition, un fichier est une suite d'octets possédant les attributs suivants :

- type ;
- masque de protection ;
- nombre de liens avec d'autres fichiers ;
- propriétaire et groupe ;
- taille ;
- date de création et de dernière modification ;
- nom .

Exemple : ***drwxrwxr-x 2 ali ali 4096 sept. 11 23:04 rep1***

### Types de fichiers

Type	Code
standard	-
répertoire	d
lien symbolique	l
fichier spécial mode bloc	b
fichier spécial mode caractère	c
fichier spécial mode réseau	n
pipe nommé	p
socket	s

## III- Commandes de manipulation de fichiers

54

- Le type "**lien symbolique**" correspond à un fichier spécial pointant sur un autre fichier ou répertoire (comme raccourci en Windows).
- Les types "**fichier spécial mode bloc**" et "**fichier spécial mode caractère**" servent à communiquer avec les périphériques (disques, terminaux, etc.) :
  - Ils ne contiennent pas de données directement, mais permettent au noyau de diriger les opérations vers les bons pilotes de périphériques.
  - Pour les afficher : `ls -l /dev/`
- Le type "**fichier spécial mode réseau**". Représente un périphérique réseau et manipule une interface réseau au niveau matériel ou logiciel (par ex., interfaces TUN/TAP pour les réseaux virtuels).
- Le type "**pipe nommé**" est un fichier spécial qui peut être créé dans le système de fichiers pour servir de canal de communication entre différents processus locaux (lecture/écriture).
- Un socket est une interface de communication qui permet à deux processus, qu'ils soient sur la même machine ou sur des machines différentes et d'échanger des données via les protocoles : TCP/IP, UDP, etc.
- La syntaxe d'un nom de fichier n'est pas stricte. Il est recommandé de :
  - Limiter le nom à **14** caractères.
  - N'utiliser que les lettres **MAJ** et **MIN**, les **chiffres** et quelques caractères ("**.**", "**-**", "**\_**").

## III- Commandes de manipulation de fichiers

55

- Nom d'un fichier :
  - Longueur minimale d'un nom : 1 caractère.
  - Caractères à éviter : \ > < \$ ? & [ ] \* ! " ' ( ) @ ~ <espace>, caractères accentués
  - Un fichier dont le nom commence par un **point** est un fichier **caché** (exple : *.profile*).
  - Le point sert aussi à suffixer des fichiers (exples : *essai.c*, *include.h*, *essai.f*, *essai.o*, ...).
- Les caractères génériques (**wildcards**) permettent d'appliquer une commande à plusieurs fichiers :
  - \* : désigne toute chaîne de 0 à n caractères ;
  - ? : désigne un caractère quelconque ;
  - [...] : désigne un caractère parmi ceux entre crochets.

### Exemples :

*fich.\** : désigne tous les fichiers de nom **fich** et ayant suffixe quelconque.

*essai?* : désigne tous les fichiers ayant 6 caractères dont les 5 premiers sont **essai**.

*[a-f]* : désigne une lettre comprise entre **a** et **f**.

*[a-z]\** : désigne les noms commençant par une lettre **minuscule**.

*fich\*[12]* : désigne les noms commençant par **fich** et se terminant par **1** ou **2**.

*\*.{txt,doc}* : désigne tous les fichiers **.txt** ou **.doc**.

**!** : pour la négation. Exemple : *[!14.]* : le nom ne doit pas contenir **1**, **4** et **.** (point).

## III- Commandes de manipulation de fichiers

56

- La commande **cat** (conCATenate) : permet d'afficher, créer, copier et concaténer des fichiers :  
Exemple1 : **cat /etc/passwd** : affiche le contenu du fichier texte **/etc/passwd**  
Exemple2 : **cat -n /home/toto/essai** permet de numéroter les lignes du fichier affiché  
Exemple3 : **cat > essai** : crée le fichier **essai** et y écrit le texte : *Bonjour ...*  
*Bonjour*  
*comment vas-tu ? <Ctrl-d>* (caractère de fin de saisie)
- La commande **cp** (CoPy) : permet de copier un fichier (ou un répertoire)  
syntaxe : **cp [option] fichier\_origine fichier\_destination**  
ou **cp [option] fichier repertoire**  
Exemple1 : **cp test1 test2** : fait une copie du fichier **test1** en un fichier **test2**.  
On a maintenant deux exemplaires de notre fichier dans le répertoire courant.  
**Remarque** : si l'on effectue une copie d'un fichier sur un fichier qui existe déjà, celui-ci sera effacé et remplacé par le nouveau fichier.



## III- Commandes de manipulation de fichiers

57

Exemple2 : `# cp test1 /home` fait une copie du fichier *test1* dans le répertoire */home* en gardant le même nom.

Exemple3 : `# cp test1 /home/test2` fait une copie de *test1* dans */home* avec un autre nom.

Exemple4 : `# cp * /home/toto/doss4` fait une copie du contenu du rep courant dans le rep : */home/toto/doss4*

Exemple5 : `cp -r Doss1 Doss2/` : Si *Doss2* n'existe pas, il sera créé et *Doss1* sera copié à l'intérieur. Si dossier2 existe déjà, dossier1 sera copié à l'intérieur de dossier2.

### Quelques options importantes de **cp** :

**cp -i** : avertit l'utilisateur de l'existence d'un fichier du même nom et lui demande s'il peut ou non remplacer son contenu.

**cp -b** : permet comme l'option -i de s'assurer que la copie n'écrase pas un fichier existant : le fichier écrasé est sauvegardé, seul le nom du fichier d'origine est modifié et cp ajoute un tilde (~) à la fin du nom du fichier.

**cp -l** : permet de faire un lien "dur" entre le fichier source et sa copie. Le fichier copié et sa copie partageront physiquement le même espace.

**cp -p** : permet lors de la copie de préserver toutes les informations concernant le fichier comme le propriétaire, le groupe, la date de création.

## III- Commandes de manipulation de fichiers

58

**cp -r** : permet de copier de manière récursive l'ensemble d'un répertoire et de ses sous-répertoires.

**cp -v** : permet d'afficher le nom des fichiers copiés. Utile si par exemple pour vérifier la copie de plusieurs fichiers (à l'aide des occurrences "\*" et/ou "?").

- La commande **rm** (ReMove) : supprime un ou plusieurs fichiers d'un répertoire :

Syntaxe : **rm fich1 fich2...**

Exemple : **rm /home/toto/lettre2**

### Quelques options importantes de **rm** :

**rm -i** : permet de demander à l'utilisateur s'il souhaite vraiment supprimer le ou les fichiers en question.

**rm -r** : permet de supprimer un répertoire et ses sous répertoires (attention très dangereux)

**rm -f** : permet de supprimer les fichiers protégés en écriture et répertoires sans que le prompt demande une confirmation de suppression (à utiliser avec précaution ...).

## III- Commandes de manipulation de fichiers

59

- La commande **mv** (MoVe) : permet de renommer et de déplacer un fichier ou un répertoire :  
= copie+suppression

Syntaxe : **mv fich1 fich2** ou **mv fichiers repertoire**

Exemple 1 : **mv lettre1 lettre2** : renomme le fichier lettre1 en lettre2.

Exemple 2 : **mv rep3 doss3** : renomme le répertoire **rep3** en **doss3**, si doss3 n'existe pas.  
Sinon déplace **rep3** dans **doss3**.

Exemple 3 : **mv test /home/toto/pgme** : déplace le fichier test du répertoire courant vers le répertoire **/home/toto/pgme**.

### Quelques options importantes de **mv** :

**mv -b** : ('b' comme "backup") va effectuer une sauvegarde des fichiers avant de les déplacer.

**mv -i** : ('i' comme «interactive») demande pour chaque fichier et chaque répertoire s'il peut ou non déplacer les fichiers et répertoires.

**mv -u** : ('u' comme «update») demande à mv de ne pas supprimer le fichier si sa date de modification est la même ou est plus récente que son remplaçant.

## III- Commandes de manipulation de fichiers

60

- La commande **find** est un outil très puissant sous Linux qui permet de rechercher des fichiers et des répertoires en fonction de divers critères comme le **nom**, la **taille**, la **date** de modification, les **permissions**, etc.

Exemple 1 : **find / -name test1 ; find /home/ali/Docs -iname "lettre.pdf"**

**/** indique que nous voulons chercher à partir de la racine notre fichier **test1**.

**-name** est l'option qui indique ici que nous voulons spécifier le nom d'un fichier.

**-iname** pour ignorer la casse des lettres.

Exemple 2 : **find . -name .profile**

recherche le fichier caché **.profile** à partir du rép. courant **"."** et l'affiche à l'écran.

Exemple 3 : **find /home/ali/pgmes 'test\*'**

recherche tous les fichiers commençant par **test** à partir du répertoire **/home/ali/pgmes**.

Exemple 4 : **find . -type f -size +4M**

recherche tous les fichiers **de taille > 4Mo** à partir du rép. **courant**.

Exemple 5 : **find /home/ali/Docs -mtime -7**

recherche les fichiers **modifiés** dans les **7** derniers jours.

## III- Commandes de manipulation de fichiers

61

Exemple 6 : `find /chemin/directory -mtime +20`

Recherche des fichiers qui **n'ont pas été modifiés** depuis plus de **20** jours

Exemple 7 : `find /usr -type d -name bin`

recherche et affiche tous les répertoires de nom **bin**, à partir du répertoire **/usr**.

Exemple 8 : `find /usr/bin -perm 755`

recherche les fichiers avec des permissions **755** (par exemple, des fichiers exécutables)

Exemple 9 : `find . -name core -exec rm {} \ ;`

recherche à partir du rep. courant, tous les fichiers de nom **core** et les supprime.

**Remarque** : La différence entre **-exec** et **-ok** est que la 2<sup>ème</sup> demandera **pour chaque fichier** trouvé si l'on souhaite réellement réaliser l'opération.

## III- Commandes de manipulation de fichiers

62

- **Lien symbolique** (ou symlinks ou soft link) : un raccourci vers un fichier ou un dossier :
  - Il fonctionne comme un pointeur, reliant le lien à un autre fichier par son chemin.
  - Si le fichier d'origine est supprimé, le lien symbolique devient cassé (broken link).
  - Peut pointer vers des fichiers sur un autre système de fichiers.
  - Se crée avec la commande : `ln -s <cible> <lien_symbolique>`
  - Permet de créer plusieurs raccourcis vers un fichier sans en dupliquer le contenu.
- **Lien dur (hard link)** : une autre référence à un fichier existant, partageant le même inode:
  - Il pointe directement vers les données sur le disque.
  - Si le fichier original est supprimé, le lien dur reste intact et accède toujours aux données.
  - Ne peut pointer qu'à des fichiers sur le même système de fichiers.
  - Se crée avec la commande : `ln <fichier> <lien_dur>`
  - Plus robuste que le lien symbolique, mais ne fonctionne pas pour les répertoires (sauf pour les répertoires spéciaux comme "." et "..").

## III- Commandes de manipulation de fichiers

63

### Utilité des liens :

- Un même fichier peut apparaître dans plusieurs endroits avec des noms différents ;
- Optimisation de l'espace disque ;
- Assurer la cohérence et la synchronisation des données de différentes copies (modification).

Exemple 1 : `ln test /home/toto/test_lien_dur` (création de lien dur).

Exemple 2 : `ln -s test /home/toto/test_lien_symb` (création de lien symbolique)

## III- Commandes de manipulation de fichiers

64

- Une commande peut être de 4 types :
  - Un programme **exécutable** (comme ceux de */usr/bin*). binaires compilés, tels que les programmes écrits en *C* ou *C++*, ou des programmes écrits dans des langages de script comme le *shell*, *Perl*, *Python*, *Ruby*, *etc.*
  - Une commande **intégrée** directement dans le shell. *bash* prend en charge plusieurs commandes internes appelées builtins du shell. Par exemple, la commande *cd* est une commande intégrée au shell.
  - Une **fonction** de shell : petit script shell intégré dans l'environnement (traitée ultérieurement).
  - Les **alias** sont des commandes que nous pouvons définir nous-mêmes, construites à partir des autres commandes.
- La commande *type* : affiche le type de la commande: Exemples : *type ls* ; *type cd*
- La commande *file* est utilisée pour déterminer le type de contenu d'un fichier. Elle analyse le contenu du fichier et affiche des informations sur son type (texte, image, binaire, etc.).
- La commande *wich* (WICH) : permet simplement de connaître le chemin d'un exécutable :  
Exemple : *which ls* → affiche */bin/ls*
- La commande *grep* (GREP) : recherche dans un ou plusieurs fichiers, toutes les lignes contenant une chaîne de caractères donnée ou un motif.  
Exemple1 : *grep bonjour fich* : recherche la chaîne "*bonjour*" dans le fichier "*fich*".



## III- Commandes de manipulation de fichiers

65

l'option **-i** peut être utilisée pour ignorer la casse.

**grep -v bonjour fich** : demande à grep d'imprimer uniquement les lignes qui ne correspondent pas au motif "bonjour".

Exemple2 : **grep -n recrutement test.html** : recherche dans le fichier *test.html*, la ligne et son numéro, contenant la chaîne "recrutement".

Exemple3 : **grep -l bonjour /home/ali/docs/\*** : permet de n'afficher que les **noms** des fichiers du répertoire */home/ali/docs* contenant la chaîne "**bonjour**".

- **sort** : utilisée pour trier (tri alphabétique ou numérique) les lignes d'un fichier texte ou l'entrée standard (stdin). (voir les options : **-r**, **-n**, **-u**, **-k**, *etc.*).
- **uniq** : permet de filtrer les lignes d'un fichier ou de l'entrée standard (stdin) en supprimant les lignes dupliquées. Elle est souvent utilisée en combinaison avec **sort**, car **uniq** ne détecte que les doublons consécutifs (voir les options : **-u**, **-d**, **-c**, *etc.*).
- **head -n** (resp. **tail -n**) permet d'afficher les **n** premières lignes (resp. les **n** dernières lignes).
- **wc** (Word Count) : permet le dénombrement des mots, lignes et caractères dans un fichier. Un mot est déterminé par des espaces, des tabulations ou une nouvelle ligne.

Syntaxe : **wc -lwc fichier** (**l** : ligne (line) **w** : mot (word) **c** : caractère (character) )

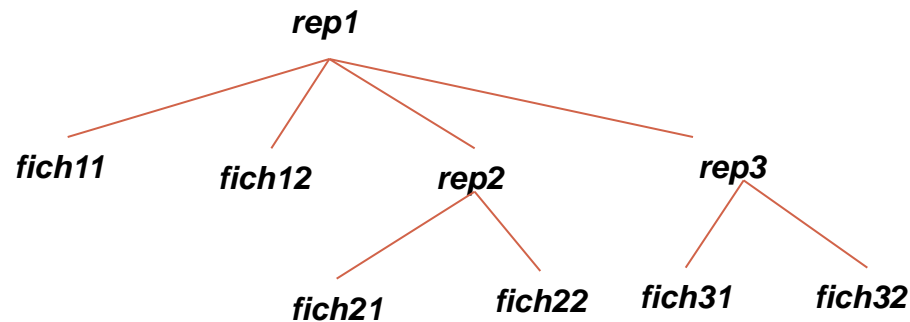
Exemple : **wc -w essai** : Affiche le nombre de mots du fichier *essai*.

## IV- Exercices

66

### Exercice 1

Dans votre répertoire d'accueil, créez l'arborescence suivante, en n'utilisant que des chemins relatifs :



Vérifiez la création.

### Exercice 2

Comment déplacer toute l'arborescence **rep3** sous le répertoire **rep2** ? Supprimez tout sauf **rep1**, **fich11** et **fich12**.

### Exercice 3

A l'aide de la commande **id**, déterminez votre **UID** et votre groupe (nom de groupe et GID). Combien y a-t-il des utilisateurs dans votre équipe ?

## IV- Exercices

67

### Exercice 4

Copier les fichiers dont l'avant dernier caractère est un 4 ou 1 dans le répertoire */tmp* en une seule commande.

### Exercice 5

Listez tous les fichiers :

- se terminant par 5
- commençant par *annee4*
- commençant par *annee4* et de 7 lettres maximum
- commençant par *annee* avec aucun chiffre numérique
- contenant la chaîne *ana*
- commençant par *a* ou *A*

### Exercice 6 (Ecrire, compiler et exécuter un programme C++ sur Linux)

1. Ecrire le programme sur un éditeur (*vi*, *vim*, *nano*, *gedit*, *etc*), le sauvegarder avec un nom et avec l'extension "*.cpp*". (exemple : *test.cpp*).
2. Le compiler à l'aide de la commande : *g++ test.cpp -o test*. Si le compilateur "*g++*" n'est pas installé par défaut, l'installer avec : *sudo apt update ; sudo apt install g++*
3. Exécuter le programme en tapant : *./test* (*test* est l'exécutable créé).

## I- Droits d'accès aux fichiers

68

- Unix/Linux possède des mécanismes permettant au propriétaire d'un fichier (celui qui l'a créé) d'en protéger le contenu : droits d'accès (permissions).
- Unix/Linux définit la notion de groupe d'utilisateurs pour permettre le partage de fichiers et faciliter le travail en équipe. Tout utilisateur appartient au moins à 1 groupe.
- Droits d'accès d'un fichier : ensemble d'indicateurs associés à un fichier.
- 3 types d'utilisateurs pour chaque fichier :
  - Le **propriétaire** du fichier ;
  - Les membres du **groupe** du propriétaire ;
  - Les **autres** utilisateurs du système .
- **id** permet d'afficher des informations sur l'identité de l'utilisateur actif ou d'un utilisateur spécifique. Elle fournit des détails tels que l'ID utilisateur (UID), l'ID de groupe primaire (GID) et les groupes auxquels l'utilisateur appartient.

Exemples : Pour un utilisateur **root** "ahmadi" et pour un utilisateur **standard** "said".

```
ahmadi@ahmadi-VirtualBox:~$ id
uid=1000(ahmadi) gid=1000(ahmadi) groupes=1000(ahmadi),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),100(users),114(lpadmin)
ahmadi@ahmadi-VirtualBox:~$ id said
uid=1002(said) gid=1002(said) groupes=1002(said),100(users)
ahmadi@ahmadi-VirtualBox:~$
```

## I- Droits d'accès aux fichiers

69

- Lorsqu'un compte utilisateur est créé, un numéro appelé ID utilisateur (**uid**) est attribué à l'utilisateur et sera associé à son nom.
- L'utilisateur se voit également attribuer un ID de **groupe primaire (gid)** et peut appartenir à d'autres groupes supplémentaires.
- 3 modes d'accès (par fichier et par type d'utilisateur) :
  - Autorisation d'**écriture (w)** ;
  - Autorisation de **lecture (r)**;
  - Autorisation d'**exécution (x)**.
- 10 attributs associés à chaque fichier Unix (1 pour le type et 9 pour la protection) :  
Exemple : **-rw- rw- r--** (- : fich. ordinaire, **rw-** : droits propriétaire, **rw-** : droits groupe, **r--** : droits autres utilisateurs).

