UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Linux Advanced

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Architecture système Linux

Architecture système Linux

Introduction

- Le système Linux est structuré selon une architecture modulaire en couches distinctes, comprenant du matériel, un noyau, et des couches logicielles qui interagissent via des interfaces standardisées.
- Cette architecture garantit sécurité, stabilité et modularité.

Architecture système Linux

Vue d'ensemble de l'architecture Linux

L'architecture générale de Linux est constituée de :

- **Matériel (Hardware)** : CPU, mémoire, périphériques d'entrée/sortie, disques durs, etc.
- **Noyau Linux (Kernel)** : couche centrale responsable de la gestion des ressources matérielles.
- **Modules du noyau (LKM)** : extensions dynamiques du noyau.
- **Shell et utilitaires système** : permettant à l'utilisateur d'interagir avec le système.
- **Bibliothèques système (glibc, libm, libpthread, etc.)** : assurent l'abstraction matérielle.
- **Applications utilisateur** : logiciels exécutés par les utilisateurs.

Architecture système Linux

Vue d'ensemble de l'architecture Linux

Schéma simplifié :

```
+-----------------------------------+
| Applications utilisateur          | Ring 3
+-----------------------------------+
| Bibliothèques système (glibc...)  | Ring 3
+-----------------------------------+
| Appels système (syscalls)         |
+-----------------------------------+
| Noyau Linux (Kernel + LKM)        | Ring 0
+-----------------------------------+
| Matériel (CPU, RAM, Disques...)   | Ring -1 (Hyperviseur/firmware)
+-----------------------------------+
```

Architecture système Linux

Vue d'ensemble de l'architecture Linux

- Le système Linux est structuré en couches :
  - en bas, le **matériel** (processeur, mémoire, périphériques) ;
  - au-dessus, le **noyau** du système (le kernel) qui s'exécute en mode privilégié ;
  - et tout en haut, l'**espace utilisateur** où tournent les applications et processus utilisateur.
- Cette séparation entre espace noyau et espace utilisateur est fondamentale pour la stabilité et la sécurité : le noyau dispose d'un accès direct au matériel et gère les ressources, tandis que les programmes utilisateurs doivent passer par des appels contrôlés (appels système) pour solliciter ses services.
- Cela offre une protection mémoire et empêche qu'un programme en espace utilisateur ne corrompe directement le noyau ou les autres processus

Architecture système Linux

Vue d'ensemble de l'architecture Linux

- En pratique, lorsqu'une application veut accéder au disque, au réseau ou à tout périphérique, elle effectue un appel système pour demander au noyau d'exécuter cette opération privilégiée.
- Le noyau joue donc le rôle d'intermédiaire entre le matériel et les logiciels :
  - il gère la **mémoire**,
  - planifie les **processus**,
  - fournit des abstractions comme les **systèmes de fichiers** et les **interfaces réseau**,
  - et expose ces fonctionnalités aux applications via une interface bien définie.

Architecture système Linux

Anneaux de protection (-1, 0, 3)

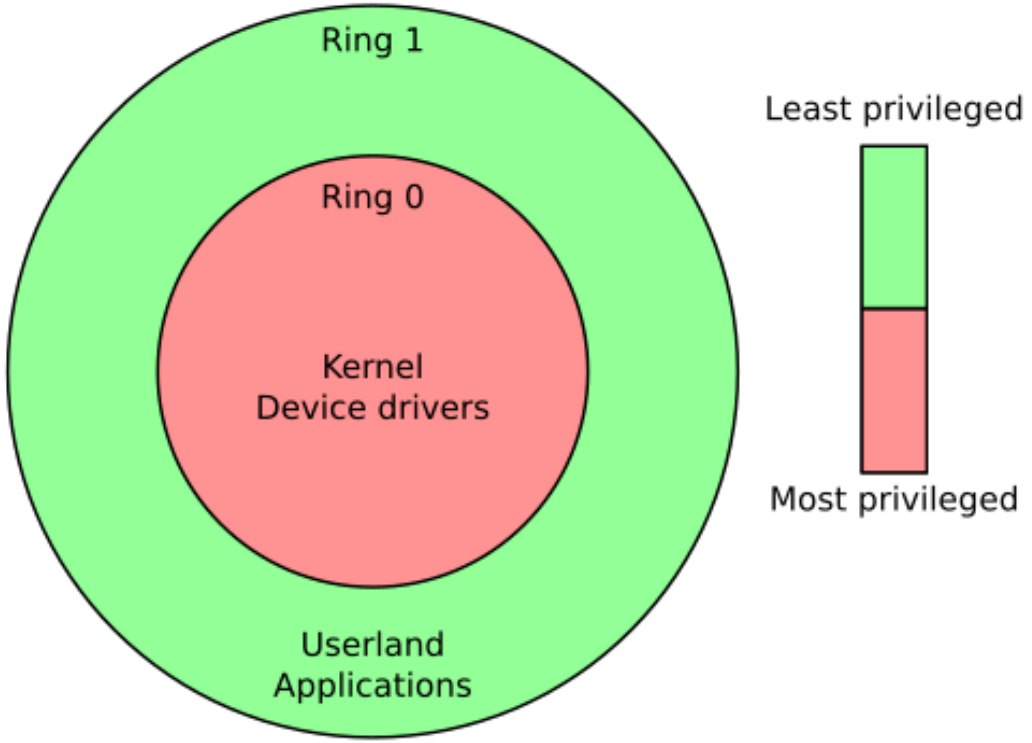
Linux tire parti des anneaux de protection pour assurer une isolation efficace entre les processus et le noyau :

