Table of Contents

Introduction - 01

Le cours de **Systèmes d’exploitation I – Théorie** a pour objectif principal de familiariser l’étudiant avec l’utilisation d’un système informatique et avec certaines fonctionnalités fondamentales du noyau d’un système d’exploitation. Les systèmes informatiques étant très variés, le cours se concentre volontairement sur un environnement de type PC classique, équipé d’un clavier, d’un écran et d’une souris, afin de fournir une base commune et concrète à tous les étudiants.

Un système d’exploitation, souvent abrégé en **OS** pour *Operating System*, est le logiciel central qui permet à l’utilisateur et aux programmes d’interagir avec le matériel de l’ordinateur. Les notions abordées dans ce cours sont communes aux principaux systèmes d’exploitation actuels tels que Windows, macOS, Linux, Unix ou encore Android. Même si ces systèmes diffèrent dans leur interface ou leur usage, ils reposent tous sur des principes similaires.

Dans le cadre de ce cours, le système **Linux** est privilégié comme support d’étude. Créé en 1991 par Linus Torvalds, Linux est un système open source, ce qui signifie que son code source est public et accessible. Cette caractéristique en fait un excellent candidat pour l’apprentissage, car il est possible d’en étudier le fonctionnement interne en détail. De plus, Linux est un système extrêmement répandu, notamment dans le monde des serveurs, où il équipe plus de 90 % des plus

grandes infrastructures informatiques mondiales. Les exemples présentés durant le cours théorique s'appuieront donc principalement sur Linux.

Il est important de distinguer certains termes essentiels du vocabulaire des systèmes d'exploitation. Le **noyau**, ou *kernel*, est la partie centrale du système d'exploitation ; il est responsable de la gestion des ressources matérielles comme le processeur (CPU), la mémoire vive (RAM) ou les périphériques. Une **distribution Linux** correspond à un ensemble cohérent de logiciels construits autour du noyau Linux, comme Ubuntu, Debian ou Fedora. Enfin, le terme système d'exploitation peut parfois désigner uniquement le noyau, et parfois l'ensemble formé par le noyau et les logiciels qui l'entourent.

Le cours est structuré en deux grandes parties. La première partie est consacrée à l'interaction avec un système d'exploitation. L'étudiant y apprend notamment à créer, modifier, supprimer et retrouver des fichiers et des dossiers. La seconde partie s'intéresse davantage au noyau du système, en abordant des notions telles que la gestion des processus, la gestion de l'espace disque, l'allocation des ressources et les problématiques de blocage. Ces notions théoriques sont complétées par des séances de laboratoire, qui permettent de mettre en pratique les concepts vus en cours dans différents environnements.

L'interaction avec le système d'exploitation se fait soit par une interface graphique, soit par une interface en ligne de commande, appelée **console**. Le cours privilégie l'utilisation de la console, car l'interface graphique est souvent plus lente et peu adaptée à l'automatisation. La ligne de commande permet de reproduire facilement des séquences d'actions, de réduire les erreurs et d'automatiser des tâches complexes à l'aide de scripts. Bien que son apprentissage puisse sembler plus difficile au début, elle devient rapidement plus efficace avec l'habitude.

La console repose sur l'utilisation d'un **terminal** et d'un programme appelé **shell**, qui interprète les commandes saisies par l'utilisateur et les exécute. Le shell le plus courant sous Linux est *bash*, mais il en existe d'autres. Les commandes suivent une syntaxe précise : elles commencent par le nom d'un programme, suivi éventuellement d'options et de paramètres. L'utilisateur dispose également de nombreux outils d'aide, comme les pages de manuel (*man*), qui constituent une ressource essentielle pour comprendre le fonctionnement des commandes.

Enfin, le cours introduit la notion fondamentale de **noyau**. Dans une vision simplifiée, le noyau est responsable de l'allocation de la mémoire aux processus, de la gestion du processeur afin que plusieurs programmes puissent s'exécuter de manière concurrente, du chargement du système et des programmes en mémoire, de la communication avec le matériel via des pilotes et de l'organisation des données sur le disque à l'aide des systèmes de fichiers. Le noyau est donc un élément central du système d'exploitation, et son étude constitue une partie essentielle du cours.

Arborescence et système de fichiers – 02

Un **système de fichiers** décrit la manière dont les fichiers et les dossiers sont stockés et organisés sur un disque. Il n'existe pas de méthode unique et naturelle pour organiser les données sur un

support de stockage, ce qui explique l'existence de plusieurs systèmes de fichiers comme FAT, NTFS, HFS ou EXT. Ces systèmes se distinguent par la taille maximale des fichiers et des partitions, le nombre maximal de fichiers, la gestion des droits, la journalisation ou encore la manière dont l'espace disque est alloué. Un même système d'exploitation peut proposer plusieurs systèmes de fichiers différents.

Du point de vue de l'utilisateur, on ne s'intéresse pas encore aux détails internes du stockage, mais plutôt à la façon dont les fichiers et dossiers sont visibles et manipulables. Les systèmes de fichiers sont généralement organisés sous forme **d'arborescence**, c'est-à-dire une structure hiérarchique partant d'un point unique appelé la **racine**, notée `/` sous Linux. À partir de cette racine, on trouve des dossiers qui peuvent contenir à la fois des fichiers et d'autres dossiers.

Pour visualiser cette structure hiérarchique, on peut utiliser la commande suivante, qui affiche graphiquement l'arborescence d'un dossier :

```
tree dossier1
```

Cette commande montre clairement que plusieurs fichiers peuvent porter le même nom tant qu'ils se trouvent dans des dossiers différents.