## II- Modification des droits d'accès

70

### 1- Modification des droits d'accès

- La protection d'un fichier ne peut être modifiée que par le propriétaire ou par le root à l'aide de la commande **chmod** (Change MODE).
- 2 modes d'utilisation de la commande **chmod** :
  - Description des protections par un nombre octal (utilisation ancienne) :  
Exemple1: **rw- rw- r-x** est représenté par le nombre octal : **765** → **111 110 101**  
Exemple2 : modification de la protection du fichier **essai** en **r-x rw- rwx** : **chmod 567 essai**
  - Description absolue ou relative des droits d'accès (mode symbolique) :  
syntaxe : **chmod [who] op [permission] fichier**
    - **who** : combinaison de lettres **u** (user=propr.), **g** (groupe), **o** (others=autres) ou **a** (all=tous)
    - **op** : opérateur : "+" ajoute un droit, "-" supprime un droit et "=" affecte un droit de manière absolue (tous les autres bits sont remis à 0).
    - **permission** : **r** (read), **w** (write), **x** (execute).  
Exemple 1 : **chmod u-w test** : supprime le droit d'écriture au *propriétaire* sur le fichier *test*  
Exemple 2 : **chmod g+r test** : ajoute le droit de *lecture* pour le *groupe* sur le fichier *test*.  
Exemple 3 : **chmod ug=x test** : accès uniquement en exécution au *propriétaire* et au *groupe*. On peut affecter plusieurs droits en les séparant par des "virgules".

## II- Modification des droits d'accès

71

### 1- Modification des droits d'accès

Autres exemples :

Droit	Signification
<b>u+x</b>	Ajouter la permission d' <b>exécution</b> pour le propriétaire.
<b>u-x</b>	Retirer la permission d'exécution au propriétaire.
<b>+x</b>	Ajouter la permission d' <b>exécution</b> pour le propriétaire, le groupe et le monde (autres). C'est équivalent à <b>a+x</b> .
<b>o-rw</b>	Retirer les permissions de <b>lecture</b> et d' <b>écriture</b> à quiconque en dehors du propriétaire et du groupe propriétaire.
<b>go=rw</b>	Définir les permissions de <b>lecture</b> et d' <b>écriture</b> pour le groupe propriétaire et quiconque en dehors du propriétaire. Si le groupe propriétaire ou le monde avaient auparavant la permission d'exécuter, elle est retirée.
<b>u+x, go=rx</b>	Ajouter la permission d' <b>exécution</b> pour le propriétaire et définir les permissions pour le groupe et les autres à <b>lire</b> et <b>exécuter</b> . Plusieurs spécifications peuvent être séparées par des virgules.

## II- Modification des droits d'accès

72

### 2- Droits d'accès à la création

- La protection d'un fichier, son propriétaire et son groupe sont établis à la création du fichier et ne peuvent être modifiés que par son **propriétaire** ou par le **root** ;
- **umask** permet de définir un masque de protection des fichiers et répertoires lors de leur création ;
- Le masque est exprimé en **base 8**. Il est soustrait des permissions maximales autorisées par le système : 666 (ou rw-rw-rw-) pour les fichiers et 777 (ou rwxrwxrwx) pour les répertoires.

Exemple : **umask 022**. Pour les répertoires, **022** est soustraite de la permission permanente **777**

$$\begin{array}{r} 111 \ 111 \ 111 \\ 000 \ 010 \ 010 \\ \hline 111 \ 101 \ 101 \end{array} = 755$$

#### Remarques :

- **umask 022** permet de définir la protection des répertoires : **rwx r-x r-x** est souvent l'opération par défaut.
- Pour les fichiers ordinaires, **umask 022** définit la protection : **rw- r-- r--** : l'exécution n'est pas autorisée sur les fichiers ordinaires lors de leur création.

Exemple : **umask 022; touch fich1; ls -l fich1** →

**-rw-rw-r-- 1 ahmadi ahmadi 0 oct. 2 13:25 fich1**



## III- Droits d'accès aux répertoires

73

- Affichage des informations d'un répertoire : `ls -dl <nom_rep>`

Exemple : `ls -dl rep` → `drwx r-x r-x 3 abdou amis 1024 Jul 16 12:45 bin`

- Signification de "r" , "w" et "x" pour les répertoires :

- **r** : permet de voir la liste des fichiers qui sont dans le répertoire ;
- **x** : autorise l'accès au répertoire (à l'aide de la commande cd) ;
- **w** : autorise la création, la suppression et le changement du nom d'un élément du répertoire.

Cette permission est indépendante de l'accès aux fichiers dans le répertoire.

Exemple1 :

**rep1** (drwx --- ---)

**fich1**

-rwx --- ---

appartient à  
l'utilisateur **ali**)

→ seul l'utilisateur **ali** pourra supprimer et modifier son fichier **fich1**.

Exemple2 :

**rep2** (dr-x --- ---)

**fich2**

-rwx --- ---

appartient à  
l'utilisateur **ali**)

→ seul l'utilisateur **ali** pourra modifier son fichier **fich2**, mais il ne pourra pas le supprimer car il n'a pas "w" sur le répertoire contenant **fich2**.

## IV- Modification du propriétaire

74

- **chown** (CHange OWNer) : utilisée (avec des **privilèges de root**) pour changer le propriétaire du fichier et/ou le propriétaire du groupe de fichier en fonction du premier argument de la commande.

Syntaxe : **chown** [**owner**][:**group**] **file...**

Exemple 1 :

Argument	Résultats
<b>ali</b>	Change le propriétaire du fichier, en passant de son propriétaire actuel à l'utilisateur <b>ali</b> .
<b>ali:users</b>	Change le propriétaire du fichier, en passant de son propriétaire actuel à l'utilisateur <b>ali</b> , et change le groupe propriétaire du fichier au groupe <b>users</b> .
<b>:admins</b>	Change le groupe propriétaire du fichier pour le groupe <b>admins</b> . Le propriétaire du fichier reste <b>inchangé</b> .
<b>ali:</b>	Change le propriétaire du fichier en passant de son propriétaire actuel à l'utilisateur <b>ali</b> , et change le groupe propriétaire pour le <b>groupe de connexion</b> de l'utilisateur <b>ali</b> .

## IV- Modification du propriétaire

75

- **chgrp** (Change GRoup) : Change la propriété du groupe. Dans les anciennes versions d'Unix, la commande **chown** ne modifiait que la propriété du fichier, pas la propriété du groupe. A cette fin, une commande distincte, **chgrp**, était utilisée.

Exemple 2 :

```
ahmadi@ahmadi-VirtualBox:~$ sudo cp fich1 ~said
[sudo] Mot de passe de ahmadi :
ahmadi@ahmadi-VirtualBox:~$ sudo ls -l ~said/fich1
-rw-r--r-- 1 root root 201 oct.  6 15:15 /home/said/fich1
ahmadi@ahmadi-VirtualBox:~$ sudo chown said ~said/fich1
ahmadi@ahmadi-VirtualBox:~$ sudo ls -l ~said/fich1

-rw-r--r-- 1 said root 201 oct.  6 15:15 /home/said/fich1
ahmadi@ahmadi-VirtualBox:~$ su said
Mot de passe :
said@ahmadi-VirtualBox:/home/ahmadi$ ls -l ~said/fich1
-rw-r--r-- 1 said root 201 oct.  6 15:15 /home/said/fich1
said@ahmadi-VirtualBox:/home/ahmadi$ chgrp said ~said/fich1
said@ahmadi-VirtualBox:/home/ahmadi$ ls -l ~said/fich1

-rw-r--r-- 1 said said 201 oct.  6 15:15 /home/said/fich1
said@ahmadi-VirtualBox:/home/ahmadi$
```

**Remarque :** L'administrateur peut accéder à tous les fichiers → il peut modifier les droits d'accès.

## V- Appartenance à plusieurs groupes

76

En Linux moderne :

- Groupe **primaire** : Le groupe par défaut associé aux fichiers créés par l'utilisateur. Il est défini dans le fichier `/etc/passwd` pour chaque utilisateur. Chaque utilisateur a un **seul** groupe **primaire**.
- Groupes **secondaires** : Des groupes supplémentaires permettent d'accorder à un utilisateur des permissions supplémentaires sur des fichiers appartenant à ces groupes. Ils sont définis dans le fichier `/etc/group`. Un utilisateur peut appartenir à **plusieurs** groupes **secondaires**.
- Groupe **effectif** : Le groupe utilisé pour déterminer les permissions actuelles sur les fichiers, modifiable dynamiquement via la commande **newgrp**.

Exemple :

```
ahmadi@ahmadi-VirtualBox:~$ id
uid=1000(ahmadi) gid=1000(ahmadi) groupes=1000(ahmadi),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),100(users),114(lpadmin)
ahmadi@ahmadi-VirtualBox:~$ id said
uid=1002(said) gid=1002(said) groupes=1002(said),27(sudo),100(users)
ahmadi@ahmadi-VirtualBox:~$ su said
Mot de passe :
said@ahmadi-VirtualBox:/home/ahmadi$ pwd
/home/ahmadi
said@ahmadi-VirtualBox:/home/ahmadi$ touch ~said/fich1
said@ahmadi-VirtualBox:/home/ahmadi$ ls -l ~said/fich1
-rw-r--r-- 1 said root 201 oct.  9 21:50 /home/said/fich1
said@ahmadi-VirtualBox:/home/ahmadi$ newgrp users
said@ahmadi-VirtualBox:/home/ahmadi$ touch ~said/fich2
said@ahmadi-VirtualBox:/home/ahmadi$ ls -l /home/said/fich2
-rw-rw-r-- 1 said users 0 oct.  9 21:53 /home/said/fich2
said@ahmadi-VirtualBox:/home/ahmadi$
```

## V- Appartenance à plusieurs groupes

77

- Les permissions de groupe d'un fichier (ou répertoire) sont applicables à tout utilisateur membre du groupe propriétaire du fichier.
- Création de fichiers et héritage de groupe :
  - Par défaut, tout fichier nouvellement créé appartient au groupe principal de l'utilisateur créateur.
  - Cependant, si le bit **SGID** (Set Group ID) est activé sur un répertoire, alors tout fichier créé dans ce répertoire héritera du groupe propriétaire du répertoire, même si l'utilisateur n'est pas membre de ce groupe.
  - Ainsi, un utilisateur peut créer un fichier appartenant à un groupe dont il n'est pas membre si le répertoire où il crée le fichier a le bit SGID activé.
- **SetGID** permet une gestion plus homogène des fichiers lorsque plusieurs utilisateurs collaborent dans un répertoire partagé, et où les fichiers doivent appartenir à un groupe commun, quel que soit le groupe effectif de l'utilisateur créateur.
- Exemple : Tous les fichiers créés dans **/projets** hériteront du groupe **amis**

```
mkdir /projets
```

```
chown :amis /projets
```

```
chmod g+s /projets
```

Vérification : *ls -ld /projets* (un "s" apparaîtra dans les droits du groupe).

## VI- ACLs (Access Control Lists)

78

- Le modèle traditionnel utilisé dans les systèmes UNIX et Linux (propriétaire, groupe et autres avec lecture/écriture/exécution) fonctionne bien dans la plupart des cas, mais il est limité dès que l'on a besoin de définir des permissions plus **complexes**.
- **ACLs** (Access Control Lists) : permettent une gestion **granulaire** et plus **fine** des permissions, en permettant de définir des droits d'accès spécifiques pour plusieurs utilisateurs ou groupes individuels sur un même fichier ou répertoire.
- Par exemple, si plusieurs utilisateurs ont besoin de permissions spécifiques sur un fichier sans pour autant modifier les permissions globales du fichier pour tous les autres utilisateurs ou groupes, les ACLs permettent cette flexibilité.
- ACLs sont utiles dans des environnements où un contrôle fin des permissions est nécessaire (comme dans certaines grandes entreprises ou réseaux mixtes UNIX/Windows).
- Inconvénients des ACLs :
  - *Administration complexe* : Plus il y a d'entrées dans une ACL, plus elle devient difficile à gérer et à maintenir.
  - *Incompatibilité potentielle* : Tous les systèmes de fichiers et outils ne gèrent pas de la même manière les ACLs, ce qui peut entraîner des problèmes d'interopérabilité.

## VI- ACLs (Access Control Lists)

79

- Principales commandes sur les ACLs :
  - **getfacl** : affiche les ACLs d'un fichier ou répertoire.
  - **setfacl** : modifie les ACLs d'un fichier ou répertoire.

### Exemples :

**setfacl -m u:said:rw fich2** : Donne à l'utilisateur **said** les droits de **lecture** et **écriture** sur **fich2**.

**getfacl fich2** : Affiche les ACLs du fichier **fich2**.

```
ahmadi@ahmadi-VirtualBox:~$ setfacl -m u:said:rw fich2
ahmadi@ahmadi-VirtualBox:~$ getfacl fich2
# file: fich2
# owner: ahmadi
# group: ahmadi
user::rw-
user:said:rw-
group::rw-
mask::rw-
other::r--
```

**setfacl -x u:said fich2** : Retire toute entrée ACL liée à **said** sur le fichier **fich2**.

**setfacl -m g:amis:rw- lettre.txt** : Attribuer des permissions de **lecture** et d'**écriture** pour le groupe **amis** sur le fichier **lettre.txt**.

**setfacl -R -m u:ali:rx Docs/** : Appliquer des permissions de **lecture** et d'**exécution** pour l'utilisateur **ali** sur le répertoire **Docs** et ses sous-répertoires.

**Remarque** : Bien que puissantes, les ACLs sont souvent réservées à des cas spécifiques. Dans la majorité des situations, le modèle traditionnel à **9 bits** est suffisant et plus facile à utiliser.

### Exercice 1

Dans votre répertoire courant, créez un répertoire courant ***essai\_rep***, par défaut ce répertoire est à **755** (***rwxr-xr-x***). Quelles sont les commandes (en notation ***symbolique et en base 8***) pour lui donner les droits suivants (on suppose qu'après chaque commande on remet le répertoire à **755**):

	Propriétaire			groupe			les autres		
	droit en lecture	droit en écriture	droit d'accès	droit en lecture	droit en écriture	droit d'accès	droit en lecture	droit en écriture	droit d'accès
commande 1	oui	oui	oui	oui	non	oui	non	non	oui
commande 2	oui	non	oui	non	oui	non	non	non	oui
commande 3	non	oui	non	non	non	oui	oui	non	non
commande 4	Non	non	oui	oui	non	oui	non	non	non

### Exercice 2

Créez un fichier ***test*** dans le répertoire ***essai\_rep*** précédent. Par défaut ce fichier est à **644** (***rw-r--r--***). En partant du répertoire courant, pour chaque commande de l'exercice précédent, essayez d'accéder au répertoire ***essai\_rep*** (commande ***cd***), de faire un ***ls*** dans ***essai\_rep*** et de ***modifier*** le fichier avec un éditeur quelconque.



## Exercices

81

### Exercice 3

Créer dans un nouveau répertoire *reptest*, le fichier “*bienvenue*” contenant la ligne de commande :  
***echo Bienvenue dans le monde UNIX***  
Exécuter ce fichier.

### Exercice 4

Créer un fichier que vous pouvez *lire*, *modifier* et *supprimer*.

### Exercice 5

Créer un fichier que vous pouvez *lire* et *supprimer* mais que vous ne pouvez pas *modifier*.

### Exercice 6

Créer un fichier que vous pouvez *lire* mais que vous ne pouvez ni *modifier* ni *supprimer*.

### Exercice 7

Dans quel cas les permissions d'un fichier à sa création sont-elles différentes des permissions fixées par *umask* ?

### Exercice 8

Supposons que vous travaillez avec un collègue appartenant au même groupe que vous. Modifiez les permissions du fichier créé à l'exercice “*Bienvenue*” ci-dessus de telle façon que votre collègue puisse le *lire* et l'*exécuter*, mais ne puisse pas le *modifier* ni le *supprimer*.

Pouvez-vous modifier les permissions de ce fichier de telle sorte que votre collègue puisse le *lire*, le *modifier* et l'*exécuter* alors que vous-même ne pouvez pas le *modifier*.

### Exercice 9

Comment est attribuée la permission *d'effacer* un fichier ? Créez un fichier que votre collègue peut *modifier* mais pas *supprimer* et un autre qu'il peut *supprimer* mais pas *modifier*. Est-il logique de pouvoir attribuer de tels droits ? Quelles sont les conséquences pratiques de cette expérience ?

### Exercice 10

Comment savoir très simplement si votre système suit la logique *BSD* ou *System V* en ce qui concerne le *GID* des fichiers à la création ?

### Exercice 11

Vous avez deux utilisateurs sur un système Linux : *ali* et *nadia*. Ils souhaitent créer un répertoire partagé *projet* où ils pourront déposer et modifier des documents relatifs à un projet commun. *ali* a accès aux privilèges *superutilisateur* via la commande *sudo*, tandis que *nadia* n'a pas de privilèges élevés. Que faire pour réaliser cette collaboration entre les 2 utilisateurs ?

# Chap 5 Gestion des Commandes et Flux

## I- Ligne de commandes séquentielles

83

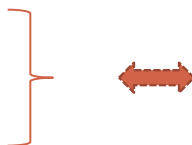
- On peut taper plusieurs commandes séparées par des "; ". Elles s'exécuteront séquentiellement et indépendamment les unes des autres.

Exemple : `pwd ; who ; ls`

## II- Commandes sur plusieurs lignes

- `\<Entrée>` : permet de revenir à la ligne pour taper la suite de la commande.

Exemple : `ls -l /home/ali/develop \<return>`  
`/essai.f \<return>`



`ls -l /home/ali/develop/essai.f \<return>`

# Chap 5 Gestion des Commandes et Flux

## III- Séparateurs conditionnels

84

- **&&** : permet d'exécuter la commande qui le suit si et seulement si la commande qui le précède a été exécutée sans erreur.
- **||** : permet d'exécuter la commande qui le suit si et seulement si la commande qui le précède a été exécutée avec erreur ;

Exemple 1 : ***cd repl && rm \****

suppression des fichiers si la commande ***cd repl*** a été correctement exécutée.