Anneau	Niveau	Description
-1	Hyperviseur	Virtualisation matérielle et firmware (BIOS/UEFI)
0	Mode noyau	Accès total au matériel, opérations critiques, gestion des processus et du matériel
1	Non utilisé	Non utilisé par Linux pour simplifier l'architecture
2	Non utilisé	Non utilisé par Linux pour simplifier l'architecture
3	Mode utilisateur	Exécution des applications et bibliothèques. Accès limité et contrôlé au matériel via appels système

Ainsi, les applications utilisateur n'ont pas d'accès direct au matériel, ce qui garantit sécurité et stabilité.

Architecture système Linux

Anneaux de protection (-1, 0, 3)



Architecture système Linux

Anneaux de protection (-1, 0, 3)

- Sur les processeurs x86, le mode de protection par **anneaux** définit plusieurs niveaux de privilège matériel.
- L'anneau le plus privilégié est le **ring 0**, utilisé par le noyau (mode superviseur), et l'anneau le moins privilégié est généralement le **ring 3**, utilisé par les applications en espace utilisateur (mode utilisateur).
- En théorie, x86 offre 4 anneaux (0 à 3), mais la plupart des OS (Unix, Windows…) n'en utilisent que deux : ring 0 pour le noyau et ring 3 pour l'espace utilisateur.
- Linux suit ce modèle, laissant les anneaux intermédiaires inutilisés.
- Cette séparation garantit qu'un programme en ring 3 ne peut pas exécuter d'instructions sensibles ni accéder directement au matériel sans passer par le ring 0 – toute tentative illégitime déclencherait une faute de protection générale interceptée par le noyau.

Architecture système Linux

Anneaux de protection (-1, 0, 3)

- Linux suit ce modèle, laissant les anneaux intermédiaires inutilisés.
- Cette séparation garantit qu'un programme en ring 3 ne peut pas exécuter d'instructions sensibles ni accéder directement au matériel sans passer par le ring 0 – toute tentative illégitime déclencherait une faute de protection générale interceptée par le noyau

Architecture système Linux

Anneaux de protection (-1, 0, 3)

- Les **hyperviseurs** (virtualisation) exploitent un niveau de privilège encore supérieur, parfois qualifié d'**anneau -1**.
- En effet, les extensions de virtualisation matérielle (Intel VT-x, AMD-V) introduisent un mode d'exécution spécial pour l'hyperviseur, plus privilégié que le noyau lui-même.
- Dans ce mode, l'hyperviseur (comme KVM ou Xen) s'exécute en *ring -1* et peut contrôler entièrement la machine, tandis que chaque système d'exploitation invité tourne en ring 0 virtuel sous son contrôle.
- Avant l'apparition de ces extensions, un hyperviseur devait ruser : par exemple, Xen exécutait les noyaux invités en ring 1 et interceptait via un mécanisme *trap-and-emulate* toutes les instructions privilégiées non autorisées.
- Désormais, grâce au *"mode root"* VT-x, l'hyperviseur dispose d'un niveau dédié (ring -1) pour arbitrer les accès, et les OS invités peuvent fonctionner en ring 0 sans compromettre l'isolement.
- En résumé, l'anneau 0 correspond au noyau Linux (et drivers) s'exécutant en mode superviseur, l'anneau 3 correspond aux programmes utilisateurs en mode non privilégié, et l'anneau -1 désigne le mode hyperviseur offert par la virtualisation matérielle pour héberger des machines virtuelles avec un niveau de privilège supérieur au noyau hôte.

Architecture système Linux

Plateformes matérielles supportées par Linux

- Une grande force du noyau Linux est sa **portabilité** : il a été porté sur un très large éventail d'architectures matérielles.
- Historiquement né sur PC Intel 80386, Linux prend aujourd'hui en charge des **douzaines d'architectures** différentes

Architecture système Linux

Plateformes matérielles supportées par Linux

- **x86 32 bits (IA-32)** et **x86 64 bits (x86_64/AMD64)** – architectures PC grand public, serveurs et laptops.
- **ARM** (32 bits ARMv7 et 64 bits ARMv8/AArch64) – omniprésent dans l'embarqué, les smartphones, tablettes et micro-ordinateurs (Raspberry Pi, etc.).
- **RISC-V** – une architecture RISC ouverte et modulaire, dont le support a été intégré au noyau Linux en 2022 (RISC-V - Wikipedia).
- **PowerPC/POWER** – architecture RISC d'IBM utilisée sur des stations de travail, serveurs et anciens Mac (Linux tourne sur des systèmes IBM Power et sur d'anciennes consoles de jeu, par ex. la PS3, basées sur PowerPC).
- **MIPS** – architecture RISC autrefois répandue dans les systèmes embarqués et stations SGI, supportée par Linux.
- **SPARC** – architecture RISC de Sun/Oracle (stations Unix), également supportée par Linux.
- **IBM S/390 (s390x)** – l'architecture des grands mainframes IBM, que Linux supporte nativement depuis les années 2000.
- Autres plateformes : **Microblaze**, **MIPS64**, **SuperH**, **Alpha**, **PA-RISC**, **Itanium**, **ARC**, etc.