Chemins absolus et chemins relatifs

Chaque fichier est identifié de manière unique par son **chemin**, c'est-à-dire la suite de dossiers qui le séparent de la racine. Lorsqu'un chemin commence par `/`, on parle de **chemin absolu**, par exemple :

`/home/mba/brol`

`/etc/passwd`

À l'inverse, un **chemin relatif** est exprimé par rapport au dossier courant et ne commence pas par `/`. Par exemple :

`brol`
`mba/brol`
`../mcd/sonprojet`

Le dossier courant correspond à l'endroit où l'on se situe dans l'arborescence. Il peut être affiché avec la commande :

`pwd`

Pour se déplacer dans l'arborescence, on utilise la commande :

`cd chemin`

Il est possible d’enchaîner plusieurs commandes sur une même ligne en les séparant par un point-virgule :

```
cd /home/mba/mondossier1 ; ls mondossier2
```

Dossier personnel et raccourcis de navigation

Chaque utilisateur possède un **dossier personnel**, appelé *home*, qui contient ses fichiers privés. Il se situe généralement dans `/home/nom_utilisateur`. Le caractère `~` est un raccourci qui représente automatiquement le dossier personnel de l’utilisateur courant. Ainsi, les commandes suivantes sont équivalentes si l’utilisateur est `mba` :

```
ls ~
ls /home/mba
```

Le symbole `.` représente le dossier courant, tandis que `..` représente le dossier parent. Ces notations permettent de construire des chemins relatifs très pratiques :

```
.
..
./rep
../mcd/rep
../..
```

Métadonnées et consultation des fichiers

Les fichiers et dossiers possèdent des **métadonnées**, c’est-à-dire des données qui décrivent les données, comme le nom, la taille, le propriétaire ou les droits d’accès. Ces informations peuvent être affichées avec la commande :

```
ls -l
```

Pour consulter l’arborescence ou le contenu des dossiers, plusieurs commandes sont couramment utilisées :

```
tree /home
ls /home
ls -lR /home
find /home
less /home/mba/brol
```

Un exemple de recherche avec `find` :

```
find /home -name a
```

Liens symboliques

Un **lien symbolique** est un fichier spécial qui pointe vers un autre fichier sans en faire une copie. Il est créé avec la commande :

```
ln -s /etc/passwd pass
```

On peut vérifier qu'il s'agit bien d'un lien symbolique avec :

```
ls -l pass
```

Les deux commandes suivantes afficheront exactement le même contenu :

```
less pass
less /etc/passwd
```

Les liens symboliques facilitent notamment le partage de fichiers entre plusieurs utilisateurs sans duplication.

Modification et manipulation de fichiers

Linux fournit de nombreuses commandes pour créer, modifier et supprimer des fichiers et des dossiers. Les principales sont :

```
touch file
mv file rep
mv file rep/nom
rm file
mkdir rep
rmdir rep
```

Pour éditer des fichiers texte, on utilise généralement des éditeurs simples comme :

```
vi file
nano file
```

Recherche avancée de fichiers

La commande `find` permet de rechercher des fichiers selon de nombreux critères, comme la date de modification ou la taille. Par exemple :

```
find ~ -mtime -1
```

pour trouver les fichiers modifiés depuis moins d'un jour dans le home, ou :

```
find / -size +1M
```

pour rechercher les fichiers de plus de 1 MiB sur tout le système.

Partitions et montage

Les systèmes de fichiers résident sur des **partitions**, qui sont des portions de disque. Sous Linux, les partitions portent des noms comme `sda1`, `sda2`, `sdb1`, etc. Chaque partition contient au maximum un système de fichiers. Pour être utilisable, celui-ci doit être **monté**, c'est-à-dire rattaché à un dossier de l'arborescence principale.

Exemples de commandes de montage :

```
mount /dev/sda6 /
mount /dev/sda8 /home
```

```
mount /dev/sda1 /windows/C
mount /dev/sda9 /home/mtk/test
```

Pour visualiser les partitions et leurs points de montage, on utilise :

```
lsblk
```

Gestion des partitions

Les systèmes d'exploitation fournissent des commandes administrateur permettant de gérer les partitions et les systèmes de fichiers. Par exemple, pour partitionner un disque :

```
fdisk /dev/sda
```

Et pour formater une partition en NTFS :

```
mkfs.ntfs /dev/sda2
```

Formater une partition consiste à y inscrire les métadonnées nécessaires au système de fichiers choisi.

Les commandes Linux – 03

Une **commande Linux** est une instruction tapée dans le shell afin de demander au système d'exécuter une action. Toute commande suit une structure bien définie composée d'un nom, éventuellement suivi d'options et d'arguments. Cette structure permet au shell d'interpréter correctement l'action demandée et de transmettre les informations nécessaires au programme concerné.

Le **nom de la commande** peut désigner soit une commande interne au shell, comme `cd`, soit le chemin vers un fichier exécutable, par exemple `/usr/bin/bash`, soit simplement le nom d'un fichier exécutable présent dans l'un des répertoires connus du système. Pour savoir où se trouve l'exécutable correspondant à une commande, on utilise la commande suivante :

```
which commande
```

La variable PATH

Le shell ne cherche pas une commande n'importe où sur le système. Il la recherche uniquement dans les répertoires listés dans une variable d'environnement appelée PATH. Cette méthode rend l'exécution plus rapide et plus sûre. Une variable d'environnement est une variable contenant une chaîne de caractères et connue par les processus du système.

Pour afficher le contenu de la variable PATH, on utilise l'expansion avec le caractère $ et la commande echo :

```
echo $PATH
```

Le résultat est une liste de dossiers séparés par des deux-points, par exemple :

```
/home/user0/bin:/usr/local/bin:/usr/bin:/bin
```

Les arguments (ou paramètres)

Les arguments, aussi appelés paramètres, précisent sur quoi la commande doit agir. Par exemple, dans la commande suivante :

```
mv oldname newname
```

oldname correspond au fichier à renommer et newname à son nouveau nom. Le nombre et l'ordre des arguments sont imposés par chaque commande. Par convention, les arguments sont placés après les options.