Exemple 2 : ***mkdir nouveau\_dossier && cd nouveau\_dossier && touch fichier.txt***

Exemple 3 : ***cd ~/doss/doss1 || echo "Le répertoire n'existe pas"***

Si ***~/doss/doss1*** échoue (parce que le répertoire n'existe pas), alors **echo "Le répertoire n'existe pas"** sera exécutée.

Exemple 4 : ***cd repl || mkdir repl***

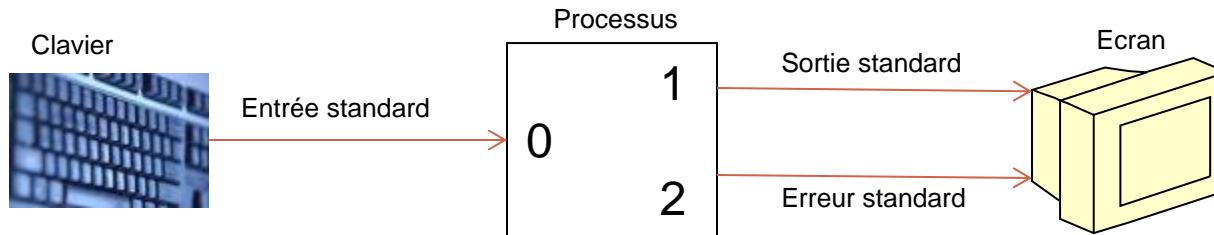
si le répertoire ***repl*** n'existe pas alors il sera créé par la commande ***mkdir***.

# Chap 5 Gestion des Commandes et Flux

## IV- Redirection des Entrées/Sorties

85

- Au lancement d'un processus, le shell ouvre une **entrée standard (stdin)** (par défaut le **clavier**), une **sortie standard (stdout)** (par défaut l'**écran**) et une sortie d'erreurs standard (**stderr**) (par défaut l'**écran**).



- Chaque flux est associé à un identifiant numérique (**0** pour **stdin**, **1** pour **stdout**, **2** pour **stderr**)
- Ces **entrées/sorties** standard peuvent être redirigées vers un **fichier**, un **tube** ou un **périphérique**.
- La redirection des sorties peut être réalisée par effacement et création du fichier ou par ajout à la fin du fichier.
- En cas de redirection d'entrée, le fichier doit exister.
- La syntaxe de redirection d'**E/S** est différente d'un shell à l'autre.

# Chap 5 Gestion des Commandes et Flux

## IV- Redirection des Entrées/Sorties

86

### 1- Redirection en Bash

- a. **commande < fe** : définition de **fe** comme fichier d'**entrée** standard.
- b. **commande > fs** : redirection de la **sortie** standard vers le fichier **fs** (possibilité d'écrasement).
- c. **commande >> fs** : **sortie** standard ajoutée en **fin** du fichier **fs**.
- d. **commande > f 2>&1** : **sortie** standard et sortie d'**erreurs** standard vers le fichier **f** (écrasement s'il existe déjà).
- e. **commande >> f 2>&1** : **sortie** standard et sortie d'**erreurs** standard sont ajoutées en **fin** du fichier **f**.
- f. **commande > fs 2> err** : redirige la **sortie** standard vers **fs** et la sortie d'**erreurs** vers **err**.

#### Exemples :

- **ls > sortie1** : Redirige la sortie standard de 'ls' vers le fichier 'sortie1'.
- **ls >> fich** : Ajoute la sortie standard de 'ls' à la fin du fichier 'fich'.
- **ls /ali\* > sortie2** : Redirige la sortie standard de 'ls /ali \*' (liste des fichiers du répertoire /ali) vers 'sortie2' sans rediriger les erreurs
- **ls /ali\* > sortie2 2>&1** : Redirige la sortie standard de 'ls /ali \*' vers 'sortie2' et la sortie d'erreur (stderr) vers le même fichier.
- **ls /ali\* > sortie3 2> err3** : Redirige la sortie standard de 'ls /ali \*' vers 'sortie3' et la sortie d'erreur vers 'err3'.

# Chap 5 Gestion des Commandes et Flux

## IV- Redirection des Entrées/Sorties

87

### 2- Redirection en C-shell

- a) `< fe` : définition de *fe* comme fichier d'entrée standard ;
- b) `> fs` : redirection de la *sortie* standard vers le fichier *fs* (possibilité d'écrasement) ;
- c) `>> fs` : *sortie* standard ajoutée en fin du fichier *fs* ;
- d) `>& f` : *sortie* standard et *sortie d'erreurs* standard vers le fichier *f* (écrasement s'il existe déjà)
- e) `>>& f` : *sortie* standard et *sortie d'erreurs* standard sont ajoutées en fin du fichier *f* ;
- f) `(commande > fs) >& err` : redirige la *sortie* standard vers *fs* et la *sortie d'erreurs* vers *err* ;

#### Exemples :

- `ls > sortie1` : range la liste des fichiers du répertoire courant dans le fichier *sortie1*
- `ls >> fich` : ajoute au fichier *fich* la liste des fichiers du répertoire *courant*
- `ls /ali* > sortie2` : range, dans *sortie2*, la liste des fichiers du rep */ali* et les messages d'erreurs
- `(ls /ali* > sortie3)>& err3` : résultat dans *sortie3* et erreurs dans *err3*

### 3- Redirection en bourne shell et en korn shell

- a) `< fe`
- b) `> fs`
- c) `>> fs`
- e) `2>f` : redirection de la *sortie d'erreurs* standard vers *f*
- d) `2>>f` : *sortie d'erreurs* standard est ajoutée au fichier *f*
- f) `commande > fs 2> err` : *sortie* → *fs* et *erreurs* → *err*

# Chap 5 Gestion des Commandes et Flux

## IV- Redirection des Entrées/Sorties

88

### 4- La commande cat et les redirections

**cat** est une commande multi-usage : affichage, création, copie et concaténation de fichiers.

#### a. Lecture au clavier et écriture sur écran :

Exemple 1 : **cat**  
*Bonjour Mr ...*  
*<ctrl-d>*

#### b. Copie d'un fichier :

Exemple 2 : **cat f1 > f2** ou **cat <f1 >f2**

#### c. Concaténation des fichiers :

Exemple 3 : **cat f1 f2 f3 > f** *f* contiendra la concaténation des contenus de *f1*, *f2* et *f3*

#### d. Ajout d'un fichier :

Exemple 4 : **cat f1 >> f2** *f1* est concaténé à la suite de *f2*. *f2* est créé s'il n'existe pas.

#### e. Création d'un fichier par saisie au clavier :

Exemple 5 : **cat > f1**  
*Bonjour ...*  
*<ctrl-d>*

#### f. Création d'un fichier avec condition de saisie :

Exemple 6 : **cat << EOT > f1** *f1* est créé par saisie de texte au clavier, jusqu'à la saisie en début de ligne de la chaîne **EOT**.  
*Au revoir ...*  
*EOT*

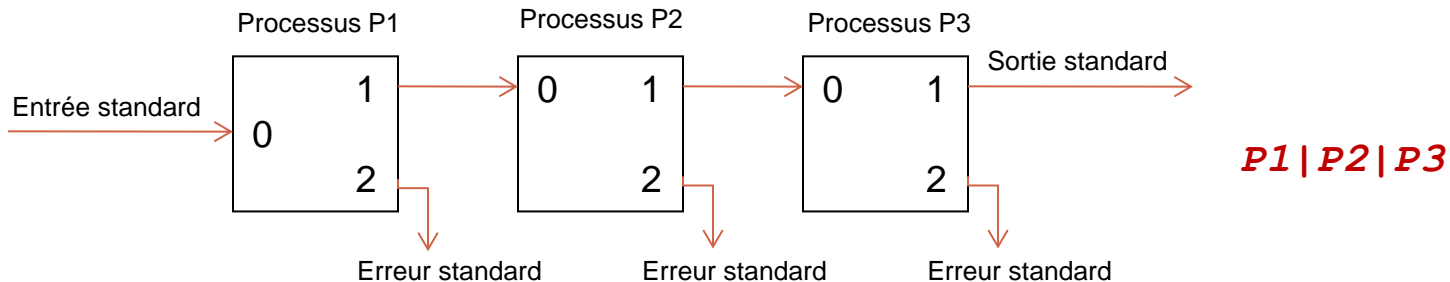


# Chap 5 Gestion des Commandes et Flux

## V- Tubes de communication (pipes)

89

- Un tube (pipe) est un flot de données qui permet de relier la sortie standard d'une commande à l'entrée standard d'une autre commande sans passer par un fichier temporaire.



Exemple 1 : **ls -l | less** : Affichage page par page du contenu du répertoire courant

Exemple 2 : **ls -l | grep "rwxr-xr-x" | less**

Affichage page par page des fichiers du répertoire courant ayant les protections **rwxr-xr-x**

↔ { **# ls -l > temp1; \<return>**  
**grep "rwxr-xr-x" <temp1 > temp2 ; \<return>** **Solution longue !!**  
**less temp2 ; rm temp1 temp2**

- Remarques** :
- Toutes les commandes liées par un tube s'exécutent en parallèle. C'est le système qui réalise la synchronisation entre le processus émetteur et le processus récepteur.
  - Il n'y pas de taille limite pour le flot de données et pas de fichier temporaire.

# Chap 5 Gestion des Commandes et Flux

## V- Tubes de communication (pipes)

90

Exemple 3 : `who | wc -l` : Affiche le nombre d'utilisateurs connectés au système

Exemple 4 : `ls | wc -w` : Affiche le nombre de fichiers dans le répertoire courant

- Un filtre est une commande qui lit les données sur l'entrée standard, les traite et les écrit sur la sortie standard. Les filtres les plus utilisés sont :
  - `grep` : recherche les occurrences d'une chaîne de caractères (ou un motif).
  - `wc` et `more/less` : déjà vues.
  - `sed` : éditeur de flot. Il applique les commandes de l'éditeur `ed` sur l'entrée standard et envoie le résultat sur la sortie standard.
  - `awk` : petit langage de manipulation de texte. Exemple 5 : `awk -F: '$5==" "' /etc/passwd` sélectionne, dans le fichier `/etc/passwd`, les lignes dont le 5ème champ est vide. Le séparateur de champ étant le caractère ":".
  - `sort` : filtre de tri. Exemple 6 : `sort -o test.tri test`. Le fichier `test.tri` contiendra le contenu trié de `test`.

Exercice : afficher le nombre d'utilisateurs de la machine dont le login shell est `c-shell`.

Solution : `cat /etc/passwd | grep /bin/csh | wc -l`

# Chap 5 Gestion des Commandes et Flux

## VI- Backquoting

91

- **Backquotes** : Permettent d'utiliser le résultat d'une commande1 comme argument d'une autre commande2.
- Syntaxe : ``commande1`` accents graves en Bash ou `$(commande1)` en korn-shell.

Exemple1 : `echo "Le répertoire actuel est : `pwd`"` affiche : **Le répertoire courant est : /home/toto** par contre `echo 'pwd'` affichera : **pwd**.

Exemple2 : `cat `grep -l 'titi' *`` : permet de visualiser le contenu des fichiers du répertoire courant contenant au moins une fois la chaîne titi.

Exemple3 : `echo il y a `who | wc -l` utilisateurs connectés`: indique le nb d'utilisateurs connectés.

Exemple4 : `# grep -n "coucou" `find . -type f -print`` : affiche les lignes avec leurs numéros de tous les fichiers (à partir du répertoire courant) contenant la chaîne **coucou**

Exemple5 : `# rm `find . -mtime +20 -print`` : permet de supprimer les fichiers n'ayant été modifiés depuis plus de 20 jours. Équivalent à : `find . -mtime +20 -exec rm {} \;`

# Chap 5 Gestion des Commandes et Flux

## VII- Commandes groupées

92

- Une commande groupée est une succession de commandes séparées par ";" et considérée comme un ensemble.
- *(commande1;commande2)* est différente de : *commande1;commande2*. En effet, la 1ère est exécutée par un sous-shell (un nouveau processus shell).

Exemple1 : *(cd doss1;rm temp)* : le répertoire courant ne change pas.

*cd doss1;rm temp* : le répertoire courant change. Il devient */.../doss1*

Exemple2 : *(echo "Aujourd'hui "; date; echo les \<return>*

*personnes suivantes; who; echo sont connectées) > fs*

Le fichier *fs* contiendra le texte : **Aujourd'hui ...sont connectés**

Exemple3 : *echo "Aujourd'hui"; date; echo les \<return>*

*personnes suivantes; who; echo sont connectés > fs*

Le fichier *fs* contiendra uniquement le texte : **sont connectées**

Exemple4 : *(cp fich1 doss1; find . -mtime +20 -exec rm {} \;) &* : Lance les 2 commandes en arrière-plan.

*cp fich1 doss1; find . -mtime +20 -exec rm {} \; &* : Lance uniquement la dernière commande en arrière-plan.

# Chap 5 Gestion des Commandes et Flux

## VIII- Caractère de neutralisation

93

- Les caractères de neutralisation permettent de neutraliser l'interprétation des caractères spéciaux (< ? \* \$ ...) par le shell. Cela sert à interpréter un caractère littéralement, même s'il a une signification particulière pour le shell.

Exemple1: `\<return>` : permet de neutraliser l'effet de `<return>` qui est la validation d'une commande.

Exemple2 : `touch f\*1` : permet de créer un fichier dont le nom est `f*1`.

Attention : `rm f*1` : supprimera tous les fichiers commençant par `f` et se terminant par `1`.

- L'utilisation du caractère `\` n'est pas pratique pour neutraliser une chaîne de caractères. On peut alors délimiter la chaîne par 2 **quotes**.

Exemple 3 : `touch 'f*?1'` : permet de créer un fichier nommé `f*?1`.

La commande : `rm f*?1` permet de supprimer tous les fichiers commençant par `f` et se terminant par `1`.

Exemple 4: `echo fichier{1,2,3}.txt` affichera :

`fichier1.txt fichier2.txt fichier3.txt.`

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

94

- le shell maintient une série d'informations pendant une session de shell, appelée l'**environnement**
- L'environnement est l'ensemble des variables et configurations qui influencent le comportement des processus en cours.
- Le shell stocke deux types de données de base dans l'environnement : **variables d'environnement** et **variables du shell**. Il stocke aussi les **alias** et les **fonctions**.
- **Variables du shell :**
  - Variables internes au shell, créées et gérées uniquement dans la session en cours du shell.
  - Elles sont temporaires et n'affectent pas les programmes ou les processus exécutés en dehors de cette session du shell, sauf si elles sont explicitement exportées.
  - Exemples : les variables créées par les scripts shell pour stocker des informations temporairement, comme **Compteur=10** ou **monFich="/Docs/Pgmes/script1.sh"**.
- **Variables d'environnement :**
  - Ce sont des variables qui ont été exportées depuis le shell et sont disponibles pour tous les processus fils (programmes et scripts) lancés depuis cette session du shell.
  - Elles sont visibles par le shell et les programmes exécutés par le shell, car elles sont héritées par les processus enfants.
  - Elles permettent de partager des informations générales sur le système ou la session, comme la langue (**LANG**), le répertoire de l'utilisateur (**HOME**), ou l'éditeur de texte préféré (**EDITOR**).

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

95

- La commande **set** (intégrée dans bash) affichera à la fois les variables du shell et de l'environnement, tandis que **printenv** affichera uniquement les variables d'environnement.

Exemples: **printenv | less** # affiche le contenu page par page

**set | less**

**printenv USER** ou bien **echo \$USER** # affiche l'utilisateur courant

- Les principales variables d'environnement :

Variable d'Environnement	Description
<b>HOME</b>	Répertoire personnel de l'utilisateur actuel.
<b>USER</b>	Nom de l'utilisateur courant
<b>SHELL</b>	Le chemin absolu du shell utilisé par l'utilisateur (par exemple /bin/bash)
<b>PATH</b>	Liste des répertoires dans lesquels les commandes sont recherchées.
<b>PWD</b>	Répertoire de travail actuel (présent dans lequel l'utilisateur se trouve)
<b>OLDPWD</b>	Ancien répertoire de travail avant le dernier changement de répertoire
<b>EDITOR</b>	Editeur de texte préféré (souvent utilisé par des applications comme crontab)

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

96

Variable d'Environnement	Description
<b><i>LOGNAME</i></b>	Nom de l'utilisateur connecté.
<b><i>LANG</i></b>	Définition de la langue du système et des paramètres de localisation (par exemple en_US.UTF-8)
<b><i>HISTSIZE</i></b>	Nombre maximum de lignes que l'historique de commandes peut contenir
<b><i>HISTFILE</i></b>	Fichier dans lequel l'historique des commandes est enregistré (par défaut ~/.bash_history).
<b><i>TERM</i></b>	Type de terminal utilisé (par exemple xterm, linux, etc.)
<b><i>MAIL</i></b>	Répertoire où les messages électroniques sont stockés pour l'utilisateur
<b><i>TMPDIR</i></b>	Répertoire où les fichiers temporaires sont stockés
<b><i>DISPLAY</i></b>	Définit l'affichage pour les applications graphiques (principalement utilisé avec X11)
<b><i>TZ</i></b>	Paramètres de fuseau horaire du système
<b><i>PS1</i></b>	Définition du prompt du shell. PS2 pour le second prompt.
<b><i>LD_LIBRARY_PATH</i></b>	Liste des répertoires où le système cherche les bibliothèques partagées.



# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

97

- Lorsqu'un utilisateur se connecte au système, le programme *bash* démarre et lit une série de scripts de configuration appelés fichiers de démarrage, qui définissent l'environnement par défaut partagé par tous les utilisateurs. Ensuite, il lit les fichiers de démarrage situés dans le répertoire personnel, qui définissent l'environnement personnel de l'utilisateur en question.
- La séquence exacte dépend du type de session shell en cours de démarrage. Il existe deux types de sessions :
  - Une session **shell de connexion** : celle où l'on doit saisir un nom d'utilisateur et un mot de passe. Cela se produit, par exemple, lorsqu'on démarre une session de console virtuelle.
  - Une session **shell sans connexion** : se produit généralement lorsqu'on lance une session de terminal dans l'interface graphique.
- Fichiers de démarrage pour les sessions **shell de connexion** :

Fichier	Contenu
<b>/etc/profile</b>	Un script de configuration global qui s'applique à tous les utilisateurs.
<b>~/.bash_profile</b>	Un fichier de démarrage personnel de l'utilisateur. Il peut être utilisé pour étendre ou remplacer les paramètres du script de configuration global.
<b>~/.bash_login</b>	Si <i>~/.bash_profile</i> n'est pas trouvé, <i>bash</i> tente de lire ce script.
<b>~/.profile</b>	Si ni <i>~/.bash_profile</i> ni <i>~/.bash_login</i> ne sont trouvés, <i>bash</i> tente de lire ce fichier. C'est le fichier par défaut dans les distributions basées sur Debian, comme Ubuntu.