Architecture système Linux

Plateformes matérielles supportées par Linux

- Par exemple, une distribution comme **Ubuntu** fournit des images officielles pour x86_64, ARM64, PowerPC64 et IBM System z , ce qui illustre la diversité du matériel pris en charge.
- Linux peut ainsi fonctionner aussi bien sur un **microcontrôleur** de quelques MHz avec peu de mémoire que sur un **supercalculateur** massif – ce qui témoigne de son extrême adaptabilité.
- Cette portabilité est rendue possible par l'abstraction que fait le noyau du matériel : une grande partie du code est indépendante de l'architecture, et seules quelques couches (gestion du CPU, interruptions, etc.) sont spécifiques à chaque architecture, souvent isolées dans des sous-répertoires du code source (par ex. `arch/x86`, `arch/arm`...).
- Le support d'une nouvelle architecture consiste à implémenter ces couches d'adaptation.

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

Noyau Linux (Kernel Linux)

- Le noyau Linux adopte une architecture dite **monolithique modulaire**. **Monolithique** signifie que le noyau constitue un seul bloc de code s'exécutant en espace noyau (par opposition à un micronoyau qui éclaterait les services en plusieurs processus séparés).
- **Modulaire** signifie que ce noyau peut être étendu à chaud via des **modules chargeables**.
- En effet, Linux est conçu de manière modulaire, permettant d'intégrer des composants du noyau sous forme de modules logiciels qu'on peut charger ou décharger dynamiquement selon les besoins.
- À la compilation du noyau, de nombreux drivers et fonctionnalités peuvent être sélectionnés soit comme faisant partie intégrante du noyau, soit comme modules séparés (.ko).

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

Noyau Linux (Kernel Linux)

Le noyau est le cœur du système Linux, responsable :

- De la **gestion des ressources matérielles** (CPU, RAM, I/O, périphériques...).
- Du **multi-tâche et ordonnancement** (scheduler).
- De la **gestion mémoire** (allocation, pagination, mémoire virtuelle...).
- Du **système de fichiers** et du stockage (FS).
- De la **gestion des périphériques**.
- De la **sécurité et contrôle d'accès** (permissions, isolation, sécurité).
- Des **communications inter-processus** (IPC).
- Du **réseau et protocoles associés**.

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

LKM (Loadable Kernel Modules)

Les modules du noyau Linux (LKM - Linux Kernel Modules) sont des composants logiciels qui peuvent être chargés ou déchargés dynamiquement pour :

- Étendre les fonctionnalités du noyau sans redémarrage (pilotes matériels, protocoles réseau, systèmes de fichiers).
- Faciliter la maintenance du système (correction de bugs, sécurité renforcée).
- Optimiser les ressources système en chargeant uniquement les modules nécessaires.

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

LKM (Loadable Kernel Modules)

- Les **LKM (Loadable Kernel Modules)** sont ces modules du noyau que l'on peut insérer ou retirer à l'exécution. Ils offrent plusieurs avantages :
  - ajouter un **module** permet d'activer une nouvelle fonctionnalité (par exemple le support d'un nouveau système de fichiers ou d'un périphérique) sans recompiler ni redémarrer le noyau ;
  - retirer un module libère les ressources associées si le matériel n'est plus utilisé.
    Cela contribue à réduire la taille du noyau en mémoire (on ne charge que les composants nécessaires) et facilite les mises à jour (on peut remplacer un module par une nouvelle version sans interrompre tout le système).
- Par exemple, les pilotes de certaines cartes réseau, imprimantes ou systèmes de fichiers (ex: **ext4**, **XFS**, **NTFS**...) sont souvent fournis sous forme de modules que l'on insère au besoin.

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

LKM (Loadable Kernel Modules)

- Un **module** est un morceau de code compilé séparément (fichier .ko) qui peut s'insérer dans l'espace noyau.
- Le **noyau** garde une table des symboles exportés auxquels les modules peuvent faire référence. Lorsqu'on charge un module (avec `insmod` ou `modprobe`), le kernel réalise un lien à chaud : il résout les dépendances du module envers les symboles du noyau ou d'autres modules déjà chargés, alloue de la mémoire kernel pour ce module et y transfère l'exécution.
- Une fois chargé, le **module** s'exécute avec les mêmes privilèges que le noyau lui-même.
- Linux fournit des APIs internes pour écrire des **modules**, par exemple pour enregistrer un nouveau pilote, un protocole réseau, etc., via des fonctions d'initialisation qui sont appelées au chargement.
- Inversement, lorsqu'on enlève un module (`rmmod`), le kernel appelle la routine de nettoyage du **module** puis libère ses ressources.
- Notons qu'un **module** mal programmé (buggé) peut potentiellement planter le système autant qu'un bug dans le noyau monolithique, puisqu'il tourne au même niveau de privilège.