Les options

Les options modifient le comportement d'une commande. Elles sont souvent précédées d'un tiret. Par exemple :

```
mv -i oldname newname
```

L'option -i demande une confirmation avant d'écraser un fichier existant. Certaines options prennent elles-mêmes un paramètre, comme dans l'exemple suivant :

```
tar -c -f backup.tar ~/Documents
```

Cette commande peut aussi s'écrire sous une forme condensée :

```
nginx
```

```
tar -cf backup.tar ~/Documents
```

Les versions modernes des options utilisent souvent la forme longue, avec deux tirets :

```
--option
```

```
--option=valeur
```

La page de manuel d'une commande, accessible avec man, décrit précisément sa syntaxe et ses options :

```
man mv
```

Les globs (jokers)

Les globs sont des caractères spéciaux interprétés par le shell permettant de désigner plusieurs fichiers en une seule fois. Par exemple, pour supprimer tous les fichiers commençant par temp, on peut utiliser :

```
rm temp*
```

Le caractère ? représente exactement un seul caractère quelconque. Par exemple :

pgsql

temp?

correspond à temp1 ou tempa, mais pas à temp10.

Le caractère * représente une suite de caractères quelconques, éventuellement vide. Par exemple :

*.txt

correspond à note.txt ou a.txt, mais pas à note.pdf.

Les globs peuvent être combinés pour former des motifs plus complexes :

n-*-s-*.txt

Traitement des globs par le shell

Lorsqu'un glob apparaît dans une commande, ce n'est pas la commande elle-même qui l'interprète, mais le shell. Celui-ci recherche les fichiers correspondant au motif dans le dossier courant ou dans le dossier indiqué, puis remplace le glob par la liste des fichiers trouvés. Si aucune correspondance n'est trouvée, l'argument est laissé tel quel. Ce mécanisme est totalement transparent pour la commande exécutée, à condition qu'elle accepte plusieurs noms de fichiers en argument.

Commandes Linux courantes

Linux fournit un grand nombre de commandes standards permettant de manipuler les fichiers, les dossiers et les processus. Parmi les plus utilisées, on trouve :

mkdir

rmdir

touch

ls

cat

less

cd

pwd

mv

cp

rm

du

date

file

diff

whoami

groups

D’autres commandes sont orientées vers la gestion des processus, des droits ou le traitement de texte :

ps

top

kill

chmod

chgrp

find

grep

head

tail

cut

sort

uniq

Pour chacune de ces commandes, il est fortement recommandé de consulter les pages de manuel afin de comprendre leur fonctionnement précis et leurs nombreuses options :

man commande

Utilisateurs et droits sous Linux – 04

Les systèmes d’exploitation modernes permettent à **plusieurs utilisateurs** de partager une même machine, parfois simultanément, comme c’est le cas sur les serveurs. Cette possibilité soulève des questions essentielles de **sécurité**, de **confidentialité** et de **contrôle d’accès**. Linux intègre donc un mécanisme complet de gestion des utilisateurs et des droits afin de protéger les données et d’éviter qu’un utilisateur ne puisse interférer avec le travail d’un autre.

La notion d’utilisateur

Un **utilisateur** sous Linux est identifié par plusieurs éléments. Il possède tout d’abord un **login**, qui est un nom lisible et mémorisable par un humain. En interne, le système utilise plutôt un **User ID (UID)**, qui est un numéro unique plus facile à manipuler pour le système d’exploitation. Chaque

utilisateur dispose également d'un **mot de passe** et d'un **dossier personnel**, appelé *home*, dans lequel sont stockés ses fichiers.

Pour connaître l'utilisateur actuellement connecté, on utilise la commande :

```
whoami
```

Pour afficher l'identifiant numérique de l'utilisateur :

```
id --user
```

Sous Linux, les utilisateurs sont définis dans le fichier `/etc/passwd`, qui est lisible par tous :

```
cat /etc/passwd
```

Sécurité des mots de passe

Historiquement, les mots de passe chiffrés des utilisateurs étaient stockés dans le fichier `/etc/passwd`. Cette solution n'étant plus suffisamment sécurisée, ils ont été déplacés dans le fichier `/etc/shadow`, qui n'est lisible que par l'administrateur du système. Un utilisateur classique n'a pas le droit de le consulter :

```
cat /etc/shadow
```

Cette commande échoue avec un message de permission refusée, ce qui montre l'importance du contrôle des accès.

Propriétaire d'un fichier

Chaque fichier et chaque répertoire appartient à **un utilisateur**, appelé le **propriétaire**. Il s'agit en général de l'utilisateur qui a créé le fichier. Le propriétaire est visible grâce à l'affichage détaillé des fichiers :

```
ls -l
```

Le propriétaire joue un rôle central dans la gestion des droits d'accès aux fichiers.

Les permissions sur les fichiers

Chaque fichier possède trois types de permissions. La permission **read (r)** permet de lire le contenu du fichier. La permission **write (w)** autorise la modification de son contenu. Enfin, la permission **execute (x)** permet d'exécuter le fichier s'il contient du code.

Les permissions sont visibles avec la commande suivante :

```
ls -l test
```

Elles peuvent être modifiées à l'aide de la commande `chmod`. Par exemple :

```
chmod g+w test
```

Cette commande ajoute le droit d'écriture au groupe associé au fichier. Il est également possible de définir les droits à l'aide d'une notation numérique :

```
chmod 640 test
```

Les groupes

Chaque utilisateur appartient à **un groupe principal** et éventuellement à plusieurs **groupes secondaires**. Les groupes permettent de gérer des droits collectifs sur les fichiers. Pour afficher les groupes auxquels appartient l'utilisateur courant, on utilise :

```
groups
```

Les groupes sont définis dans le fichier `/etc/group` :

```
cat /etc/group
```

Chaque fichier appartient également à **un seul groupe**, et les membres de ce groupe bénéficient des droits définis pour celui-ci. Le groupe associé à un fichier peut être modifié avec la commande :

```
chgrp groupe fichier
```

Droits sur les répertoires

Les répertoires possèdent eux aussi des permissions, mais leur signification est légèrement différente. Le droit **read** permet de connaître le contenu du répertoire, par exemple avec `ls`. Le droit **write** autorise la modification du contenu du répertoire, c'est-à-dire l'ajout ou la suppression de fichiers. Le droit **execute** permet d'entrer dans le répertoire et de le parcourir.