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

98

- Fichiers de démarrage pour les sessions *shell sans connexion* :

Fichier	Contenu
<code>/etc/bash.bashrc</code>	Un script de configuration global qui s'applique à tous les utilisateurs.
<code>~/.bashrc</code>	Un fichier de démarrage personnel de l'utilisateur. Il peut être utilisé pour étendre ou remplacer les paramètres du script de configuration global.

**Remarque** : En plus de lire les fichiers de démarrage du tableau précédent, les shells sans connexion héritent de l'environnement de leur processus parent, habituellement un shell de connexion.

- Le fichier `~/.bashrc` est le fichier de démarrage le plus important du point de vue de l'utilisateur ordinaire, car il est presque toujours lu :
  - Les shells sans connexion le lisent par défaut ;
  - la plupart des fichiers de démarrage pour les shells de connexion sont écrits de manière à lire également le fichier `~/.bashrc`.
- En règle générale, pour ajouter des répertoires à la variable *PATH* ou définir des variables d'environnement supplémentaires, on le fait dans `.bash_profile` (ou équivalent selon la distribution ; par exemple, Ubuntu on utilise `.profile`). Pour tout le reste, on place les modifications dans `.bashrc`.

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux

99

Exemple d'utilisation : le script ci-dessous accède aux variables USER et HOME pour afficher le nom d'utilisateur et le répertoire personnel.

```
#!/bin/bash
echo "Utilisateur : $USER"
echo "Répertoire : $HOME"
```

- La commande **export** en Linux est utilisée pour définir ou marquer des variables d'environnement afin qu'elles soient accessibles non seulement dans le shell actuel, mais aussi par tous les processus enfants (shells fils et autres commandes) démarrés par ce shell.
- Sans **export**, une variable définie dans un shell reste *locale* et n'est pas accessible par les sous-processus.
- Peut être utilisée pour **modifier** ou **ajouter** des variables d'environnement qui influencent le comportement de programmes ou de commandes dans les shells fils.
- Pour vérifier les variables d'environnement exportées, on utilise la commande **printenv** ou **export -p**

Exemples : **export PATH=\$PATH:/home/ahmadi/bin/scripts**  
**export LANG=fr\_FR.UTF-8**

# Chap 6 Introduction à l'environnement en Linux

## I- Notion d'environnement en Linux



- On peut aussi exporter une variable temporairement pour une commande unique en plaçant *export* directement avant la commande :

Exemple : (**export LANG=en\_US.UTF-8; date**) # Cette commande affichera la date en *anglais*.

ou bien : **LANG=en\_US.UTF-8 date**

- Un **shell fils** est un shell lancé par un autre shell, appelé le **shell parent** : lorsqu'un utilisateur ou un programme exécute un nouveau shell (par exemple, en lançant une nouvelle instance de bash depuis une session existante).
- Le **shell fils** hérite des variables d'environnement du **shell parent**, ce qui lui permet de disposer des mêmes informations de configuration (comme PATH, HOME, USER, etc.) au démarrage.
- Les modifications que le **shell fils** apporte à ses propres variables n'affectent pas celles du **shell parent**.
- Les variables du shell qui n'ont *pas été exportées* par le **shell parent** ne sont pas transmises au **shell fils**.
- Pour exécuter un script dans le **shell courant** (pas de création de **shell fils**) :  
**source Pgmes/monScript1.sh** ou bien : **. Pgmes/monScript1.sh**
- Pour exécuter un script dans un **shell fils** : **./Pgmes/monScript1.sh** ou **Pgmes/monScript1.sh**  
ou **bash Pgmes/monScript1.sh**

# Chap 6 Introduction à l'environnement en Linux

## II- Personnalisation de l'invite (prompt)

101

- PS1 est une variable puissante pour personnaliser l'apparence du terminal.

Exemple 1 : `PS1="\u@\h:\w$ "`, où :

`\u` affiche le nom de l'utilisateur, `\h` affiche le nom de l'hôte et `\w` affiche le répertoire de travail courant.

Exemple 2 : `PS1="\[\e[34m\]\u@\h\[\e[m\]:\[\e[32m\]\w\[\e[m\]]$ "`, où

`\[\e[34m\]` : Ce code applique la couleur **bleue** au texte qui suit.

`\e[34m` : Code ANSI pour le bleu (34). `\[ et \]` : Ces crochets encadrent les codes de couleur

`\u` : Affiche le nom de l'utilisateur actuel

`@` : Simple symbole pour séparer le nom de l'utilisateur et le nom de l'hôte.

`\h` : Affiche le nom de l'hôte (nom de la machine).

`\[\e[m\]` : Réinitialise la couleur du texte, revenant à la couleur par défaut du terminal.

`\e[m` : Code ANSI pour réinitialiser la couleur.

`:` : Simple symbole : pour structurer visuellement le prompt et séparer le nom de l'hôte du répertoire de travail.

`\[\e[32m\]` : Applique la couleur verte au texte qui suit. `\e[32m` : Code ANSI pour le vert (32).

`\w` : Affiche le chemin du répertoire courant de manière relative.

`\[\e[m\]` : Réinitialise la couleur du texte pour revenir à la couleur par défaut.

`$` : Affiche le symbole \$ pour indiquer que le prompt est prêt à recevoir une nouvelle commande. Ce symbole est généralement \$ pour les utilisateurs ordinaires et # pour l'utilisateur root (administrateur).

Le nouveau prompt (pour mon cas) sera : `ahmadi@ahmadi-VirtualBox:~$`

# Chap 6 Introduction à l'environnement en Linux

## II- Personnalisation de l'invite (prompt)

102

Exemple 3 : `PS1="\d \u \w$ "` : afficher la date et le nom de l'utilisateur.

Exemple 4 : Ce prompt devient **vert** pour une commande réussie, **rouge** en cas d'erreur.

```
PS1='$ (if [[ $? == 0 ]]; then echo "\[\e[32m\]:)"; else echo "\[\e[31m\]:("; fi)
\[\e[m\] \u@\h:\w$ '
```

- Modification **temporaire** : Les changements faits directement dans le terminal sont valables uniquement pour la session en cours.
- Modification **permanente** : Pour conserver le prompt personnalisé, ajoutez la commande `PS1="..."` dans le fichier `~/ .bashrc`, puis rechargez le fichier avec : `source ~/ .bashrc`

# Chap 6 Introduction à l'environnement en Linux

## III- Alias

103

- Un **alias** permet de créer un raccourci pour une commande ou une série de commandes. Ils rendent l'utilisation du terminal plus rapide et permettent de personnaliser des commandes longues ou complexes.
- Syntaxe: `alias nom_alias='commande'` ou `nom_alias="commande"` (si commande contient des variables)
- Exemples d'alias courants :
  - `alias ll='ls -l'` pour simplifier l'utilisation de la commande `ls -l`.
  - `alias psg='ps aux | grep'` : Recherche un processus par nom, par exemple `psg nginx`.
  - `alias ping='ping -c 5'` : Limite le nombre de pings envoyés à 5 pour éviter une boucle infinie.
  - `alias cls='clear'` : Efface l'écran du terminal.
  - `alias reload='source ~/.bashrc'` : Recharge les paramètres du fichier `.bashrc` sans redémarrer le terminal.
  - `alias update='sudo apt update && sudo apt upgrade'` : Met à jour le système (utile sur les distributions basées sur Debian/Ubuntu).
  - `alias activate='source venv/bin/activate'` : Active un environnement virtuel Python.
  - `alias py='python3'` : Lance Python 3 (utile sur les syst. où python appelle encore Python 2).
- Suppression d'un alias : Utilisez `unalias nom_alias` pour supprimer un alias.
- Alias **permanents** : Ajouter les alias dans le fichier `~/.bashrc` pour les rendre persistants à chaque nouvelle session.

# Chap 6 Introduction à l'environnement en Linux

## IV- Fonctions

104

- Une fonction est un bloc de commandes regroupées sous un nom, permettant d'exécuter des tâches plus complexes qu'un alias.

- Syntaxe :

```
nom_fonction() {  
    # Commandes  
}
```

- Exemple 1 :

```
sauvegarde() {  
    cp "$1" "$1.bak"  
    echo "Backup de $1 créé avec succès."  
}  
# $1 est le premier argument lors de l'appel après le nom de la  
# fonction. $2: 2ème argument, etc.
```

Utilisation :

**Sauvegarde pgme1.cpp**

- Exemple 2 :

```
search_log() {  
    grep "$1" /var/log/syslog  
}
```

Permet de faire une recherche dans les logs.

- Pour rendre une fonction disponible dans chaque session on l'ajoute dans **~/.bashrc** et on utilise **source ~/.bashrc** pour appliquer immédiatement les changements sans redémarrer la session.
- Si de nombreux alias et fonctions sont créés, il est recommandé de les organiser dans un fichier séparé, comme **~/.mes\_alias** ou **~/.mes\_fonctions**, puis de les inclure dans **~/.bashrc** avec :

```
if [ -f ~/.mes_alias ]; then  
    source ~/.mes_alias  
fi
```



## I- Processus

105

- Un processus est une instance d'un programme en cours d'exécution. Il représente l'état dynamique d'un programme, y compris son code, ses données, et ses ressources.
  - Chaque processus possède des ressources isolées (mémoire, espace d'adressage) pour éviter les conflits entre programmes.
  - Composants d'un processus :
    - **PID** (Process Identifier) : Un identifiant unique pour chaque processus dans le système.
    - **UID** et **GID** : Identifiants d'utilisateur et de groupe qui permettent de gérer les permissions.
    - **Segments** de mémoire du processus :
      - *Code* : Contient le programme exécutable.
      - *Données* : Stocke les variables globales et statiques.
      - *Pile* (Stack) : Gère les appels de fonctions, les variables locales et les adresses de retour.
      - *Tas* (Heap) : Alloue de la mémoire dynamique pendant l'exécution.
- NB** : La séparation des segments mémoire améliore la sécurité et l'isolation entre les processus.
- Cycle de vie d'un processus :
    1. **Création** : Le processus est initialisé par une commande ou un autre processus (appelé processus parent).
    2. **Exécution** : Le processus est en cours d'exécution par le CPU.
    3. **Attente** : Le processus est suspendu en attente d'un événement (ex. fin d'une opération d'E/S).
    4. **Reprise** : Le processus retourne en exécution dès que la ressource est disponible.
    5. **Fin** : Le processus se termine lorsque son exécution est complétée ou qu'il est interrompu.

## I- Processus

106

- Les processus sont souvent créés via des appels système comme **fork()** sous Unix/Linux, qui génère un processus enfant à partir d'un parent.
- Principaux **états** d'un processus :
  1. **Nouveau** : Le processus est en cours de création.
  2. **Prêt** : Le processus est prêt à être exécuté par le CPU mais en attente de son tour.
  3. **Exécution** : Le processus est actif et utilise le CPU.
  4. **Attente** : Le processus est en pause, attendant une ressource.
  5. **Terminé** : Le processus a fini son exécution et libère ses ressources.
- Les processus passent d'un état à l'autre en fonction des conditions système, telles que la disponibilité des ressources et les priorités des tâches.
- Ordonnanceur de processus : responsable de l'allocation du temps CPU aux processus pour garantir un fonctionnement fluide du système. Il utilise divers algorithmes pour optimiser la gestion du temps et des ressources.
- Algorithmes d'ordonnancement :
  - **FIFO** (First In, First Out) : Les processus sont exécutés dans l'ordre de leur arrivée.
  - **Round Robin** : Chaque processus obtient une portion de temps fixe (quantum), puis passe au suivant.
  - **Priorité** : Les processus sont ordonnés en fonction de leur priorité.
  - **SJF** (Shortest Job First) : Le processus ayant la plus courte durée d'exécution est exécuté en premier.

**NB** : Certains algorithmes d'ordonnancement peuvent être préemptifs (interrompant les processus) pour une meilleure réactivité.

- Le **contexte d'un processus** contient toutes les informations nécessaires pour *sauvegarder* et *restaurer* l'état d'un processus. Il inclut : le contenu des **registres** du CPU, le **compteur** de programme, les **segments** de mémoire, et l'état des **E/S**.
- Changement de contexte (context switch) : Lorsqu'un processus est suspendu, son contexte est sauvegardé pour que le CPU puisse exécuter un autre processus. Bien que coûteux en ressources, le changement de contexte est essentiel pour le **multitâche**.
- **Processus vs Threads** :
  - Processus : Unités indépendantes avec des espaces *mémoire distincts*.
  - Threads : "Légers" sous-composants d'un processus partageant la *même mémoire*, mais pouvant s'exécuter en *parallèle*.
- Avantages des threads :
  - Ils consomment moins de mémoire car ils partagent la même adresse mémoire.
  - Les threads permettent des opérations *multitâches* plus efficaces à l'intérieur d'un même processus.
- Types de threads :
  - Threads au niveau utilisateur : Gérés par des bibliothèques au niveau de l'application.
  - Threads au niveau noyau : Gérés directement par le système d'exploitation.

## II- Surveillance et gestion des processus

108

- **ps** (Process Status) : commande utilisée pour afficher les informations des processus en cours d'exécution. Elle capture un instantané des processus au moment où elle est exécutée.
  - **ps -e** : Affiche tous les processus.
  - **ps -f** : Affiche des informations détaillées (format complet) sur chaque processus.
  - **ps -u ali** : Affiche les processus appartenant à un utilisateur **ali**.
  - **ps aux** : Affiche tous les processus en cours avec des informations détaillées.
  - **ps -ef** : Affiche tous les processus au format complet, incluant les **PPID** (Parent Process ID).

### Exemples :

**ps -ef | grep <PID>** : Cherche le processus en cours ayant un PID spécifique.

**ps -aux | grep nginx** : Filtre les processus pour ne montrer que ceux relatifs à "nginx".

- **top** (Table of Processes) : affiche une vue en **temps réel** des processus en cours d'exécution, en mettant en évidence l'utilisation du CPU, de la mémoire, et d'autres informations critiques.
  - **top -u <utilisateur>** : Montre uniquement les processus d'un utilisateur spécifique.
  - **top -n <nombre>** : Exécute top pour un nombre d'itérations donné, puis quitte.
  - **top -d <seconde>** : Définit la fréquence de rafraîchissement (en secondes).
  - **P** : Trie les processus par utilisation **CPU** (touche en mode interactif).
  - **M** : Trie les processus par utilisation de la **mémoire** (touche en mode interactif).

## II- Surveillance et gestion des processus

109

- **htop** (Process Status) : C'est une version améliorée de **top** avec une interface visuelle plus intuitive et interactive. Elle propose des couleurs et une navigation facile.
  - **htop -u <utilisateur>** : Affiche les processus pour un utilisateur spécifique.
  - **htop -p <PID1>,<PID2>** : Affiche uniquement les processus avec les PIDs spécifiés.
  - **F5** : Affiche les processus sous forme d'arborescence (touche en mode interactif)..
  - **F6** : Permet de trier les processus par colonne (CPU, mémoire, PID, etc.) (touche en mode interactif).
- **kill** : envoie des signaux aux processus, généralement pour les terminer ou les interrompre. En fonction du signal, on peut interrompre, arrêter temporairement ou définitivement un processus.
  - **kill -9 <PID>** : Envoie le signal **SIGKILL** pour forcer la terminaison **immédiate** du processus.
  - **kill -15 <PID>** : Envoie le signal **SIGTERM**, demandant une terminaison "**propre**" du processus.
  - **kill -STOP <PID>** : Suspend temporairement le processus.
  - **kill -CONT <PID>** : Reprend un processus précédemment suspendu.
  - **killall [-u user] [-signal] name ...** : envoyer des signaux à plusieurs processus.
- **nice** : permet de démarrer un processus avec une priorité de CPU spécifique. Les valeurs **nice** vont généralement de **-20** (priorité élevée) à **+19** (priorité faible).
  - **nice -n <valeur> <commande>** : Définit la priorité pour une nouvelle commande.
  - **nice <commande>** : Exécute la commande avec la priorité par défaut (**+10** pour les utilisateurs normaux).

## II- Surveillance et gestion des processus

110

- **renice** (Process Status) : utilisée pour ajuster la priorité des processus déjà en cours. Elle est souvent utilisée en conjonction avec **ps** ou **top** pour repérer un processus et ajuster son niveau de priorité.
  - **renice <valeur> -p <PID>** : Change la priorité d'un processus spécifique.
  - **renice <valeur> -u <utilisateur>** : Change la priorité de tous les processus appartenant à un utilisateur.
- Un processus sous UNIX est créé en faisant appel (directement ou indirectement) à la fonction système (ou primitive système) **fork**.
- Lorsqu'un processus est créé, son numéro est enregistré dans une table système : table des processus. Une fois le processus est terminé, son numéro est supprimé de cette dernière.
- Lorsqu'un processus se termine, il retourne une valeur (status). **0** s'il se termine correctement et une valeur **non nulle** sinon.
  - En sh et bash : **echo \$?**
  - En C-shell : **echo \$status**
- Un processus peut être lancé en avant-plan (foreground) ou en arrière-plan (background)
  - En **avant-plan** : toutes les commandes vues jusqu'à présent ;
  - En **arrière plan** : On rajoute le symbole "&" à la fin de la commande.

## II- Surveillance et gestion des processus

111

Exemple : `find / -name test -print &`

On voit alors apparaître un numéro **PID** du processus qui vient d'être créé et un autre numéro qui est le *numéro de job*. Dans ce cas, le processus s'exécute en arrière-plan et le système donne la main pour exécuter d'autres processus.

- Les processus en avant-plan sont traités de manière **séquentielle** (ou synchrone).
- Les processus en arrière-plan s'exécutent en **parallèle**, plus précisément en multitâche (ou asynchrone)

Remarque : plusieurs processus en arrière-plan → redirection pour éviter la confusion de résultats.