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

En résumé

Linux combine le meilleur des deux approches :

- un **noyau monolithique** performant où tout le code tourne en mode superviseur, et une **extensibilité modulaire** permettant une grande flexibilité d'utilisation.
- La plupart des distributions livrent un noyau générique comportant un minimum de fonctionnalités en dur, et tout le reste en modules (fichiers situés sous `/lib/modules/` correspondant à la version du noyau).
- Au démarrage, seul le noyau de base est chargé ;
- puis les modules nécessaires (pilotes, etc.) sont insérés à la volée (souvent automatiquement via udev ou *scripts* de démarrage) en fonction du matériel détecté ou des besoins du système

Architecture système Linux

Noyau Linux et LKM (Loadable Kernel Modules)

Commandes clés pour la gestion des modules :

```
lsmod              # Affiche les modules actuellement chargés
modprobe <module> # Charge un module spécifique
rmmod <module>     # Décharge un module
insmod <module>    # Charge directement un module (.ko)
modinfo <module>   # Affiche les informations sur un module
```

Exemple de gestion de module réseau :

```
sudo modprobe e1000e        # charger module pour carte réseau Intel
sudo rmmod e1000e           # décharger module
```

Architecture système Linux

Le système de fichiers racine (Root Filesystem `/`)

- Sous Linux, l'ensemble du système de fichiers s'organise autour d'une racine unique notée `/` (le *root filesystem*).
- Cette arborescence respecte la norme **FHS (Filesystem Hierarchy Standard)**, un standard qui définit de manière cohérente les emplacements des fichiers et répertoires dans les systèmes de type Unix

Architecture système Linux

Le système de fichiers racine (Root Filesystem `/`)

Le système de fichiers racine est la base hiérarchique d'un système Linux, à partir de laquelle tous les autres systèmes de fichiers sont montés.

Structure typique :

```
/
├── /bin      (binaires essentiels)
|── /boot    (fichiers liés au boot)
├── /etc       (configurations système)
├── /home       (répertoires utilisateurs)
├── /var       (fichiers variables, logs, bases de données)
├── /lib       (bibliothèques partagées essentielles)
├── /usr       (applications, bibliothèques, ressources)
├── /bin, /sbin (commandes essentielles du système)
├── /tmp       (fichiers temporaires)
└── /dev       (périphériques matériels)
```

Architecture système Linux

Le système de fichiers racine (Root Filesystem `/`)

- Cette organisation standardisée (définie par FHS) permet aux administrateurs et utilisateurs de s'y retrouver facilement, et aux logiciels de savoir où placer ou chercher leurs fichiers.
- Ainsi, un programme pourra supposer que les configurations système sont dans `/etc`, que les logs vont dans `/var/log`, que les exécutables sont dans `/usr/bin` ou `/usr/local/bin` selon qu'ils sont distribués ou locaux, etc.
- La plupart des distributions Linux respectent scrupuleusement FHS, avec parfois quelques légères variantes ou liens symboliques (ex : certaines distributions unifient `/bin` et `/usr/bin` en faisant de l'un un lien vers l'autre pour simplifier l'arborescence).
- Mais globalement, un utilisateur passant d'une distribution à l'autre retrouvera la même structure de base, héritée de Unix.

Architecture système Linux

Pilotes de périphériques

- Les **pilotes de périphériques** (device drivers) sont des composants du noyau chargés de la communication avec le matériel ou avec des périphériques virtuels.
- Leur rôle est de fournir une interface standard aux programmes pour utiliser un matériel donné, en se chargeant de tous les détails bas-niveau spécifiques à ce matériel.
- Sous Linux, quasiment tout est piloté par des drivers : les disques, les clés USB, les cartes réseau, les cartes graphiques, le son, mais aussi des éléments plus virtuels comme le système de fichiers pseudo `proc` ou les terminaux virtuels.

Architecture système Linux

Pilotes de périphériques

Du point de vue du noyau, un driver s'enregistre généralement comme un certain **type de périphérique** :

- **Pilotes de caractère** (**char drivers**) – ils gèrent des périphériques accessibles comme un flux de bytes, octet par octet. Typiquement, ce sont les ports série, les terminaux, les périphériques d'entrée (clavier) ou encore `/dev/tty`, `/dev/random`, etc. Les opérations se font via des appels système comme `read()`/`write()` qui transfèrent des octets en séquence.
- **Pilotes de bloc** – ils gèrent des périphériques organisés en blocs adressables (typiquement 512 octets ou plus). Ce sont principalement les **disques** et autres supports de stockage (SSD, CD-ROM). Le noyau les utilise via le cache de blocs et les appels comme `read()/write()` en bloc, et ils permettent notamment de monter des systèmes de fichiers.
- **Pilotes réseau** – ils ne s'exposent pas comme des fichiers dans `/dev` mais dans la pile réseau du noyau. Leur interface est plus complexe : ils échangent des **trames** (frames) ou paquets réseau plutôt que de simples octets ou blocs. Exemples : driver d'interface Ethernet, Wi-Fi (émission/réception de paquets), interface loopback. L'accès se fait via l'API socket/Berkeley du côté user, qui s'appuie en interne sur ces drivers réseau.