Une erreur fréquente consiste à croire que supprimer un fichier nécessite un droit d'écriture sur le fichier lui-même. En réalité, la suppression d'un fichier dépend du **droit d'écriture sur le répertoire qui le contient**, et non sur le fichier.

Le super utilisateur (root)

Linux possède un utilisateur spécial appelé **super utilisateur**, **administrateur** ou **root**. Son login est `root` et il dispose de droits illimités sur le système, indépendamment des permissions des fichiers. Il est chargé de l'administration de la machine, comme la création d'utilisateurs, l'installation de logiciels ou le formatage des disques.

Il existe deux manières principales d'obtenir des privilèges administrateur. La première consiste à devenir root avec la commande :

```
su root
```

Cette commande nécessite le mot de passe de root. La seconde méthode, plus courante et plus sûre, consiste à exécuter une seule commande avec les droits administrateur via :

```
sudo commande
```

Dans ce cas, c'est le mot de passe de l'utilisateur courant qui est demandé, à condition qu'il soit autorisé à utiliser `sudo`.

Exemple d’utilisation de sudo et su

Un utilisateur classique ne peut pas lire le fichier `/etc/shadow` :

```
cat /etc/shadow
```

En revanche, avec `sudo`, il peut exécuter la commande avec les droits administrateur :

```
sudo cat /etc/shadow
```

Il est également possible de devenir root temporairement :

```
su root
```

Puis de quitter la session administrateur avec :

```
exit
```

Cette gestion des utilisateurs, des groupes et des droits constitue l’un des piliers de la **sécurité sous Linux** et permet un partage sûr et contrôlé des ressources sur une même machine.

Redirections et tuyaux sous Linux – 04

Sous Linux, chaque programme s’exécute en communiquant avec son environnement à travers des **entrées et sorties standards**. Par défaut, l’entrée standard correspond au clavier, tandis que la sortie standard et la sortie d’erreur sont affichées à l’écran. Le shell permet de modifier ce comportement grâce aux **redirections** et aux **tuyaux**, qui constituent des mécanismes fondamentaux pour automatiser et combiner des commandes.

Entrée et sorties standards

Tout processus dispose de trois canaux de communication. L’**entrée standard**, appelée *stdin*, est utilisée pour recevoir des données, généralement depuis le clavier. La **sortie standard**, ou *stdout*, est utilisée pour afficher les résultats normaux du programme. Enfin, la **sortie d’erreur**, ou *stderr*, est réservée aux messages d’erreur. Par défaut, ces trois flux sont reliés au terminal.

Ces flux sont identifiés par des numéros :

- 0 pour l’entrée standard,
- 1 pour la sortie standard,
- 2 pour la sortie d’erreur.

Lorsqu’un programme écrit sur sa sortie standard, il n’a aucune connaissance de l’endroit réel où cette sortie est envoyée. C’est le shell qui décide si les données vont vers l’écran, un fichier ou un autre programme.

Principe de la redirection

La **redirection** consiste à demander au shell de connecter une entrée ou une sortie standard à un fichier au lieu du terminal. Cette opération est totalement transparente pour le programme exécuté, qui ne sait pas qu'il est redirigé.

Par exemple, la commande suivante redirige la sortie de `ls` vers un fichier :

```
ls > out
```

Le fichier `out` est alors créé (ou vidé s'il existe déjà) et contient le résultat de la commande. Si l'on souhaite ajouter le résultat à la fin du fichier sans l'effacer, on utilise :

```
ls >> out
```

Il est également possible de rediriger l'entrée standard depuis un fichier :

```
commande < file
```

Dans ce cas, le programme lit ses données dans le fichier au lieu du clavier.

Redirection des erreurs

Par défaut, les messages d'erreur s'affichent à l'écran. Il est possible de les rediriger vers un fichier spécifique à l'aide de la redirection associée à *stderr* :

```
commande 2> erreurs
```

Pour rediriger à la fois la sortie standard et la sortie d'erreur vers un même fichier, on utilise :

```
commande &> fichier
```

Fichier ou entrée standard

Certaines commandes peuvent fonctionner soit avec un fichier passé en paramètre, soit en lisant directement l'entrée standard. C'est le cas de la commande `cat`. Par exemple :

```
cat fichier
```

affiche le contenu du fichier sur la sortie standard. En revanche :

```
cat
```

lit les données saisies au clavier jusqu'à ce que l'utilisateur indique la fin de l'entrée avec la combinaison de touches `<Ctrl>d`.

La commande suivante a le même effet que `cat fichier`, mais utilise une redirection :

```
cat < fichier
```

Dans ce cas, c'est le shell qui ouvre le fichier et le connecte à l'entrée standard avant de lancer la commande.

Exemples de redirections

Les redirections permettent de réaliser facilement des opérations courantes sur les fichiers. Par exemple, pour copier le contenu d'un fichier dans un autre :

```
cat < file1 > file2
```

Pour ajouter le contenu d'un fichier à la fin d'un autre :

```
cat < file1 >> file2
```

Pour séparer les résultats normaux et les messages d'erreur d'une commande :

```
find / -name test > res 2> err
```

Pour ajouter une ligne à la fin d'un fichier :

```
echo "- Étudier" >> todo
```

Les tuyaux (pipes)

Les **tuyaux**, représentés par le caractère `|`, permettent de relier la sortie standard d'une commande à l'entrée standard d'une autre. Ce mécanisme est particulièrement puissant car il permet de combiner plusieurs commandes simples pour réaliser une tâche complexe.

Un tuyau n'a de sens que si la première commande produit des données sur *stdout* et si la seconde est capable de lire des données sur *stdin*. Par exemple :

```
find ~ -name test | less
```

Ici, la liste produite par `find` est transmise directement à `less` pour un affichage paginé.