- **wait** oblige l'attente de la fin d'un ou plusieurs processus fils lancés en arrière-plan dans le shell.  
→ Cela permet de synchroniser l'exécution de commandes en veillant à ce qu'une commande s'exécute seulement après que les processus fils spécifiés soient terminés.

```
#!/bin/bash
# Lancer deux commandes en arrière-plan
sleep 30 &
sleep 60 &
# Attendre que toutes les commandes en
# arrière-plan se terminent
wait
echo "Tous les processus fils sont terminés."
```

Exemple 1

```
#!/bin/bash
# Lancer une commande en arrière-plan
sleep 30 &
PID=$!
echo "Attente du processus PID: $PID"
wait $PID
echo "Le processus $PID est terminé."
```

Exemple 2

## III- Tâches en arrière-plan

112

- Les commandes longues ne nécessitant pas de dialogue peuvent être lancées en arrière plan et le système donne la main pour d'autres tâches → **Multitâche** d'Unix/Linux  
Exemple : `cat < entree1 > sortie1 &`
- Les processus en arrière plan ne sont pas interrompus par <Ctrl-C>. → 2 solutions :
  - Sortir de sa session, et dans ce cas tous les processus seront interrompus ;
  - Chercher le numéro de processus, puis taper : `kill %num`
- La commande `nohup` permet de ne pas interrompre un processus lorsqu'on quitte sa session/terminal :  
Exemple : `nohup cat < entree1 > sortie1 &`
- `jobs` : liste les travaux en arrière plan avec leurs numéros (running ou stopped)
- `fg %num_job` : ramène le processus de numéro `num_job` en **avant-plan**.
- `<Ctrl-Z>` : **suspend** (pause) un processus en avant-plan si le terminal le permet (`tty isig`). On peut définir la touche de suspension : `stty susp <Ctrl-Z>`. Dans ce cas, le job suspendu est déplacé en arrière-plan.
- `bg %num_job` : **réactive** le processus suspendu de numéro `num_job` mais reste en **arrière-plan**.
- `fg %num_job` : **réactive** le processus suspendu de numéro `num_job` et devient en **premier-plan**.



## III- Tâches en arrière-plan

113

- **<Ctrl-Z>** permet de déplacer un job en avant-plan de l'état exécution à l'état arrêt.
- **kill -SIGSTOP %num\_job** permet de déplacer un job en arrière-plan de l'état exécution à l'état arrêt.
- **kill -SIGKILL %<numéro\_job>** : Force l'arrêt immédiat (en cours d'exécution ou suspendu).
- **kill %<numéro\_job>** ou **kill PID** : Arrêt normal (envoie le signal **SIGTERM**).
- Pour faire passer un job de l'état avant-plan à l'état arrière-plan, on procède comme suit :
  - On arrête d'abord le job en *avant-plan* par : **<Ctrl-Z>**
  - On détermine ensuite son numéro de job par : **jobs**
  - On ramène enfin, ce job en *arrière-plan* par : **bg %num\_job**
- Pour tuer un job en avant-plan : **<Ctrl-C>** ou **<Ctrl-\\>**.

**Remarque** : On peut désigner un job par le **nom** de la commande ou par une **chaîne** de caractères **include** dans le nom de la commande, au lieu de son numéro.

Exemple : **fg %?sleep**

## IV- Traitement en temps différé

114

- Traitement **programmé** (ou en **temps différé**) : exécution de commandes ou tâches à des moments prédéfinis.
  - Utilité : automatisation, optimisation des ressources, tâches administratives répétitives.
- Unix/Linux dispose d'outils variés pour réaliser un traitement en différé.
- La commande **sleep** permet de suspendre toute exécution pour un utilisateur :
  - En *avant-plan* : **sleep 20 ; ./pgme1** : Pour arrêter l'attente : **<Ctrl-C>**
  - En *arrière-plan* : **(sleep 2d; ./pgme1) &** : Pour arrêter l'attente: on **tue** le processus (ou le job) correspondant mais on supprimera aussi l'exécution de la commande **./pgme1**.
  - **sleep 1h ...** (attente d'une heure), **sleep \$((3\*24 + 5))h** (attente de 3 jours et 5heures)
- La commande **at** permet de lancer l'exécution d'une commande à une **date et/ou heure** donnée :
  - Si **at** n'est pas installée : **sudo apt update** puis **sudo apt install at**

Exemple 1 :

```
at now + 1 min
pwd > fich1
<Ctrl-D>
```

On reçoit un *numéro* de job correspondant ainsi que la *date* et l'*heure* d'exécution prévue.

Exemple 2 :

```
at now + 8 days
./somme < fe > fs
<Ctrl-D>
```

Exemple 3 :

```
atrm 3
```

Permet de supprimer le job *numéro 3* en attente.

## IV- Traitement en temps différé

115

Exemple 4 : `at -l` ou `atq` affiche la liste des processus en différé.

Exemple 5 : `at 17:00 2024-11-23`  
`./pgme1 > fich2`  
`<Ctrl-D>` Le script ***pgme1*** sera exécuté le 23 novembre à 17h00.  
Si fichier de sortie n'est pas indiqué, le résultat sera  
envoyé par *mail*.

**Remarque:** si l'on indique une heure ou une date antérieure à la date courante, on est automatiquement reporté au cycle suivant (jour, semaine, mois ou année suivante).

Exemple 6 : `at 2335`  
... exécution à 23h35  
du jour courant.

Exemple 7 : `at time + val incrément`  
...  
time = *date* ou *now*  
val = valeur entière  
incrément = *minutes, hours, days, weeks, months, years*

par exemple :

`at now + 7 days`  
...

`at 0000 Jan 1 + 9 years`  
...

## IV- Traitement en temps différé

116

- La commande **batch** est utilisée pour exécuter des tâches différées, mais avec une particularité : elle exécute les tâches lorsqu'il y a une **faible charge système**. Elle est similaire à **at**, mais avec une gestion automatique de la charge.

Exemple :

```
batch  
> ./pgme1 > fich2  
> <Ctrl-D>
```

### Droits d'utilisation de la commande **at**

- Le **root** pourra toujours utiliser **at/batch** pour lancer une exécution quelconque.
- Le fichier **/etc/at.allow** contient la liste des utilisateurs éventuels de la commande **at/batch**. Cependant, si un utilisateur n'y figure pas, cela n'applique pas obligatoirement qu'il n'est pas autorisé à utiliser la commande **at**.
- Si un utilisateur qui n'est pas dans **/etc/at.allow**, figure par contre dans le fichier **/etc/at.deny**, ceci implique qu'il n'est pas autorisé à utiliser la commande **at/batch**.
- Si un utilisateur ordinaire ne figure ni dans **/etc/at.allow** ni dans **/etc/at.deny** cela signifie qu'il peut utiliser la commande **at**.
- C'est le **root** qui peut inscrire des noms dans chacun de ces 2 fichiers, à raison d'un nom d'utilisateur par ligne. Si les 2 fichiers **at.allow** et **at.deny** n'existent pas le root pourra les créer.

## V- Mode Cyclique

117

- Le mode cyclique permet une exécution périodique d'une tâche à l'aide de la commande **crontab**.
- Autorisation d'utilisation de **crontab** : 2 fichiers **cron.allow** et **cron.deny**
- L'utilisateur crée un fichier **fich** comprenant les indications de la répétition d'exécution et de la tâche elle-même. **crontab fich** copiera le contenu de fich dans la zone spool du **crontab**.

**Structure d'une ligne dans crontab (6 champs) : \* \* \* \* \* commande**

- Champ1 : la minute (0-59)
- Champ2 : l'heure (0-23)
- Champ3 : le jour du mois (1-31)
- Champ4 : le mois de l'année (1-12)
- Champ5 : le jour de la semaine (0-6 0=dimanche...)
- Champ6 : tâche à exécuter

Chaque champ peut contenir :

- Un **nombre exact** (exemple : 5 pour 5 minutes).
- Un **intervalle** (exemple : 5-10 pour les minutes 5 à 10).
- Un **modulo** (exemple : \*/15 pour toutes les 15 minutes).
- Une **liste de valeurs** séparées par des virgules (exemple : 1,15,30).
- \* : veut dire une valeur quelconque

Exemple : **59 0 \* \* 1-6 tâche1** permet d'exécuter tâche1 tous les jours du **lundi** au **samedi** à **0h59**. Les résultats sont envoyés par mail sauf en cas de redirection.

- **crontab -e** : édite directement le **crontab** de l'utilisateur.

## V- Mode Cyclique

118

- L'éditeur par défaut de *crontab* est "*vim*" ou "*nano*" mais peut le modifier par :  
*export EDITOR=gedit* et pour voir/éditer les commandes cycliques : *crontab -e* puis voir ou éditer le fichier. On l'enregistre avant de le fermer.
- Le root peut programmer une tâche périodique pour un utilisateur donné:

*sudo crontab -u said -e* puis saisir par exemple :

*00 10 \* \* \* root echo "Coucou. Ceci est test de cron"*

Enregistrer le fichier avant de le fermer.

Quand *said* ouvre sa session et tape, dans un terminal, la commande *mail*, il trouvera la liste des mails reçus. Pour lire un mail, il suffit de taper son numéro et de valider par *<Entrée>*. Pour quitter, il suffit de taper la lettre *q*.

## VI- Minuterie systemd

119

- Les **minuterie systemd** permettent de planifier des tâches, tout comme **cron**, mais avec des options supplémentaires comme les déclenchements relatifs aux événements système..
- Elles consistent en deux fichiers principaux :
  - un fichier "**.timer**" (définit quand exécuter la tâche)
  - un fichier "**.service**" (définit quelle tâche exécuter).
- Avantages :
  - Les *minuterie systemd* supportent des spécifications temporelles complexes (ex., une tâche qui s'exécute après le démarrage du système ou après une autre tâche).
  - Les tâches *manquées* (à cause d'une panne ou d'un arrêt du système) peuvent être rattrapées si elles sont marquées comme *persistantes*.
- Les minuterie systemd sont puissantes et flexibles, adaptées aux *besoins modernes*.
- Pour plus de détail consulter le Help : **man systemd.timer**

## I- Introduction

120

- Le **shell** : c'est un **interpréteur** de commandes et un **langage** de programmation puissant.
- Plusieurs shells → plusieurs langages différents.
- Un **Script** est fichier contenant des commandes à exécuter séquentiellement par un shell. Il peut inclure :
  - *des variables, conditions, boucles, et fonctions.*
  - *des instructions pour manipuler des fichiers, gérer des processus, effectuer des calculs, ou même interagir avec l'utilisateur.*
- **Utilité** des scripts shell :
  - **Automatisation des tâches répétitives** : regrouper des commandes répétitives dans un fichier unique, évitant ainsi une saisie manuelle à chaque fois (Sauvegarde automatique, nettoyage automatique, etc.).
  - **Administration système** : Gestion efficace des utilisateurs, processus et serveurs par les administrateurs système.
  - Traitement et analyse des fichiers et des données.
  - **Planification des tâches** : Les scripts shell peuvent être utilisés avec des planificateurs comme **cron** pour exécuter des tâches à des intervalles réguliers.
  - **Sécurité et audit** : Les scripts shell sont utiles pour surveiller les systèmes et détecter des anomalies.
  - **etc.**



## I- Introduction

121

- Jeu d'instructions d'un script :
  - Toutes les commandes Unix ;
  - Instructions d'affectation des variables ;
  - Invocation de programmes exécutables (ou scripts) avec arguments ;
  - Instructions de contrôle (instructions conditionnelles et boucles) ;
  - Instructions d'Entrée/Sortie.
- Il est possible d'écrire et d'invoquer des scripts dans un certain shell tout en utilisant un autre shell en interactif.
- La majorité des instructions écrites pour un script en **Bourne shell (sh)** peuvent être exécutées dans **Bash** (Bourne Again Shell) car **Bash** est conçu pour être compatible avec le Bourne shell, tout en offrant des améliorations et des fonctionnalités supplémentaires.
- *Bash* offre des fonctionnalités supplémentaires qui ne sont pas disponibles dans le Bourne shell, telles que les *extensions de paramètres*, les *structures de contrôle avancées*, les *commandes intégrées supplémentaires*, les *fonctions* puissantes en Bash, etc.
- Si un script commence par **#!/bin/xxx**, il est interprété par le shell **/bin/xxx**.

Exemple : **#!/bin/bash**

### Bonnes pratiques pour écrire un script shell sous Unix/Linux :

1. Définir un Shebang : Spécifier le shell utilisé en tête du script avec `#!/bin/sh` ou `#!/bin/bash`.
2. Ajouter des **commentaires** et rendre le script **lisible** :
  - Ajouter des commentaires pour expliquer les sections et les parties complexes.
  - Utiliser des noms explicites pour les variables et fonctions.
  - Structurer le script avec une indentation cohérente et des lignes vides pour séparer les blocs logiques.
3. Manipuler les **variables** avec précaution :
  - Entourer les variables par des guillemets pour éviter les erreurs avec les espaces ou caractères spéciaux.
  - Utiliser des variables locales dans les fonctions pour éviter les conflits.
4. Gérer les **erreurs** :
  - Activer un comportement strict avec `set -e`, `set -u`, et `set -o pipefail`.
  - Vérifier explicitement les résultats des commandes critiques.
  - Fournir des messages clairs en cas d'échec.
5. Utiliser des **fonctions** pour modulariser
  - Diviser le script en fonctions distinctes et réutilisables.
  - Placer les fonctions en haut ou dans une section dédiée pour une meilleure organisation.

### Bonnes pratiques pour écrire un script shell sous Linux :

6. Vérifier les **entrées** utilisateur :
  - Valider les arguments avant utilisation.
  - Vérifier l'existence des fichiers ou répertoires nécessaires.
  - Fournir des messages d'erreur ou d'aide en cas d'entrées incorrectes.
7. Suivre les standards de **sortie** :
  - Utiliser *stdout* pour les messages normaux et *stderr* pour les erreurs.
  - Respecter les codes de sortie standards (**0** pour le **succès**, **1** ou plus pour les **échecs**).
8. **Sécuriser** le script :
  - Limiter les *permissions* des fichiers pour protéger le script.
9. **Tester** et **déboguer** :
  - Activer le mode débogage avec **set -x** pour tracer les commandes exécutées.
  - Tester le script dans différents environnements et avec plusieurs interpréteurs.
10. **Documenter** le script :
  - Ajouter un **en-tête** descriptif avec le nom, l'**auteur**, et l'**objectif** du script.
  - Implémenter une option **--help** ou fournir une documentation utilisateur.

## II- Programmation en Bourne Shell

124

- Un script Bourne-shell est **portable** sur tout système UNIX ;
- Inconvénient : Les variables ne peuvent être que de type **chaîne de caractères**. Les variables *numériques* et les *tableaux* n'existent pas en Bourne-shell.
- On commence par la ligne `#!/bin/sh`, puis on ajoute les éventuels commentaires dans les lignes suivantes.
- Exemple de script : création avec `vi/nano/gedit/...` du fichier **listf** contenant la commande :

**ls -aCF**

```
chmod a+x listf    # Ajout du droit d'exécution à tout le monde
./listf            # Exécution du script en Bourne Shell et en Bash
sh listf           # une autre manière d'exécuter le script en Bourne shell
                  # indiquer le chemin relatif/absolu si le script n'est pas dans le répertoire courant
```

```
sh -x listf : demande au shell de tracer (tracing) le déroulement du script.
              C'est utile dans le débogage.
```

```
sh -v listf : mode verbeux (verbose). Le shell affichera chaque ligne du script
              sur le terminal avant de l'exécuter
```

**Remarque** : On peut aussi, activer le traçage en haut du script à l'aide de la commande **set -x** et le désactiver à n'importe quel endroit du script à l'aide de la commande **set +x**.

# Chap 8 La programmation shell

## II- Programmation en Bourne Shell

125

### 1- Le passage des paramètres

- Le script **lstf** contient :

```
#!/bin/sh
echo "contenu du répertoire $1"
ls -aCF $1
```

→ Exécution : **lstf /tmp**

- On a 9 variables : **1**, **2**, ..., **9** qui permettent de désigner les paramètres associés à l'invocation du script.
- La commande **shift** permet d'utiliser plus de 9 paramètres :

Exemple 1 :

```
echopar1
#!/bin/sh
echo $1 $2 $3
P1=$1
shift
echo $1 $2 $3
echo $P1
```

→ Exécution :

```
echopar1 un deux trois
un deux trois
deux trois
un
```

Exemple 2 :

```
echopar2
#!/bin/sh
echo $1 $2 $3 $4 $5 $6 $7 $8 $9
Shift
Echo $1 $2 $3 $4 $5 $6 $7 $8 $9
```

→ exécution:

```
echopar2 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 10
```

## II- Programmation en Bourne Shell

126

### 2- Les variables spéciales

- Variables facilitant la programmation :
  - **0** : contient le nom sous lequel le script est invoqué ;
  - **#** : contient le nombre de paramètres passés en argument ;
  - **\*** : contient la liste de paramètres passés en argument ;
  - **?** : contient le code de retour de la dernière commande exécutée ;
  - **\$** : contient le numéro de processus (PID) du shell (en base **10**).

Exemple 1: *echopara un deux trois* → *\$0=echopara, \$1=un, \$2=deux, \$3=tois et \$\* = tous le paramètres.*

Exemple2 : *echopara* contient :

```
#!/bin/sh
echo $0 a été appelé avec $# paramètres
echo qui sont : $*
```

→ exécution:

```
echopara a b c d
echopara a été appelé avec 4 paramètres
qui sont : a b c d
```

# Chap 8 La programmation shell

## II- Programmation en Bourne Shell

127

### 3- Les caractères spéciaux

Ce sont des caractères générateurs de fichiers (\* ? [] {} déjà vus) et sont utilisés dans le passage des paramètres.

Exemple 1 : Si le répertoire courant contient uniquement les fichiers *fich1* et *fich2*, lors de l'exécution de la commande : *ls fi\**, le shell génère la liste *fich1* et *fich2* et la passe en argument à la commande *ls*.

Exemple 2 : répertoire courant contient 2 fichiers *fich1* et *fich2* et un répertoire *fifou* :

```
#!/bin/sh  
echo $1 $2 $3 $4 $5
```

→ exécution

```
test1 fi*  
fich1 fich2 fifou
```

### 4- La commande *test*

Complément de l'instruction *if*. Elle permet de :

- reconnaître les caractéristiques des fichiers et des répertoires ;
- de comparer des chaînes de caractères ;
- comparer algébriquement des nombres.