Architecture système Linux

Pilotes de périphériques

- En pratique, les drivers Linux peuvent être soit **intégrés statiquement** au noyau, soit compilés en **modules** chargeables (LKM) comme vu plus haut.
- La plupart des distributions choisissent de livrer la majorité des drivers sous forme de modules, afin qu'ils ne soient chargés que si le matériel correspondant est présent.
- Lorsque le système détecte un nouveau matériel (par exemple insertion d'une clé USB), un mécanisme comme **udev** va éventuellement charger le module kernel approprié pour le gérer.
- Inversement, un module de pilote inutilisé peut être déchargé pour libérer de la mémoire.
- On peut lister les drivers (modules) actuellement chargés avec la commande `lsmod`, qui affiche la liste des modules du noyau en mémoire.
- Par exemple, on y verra des entrées comme `usb_storage`, `i915` (driver GPU Intel), etc., avec leur taille et combien de fois ils sont utilisés.
- Techniquement, `lsmod` ne fait qu'afficher le contenu de `/proc/modules` de façon formatée.

Architecture système Linux

Pilotes de périphériques

En résumé :

- Les pilotes de périphériques sont le code du noyau qui fait le lien avec le matériel.
- Ils exposent souvent une abstraction (par ex. un fichier dans `/dev` ou une interface réseau) que les programmes peuvent utiliser via les appels système standard.
- Grâce aux drivers, les applications n'ont pas besoin de connaître les détails du fonctionnement d'une carte réseau ou d'un disque : elles utilisent des appels génériques (`open`, `ioctl`, `read`, `write`…), et c'est le driver qui, en coulisse, traduira ces requêtes en opérations concrètes sur le matériel.

Architecture système Linux

Pilotes de périphériques

En résumé :

- La qualité et la richesse de la logithèque de drivers font la force de Linux : aujourd'hui, le noyau supporte une quantité immense de matériels différents.
- Des commandes utiles pour interagir avec les drivers sont par exemple :
  - `lsmod` – lister les modules chargés (donc les drivers actifs).
  - `modinfo <module>` – afficher des informations sur un module (version, description, licence, alias de matériel pris en charge).
  - `lspci`, `lsusb` – lister les périphériques PCI/USB connectés (pour savoir quels matériels sont présents et quels modules peuvent être associés).
  - Fichiers dans `/proc` ou `/sys` – ex : `/proc/interrupts` pour voir quels drivers utilisent quelles IRQ, `/sys/bus/usb/devices/.../driver` pour voir quel driver gère un périphérique USB spécifique, etc.

Architecture système Linux

Bibliothèques partagées et statiques

- Les **bibliothèques** sont des collections de fonctions/utilitaires partagées par plusieurs programmes.
- Sous Linux (et Unix en général), il existe deux modes principaux de liaison aux bibliothèques : la liaison **statique** et la liaison **dynamique** (*partagée*).

Architecture système Linux

Bibliothèques partagées et statiques

- **Bibliothèques statiques** :
  - ce sont des fichiers `.a` (archive) qui contiennent du code objet.
  - Lors de la compilation/édition de liens d'un programme, on peut lier statiquement certaines libs – le code de la bibliothèque nécessaire est alors copié **dans l'exécutable** final.
  - L'avantage est que l'exécutable n'a pas besoin de dépendance externe à l'exécution (il est auto-suffisant), mais l'inconvénient est une taille plus grande et une duplication du même code en mémoire si plusieurs programmes utilisent la même lib.

Architecture système Linux

Bibliothèques partagées et statiques

- **Bibliothèques dynamiques/partagées** :
  - ce sont les fichiers `.so` (*shared object*).
  - Ici, l'exécutable n'embarque pas le code de la lib ;
  - il contient seulement une référence indiquant "j'aurai besoin de telle bibliothèque partagée".
  - À l'exécution, le système va **charger** la bibliothèque partagée en mémoire (une seule fois même si plusieurs processus en ont besoin) et la lier au programme.
  - Ainsi, plusieurs applications peuvent **partager** une même bibliothèque en mémoire, ce qui réduit significativement l'empreinte globale.
  - De plus, les mises à jour sont centralisées : corriger un bug dans la bibliothèque profite immédiatement à tous les programmes qui l'utilisent (il n'est pas nécessaire de recompiler chaque application)
  - Ces raisons – **réduction de la taille sur disque**, **réduction de l'usage mémoire** et **mutualisation des mises à jour** – ont rendu la liaison dynamique largement majoritaire dans les systèmes modernes

Architecture système Linux

Bibliothèques partagées et statiques

- Sous Linux, presque toutes les applications utilisent des bibliothèques partagées (à commencer par la **glibc**, la bibliothèque C standard, qui fournit les appels système, gestion de mémoire, fonctions standard, etc.).
- Lorsqu'un programme est lancé, c'est le **linker dynamique** (`ld-linux.so`, aussi appelé *runtime loader*) qui se charge de trouver et charger les `.so` requis en mémoire.
- Il consulte pour cela les chemins standards (`/lib`, `/usr/lib`…) et éventuellement la variable d'environnement `LD_LIBRARY_PATH` ou le cache `/etc/ld.so.cache` généré par `ldconfig`.
- Si une bibliothèque nécessaire est manquante, le programme ne pourra pas démarrer (erreur *"libXYZ.so.1 not found"*).