Enchaînement de commandes et filtres

Linux repose sur le principe **KISS** (*Keep It Simple, Stupid*). De nombreuses commandes sont conçues comme des **filtres** : elles lisent des données sur l'entrée standard, les transforment, puis les renvoient sur la sortie standard. Chaque commande fait une tâche simple, mais leur enchaînement permet de résoudre des problèmes complexes.

Par exemple :

```
cat test | grep mot | grep autre
```

Chaque commande filtre progressivement les données produites par la précédente.

Exemples complets avec tuyaux

Pour compter le nombre d'utilisateurs actuellement connectés :

```
who | cut -d " " -f 1 | sort | uniq | wc -l
```

Pour afficher les trois dossiers les plus volumineux du dossier personnel :

```
du -h ~/* | sort -hr | head -3
```

Dans ces exemples, chaque commande joue un rôle précis, et les tuyaux permettent de transmettre les données d'une étape à la suivante.

Les **redirections** et les **tuyaux** sont des outils essentiels sous Linux. Ils permettent d'exploiter pleinement la puissance du shell en automatisant le traitement de données, en combinant des commandes simples et en construisant des chaînes de traitement efficaces et élégantes.

Le noyau d'un système d'exploitation – 05

Le **noyau** est la composante logicielle centrale d'un système d'exploitation. Il s'agit d'un programme chargé en mémoire vive au démarrage de la machine et qui reste actif tant que le système fonctionne. Les termes *système d'exploitation* et *noyau* sont parfois utilisés comme des synonymes, mais en réalité, Linux désigne strictement le noyau, tandis que **GNU/Linux** désigne l'ensemble formé par le noyau et les outils logiciels qui l'entourent.

On peut définir le noyau à travers les services fondamentaux qu'il rend : il agit comme une **machine étendue**, un **gestionnaire de ressources**, il permet le **démarrage** du système, fournit des **appels système**, gère les **interruptions**, les **processus** et les **démons**, et garantit la sécurité grâce aux niveaux de privilèges.

Le noyau comme machine étendue

À l'origine, une machine informatique simplifiée se compose uniquement d'un processeur (CPU) et de mémoire (RAM). Le noyau transforme cette machine minimale en une **machine étendue** en y intégrant la gestion des périphériques comme le clavier, la souris, l'écran, la carte réseau, le disque dur ou encore l'imprimante.

Cette gestion repose sur des **pilotes** (*drivers*), qui sont des morceaux de code intégrés au système. Les pilotes servent d'intermédiaires entre les programmes et le matériel. Ils facilitent le dialogue, vérifient que l'accès est autorisé, préservent l'état du périphérique et masquent les détails techniques du matériel. Sans pilotes, un programme pourrait écrire n'importe où sur le disque ou interagir dangereusement avec le matériel.

Appels système et interruptions

Lorsqu'un programme souhaite accéder à un périphérique, il ne peut pas le faire directement. Il effectue un **appel système**, c'est-à-dire une demande officielle au noyau. À l'inverse, lorsqu'un événement matériel survient (pression d'une touche, fin d'une lecture disque), le matériel déclenche une **interruption**, qui force l'exécution d'un code du noyau.

Dans les deux cas, c'est toujours le noyau qui prend la main. Par exemple, lors de la lecture d'un fichier, le processus demande la lecture via un appel système, puis est bloqué pendant que le disque travaille. Lorsque la lecture est terminée, le périphérique déclenche une interruption, le noyau débloque le processus, et celui-ci peut reprendre son exécution.

Les appels système

L'utilisateur n'interagit jamais directement avec le noyau. Il passe par des programmes, qu'ils soient graphiques ou en ligne de commande. Ces programmes utilisent des **appels système**, qui sont des fonctions fournies par le noyau et identifiées par un numéro.

Les appels système sont documentés dans les pages de manuel de section 2. Par exemple :

```
man 2 write
```

Quelques appels système courants sont :

- `read` : lire des données (fichier, clavier)
- `open` : ouvrir ou créer un fichier
- `fork` : créer un nouveau processus
- `exit` : terminer un processus
- `mkdir` : créer un répertoire
- `chmod` : modifier les droits d'accès

Pour lire un fichier, un programme effectue typiquement les appels suivants :

```
fd = open(pathname, flags)
read(fd, buf, count)
close(fd)
```

Les descripteurs `0`, `1` et `2` correspondent respectivement à l'entrée standard, la sortie standard et la sortie d'erreur, que tu as déjà rencontrées avec les redirections.

Le noyau comme gestionnaire de ressources -

Le noyau joue également le rôle de **gestionnaire de ressources**. Certaines ressources sont dites **non partageables**, car elles ne peuvent être utilisées simultanément par plusieurs processus. C'est le cas du processeur, de la mémoire vive ou encore d'une imprimante.

Pour gérer ces ressources, le noyau utilise différents mécanismes. L'**ordonnanceur** attribue le processeur aux processus de manière équitable. Le **chargeur** place les programmes en mémoire. Le **spooler d'impression** gère les files d'attente pour l'impression. Ces composants garantissent que chaque processus obtient les ressources nécessaires sans bloquer les autres.

Le démarrage du système

Au démarrage de la machine, la mémoire vive est vide. Or, aucun programme ne peut s'exécuter s'il n'est pas chargé en RAM. Le démarrage se fait donc en plusieurs étapes.

La première étape repose sur le **firmware**, appelé BIOS (ancien) ou UEFI (récent). Ce programme est stocké dans une mémoire non volatile et initialise le matériel avant de charger la suite du processus.

La deuxième étape est assurée par le **bootloader**, comme GRUB ou LILO sous Linux. Il est chargé de proposer éventuellement un choix de système d'exploitation, puis de charger le noyau en mémoire et de lui transmettre le contrôle.

Explorer le noyau et le système

Il est possible d'interroger le système pour obtenir des informations sur le matériel et le noyau. Sous Linux, le répertoire virtuel `/proc` fournit une interface vers l'état interne du système.