Syntaxe1 : *test expression*  
ou  
*[ expression ]*

Syntaxe2 : *if [ expression ]*  
*then commandes*  
*fi*

ou *if [ expression ];then*  
*commandes*  
*fi*

**Remarque** : il faut laisser un espace après "[" et avant "]".

## II- Programmation en Bourne Shell

128

### Les expressions les plus utilisées :

- d rep** : vrai si le répertoire **rep** *existe* ;
- f fich** : vrai si le fichier **fich** *existe* ;
- s fich** : vrai si le fichier **fich** *existe* et *non vide* ;
- r fich** : vrai si le fichier **fich** *existe* et accessible en *lecture* ;
- w fich** : vrai si le fichier **fich** *existe* et accessible en *écriture* ;
- x fich** : vrai si le fichier **fich** *existe* et est *exécutable* ;
- z ch** : vrai si la chaîne de caractères **ch** est *vide* ;
- n ch** : vrai si la chaîne de caractères **ch** est *non vide* ;
- c1=c2** : vrai si les 2 chaînes **c1** et **c2** sont *identiques* ;
- c1!=c2** : vrai si les 2 chaînes **c1** et **c2** sont *différentes* ;
- n1 -eq n2** : vrai si les 2 **entiers** sont *égaux*.

Les autres opérateurs relationnels : **-ne**, **-lt**, **-le**, **-gt**, **-ge**

**Remarque** : Les expressions peuvent être niées par "**!**" et combinées par les opérateurs logiques :

**-o** (OU) et **-a** (ET).



### 5- L'instruction case

Elle permet d'effectuer un choix multiple

Syntaxe:

```
case chaine in
motif1)  commande1
;;
motif2)  commande2
;;
...
motifn)  commanden
;;
esac
```

**chaine** peut prendre diverses valeurs :

- un chiffre ;
- une lettre ou un mot ;
- des caractères spéciaux du shell
- une combinaison de ces éléments
- résultat d'une ``commande``
- motif : peut utiliser `*`, `?`, ...

Exemple 1 : script **comptepara**

```
Case $# in
0) echo $0 sans arguments ;;
1) echo $0 possède un argument ;;
2) echo $0 possède deux arguments ;;
*) echo $0 a plus d'arguments ;;
esac
```

### 5- L'instruction case

Exemple 2 : script *ajout*

```
Case $# in
2) if [ ! -f $1 ]
then
    echo le fichier $1 n'existe pas dans le répertoire
elif [ ! -s $1 -o ! -r $1 ]
then
    echo le fichier $1 est vide ou protégé en lecture
elif [ ! -f $2 ]
then
    cat $1 > $2
    echo $0 a copié $1 dans $2
elif [ ! -w $2 ]
then
    echo le fichier $2 est protégé en écriture
else
    cat $1 >> $2
    echo $0 a rajouté $1 en fin de $2
fi
;;
*) cat << EOT
    ajout
    Fonction : le fichier fsource est ajouté au fichier
    fcible si fcible n'existe pas, fsource est copié
    en fcible
    syntaxe : ajout fsource fcible

    EOT
    ;;
esac
```

## II- Programmation en Bourne Shell

131

### 6- Les itérations

#### a- la boucle for

Forme1

```
for variable in ch1 ch2 ... chn
do
    commandes
done
```

Forme2

```
for variable in
do
    commandes
done
```

Forme3

```
for variable in *
do
    commandes
done
```

Forme1 : variable prend ses valeurs dans : *ch1*, *ch2*, ... *chn*.

Forme2 : variable prend ses valeurs dans la liste des *paramètres* du script.

Forme3 : variable prend ses valeurs dans la liste des *fichiers* du répertoire.

Exple1 : echofor1

```
#!/bin/sh
for i in un deux trois
do
    echo $i
done
```

```
./echofor1
un
deux
trois
```

Exple2 : echofor2

```
#!/bin/sh
for i in
do
    echo $i
done
```

```
./echofor2 le
systeme unix
le
systeme
unix
```

Exple3 : echofor3

```
#!/bin/sh
for i in *
do
    echo $i
done
```

```
./echofor3
fich1
fich2
fich3
```

On suppose que le répertoire courant contient les fichiers : *fich1*, *fich2* et *fich3*

## II- Programmation en Bourne Shell

132

### 6- Les itérations

#### a- la boucle for

Exemple 4 : script **lsd**

```
echo "liste des repertoires sous `pwd` "  
echo "=====  
for i in *  
do  
    if [ -d $i ]  
    then  
        echo $i " : repertoire "  
    fi  
done  
echo "=====
```

```
./lsd  
liste des repertoires sous /home/ahmadi/Doss  
=====  
fifou : repertoire  
=====
```

## II- Programmation en Bourne Shell

133

### 6- Les itérations

#### b- Itérations non bornées : **while** et **until**

##### Syntaxe

```
while commandealpha
do
    commandesbeta
done
```

```
until commandealpha
do
    commandesbeta
done
```

Condition vraie si *commandealpha*  
retourne un code *nul*

Exemple1 : le script **wilea** listant les paramètres qui lui sont passés en argument jusqu'à ce qu'il rencontre le paramètre **fin**.

##### Whilea

```
#!/bin/sh
# ce script montre le
# fonctionnement de
# l'instruction while
while [ $1 != fin ] ; do
    echo $1
    shift
done
```

```
./wilea 1 2 3 4 fin 5 6 7
1
2
3
4
```

## II- Programmation en Bourne Shell

134

### 6- Les itérations

#### b- Itérations non bornées : **while** et **until**

Exemple2 : script *untila*

```
#!/bin/sh
# ce script montre l'utilisation
# de l'instruction until
until [ $1 = fin ] ; do
    echo $1
    shift
done
```

```
./untila 1 2 3 fin 4 5
1
2
3
```

Exemple3 : script *while\_until*

```
#!/bin/sh
# ce script illustre l'usage combiné
# de while et de until
while [ $1 != debut ] ; do
    shift
done
shift
until [ $1 = fin ] ; do
    echo $1
    shift
done
```

```
./while_until 1 2 3 debut 4 5 6 fin 7 8
4
5
6
```

## III- Programmation Bash

135

- Le shell **Bash** (Bourne-Again Shell) est un interpréteur de commandes largement utilisé dans l'univers UNIX/Linux. Il a été développé, en **1989**, par **Brian Fox** pour le projet **GNU**.
- Bash combine le meilleur des anciens shells (sh, csh, ksh) tout en surmontant leurs *limitations*. Sa puissance réside dans son équilibre entre des *fonctionnalités avancées*, une *compatibilité* avec les scripts hérités, et une *interactivité* conviviale.
- Bash s'est imposé comme un *shell universel*, adapté à une grande variété de tâches et capable de répondre aux besoins des utilisateurs modernes.
- *Bash* est entièrement compatible avec le *Bourne Shell* (sh) en termes de commandes et de syntaxe. i.e., tous les scripts écrits pour le Bourne Shell peuvent être exécutés dans *Bash* sans modification. Cependant, il y a quelques petites différences :
  - Bash prend en charge les **tableaux**, absents dans sh.
  - Expansions avancées : Les fonctionnalités comme `$(commande)` ou `${variable#pattern}` sont absentes ou limitées dans **sh**.
  - Structures plus complexes : Bash supporte des **structures conditionnelles** et **des boucles plus puissantes**.
  - **Variables et options par défaut** : Dans certains cas, des options comme `set -o` peuvent avoir des comportements différents.
  - **Fonctionnement des erreurs** : Bash peut être plus strict ou fournir des diagnostics plus détaillés.

## III- Programmation Bash

136

**NB:** Si l'on exécute un script strictement conçu pour **sh**, il est recommandé d'invoquer explicitement **Bash** en mode compatibilité (avec **#!/bin/sh** au lieu de **#!/bin/bash**) pour éviter que des extensions spécifiques à *Bash* ne perturbent le fonctionnement attendu.

### 1. Commentaires

- **#** est utilisé pour mettre en commentaire la ligne qui la suit.
- Pour commenter *plusieurs lignes* : Délimiter le texte à commenter à l'aide de **" : << 'FIN' .....FIN"**

```
#!/bin/bash
echo 'Bonjour ILSI!'
: << 'FIN'
Ceci est un programme Multithreading.
Il s'agit d'un script Shell.
Qui permet de créer 2 threads indépendants pour
Realiser un calcul parallèle
...
FIN
...
```



## III- Programmation Bash

137

### 2. Variables shell

- Les variables Bash sont essentiellement des chaînes de caractères, mais, selon le contexte, Bash autorise les opérations arithmétiques et les comparaisons sur les variables.
  - Le facteur déterminant est de savoir si la valeur d'une variable contient uniquement des chiffres.

```
#!/bin/bash
x=2334          # Entier.
let "x += 1"
echo "x = $x"    # x = 2335
echo            # Toujours un entier.
y=${x/23/BB}     # Substitue "BB" à "23".
                # Cela transforme $b en une chaîne.
echo "y = $y"    # y = BB35
declare -i y     # Le déclarer comme entier ne change rien.
echo "y = $y"    # y = BB35
let "y += 1"     # BB35 + 1
echo "y = $y"    # y = 1
echo            # Bash attribue la valeur "entière" d'une chaîne à 0.
z=BB34
echo "z = $z"    # z = BB34
```

# Chap 8 La programmation shell

## III- Programmation Bash

138

- Pour définir une variable X : **X="valeur"** ou **X='valeur'**, **Y=`ls -l`** (résultat de la commande)
- Pour afficher le contenu de X : **echo \$X** ou **echo "\$x"** (si x contient des espaces) ou **\${x}** si **x** est suivi d'autres caractères (exemple : **echo "Le fichier est \${fichier}.txt"**)

Quand utiliser les guillemets doubles ou simples ?

Type de guillemet	Comportement	Quand l'utiliser
Guillemets doubles (")	Permet les expansions (\$, `, \$( )) et les caractères spéciaux.	Lorsque le texte contient des variables ou des commandes à évaluer.
Guillemets simples (')	Traite le contenu comme une chaîne brute.	Lorsque l'on veut protéger tout le contenu de l'interprétation.

Exemple :

```
USER="Said"
x="Bonjour $USER"
y='Bonjour $USER'
Z='Coucou'
A="$Z $USER"
echo $x # Résultat : Bonjour Said
echo $y # Résultat : Bonjour $USER
echo $A # Résultat : Coucou Said
```

- Règles sur les noms des variables :
  - Seules les lettres alphabétiques, les chiffres et les traits de soulignement peuvent être utilisés pour nommer des variables.
  - Le premier caractère du nom de la variable ne peut pas être un chiffre.
  - Il ne peut y avoir d'espaces au milieu d'un nom de variable.
  - Les noms de variables ne peuvent pas contenir de signes de ponctuation.
  - Les mots-clés de Bash ne peuvent pas être utilisés comme noms de variables.
- La commande **readonly** peut définir une variable comme une variable en *lecture seule*, dont la valeur ne peut pas être modifiée.

Exemple :

```
msg='Coucou'
readonly msg
msg='Salut'           # Ne sera pas accepté par le shell
```

- La commande **unset** permet de supprimer des variables.

Exemple :

```
msg='Coucou'
unset msg           # Supprime la variable msg et echo $msg n'affichera rien
```

- Extraction d'une sous-chaîne d'une chaîne : **`${variable:position:longueur}`**

Exemple :

```
chaîne="Bonjour tout le monde"
ss_ch1=${chaîne:8:4}      # Extrait 4 caractères à partir de la position 8 (0-indexé)
echo "$ss_ch1"           # affichera : tout
ss_ch2=${chaîne:8}        # Extrait tous les caractères à partir de la position 8
ss_ch3=${chaîne: -5:5}    # Extrait les 5 derniers caractères
fichier="Doc.pdf"
extension=${fichier: -3}  # Extrait l'extension
echo "$extension"         # affichera : pdf
```

- Variables locales** : Variables visibles uniquement à l'intérieur d'un *bloc* de code ou d'une *fonction*.
- Variables d'environnement** : Si un script définit des variables d'environnement, elles doivent être "exportées" (à l'aide de **`export`**), c'est-à-dire reportées dans l'environnement *local* au script.
  - Un script peut exporter des variables uniquement vers des processus enfants, c'est-à-dire uniquement vers des commandes ou processus que ce script particulier initie.
  - Les processus enfants ne peuvent pas exporter de variables vers les processus parents qui les ont engendrés.

## III- Programmation Bash

141

- Un script shell interprète un nombre comme étant en **base décimale** (base 10), sauf si ce nombre utilise un préfixe ou une notation spéciale.
  - Un nombre précédé de **0** est interprété comme un nombre **octal** (**base 8**).
  - Un nombre précédé de **0x** est interprété comme un nombre **hexadécimal** (**base 16**).
  - Un nombre contenant un caractère **#** est interprété comme **BASE#NOMBRE** (avec des restrictions de plage et de notation).

```
#!/bin/bash
# Bases.sh : Représentation de nombres dans différentes bases.
# Décimal : par défaut
let "dec = 32"
echo "nombre décimal = $dec" # Rien d'inhabituel ici.
# Octal : nombres précédés de '0' (zéro)
let "oct = 032"
echo "nombre octal = $oct" # 26    # Résultat exprimé en base décimale.
# Hexadécimal : nombres précédés de '0x' ou '0X'
let "hex = 0x32"
echo "nombre hexadécimal = $hex" # 50
echo "${((0x9abc))} # 39612    # expansion/évaluation arithmétique avec doubles parenthèses et
                                # le résultat exprimé en base décimale.
```

```
# Autres bases : BASE#NOMBRE
# BASE entre 2 et 64.
# NOMBRE doit utiliser des symboles dans la plage de la BASE, voir ci-dessous.
let "bin = 2#111100111001101"
echo "nombre binaire = $bin"          # 31181
let "b32 = 32#77"
echo "nombre en base-32 = $b32"      # 231
let "b64 = 64#@_"
echo "nombre en base-64 = $b64"      # 4031
# Cette notation ne fonctionne que pour un intervalle limité (2 - 64) de caractères ASCII.
# 10 chiffres + 26 lettres minuscules + 26 lettres majuscules + '@' + '_'
echo
echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA)) # 1295(35*36+35) 170 44822 3375

# Note importante :
# Un chiffre hors plage pour la notation de base spécifiée génère un message d'erreur.
let "bad_oct = 081"
# Message d'erreur partiel :
# bad_oct = 081 : valeur trop grande pour la base (erreur : "081").
# Les nombres octaux utilisent uniquement des chiffres compris entre 0 et 7.
exit $? # Valeur de sortie = 1 (erreur).
```

- Lecture d'une variable : **Syntaxe** : `read [options] [variable1 variable2 ...]`

### Exemples :

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

```
# Exemple 1 : Lecture simple avec une variable
```

```
echo "Exemple 1 : Lecture simple"
```

```
read nom
```

```
echo "Bonjour, $nom"
```

```
# Exemple 2 : Afficher un message avec -p
```

```
echo "Exemple 2 : Lecture avec message"
```

```
read -p "Entrez votre nom : " nom
```

```
echo "Bonjour, $nom"
```

```
# Exemple 3 : Lecture silencieuse (mot de passe)
```

```
echo "Exemple 3 : Lecture silencieuse"
```

```
read -s -p "Entrez votre mot de passe : " motdepasse
```

```
echo -e "\nMot de passe enregistré (mais pas affiché)."
```

```
# Exemple 4 : Lecture limitée à 5 caractères
```

```
echo "Exemple 4 : Lecture limitée à 5 caractères"
```

```
read -n 5 -p "Entrez un code (5 caractères max) : " code
```

```
echo -e "\nCode saisi : $code" # -e pour interpréter les caract d'échappé
```

### Exemples :

#### **# Exemple 5 : Lecture avec un délai (timeout)**

```
echo "Exemple 5 : Lecture avec un délai"
read -t 10 -p "Entrez votre nom (vous avez 10 secondes) : " nom
echo "Nom saisi : ${nom:-Non saisi}"
```

#### **# Exemple 6 : Lecture ligne par ligne avec un tableau**

```
echo "Exemple 6 : Lecture dans un tableau"
echo "Entrez plusieurs valeurs séparées par des espaces : "
read -a valeurs # valeurs est un tableau
echo "Valeurs saisies : ${valeurs[@]}"
```

#### **# Exemple 7 : Lecture avec un délimiteur personnalisé**

```
echo "Exemple 7 : Lecture jusqu'à un délimiteur"
read -d ':' -p "Entrez une valeur (fin avec :) : " valeur
echo "Valeur saisie : $valeur"
```

#### **# Exemple 8 : Lecture multiple**

```
echo "Exemple 8 : Lecture de plusieurs variables"
echo "Entrez votre prénom et nom séparés par un espace : "
read prenom nom
echo "Prénom : $prenom, Nom : $nom"
```



- **Paramètres positionnels** : Arguments passés au script depuis la ligne de commande : **\$0**, **\$1**, **\$2**, **\$3**, etc.
  - **\$0** est le nom du script lui-même, **\$1** est le premier argument, **\$2** le deuxième, **\$3** le troisième, et ainsi de suite (pas de limite stricte prédéfinie)
  - Après **\$9**, les arguments doivent être entourés d'accolades, par exemple, **\${10}**, **\${11}**, **\${12}**. Les variables spéciales **\$\*** et **\$@** désignent tous les paramètres positionnels.

### 3. Calcul des expressions arithmétiques

- En Bash, on peut calculer des expressions arithmétiques à l'aide de plusieurs méthodes, notamment **\$((...))**, **expr**, et **bc** pour des calculs plus complexes.