Architecture système Linux

Bibliothèques partagées et statiques

- Pour savoir de quelles bibliothèques partagées dépend un exécutable, on utilise la commande **`ldd`** (pour **l**i**d** Dynamic Dependencies).
- Par exemple, `ldd /bin/ls` listera toutes les libs nécessaires à `/bin/ls` (libc, libpthread, libselinux, etc.) et où elles sont trouvées dans le système.
- En interne, `ldd` fonctionne en exécutant le loader dynamique en mode spécial (avec `LD_TRACE_LOADED_OBJECTS`) afin d'afficher ces informations sans réellement lancer le programme.
- C'est un outil très utile pour diagnostiquer des problèmes de dépendances manquantes ou simplement pour comprendre quelles bibliothèques un binaire utilise.
- Un autre outil précieux est **`strace`**, qui permet de tracer les appels système effectués par un programme.
- Bien que strace ne soit pas spécifique aux bibliothèques, on peut s'en servir pour voir le processus de chargement : en lançant une application avec `strace`, on verra notamment des appels `open("/lib/xyz.so", ...)` cherchant les bibliothèques, puis des `mmap()` qui les mappent en mémoire, etc. Strace montre *la fine couche entre un processus utilisateur et le noyau*.
- ce qui inclut l'ouverture des libs, la recherche de fichiers de config (par ex. `/etc/ld.so.cache`), etc.
- En cas d'erreur de chargement, strace peut révéler par exemple que le processus cherche une lib dans tel répertoire et obtient `ENOENT` (fichier non trouvé), aidant ainsi à corriger le problème (installer la lib manquante, ajuster `LD_LIBRARY_PATH`, etc.).

Architecture système Linux

Bibliothèques partagées et statiques

En résumé

- Les bibliothèques partagées (.so) sont un pilier des systèmes Linux, permettant un **partage de code** efficace entre applications. Des outils comme `ldd` permettent d'auditer les dépendances d'un programme, et `strace` d'observer dynamiquement le fonctionnement d'un processus (y compris le chargement de ces dépendances).
- Il est généralement recommandé de lier dynamiquement sauf cas d'usage spécifiques (logiciel embarqué ultra-portable, ou exigence d'avoir un binaire unique ne dépendant de rien).
- À l'inverse, la liaison statique peut être utile pour des environnements minimalistes ou pour s'affranchir de bibliothèques système (au prix d'une duplication du code).
- Linux supporte les deux, mais la plupart des distributions n'utilisent que très rarement les .a (parfois même ne fournissent pas les .a par défaut, uniquement les .so et les headers de dev, afin de pousser à la dynamique).

Architecture système Linux

Appels systèmes

- Les appels système (ou "syscalls") sous Linux sont le mécanisme par lequel les applications en mode utilisateur peuvent interagir avec le noyau pour demander des services, tels que l'accès aux ressources matérielles ou la gestion des processus.
- Lorsqu'un programme a besoin d'un service que seul le noyau peut rendre (par exemple lire un fichier, allouer de la mémoire, créer un processus), il exécute un appel système.
- Un syscall s'apparente à un appel de fonction, à la différence qu'il entraîne un passage du **mode utilisateur au mode noyau** via un mécanisme de trap/interrupt matériel

Architecture système Linux

Appels systèmes

Voici quelques points clés :

1. **Interface entre utilisateur et noyau**
   - Les appels système fournissent une API standardisée pour que les programmes puissent exécuter des opérations critiques sans avoir besoin d'un accès direct au matériel.
   - Cela garantit que les opérations sensibles (comme la lecture/écriture de fichiers, la gestion de la mémoire ou la communication entre processus) sont réalisées de manière sécurisée.
2. **Mécanisme d'invocation**
   - Sous Linux, les appels système sont généralement implémentés via des instructions spécifiques du processeur (par exemple, `syscall` sur les architectures x86_64 ou `int 0x80` sur certaines versions plus anciennes).
   - Le passage en mode noyau est déclenché, permettant au noyau d'exécuter le code correspondant à l'appel système.

Architecture système Linux

Appels systèmes

Voici quelques points clés :

3. **Exemples d'appels système**
   - **Gestion des fichiers :** `open`, `read`, `write`, `close`
   - **Gestion des processus :** `fork`, `execve`, `waitpid`, `exit`
   - **Gestion de la mémoire :** `mmap`, `munmap`
   - **Communication inter-processus :** `pipe`, `socket`, `bind`, `listen`, `accept`
4. **Sécurité et stabilité**
   - En isolant les applications du matériel et en passant par le noyau pour les opérations sensibles, Linux améliore la sécurité et la stabilité du système.
   - Le noyau peut ainsi appliquer des politiques de sécurité, de gestion des ressources et de contrôle d'accès pour éviter les comportements malveillants ou erronés.

Architecture système Linux

Appels systèmes

En résumé

- Les appels système constituent le **contrat d'interface** entre le noyau Linux et les applications.
- Ils fournissent un ensemble de services (gestion des processus, mémoire, fichiers, communication inter-processus, réseau, etc.) qu'un programme peut invoquer.
- Cette interface est relativement stable et standardisée (POSIX), ce qui permet à des logiciels d'être portables : recompiler un programme POSIX sur Linux utilisera les appels système Linux correspondants via la glibc.
- La performance des appels système est critique, c'est pourquoi le noyau et les CPU offrent des optimisations (par ex., sur x86_64 l'instruction `syscall` est bien plus rapide que l'ancienne interruption logicielle).
- Linux fournit également des mécanismes comme le *vdso* (Virtual Dynamic Shared Object) qui permet d'exposer certaines routines noyau en espace utilisateur sans trap (par ex, gettimeofday peut se faire sans appel système en lisant une zone mémoire partagée, pour éviter le coût du trap).
- Néanmoins, pour la plupart des opérations, le passage en mode noyau est indispensable.
- Strace, en tant qu'outil d'observation, nous rappelle à quel point chaque action d'un programme utilisateur repose sur ces appels au noyau – c'est le garde-fou qui sépare le monde utilisateur (restreint) des ressources réelles du système.