Par exemple, pour connaître la quantité de mémoire vive :

```
cat /proc/meminfo | grep MemTotal
```

Pour connaître le processeur :

```
cat /proc/cpuinfo | grep "model name"
```

Pour afficher le nom et la version du noyau :

```
uname -sr
```

Les rings et les privilèges

Le noyau doit disposer de plus de privilèges que les programmes utilisateurs. Cette séparation est assurée par le mécanisme des **rings** du processeur. Le code du noyau s'exécute en **ring 0**, le mode le plus privilégié, avec accès total à la mémoire et au matériel. Les programmes utilisateurs s'exécutent en **ring 3**, un mode non privilégié, qui limite leur accès aux ressources.

Cette séparation garantit la stabilité et la sécurité du système. Les programmes doivent passer par les appels système pour accéder au matériel, ce qui permet au noyau de contrôler toutes les opérations sensibles.

Les démons

Les **démons** sont des programmes qui font partie du système d'exploitation et qui s'exécutent en permanence en arrière-plan. Ils démarrent avec le système et fournissent des services essentiels, comme la gestion du réseau, des impressions ou des tâches planifiées.

Pour afficher les démons en cours d'exécution :

```
ps -eo "euser,pid,comm"
```

Ces processus sont généralement exécutés avec des privilèges élevés et assurent le bon fonctionnement continu du système.

Récapitulatif

En résumé, le système d'exploitation repose sur deux grands éléments. D'une part, du **code**, comprenant les appels système, les gestionnaires d'interruptions, l'ordonnanceur, les chargeurs et les démons. D'autre part, des **données**, stockées en mémoire et sur disque, comme les tables internes et les structures de gestion.

Le noyau est donc le cœur du système : il relie le matériel aux logiciels, gère les ressources, assure la sécurité et rend possible l'exécution de tous les programmes.

Technologies de stockage : HDD et SSD – 07

Il existe principalement deux grandes technologies de stockage. Les **HDD** (*Hard Disk Drive*) sont des disques magnétiques composés de plateaux en rotation rapide. Des têtes de lecture et d'écriture se déplacent mécaniquement au-dessus de ces plateaux pour accéder aux données. Cette mécanique induit des temps de latence, rend les disques sensibles aux chocs, mais permet un coût par gigaoctet relativement faible.

Les **SSD** (*Solid State Drive*), quant à eux, sont constitués de mémoires à semi-conducteurs. Ils ne comportent aucune pièce mobile, ce qui les rend beaucoup plus rapides, silencieux et résistants aux chocs. En contrepartie, leur coût est plus élevé et les cellules mémoire ont une durée de vie limitée en nombre d'écritures. Il est courant de combiner les deux technologies, par exemple un SSD pour le système et un HDD pour le stockage massif des données.

Organisation physique d'un disque dur

Un disque dur est constitué de plusieurs **plateaux** fixés sur un axe central, chacun recouvert d'une surface magnétique. Les données sont organisées en **pistes**, qui sont des cercles concentriques, eux-mêmes découpés en **secteurs**, qui constituent la plus petite unité physique de stockage, généralement de 512 octets. Un **cylindre** correspond à l'ensemble des pistes accessibles sans déplacer les têtes de lecture.

Accéder à des données situées sur le même cylindre est plus rapide, car cela évite les déplacements mécaniques des têtes. Pour cette raison, les systèmes cherchent à regrouper les données liées afin d'améliorer les performances.

Abstraction du stockage : le LBA

Quelle que soit la technologie utilisée, les secteurs du disque sont numérotés à partir de zéro. Cette numérotation s'appelle le **LBA** (*Logical Block Address*). Le système d'exploitation travaille avec ces adresses logiques, qui sont ensuite traduites en adresses physiques adaptées au matériel sous-jacent.

Le rôle du système de fichiers

Un disque ou une partition n'est fondamentalement qu'une suite de bytes regroupés en secteurs. Le **système de fichiers** définit comment ces bytes sont organisés pour former des fichiers et des répertoires. Il fournit une vue logique du disque et doit assurer trois fonctions essentielles : allouer l'espace aux fichiers, localiser les fichiers existants et gérer l'espace libre.

Allocation contiguë

L'**allocation contiguë** consiste à stocker les données d'un fichier dans une suite de secteurs consécutifs sur le disque. Cette méthode permet des lectures très rapides, car les déplacements des

têtes sont minimisés. Cependant, elle souffre fortement de la **fragmentation externe** : à mesure que des fichiers sont créés, supprimés et modifiés, l'espace libre se fragmente, rendant difficile l'allocation de grands fichiers sans déplacer des données existantes. Cette approche est donc surtout adaptée aux supports en lecture seule comme les CD ou les DVD.

Allocation par blocs

La majorité des systèmes de fichiers modernes utilisent une **allocation par blocs**. Un **bloc** (ou cluster) est une unité logique composée d'un nombre fixe de secteurs. Les blocs sont numérotés et constituent l'unité d'allocation et d'accès des fichiers. Les blocs d'un fichier ne sont pas nécessairement contigus sur le disque, ce qui simplifie l'ajout et la suppression de données.

Par exemple, un fichier de 1 byte occupe malgré tout un bloc entier sur le disque. On peut l'observer avec les commandes suivantes :

```
echo -n a > f
ls -l f
du -h f
```

Même si le fichier est très petit, il consomme un bloc complet, typiquement de 4 KiB. Lorsqu'un fichier dépasse la taille d'un bloc, un nouveau bloc est alloué, ce qui peut entraîner une **fragmentation interne** (espace inutilisé dans le dernier bloc) et une **fragmentation des fichiers** (blocs dispersés).

Choix de la taille des blocs

La taille des blocs est un compromis. De grands blocs réduisent la taille des structures d'index mais augmentent la perte d'espace interne. De petits blocs limitent la perte d'espace mais augmentent la fragmentation et le nombre d'accès disque. Ce choix est effectué lors du **formatage** et dépend du type d'utilisation prévu.