Exemple 1 :

```
# $((...)) est utilisé dans des calculs simples
a=5
b=3
som=$((a + b)) prod=$((a * b)) diff=$((a - b)) div=$((a / b))
reste=$((a % b))
```

## III- Programmation Bash

146

```
a=2+3
echo $a # affichera: 2+3
echo $((a)) ou echo $(($a)) affichera 5
```

```
a=2+3
echo $(($a*4)) # affichera: 14
echo $((a*4)) # affichera : 20
```

- La variable précédée du **\$** est remplacée par sa valeur littérale avant de faire l'évaluation globale. La variable sans **\$** est évaluée, puis son résultat est placé dans l'expression avant le calcul global.
- Si une variable n'est *pas définie*, est *vide* ou contient une chaîne *non numérique*, elle est interprétée comme une valeur **nulle**.
- Il est aussi possible de réaliser, au sein de la structure **\$(( ))**, une ou plusieurs *affectations* de variables à l'aide de l'opérateur **=**

Exemple :

```
echo $((a = 15 - 6))
echo $a # affichera 9
echo $((b = a * a + a + 1))
echo $b # affichera 91
```

- Les affectations peuvent aussi se faire avec les raccourcis **+=**, **-=**, **=**, **/=**, **<<=**, **>>=**, **&=**, **|=**, **^=**.  
Exemple : **\$((a+=4)) ⇔ \$((a=a+4)) ; \$((a\*=2)) ⇔ \$((a=a\*2))**

## III- Programmation Bash

147

- La structure **(( ))** est utilisée aussi pour évaluer une condition. Elle renvoi **1** si la condition est vraie, et **0** sinon. Les opérateurs de comparaison sont : **>**, **>=**, **<**, **<=**, **==**, **!=**

Exemple : **echo \$((10+7) < 20)** affichera **1**

- Les conditions peuvent être liées par un **ET** logique (**&&**) ou un **OU** logique (**||**) ou encore être niées par **!**

Exemple :

```
a=3 ...  
if (( a < b && b < c )); then  
    echo "$a < $b et $b < $c"  
fi
```

Ou

```
a=3 ...  
if [ $a -lt $b -a $b -lt $c ]; then  
    echo "$a < $b et $b < $c"  
fi
```

**NB :**

- Pour les nombres : On utilise **(( ... ))** ou **[ ... ]** avec des opérateurs comme **-lt**, **-gt**, **-eq**, **etc.**
- Pour les chaînes de caractères : On utilise **[ ... ]** avec des opérateurs comme **=** ou **!=**.
- Pour tester des fichiers : On utilise **[ -e fichier ]** pour vérifier si un fichier existe, ou **[ -d dossier ]** pour vérifier un répertoire.

### Exemple 2 : Utilisation de l'ancienne méthode **expr**

```
a=5
b=3
som=$(expr $a + $b)
prod=$(expr $a \* $b) # Noter l'échappement du symbole '*'
dif=$(expr $a - $b)
div=$(expr $a / $b)
reste=$(expr $a % $b)
```

### Exemple 3 : Utilisation de **bc** (Basic Calculator) pour des *calculs avancés* (nombres **flottants** ou des expressions **complexes**.)

```
a=5
b=3
S=$(echo "$a + $b" | bc)
division=$(echo "scale=2; $a / $b" | bc) # 'scale' définit le nombre de décimales
echo "Somme : $S" # Affichera : Somme : 8
echo "Division(flottante):$division" # Affichera : Division (flottante) : 1.66
```

### Exemple 4 : Calculs dans des scripts *conditionnels*

```
a=7
b=4
if (( a > b )); then
    echo "$a est plus grand que $b"
else
    echo "$b est plus grand que $a"
fi
```

**NB** : Il est également possible d'utiliser la commande **let** pour effectuer des opérations arithmétiques sur des variables :

```
let "a = 5 + 3"
let "a = 10"
let "a += 1"
let "a = 15 > 10"    # a contiendra la Valeur 1 (vrai)
```

### 4. Structures de Tests et Conditions

- Bash dispose de la commande **test**, des opérateurs **[** et **[[** ainsi que de la structure **if/then**.
- Une structure **if/then** teste si le code renvoie un **statut** de sortie égal à **0** (qui signifie "succès" selon la convention **UNIX**). Si c'est le cas, elle exécute une ou plusieurs commandes.
  - La commande Bash **[** est un synonyme de **test** et interprète ses arguments comme des expressions de *comparaison* ou des *tests de fichiers* et renvoie un *statut* de sortie (**0** pour **vrai**, **1** pour **faux**).
  - la commande étendue **[[ ... ]]** effectue des comparaisons de manière plus intuitive.
  - Les constructions **(( ... ))** et **let** évaluent des expressions *arithmétiques* et renvoient un *statut* de sortie **0** (**Vrai**) si la valeur de l'expression est *non nulle*.

- Exemples :

```
(( 0 && 1 ))      # ET logique : résultat=0 (faux)
echo $?          # 1 (faux)

let "val = (( 0 && 1 ))"
echo $val        # 0 (faux)
echo $?          # 1

(( 200 || 11 ))  # OU logique : résultat=1 (vrai)
echo $?          # 0 (vrai)

let "val = (( 5 | 9 ))" # 'OU' binaire : 0101|1001=1101=13
echo $val        # 13
echo $?          # 0
```

### 4. Structures de Tests et Conditions

- **test** : Syntaxe : **test condition** : retourne *vrai* (**0**) si la condition est vraie, sinon *faux* (**1**).

Exemples :

```
test 20 -gt 10          # Retourne 0 (vrai) car 20 est bien plus grand que 10.
echo $?                # affichera 0
test -f /etc/passwd
echo $?                # affichera 0 (vrai) car "/etc/passwd" est un "fichier" existant
```

- **[...]** (crochets simples) : Syntaxe : **[ condition ]**. Elle est équivalente à **test** et est plus lisible et plus courante dans les scripts Bash.

```
[ 20 -gt 10 ]          # Retourne 0 (vrai) car 20 est bien plus grand que 10.
echo $?                # affichera 0

[ "$filiere" = "ILSI" ] && echo "C'est la filière informatique ILSI"
# si la variable filiere est="ILSI" le message "C'est la ... ILSI" est affichée et le
# code de sortie est 0. Sinon rien n'est affiché et le code de sortie est 1.
```

Ici, `$filiere` est mis entre *guillemets* pour éviter une éventuelle erreur, au cas où le contenu de la variable chaîne contient des espaces.

### 4. Structures de Tests et Conditions

- **[[...]]** (crochets doubles) : Syntaxe avancée pour les tests dans Bash.
  - Permet de tester des conditions logiques complexes et offre plus de flexibilité.
  - Supporte des opérateurs comme **&&** (et logique) et **||** (ou logique).
  - Comparaison de chaînes sans échapper les caractères spéciaux.

Exemple :

```
[[ "$string1" == "$string2" ]]  
[[ "$string1" != "$string2" ]]  
[[ "$name" == "ILSI" ]] && echo "C'est la filiere ILSI"
```

#### Avantages de **[[ ... ]]** par rapport à **[ ... ]**

1.

```
str="Bonjour monsieur"  
if [ $str = "Bonjour monsieur" ]; then  
    echo "Les chaînes sont égales."  
else  
    echo "Les chaînes sont différentes."  
fi  
  
# Cela provoque une erreur de syntaxe, car $str est interprété comme deux  
# arguments à cause de l'espace. Avec [[ $str = "Bonjour monsieur" ]] pas d'erreur
```



2.

```
str="linux123"

if [ $str =~ ^linux ]; then
    echo "La chaîne commence par 'linux'."
else
    echo "La chaîne ne commence pas par 'linux'."
fi
# Cela provoque une erreur de syntaxe car [ ... ] ne supporte pas l'opérateur
# =~. Avec [[ $str =~ ^linux ]] pas d'erreur.
```

3.

```
if [ $x = "coucou" ]; then
    echo "La variable est égale à 'coucou'."
else
    echo "La variable est différente."
fi
# Cela provoque une erreur si x n'est pas définie. Avec [[ $x = "coucou" ]] la
# variable non définie sera traitée comme une chaîne vide sans provoquer
# d'erreur. Le message "La variable est différente." sera alors affiché
```

4.

```
if [ $x -lt 10 ] && [ $y -gt 5 ]; then
    echo "Conditions remplies."
else
    echo "Conditions non remplies."
fi
# Multiplication obligatoire des crochets [ ... ] pour des conditions avec des
opérateurs logiques (&& ou ||). Avec [[ $x -lt 10 && $y -gt 5 ]] c'est plus pratique
```

### 4. Structures de Tests et Conditions

- La structure **if** peut tester n'importe quelle **commande**, pas seulement des conditions entre crochets.

Forme 1 :

```
if [ condition ]
then
    statement(s)
fi
```

Exemple :

```
#!/bin/bash
cat "$1"
if [ "$?" -ne "0" ]; then
    echo "Erreur : Impossible de lire $1."
fi
```

À part le saut de ligne après **then**, tous les autres sauts de ligne sont obligatoires ou peuvent être remplacés par des **points-virgules**. Les **espaces** autour des crochets **[ ]** sont également requis.



```
if [ condition ] ; then statement(s) ; fi
```

Forme 2 :

```
if [ condition ]
then
    Commandes1
else
    Commandes2
fi
```

Exemple :

```
#!/bin/bash
if [ ! -r "$1" ]; then
    echo "Erreur : $1 n'est pas un lisible."
else
    cat "$1"
fi
```

**elif** est une contraction de **else if**, et permet de tester plusieurs conditions successives :

Exemple :

```
#!/bin/bash
OS=$(uname -s)
if [ "$OS" = "FreeBSD" ]; then
    echo "Ceci est FreeBSD"
elif [ "$OS" = "CYGWIN_NT-5.1" ]; then
    echo "Ceci est Cygwin"
elif [ "$OS" = "SunOS" ]; then
    echo "Ceci est Solaris"
elif [ "$OS" = "Darwin" ]; then
    echo "Ceci est MacOS"
elif [ "$OS" = "Linux" ]; then
    echo "Ceci est Linux"
else
    echo "Système d'exploitation inconnu"
fi
```

**NB :** La condition après **if** peut être une commande. Si la commande s'exécute sans erreur la commande retourne un *statut* **0** et la valeur de la condition sera alors "Vrai" .

Exemple :

```
if grep -q Coucou fichier # -q : quiet (grep retourne 0 ou 1 sans rien afficher)
then
    echo "Le fichier contient au moins une occurrence de 'Coucou'."
fi
```

## III- Programmation Bash

156

- **Opérateurs de test** de fichiers :

Opérateur	Description	Exemple
-e	Vérifie si le fichier existe.	[ -e "fichier.txt" ]
-f	Vérifie si le fichier existe et est un fichier régulier (non répertoire).	[ -f "fichier.txt" ]
-d	Vérifie si le fichier existe et est un répertoire.	[ -d "/chemin/du/repertoire" ]
-r	Vérifie si le fichier est lisible.	[ -r "fichier.txt" ]
-w	Vérifie si le fichier est inscriptible (possède des permissions d'écriture).	[ -w "fichier.txt" ]
-x	Vérifie si le fichier est exécutable.	[ -x "script.sh" ]
-s	Vérifie si le fichier existe et n'est pas vide (taille > 0).	[ -s "fichier.txt" ]
-L	Vérifie si le fichier est un lien symbolique.	[ -L "lien_symbolique" ]
-h	Vérifie si le fichier est un lien symbolique (synonyme de -L).	[ -h "lien_symbolique" ]
-p	Vérifie si le fichier est un pipe nommé (FIFO).	[ -p "fichier.pipe" ]
-S	Vérifie si le fichier est un socket.	[ -S "socket" ]
-c	Vérifie si le fichier est un fichier spécial de caractère.	[ -c "device" ]
-b	Vérifie si le fichier est un fichier spécial de bloc.	[ -b "device" ]

- **case** fournit une alternative pratique et lisible à l'instruction **if/then/else**, lorsqu'il y a beaucoup de valeurs possibles à tester.

Syntaxe :

```
case "$variable" in
  pattern1)
    # Instructions pour pattern1
    ;;
  pattern2)
    # Instructions pour pattern2
    ;;
  pattern3|pattern4)
    # Instructions pour pattern3 ou pattern4
    ;;
  *)
    # Instructions par défaut (aucune correspondance)
    ;;
esac
```

- **\$variable** : La valeur ou variable à tester.
- **pattern1, pattern2**, etc. : Les motifs (patterns) à comparer. Ils peuvent inclure des métacaractères (par exemple, **\***, **?**, **[...]**) pour des correspondances flexibles. **|** : Permet de spécifier plusieurs motifs alternatifs.
- **\***) : Le cas par défaut si aucun motif ne correspond. Cette section est facultative mais souvent utilisée.
- **;;** : Termine chaque cas.

Exemple :

```
case "$1" in
  *.txt)
    echo "C'est un fichier texte."
    ;;
  *.jpg|*.png)
    echo "C'est une image."
    ;;
  [0-9]*)
    echo "Cela commence par un chiffre."
    ;;
  *)
    echo "Type non reconnu."
    ;;
esac
```

### 5. Boucles

- Bash possède **quatre** structures de boucle différentes : *for*, *while*, *until* et *select*. Chacune d'entre elles a son propre objectif et ses propres forces et faiblesses.
- Boucle *for*

Syntaxe	Description
<code>for var in liste</code>	Itère sur une liste explicite.
<code>for var in {début..fin..incrément}</code>	Itère sur une plage de nombres.
<code>for var in \$(commande)</code>	itère sur les résultats d'une commande.
<code>for ((init; condition; incr))</code>	Boucle style C avec initialisation et incrément.
<code>for var in "\$@"</code>	Itère sur les arguments d'un script.

#### Exemples :

```
#!/bin/bash

# Boucle sur une liste explicite
for couleur in rouge bleu vert
do
    echo "Couleur : $couleur"
done
```

Exemples :

```
# Boucle sur une plage de nombres
for i in {1..5}
do
    echo "Nombre : $i"
done

# Boucle sur les fichiers dans un répertoire
for fichier in $(ls *.sh)
do
    echo "Fichier script : $fichier"
done

# Boucle style C
for ((j=1; j<=3; j++))    # Noter les doubles parenthèses
do
    echo "C-Style boucle : $j"
Done

# Boucle une séquence générée
for i in $(seq 1 2 10)    # De 1 à 10 avec un pas de 2
do
    echo "Valeur : $i"
done
```



- Boucle **while**

Syntaxe	Description
<code>while [ condition ]</code>	Teste une condition avant chaque itération.
<code>while commande</code>	Exécute une boucle tant que la commande réussit.
<code>while true</code> ou <code>while :</code>	Crée une boucle infinie.
<code>while IFS= read -r ligne</code>	Lit un fichier ligne par ligne.
<code>while [ condition1 ] &amp;&amp; [ condition2 ]</code>	Combine plusieurs conditions avec AND/OR.

### Exemples :

```
#!/bin/bash

compteur=1
fichier="test.txt"

# 1. Boucle avec condition simple
while [ $compteur -le 5 ] # ou bien : while (( $compteur <=5 ))
do
    echo "Compteur : $compteur"
    ((compteur++))
done
```

Exemples :

**# 2. Boucle pour attendre l'existence d'un fichier**

```
while [ ! -f "$fichier" ]; do  
    echo "Le fichier $fichier n'existe pas. Création en cours..."  
    sleep 2  
    touch "$fichier" # Simuler la création du fichier  
done  
echo "Le fichier $fichier a été trouvé !"
```

**# 3. Lire un fichier ligne par ligne**

```
while IFS= read -r ligne; do  
    echo "Ligne lue : $ligne"  
done < "$fichier"
```

**# 4. Boucle infinie avec une condition d'arrêt**

```
compteur=1  
while true; do  
    echo "Itération $compteur"  
    ((compteur++))  
    if [ $compteur -gt 3 ]; then  
        echo "Condition remplie. Arrêt de la boucle."  
        break  
    fi  
done
```

**# Exemple de Boucle infinie**

```
while :  
do  
    echo "Boucle infinie."  
    sleep 1  
done
```

- Boucle **until**

Syntaxe	Description
<code>until [ condition ]</code>	Exécute les commandes tant que la condition est <b>fausse</b> .
<code>until commande</code>	Exécute tant que la commande retourne un statut de <b>non zéro</b> (c'est-à-dire une erreur).
<code>until false</code>	Crée une <b>boucle infinie</b> (elle s'arrête uniquement lorsqu'on utilise <code>break</code> ).
<code>until [ condition ] ; do ... done</code>	Syntaxe alternative pour écrire des boucles <b>en ligne</b> avec une seule commande.

### Exemples :

```
#!/bin/bash
# Initialisation
compteur=1
fichier="test.txt"

# 1. Boucle avec une condition simple
until [ $compteur -gt 5 ]
do
    echo "Compteur : $compteur"
    ((compteur++))
done
```

## III- Programmation Bash

164

- Boucle **until**

### Exemples :

```
# 2. Attente jusqu'à ce qu'un fichier existe
echo "=== Boucle 2 : Attendre qu'un fichier soit créé ==="
until [ -f "$fichier" ]
do
    echo "En attente de $fichier..."
    sleep 2
    touch "$fichier" # Création du fichier pour tester
done
echo "Le fichier $fichier a été trouvé."

# 3. Boucle infinie avec condition d'arrêt
compteur=1
until false
do
    echo "Itération $compteur"
    ((compteur++))
    if [ $compteur -gt 3 ]; then
        echo "Condition remplie, arrêt de la boucle."
        break
    fi
done
```

- Commande ***select*** : permet d'afficher un ***menu interactif*** à l'utilisateur. Elle permet de choisir parmi plusieurs options de manière simple et rapide. C'est un outil particulièrement pratique pour les scripts interactifs.

Syntaxe	Description
<code>select variable in liste</code>	Affiche un menu avec les éléments de liste.
<code>select variable in \$(commande)</code>	Utilise la sortie d'une commande comme liste.
<code>select variable in "\${tableau[@]}"</code>	Utilise un tableau Bash pour la liste des options.
<code>break</code>	Sortir de la boucle select.
<code>case</code>	Utilisé pour gérer chaque option du menu de manière détaillée.

### Exemple 1 :

```
#!/bin/bash
# Le choix peut être fait plusieurs fois <Ctrl+C> pr sortir
echo "Choisissez un fruit :"
select fruit in Pomme Orange Banane Fraise
do
    echo "Vous avez choisi : $fruit"
done
```

### Exécution :

```
Choisissez un fruit :
1) Pomme
2) Orange
3) Banane
4) Fraise
#? 2
Vous avez choisi : Orange
#?
```

## III- Programmation Bash

166

### Exemple 2 :

```
#!/bin/bash
echo "Choisissez un fruit : "
select fruit in Pomme Orange Banane Fraise
do
    echo "Vous avez choisi : $fruit"
    break # Sortir de la boucle après sélection
done
```

1. Un menu interactif avec 4 options est affiché.
2. L'utilisateur entre le numéro de l'option souhaitée.
3. La variable fruit contient la valeur choisie.
4. La boucle s'arrête après la première sélection grâce à break.

### Exemple 3 :

```
#!/bin/bash
echo "Choisissez une couleur : "

select couleur in Rouge Bleu Vert Jaune Quitter
do
    if [ "$couleur" == "Quitte" ]; then
        echo "Sortie du programme."
        break
    fi
    echo "Vous avez choisi : $couleur"
done
```

1. Un menu est affiché avec une option supplémentaire "Quitte".
2. La boucle continue jusqu'à ce que l'utilisateur sélectionne "Quitte".
3. Le programme affiche chaque choix sélectionné.