Architecture système Linux

Différents *Shells* (Bash, Zsh, Fish, etc.)

- Le *shell* est l'interpréteur de commandes dans un système Unix/Linux.
- Il en existe plusieurs, avec des fonctionnalités et philosophies légèrement différentes, tout en remplissant le même rôle de base :
  - permettre à l'utilisateur d'exécuter des commandes,
  - d'écrire des scripts,
  - d'automatiser des tâches.
- Sous Linux, le shell par défaut est souvent **Bash** (Bourne Again Shell), mais d'autres shells populaires incluent **Zsh** (Z Shell), **Fish** (Friendly Interactive Shell), **Tcsh**, **Ksh**, etc.

Architecture système Linux

Différents *Shells* (Bash, Zsh, Fish, etc.)

Principaux Shells Linux :

- **Bash** (`/bin/bash`) : le plus courant, standard
- **Sh** (Bourne Shell) : ancien mais très répandu, minimaliste
- **Zsh** : puissant, supporte les plugins, autocomplete avancée
- **Fish** : Shell ergonomique, auto-complétion visuelle intégrée
- **Ksh (Korn Shell)** : robuste et performant pour scripting complexe

Architecture système Linux

Différents *Shells* (Bash, Zsh, Fish, etc.)

Chacun de ces shells a ses adeptes et ses cas d'usage :

- **Bash** reste le standard de facto pour l'écriture de scripts et l'administration système, grâce à sa disponibilité universelle et sa conformité POSIX. Même un utilisateur de Zsh ou Fish utilisera souvent Bash en arrière-plan pour exécuter des scripts `/bin/sh`.
- **Zsh** est souvent choisi par les utilisateurs cherchant à optimiser leur environnement de travail avec des prompts améliorés, des complétions intelligentes et un haut degré de personnalisation. Avec les bons réglages, Zsh peut considérablement accélérer la saisie de commandes complexes.
- **Fish** convient bien à ceux qui veulent un shell interactif efficace sans effort de configuration, ou aux débutants pour qui les aides visuelles de Fish rendent l'utilisation du terminal plus accessible.

Architecture système Linux

Différents *Shells* (Bash, Zsh, Fish, etc.)

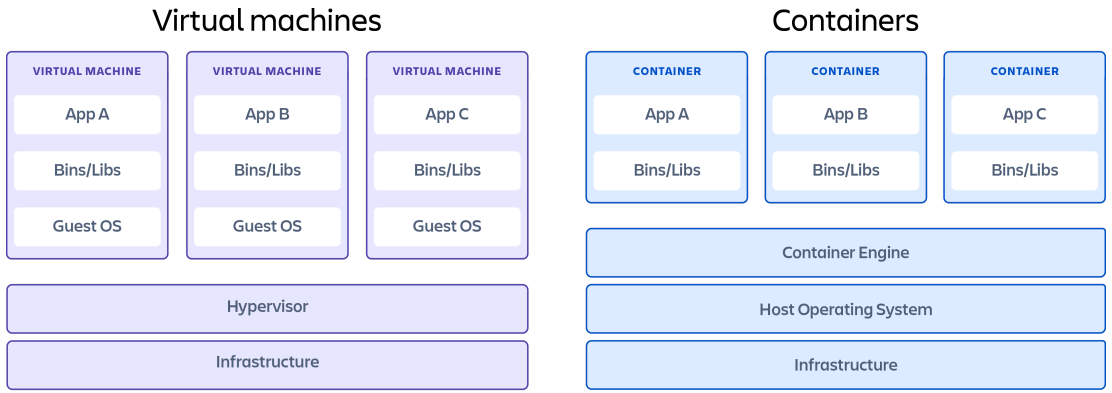
En résumé

- Tous ces shells remplissent au fond le même rôle et permettent d'accomplir les mêmes tâches ;
- Passer de l'un à l'autre est surtout affaire de confort et d'habitude.
- Un script Bash classique pourra généralement être exécuté sous Zsh (qui est largement compatible) mais pas tel quel sous Fish (non compatible par défaut).
- Certains utilisateurs avancés utilisent même plusieurs shells selon les contextes (par ex, Bash pour les scripts, Zsh pour l'interactif sur leur PC, Fish sur une autre machine pour tester).
- Grâce à l'abstraction du shell, l'environnement utilisateur peut être modulé sans toucher aux couches inférieures du système.

Architecture système Linux

La virtualisation sous Linux

- Dans un scénario virtualisation traditionnelle, un hyperviseur (ou un noyau hôte dans le cas de KVM) permet d'exécuter plusieurs OS invités complets (chaque VM ayant son propre kernel et ses bibliothèques).
- À droite, avec la conteneurisation, un seul noyau hôte est partagé, et un *moteur de conteneurs* isole plusieurs applications dans des environnements séparés mais plus légers (pas de duplicata du système d'exploitation complet).