Formatage et répertoires

Le **formatage** consiste à créer un système de fichiers sur une partition. Cette opération ajoute des métadonnées décrivant la structure du système, comme la taille des blocs ou l'emplacement des tables internes. Une partie de l'espace disque est donc réservée à ces métadonnées.

Les **répertoires** sont des structures particulières qui contiennent non pas les données des fichiers, mais leurs métadonnées : nom, attributs et informations de localisation. Ils jouent un rôle central dans la navigation et la localisation des fichiers.

Localisation des fichiers : FAT et inodes

Pour retrouver les blocs d'un fichier, deux grandes techniques existent. La première est la **table d'index FAT**, utilisée notamment sous Windows. Une table unique décrit pour chaque bloc le numéro du bloc suivant dans le fichier. Cette table peut devenir très volumineuse pour de grandes partitions.

La seconde technique est utilisée par Unix et Linux à travers les **inodes**. Un inode est une structure de données associée à chaque fichier et contenant ses métadonnées, y compris la liste des blocs utilisés. Les inodes sont stockés dans une table et sont beaucoup plus compacts que les tables FAT.

Gestion de l’espace libre

Lorsqu’un fichier est créé ou agrandi, le système doit identifier des blocs libres. Deux méthodes principales existent. La **liste de blocs libres** maintient une liste des numéros de blocs disponibles, mais devient désordonnée avec le temps. La **table de bits**, utilisée par ext2 ou exFAT, représente l’état de chaque bloc par un bit indiquant s’il est libre ou occupé. Cette méthode facilite la recherche de blocs contigus et améliore les performances.

Fragmentation

La gestion de l’espace libre est étroitement liée à la **fragmentation**. La fragmentation externe concerne la dispersion de l’espace libre, la fragmentation des fichiers correspond à la dispersion des blocs d’un même fichier, et la fragmentation interne représente l’espace perdu à l’intérieur des blocs partiellement utilisés. Une bonne gestion vise à limiter ces phénomènes afin de maintenir de bonnes performances.

Arborescence unique sous Unix

Contrairement à d’autres systèmes, Unix et Linux utilisent une **arborescence unique** pour tous les systèmes de fichiers. Plusieurs partitions ou périphériques peuvent être intégrés à cette arborescence grâce à l’outil de montage. Ainsi, tout le stockage est accessible depuis une seule hiérarchie de dossiers, ce qui simplifie l’utilisation et l’administration du système.

Les processus sous Linux – 08

Un **programme** est une suite d’instructions stockée dans un fichier exécutable. Tant qu’il n’est pas lancé, il reste une entité passive. Lorsqu’il est chargé en mémoire vive pour être exécuté, il devient un **processus**. Un processus est donc un programme en cours d’exécution, associé à un contexte d’exécution comprenant notamment de la mémoire, des ressources et un identifiant unique.

Identifiant de processus (PID)

Sous Linux, chaque processus est identifié de manière unique par un **PID** (*Process ID*), qui est un nombre entier attribué par le système. Cet identifiant permet au noyau et aux utilisateurs de référencer précisément un processus donné.

Pour afficher les processus en cours d’exécution avec leur PID et le nom de la commande associée, on utilise :

```
ps -o pid,comm
```

L’option `-o` permet de configurer les colonnes affichées. Ici, `pid` correspond à l’identifiant du processus et `comm` au nom de la commande exécutée.

Relation parent–enfant entre processus

Un processus n’apparaît jamais spontanément : il est toujours créé par un **autre processus**, appelé son **parent**. On parle alors de relation parent–enfant, ou de processus père et fils. Cette relation peut être observée grâce aux options appropriées de la commande `ps`.

Par exemple :

```
ps -o ppid,pid,comm
```

La colonne `ppid` indique le PID du processus parent. Dans un cas typique, le shell `bash` est le parent de nombreuses commandes exécutées par l’utilisateur, comme `ps` lui-même.

Le premier processus : init / systemd

Lors du démarrage d’un système Linux, un tout premier processus est chargé en mémoire. Ce processus n’a pas de parent et constitue la racine de tous les autres. Historiquement, il s’appelait **init** ; aujourd’hui, il est le plus souvent remplacé par **systemd**. Ce processus possède généralement le **PID 1**.

`systemd` est responsable du lancement des services du système et de la création de nouveaux processus, notamment ceux associés aux terminaux. Lorsqu’un utilisateur se connecte, un processus shell est créé pour lui, comme défini dans le fichier `/etc/passwd`, généralement `/bin/bash`.

Hiérarchie des processus

L’existence de relations parent–enfant entre processus entraîne une **hiérarchie**, organisée sous forme d’arborescence. Tous les processus descendent du processus initial `systemd`. Cette hiérarchie peut être visualisée à l’aide de la commande suivante :

```
ps -axfo ppid,pid,comm
```

Les options utilisées ont les significations suivantes :

- `a` affiche les processus de tous les utilisateurs,
- `x` inclut les processus non liés à un terminal,
- `f` affiche les processus sous forme d’arbre,
- `-o` permet de choisir les colonnes affichées.

Cette représentation montre clairement comment les processus sont liés entre eux, par exemple un shell lancé par un terminal, puis les commandes exécutées depuis ce shell.

Influence du type de terminal

La hiérarchie observée dépend du type de terminal utilisé. Sur un terminal texte accessible via les combinaisons de touches comme `Ctrl+F1` à `Ctrl+F6`, la hiérarchie est généralement plus simple. En revanche, dans un terminal ouvert depuis l'interface graphique, des processus intermédiaires supplémentaires apparaissent, liés au gestionnaire graphique et aux émulateurs de terminal.

Les processus constituent un élément fondamental du fonctionnement d'un système d'exploitation. Comprendre leur identification, leur création et leur hiérarchie permet de mieux appréhender la gestion des ressources et l'exécution des programmes sous Linux.