## III- Programmation Bash

167

### Exemple 4 :

```
#!/bin/bash
echo "Choisissez un fichier dans le répertoire courant :"
select fichier in $(ls)
do
    echo "Vous avez sélectionné le fichier : $fichier"
    break
done
```

1. ***\$(ls)*** génère dynamiquement une liste des fichiers dans le répertoire courant.
2. L'utilisateur peut choisir un fichier par son numéro.

### Exemple 5 :

```
#!/bin/bash
# Déclarer un tableau (voir la section 6 sur les tableaux)
Noms=("Amine" "Saïd" "Nadia" "Ihssane")
echo "Choisissez un nom :"
select nom in "${Noms[@]}"
do
    echo "Vous avez choisi : $nom"
    break
done
```

1. ***Noms*** est un tableau contenant plusieurs chaînes.
2. ***\${Noms[@]}*** permet de parcourir tous les éléments du tableau.

### Exemple 6 :

```
#!/bin/bash

echo "Choisissez une option :"

select option in Démarrer Arrêter Redémarrer Quitter
do
    case $option in
        "Démarrer")
            echo "Vous avez choisi : Démarrer"
            ;;
        "Arrêter")
            echo "Vous avez choisi : Arrêter"
            ;;
        "Redémarrer")
            echo "Vous avez choisi : Redémarrer"
            ;;
        "Quitter")
            echo "Sortie du script."
            break
            ;;
        *)
            echo "Option invalide. Veuillez réessayer."
            ;;
    esac
done
```

1. L'utilisation de case permet de traiter chaque option de manière spécifique.
2. Si l'utilisateur entre un numéro invalide, un message d'erreur est affiché.



## III- Programmation Bash

169

### 6. Tableaux

- En Bash, les tableaux sont des structures de données permettant de stocker plusieurs valeurs dans une seule variable. Ils sont très utiles pour organiser et gérer des listes de données.
- **Déclaration explicite** : On peut déclarer un tableau vide en utilisant la commande intégrée **declare** avec l'option **-a** : **declare -a tab**
- **Création implicite** avec des éléments : On peut directement attribuer des valeurs à un tableau :  
**tab=(valeur1 valeur2 valeur3)**

Les indices (index) sont automatiquement attribués, en commençant par **0**.

- **Création avec un index spécifique** : On peut attribuer des valeurs à des indices spécifiques d'un tableau :

**NB** : Les indices n'ont pas besoin d'être contigus :

```
tab[0]="première"
```

```
tab[2]="troisième"
```

- **Tableau associatif** (à partir de Bash version **4.0**) : Un tableau associatif utilise des clés au lieu d'indices numériques :

```
declare -A tab_assoc
```

```
tab_assoc[cle1]="valeur1"
```

```
tabl_assoc[cle2]="valeur2"
```

## III- Programmation Bash

170

### 6. Tableaux

- Pour afficher la valeur d'un élément du tableau, on utilise l'indice correspondant :  
`echo ${tab[0]} # Affiche la première valeur`
- Pour Afficher tous les éléments, on utilise @ ou \* :  
`echo ${tab[@]} # Affiche tous les éléments`  
`echo ${tab[*]} # Affiche tous les éléments`
- Pour afficher uniquement les indices :  
`echo ${!tab[@]} # Affiche tous les indices du tableau`
- Pour connaître la taille d'un tableau :  
`echo ${#tab[@]} # Nombre total d'éléments`
- Pour ajouter un élément à la fin d'un tableau :  
`tab+=(valeur4 valeur5) # Ajoute des valeurs au tableau existant`
- On modifie un élément en réassignant sa valeur avec l'indice correspondant :  
`tab[1]="nouvelle valeur"`
- Pour supprimer un élément précis : `unset tab[1] # Supprime l'élément avec l'indice 1`
- Pour vider complètement un tableau : `unset tab`

## III- Programmation Bash

171

### 6. Tableaux

Exemple 1 :

```
#!/bin/bash
# Exemple simple : manipulation de fichiers
fichiers=("fichier1.txt" "fichier2.txt" "fichier3.txt")
for fichier in "${fichiers[@}"; do
    touch "$fichier"
    echo "Création de $fichier"
done
```

Exemple 2 :

```
#!/bin/bash
# Exemple d'un tableau associatif
declare -A capitales
capitales["Maroc"]="Rabat"
capitales["Canada"]="Ottawa"
capitales["Espagne"]="Madrid"
for pays in "${!capitales[@}"; do
    echo "La capitale de $pays est ${capitales[$pays]}"
done
```

## III- Programmation Bash

172

### 6. Tableaux

#### Limitations des tableaux en Bash :

- Les tableaux Bash ne supportent que des valeurs **simples** (chaînes de caractères ou nombres).
- Bash n'a pas de types complexes comme les objets ou les tableaux multidimensionnels nativement.
- Pour les tableaux à plusieurs dimensions, on utilise une approche de *simulation* avec des **clés** sous forme de chaînes.

Exemple :

```
tableau_2D["ligne1,colonne1"]="valeur1"  
tableau_2D["ligne1,colonne2"]="valeur2"  
echo ${tableau_2D["ligne1,colonne1"]} # Affiche "valeur1"
```

## III- Programmation Bash

173

### 7. Fonctions

- Comme les "vrais" langages de programmation, Bash possède des **fonctions**, bien que dans une implémentation quelque peu limitée.
- Une fonction est une sous-routine, un bloc de code qui implémente un ensemble d'opérations, une "boîte noire" qui exécute une tâche spécifiée.

- Syntaxe :

```
function nom_de_fonction {  
    # Corps de la fonction  
}
```

ou

```
nom_de_fonction() {  
    # Corps de la fonction  
}
```

**NB :** Il est aussi possible de définir une fonction sur une seule ligne, mais cela peut nuire à la lisibilité. Exemple : `Salut () { echo "Bonjour tout le monde"; }` (point virgule à la fin)

- Les fonctions sont **appelées/déclenchées**, simplement en invoquant leurs *noms*. Un appel de fonction est équivalent à une commande.
- La **définition** de la fonction doit *précéder* son premier appel. Il n'existe pas de méthode de "déclaration" de la fonction, comme par exemple en C/C++.
- Si des **arguments** sont nécessaires, ils sont fournis après le nom de la fonction :  
`nom_de_fonction arg1 arg2 arg3`
- Les **arguments** passés à une fonction sont accessibles de la manière suivante :  
`$1, $2, $3, ...` : représentent les arguments individuels. `$@` : représente tous les arguments comme une liste.  
`$#` : donne le nombre d'arguments passés à la fonction.

## III- Programmation Bash

174

### 7. Fonctions

- En Bash, les fonctions peuvent être définies et utilisées de différentes manières en fonction du contexte : soit directement dans un **terminal** pour un usage interactif, soit dans des **scripts** pour un usage automatisé :
  - Une fonction définie directement dans le *terminal* interactif peut être appelée plusieurs fois (comme une commande) et reste en mémoire uniquement pour la durée de la session shell active.
  - Pour rendre une fonction disponible dans toutes les sessions, on l'ajoute dans le fichier `~/.bashrc` ou `~/.bash_profile` et on recharge la configuration après modification à l'aide de la commande : `source ~/.bashrc`
  - Les fonctions définies dans un *script* ne sont accessibles que dans ce script. Elles sont utilisées pour réduire la redondance et structurer le code.
  - Pour rendre une fonction définie dans un *shell* accessible à des **sous-shells** ou à des **scripts** exécutés dans ce shell, on utilise la commande : `export -f nom_de_fonction` juste après la définition de la fonction.
  - Une pratique courante est de définir des fonctions dans un **fichier distinct** et de les charger dans d'autres **scripts** ou dans l'environnement **shell** via la commande `source`. Cela permet de :
    - Favoriser la **réutilisation** et la **modularité**.
    - **Centraliser** des fonctions fréquemment utilisées dans un **fichier unique**.

### 7. Fonctions

#### Exemple 1 :

```
# Définition et utilisation dans le terminal
salut() {
    echo "Bonjour, $1 !"
}
# Pour l'appel, on tape dans le terminal :
salut "Mohamed" # ceci affichera : Bonjour Mohamed !
```

#### Exemple 2 :

```
# Pour rendre une fonction définie dans un shell accessible à des sous-shells
# On tape dans le terminal le code suivant :
ma_fonction() {
    echo "Fonction exportée !"
}
# puis on tape dans le terminal la commande suivante :
export -f ma_fonction
```

#### Exemple 3 :

```
# Pour rendre une fonction persistante on l'ajoute au fichier ~/.bashrc
# Dans ~/.bashrc ajouter la code suivant :
salut() {
    echo "Bonjour, $1 !"
}
# puis on tape dans le terminal la commande: source ~/.bashrc
```

### Exemple 4 :

```
# Définition d'une fonction dans un script
#!/bin/bash

sauvegarder_fichier() {      # Définir une fonction
    local fichier=$1
    local destination=$2
    cp "$fichier" "$destination"
    echo "Le fichier $fichier a été sauvegardé dans $destination."
}

sauvegarder_fichier "/etc/hosts" "/tmp"    # Appel de la fonction
```

### Exemple 5 :

```
# On crée le fichier mes_fonction.sh qui contiendra les 2 fonctions suivantes :

salut() {
    echo "Bonjour, $1 !"
}

somme() {
    echo $(( $1 + $2 ))
}
```

```
# Dans un script ou un shell interactif on tape les instructions suivantes :

source fonctions_utiles.sh
salut "Amine"          # Appel de la 1ère fonction
resultat=$(somme 5 10)  # Appel de la 2ème fonction
echo "Résultat : $resultat"
```



# Chap 7 Les Editeurs de Textes

## I-Présentation générale

177

### 1- Editeurs de texte ligne

- L'éditeur *ed* est le plus ancien. Il a été conçu aux *Bell Laboratories*
- L'éditeur *ex* est une extension de *ed*, développé à *Berkeley*

**Rque** : En fait on n'utilise plus les éditeurs ligne de nos jours. Toutefois, ces 2 éditeurs interviennent encore , indirectement, dans l' éditeurs page *vi* et dans l'éditeur non interactif *sed*.

### 2- Editeur non interactif

- L'éditeur non interactif (stream editor) standard *sed* est très utilisé, non pas spécialement en tant d'éditeur de texte classique mais plutôt comme outils de manipulation(ou filtre).

### 3- Editeurs pleine page

- L'éditeur plein écran *vi* (de Berkeley) est livré avec tous les systèmes UNIX actuels→standard
- *Emacs* est un autre éditeur pleine page assez répandu et plus élaboré que *vi*. Il a été développé par *MIT*.

# Chap 7

# Les Editeurs de Textes

## II-L'éditeur vi

178

- La commande : `# vi fich1` permet d'afficher le fichier fich1 s'il existe sinon elle affichera un écran vide. Dans les 2 cas le curseur sera positionné en haut à gauche.
- Sous vi on est toujours sous l'un des 2 modes : *commande* (par défaut) ou *insertion*.
- Pour insérer un texte on doit se mettre en mode *insertion* ;
- Pour passer du mode commande au mode *insertion*, on tape l'une des touches :

**i** : insertion avant le curseur

**I** : insertion au début de la ligne courante

**o** : insertion après la ligne courante

**a** : insertion après le curseur

**A** : insertion à la fin de la ligne courante

**O** : insertion avant la ligne courante

### Rques :

- On entre les caractères graphiques normalement au clavier et barre d'espace pour un *blanc* ;
- On rentre les caractères non graphiques en les faisant précéder de `<Ctrl-v>`
- On efface le dernier caractère avant le curseur, par `<BACK SPACE>` ou `<DEL>`
- On va à la ligne avec la touche `<RETURN>`
- On passe en mode commande avec `<ESC>` ou `<Ctrl-c>`

- Comment sortir du fichier et revenir au shell ? 2 cas :

i) Avec sauvegarde du contenu de la mémoire tampon : `ZZ` ou `:wq` ou `:x`

Rques: - les commandes précédées de ":" sont empruntées à l'éditeur ligne *ex*.  
- *vi* utilise une mémoire tampon → toutes les modifications peuvent être sauvegardées ou annulées.

ii) Sans sauvegarde du contenu du tampon : `:q!` (dans tous les cas) ou  
`:q` (si le fichier n'est pas modifié) .

- On peut sauvegarder le contenu du tampon sans quitter *vi* :

`:w` le contenu du tampon sera sauvegardé dans le fichier ouvert par *vi*

`:w fich_bis` enregistre le tampon sous *fich\_bis* (utilisé quand *vi* est lancé sans argument ou lorsqu'on veut créer une autre copie dans *fich\_bis* qui n'existait pas)

`:w! fich_bis` remplace l'ancien contenu de *fich\_bis* par le contenu du tampon

`:5,18 w fich` sauvegarde partiellement le contenu du tampon (lignes de 5 à 18) dans *fich*

### ➤ Déplacement du curseur :

Déplacement	Touche
Gauche-droite	flèche droite ou l ou <espace>
Droite-gauche	flèche gauche ou h ou <backspace>
Haut-bas	flèche basse ou j
Bas-haut	flèche haute ou k
Fin de ligne courante	\$
Début de la ligne suivante	+ ou <return>
Début de ligne courante	o
Début de la ligne précédente	-
La ligne numéro n	nG
Début du text en entier	1G
Dernière ligne	G
Fin d'une phrase	)
Début d'une phrase	(
Fin d'un paragraphe	}
Début d'un paragraphe	{

#### Fin de phrase :

. <espace><espace>  
 ?<espace><espace>  
 !<espace><espace>  
 Ligne blanche

#### Fin de paragraphe :

Ligne blanche

#### Positionnement sur le mot suivant : w

#### Positionnement sur le mot précédent : b

#### Positionnement à la fin du mot courant : e

- La commande `<Ctrl-g>` ou `":f"` indique :
  - le nom du fichier utilisé ;
  - Eventuellement si le fichier vient d'être modifié ;
  - le numéro de ligne courante ;
  - le numéro total de lignes du fichier ;
  - le pourcentage de lignes, ramené à la ligne courante

Exple : `« fich1 » [modified] line 2 of 5 --40%--`

- La commande `":set nu"` permet de numéroter les lignes du fichier et `":set nonu"` réalise l'opération inverse
- La commande `"u"` permet d'annuler l'effet de la dernière commande ;
- La commande `"U"` permet de remettre une ligne qui vient d'être modifiée en son état initial
- Pour appeler une commande shell sans quitter `vi` : `":!commande"` . Exple : `:!date`  
on reviendra à l'éditeur en tapant `<Entrée>`
- Pour aller dans le shell sans quitter : `":!sh"` . On reviendra au `vi` en tapant : `<Ctrl-d>`
- Pour annuler une requête non encore exécutée : `<ESC>` ou `<DEL>`

### ➤ Suppressions :

Suppression	Touches
Du caractère repéré par le curseur	x
Du caractère à gauche du curseur	X
De n caractères à partir du curseur	nx
De la ligne courante	dd
De n lignes à partir de la ligne courante	ndd
Du reste de la ligne à partir du curseur	D
Du mot suivant le curseur	dw
Du mot précédant le curseur	db
De la fin de la phrase	d)
Du début de la phrase	d(
De la fin du paragraphe	d}
Du début du paragraphe	d{

**dw** et **db** tiennent compte de la ponctuation, alors que **dW** et **dB** n'en tiennent pas

**Pour supprimer une partie de texte, de la ligne n1 à la ligne n2 :**  
*:n1,n2 d*

**Suppression de ligne courante et passage automatique en mode insertion :** *cc*

**Suppression du caractère courant et passage automatique en mode insertion :** *s*

**On peut récupérer les 9 dernières suppressions par :** *" "1p"* , *" "2p"* , ...

### ➤ Récupération des suppressions :

La commande **p** permet de récupérer :

- après la position courante du curseur, les éléments supprimés par une commande **x**, **X**, **nx**, **dw**, **db**, **dW**, ou **dB**, **d** , **d(** , **d}** , **d{** , précédente
- après la ligne courante, les éléments supprimés par une commande **dd** , **ndd** ou encore **:n1,n2 d** précédente

La commande **P** a le même effet que **p**, mais les éléments seront récupérés, soit avant la position courante du curseur, soit avant la ligne courante.

### ➤ Recherches :

- La commande **/coucou** permet de rechercher la chaîne **"coucou"** à partir de la position du curseur, vers l'avant. **?coucou** fait la recherche vers l'arrière (majus différent de minus)
- La commande **fc** permet de rechercher, en avant, dans la ligne courante le caractère **"c"**
- La commande **FC** permet de rechercher, en arrière, dans la ligne courante le caractère **"c"**
- Si l'on veut exécuter de nouveau la commande de recherche dans le sens initialement prévu, on tape **"n"** (pour le sens inverse on tape **"N"**)
- La commande **:set ic** permet de confondre les maj et les min à la recherche et **:set noic** annule cet effet.

### ➤ Substitutions :

- la commande "**r***C*" permet de remplacer le caractère pointé par la souris par le caractère "*C*" et on reste dans le mode *commande* ;
- la commande "**R**" permet de remplacer tous les caractères, à partir de la position du curseur, par les caractères saisis. On met fin à cette commande en tapant **<ESC>**
- la commande "**cw** *mot* **<ESC>**" permet de remplacer le mot pointé par la souris par le mot "*mot*"
- la commande "**c** *texte* **<ESC>**" permet de remplacer le reste de la ligne qui suit le curseur par le texte saisi "*texte*". Pour remplacer toute la ligne courante : "**cc** *ligne***<ESC>**"
- La commande "**:s/chaîne1/chaîne2/**" permet de remplacer la première occurrence de la chaîne *chaîne1*, dans la ligne courante, par *chaîne2*
- La commande "**:s/chaîne1/chaîne2/g**" permet de remplacer toutes les occurrences de la chaîne *chaîne1*, dans la ligne courante, par *chaîne2*
- La commande "**:n,\$s/chaîne1/chaîne2/g**" permet de remplacer toutes les occurrences de la chaîne *chaîne1*, du texte à partir de la ligne *n*, par *chaîne2*



### ➤ Copie de texte :

- la commande "**Y**" ou "**yy**" permet de copier, dans un tampon particulier, la ligne courante
- la commande "**nY**" permet de copier, dans un tampon particulier, n lignes du texte à partir de la position courante ;
- La commande "**:n1,n2 y**" permet de copier dans le tampon, les lignes **n1** à **n2**
- On récupère le contenu du tampon par la commande "**p**" ou "**P**"
- La commande "**:nr fich\_ext**" permet d'introduire le contenu du fichier externe **fich\_ext**, dans vi à partir de la ligne **n**
- La commande "**:.r fich\_ext**" permet d'introduire le contenu du fichier externe à partir de la ligne courante.