L'ordonnanceur du système d'exploitation – 09

L'**ordonnanceur** (*scheduler*) est une composante essentielle du noyau. Son rôle est d'attribuer le **processeur (CPU)** aux différents processus du système. Un processeur ne peut exécuter qu'un seul processus à la fois, alors que plusieurs processus coexistent en permanence. L'ordonnanceur doit donc organiser l'accès au CPU de manière à donner l'illusion que tous les processus progressent simultanément. Sur un processeur multi-cœur, ce principe s'applique indépendamment à chaque cœur.

Techniques de partage du processeur

Une première approche est la **multiprogrammation pure**. Dans ce cas, l'ordonnanceur n'intervient que lorsqu'un processus se bloque ou lorsqu'une interruption matérielle survient, généralement à cause d'entrées-sorties. Cette méthode pose problème pour les processus de calcul intensif, qui utilisent peu ou pas d'entrées-sorties et risquent donc de monopoliser le processeur, empêchant les autres processus de progresser correctement.

Pour résoudre ce problème, les systèmes modernes utilisent le **time slicing**. L'ordonnanceur est alors appelé régulièrement grâce à une interruption d'horloge. Chaque processus élu reçoit le CPU pour une durée limitée appelée **quantum**. Lorsque ce quantum est écoulé, le processus peut être interrompu, même s'il n'a pas volontairement rendu la main. Cette technique permet la **préemption** et garantit un partage plus équitable du processeur, particulièrement important pour les systèmes interactifs.

États des processus

Pour prendre ses décisions, l'ordonnanceur doit connaître l'**état** de chaque processus. Les trois états principaux sont l'état **élu** (*running*), dans lequel le processus utilise le CPU, l'état **prêt** (*ready*), dans lequel le processus attend de pouvoir être exécuté, et l'état **bloqué** (*blocked*), dans lequel le processus attend une ressource indisponible, comme la fin d'une lecture disque.

L'état des processus peut être observé à l'aide de la commande suivante :

```
ps -o pid,stat,comm
```

Dans cet affichage, l'état R indique un processus en cours d'exécution ou prêt à s'exécuter, tandis que l'état S indique un processus en sommeil interruptible, généralement en attente d'un événement.

Transitions entre états

Les processus changent d'état selon différentes transitions. Un processus passe de l'état élu à l'état prêt lorsqu'il a épuisé son quantum de temps processeur. Il passe de l'état prêt à l'état élu lorsqu'il est sélectionné par l'ordonnanceur. Un processus élu devient bloqué lorsqu'il demande une ressource indisponible. Enfin, un processus bloqué redevient prêt lorsque la ressource devient disponible, souvent suite à une interruption matérielle.

Certaines transitions directes sont impossibles. Par exemple, un processus prêt ne peut pas devenir bloqué sans avoir été élu, puisqu'il ne s'exécute pas encore.

Le contexte d'exécution

Pour pouvoir interrompre et reprendre l'exécution des processus, le noyau doit gérer leur **contexte d'exécution**. Ce contexte correspond à l'état du processeur à un instant donné : registres, pointeurs mémoire, compteur d'instructions, etc. Lorsqu'un processus perd le CPU, son contexte est sauvegardé dans la **table des processus**. Lorsqu'il est à nouveau élu, ce contexte est restauré afin que l'exécution reprenne exactement au même point.

Les changements de contexte ont un coût en temps. Si le processeur passe trop de temps à sauvegarder et restaurer des contextes, l'**efficacité du CPU** diminue. Par exemple, si la moitié du temps processeur est consacrée aux changements de contexte, l'efficacité n'est que de 50 %.

Algorithmes d'ordonnancement

Le choix du prochain processus à exécuter repose sur un **algorithme d'ordonnancement**. Cet algorithme doit satisfaire plusieurs critères, comme l'équité entre les processus, de bons temps de réponse pour l'utilisateur, un rendement correct du système et un équilibre global des ressources. Il n'existe pas de solution universelle parfaite, chaque algorithme favorisant certains aspects au détriment d'autres.

Le tourniquet (Round Robin)

L'algorithme du **tourniquet** repose sur une file de processus prêts. Chaque processus reçoit le CPU à tour de rôle pour un quantum de temps déterminé. Lorsqu'un processus devient prêt, il est placé en fin de file. Il perd le CPU lorsqu'il se termine, lorsqu'il se bloque pour une entrée-sortie ou lorsqu'il a épuisé son quantum.

Le principal défi de cet algorithme est le choix de la durée du quantum. Un quantum trop long détériore les temps de réponse, tandis qu'un quantum trop court augmente le nombre de changements de contexte et réduit l'efficacité du CPU.

Ordonnancement par priorité

Dans un **ordonnancement par priorité**, chaque processus se voit attribuer une priorité. L'ordonnanceur sélectionne toujours le processus prêt ayant la priorité la plus élevée. Si les priorités sont **statiques**, elles ne changent jamais, ce qui peut conduire à la **famine** des processus de faible priorité.

Pour éviter ce problème, on utilise des **priorités dynamiques**. L'idée est de favoriser les processus qui se bloquent souvent, par exemple pour des entrées-sorties. Un processus qui utilise peu son quantum reçoit une priorité plus élevée qu'un processus qui consomme tout son temps processeur pour des calculs.

Files multiples et classes de processus

Une approche plus avancée consiste à classer les processus en différentes **classes** selon leur comportement. Les processus orientés calcul reçoivent des quanta longs mais des priorités faibles, tandis que les processus orientés entrées-sorties reçoivent des quanta courts mais des priorités élevées. Chaque classe est gérée par un algorithme de type tourniquet, et un processus d'une classe inférieure n'est élu que s'il n'existe aucun processus prêt dans une classe supérieure.

La classe d'un processus peut être ajustée dynamiquement en fonction de son comportement, ce qui permet un bon compromis entre réactivité et rendement.

Conclusion

L'ordonnanceur joue un rôle stratégique dans un système d'exploitation. Il conditionne les performances, la réactivité et l'équité du système. Les systèmes modernes ont évolué de la mono-programmation vers le **temps partagé**, en utilisant le time slicing et des algorithmes sophistiqués pour répondre aux exigences des environnements interactifs et, dans certains cas, temps réel.