Architecture système Linux

La virtualisation sous Linux

Virtualisation matérielle (Anneau -1)

- **Hyperviseurs type 1 (bare-metal)** :
  - VMware ESXi, XenServer, Proxmox VE
- **Hyperviseur type 2 (hosted)** :
  - VirtualBox, VMware Workstation, KVM/QEMU

Architecture système Linux

La virtualisation sous Linux

Virtualisation complète avec hyperviseur (KVM, Xen, etc.) :

Linux peut servir aussi bien d'**hyperviseur de type 1** que de type 2. Le projet le plus emblématique est **KVM (Kernel-based Virtual Machine)**, intégré directement au noyau Linux.

- KVM transforme le noyau en hyperviseur :
  - il utilise un module noyau (`kvm.ko`) qui exploite les extensions de virtualisation matérielle des CPU (Intel VT-x, AMD-V) pour créer des machines virtuelles.
  - Chaque VM KVM est en fait un processus Linux (géré par le planificateur standard) associé à un espace d'adressage séparé pour l'OS invité.
  - Le noyau Linux hôte joue alors le rôle d'hyperviseur en arbitrant l'accès au CPU, à la mémoire et aux périphériques entre les différentes VMs.
  - L'avantage de KVM est qu'il bénéficie de tout l'écosystème Linux (ordonnanceur efficace, pilotes existants pour gérer les I/O virtuelles via virtio, etc.).
  - En somme, KVM est vu comme un hyperviseur **intégré au kernel** (certains le qualifiaient de *type 2* car il s'appuie sur un OS hôte, mais étant dans le noyau, il s'apparente à du type 1 dans les faits).

Architecture système Linux

La virtualisation sous Linux

Exemple rapide avec KVM :

Installer KVM :

```
sudo apt install qemu-kvm libvirt-daemon bridge-utils virt-manager
```

Créer une VM avec KVM :

```
virt-install --name vm-test --ram 2048 --disk size=10G --os-variant ubuntu22.04 --cdrom ubuntu.iso
```

Architecture système Linux

La virtualisation sous Linux

- À l'inverse, **Xen** est un hyperviseur historique de type 1 qui fonctionne en surcouche du matériel, indépendamment d'un OS hôte.
- Au démarrage, le hyperviseur Xen se charge en premier (à la place d'un noyau classique) et crée un premier environnement privilégié (le *dom0*) qui exécute Linux pour piloter le matériel.
- Les autres VMs (domU) s'exécutent soit via **paravirtualisation** (l'OS invité sait qu'il est virtualisé et appelle Xen pour certaines opérations, ce qui nécessite un OS modifié, mais offre de hautes performances, soit via **full virtualisation** avec extensions CPU (Xen émule alors un matériel standard si l'OS n'est pas modifié, au prix d'un peu plus de overhead).
- Xen offre un haut degré d'isolement et a longtemps été prisé dans les environnements serveurs, bien qu'il soit plus complexe à mettre en place (puisqu'il implique un hyperviseur dédié).

Architecture système Linux

La virtualisation sous Linux

- Aujourd'hui, **KVM** a largement gagné en popularité car il est directement inclus dans Linux (pas besoin d'un hyperviseur externe), bénéficiant du développement communautaire et d'un entretien plus aisé (pas de patchs externes au kernel à maintenir).
- En pratique, des solutions de virtualisation comme **QEMU/KVM** (libvirt, etc.) utilisent **KVM** pour la performance et **QEMU** pour émuler les périphériques, fournissant une expérience équivalente à celle de VirtualBox, VMWare, etc., entièrement sur Linux.

Architecture système Linux

La virtualisation sous Linux

Conteneurs (LXC, Docker, etc.) :

- Plutôt que d'émuler une machine complète, Linux propose l'**isolement par conteneur**, souvent appelé virtualisation au niveau OS. Cette approche utilise les fonctionnalités du noyau telles que les **namespaces** (espaces de noms) et les **cgroups** (groupes de contrôle des ressources) pour isoler des processus les uns des autres comme s'ils tournaient sur des systèmes différents, alors qu'ils partagent le même noyau.
- Des projets comme **LXC (Linux Containers)** ont été les pionniers, et plus récemment l'écosystème **Docker** a popularisé cette approche en facilitant le déploiement d'applications dans des conteneurs légers.
- Dans un conteneur, on va isoler les vues des **processus**, du **système de fichiers**, du **réseau**, des **utilisateurs**, etc., de sorte qu'un processus dans le conteneur pense être seul sur la machine.
- Cependant, tous les conteneurs partagent le noyau **Linux hôte** : ils n'embarquent pas de kernel propre.
- Ainsi, lancer 10 conteneurs n'instancie pas 10 noyaux, juste 10 groupes de processus isolés.
- Cela apporte une énorme **légèreté** : on peut démarrer un conteneur en quelques millisecondes, consommer très peu de RAM supplémentaire (juste les processus en plus), et on évite la redondance des OS invités.

Architecture système Linux

La virtualisation sous Linux

Conteneurs (LXC, Docker, etc.) :

La contrepartie des conteneurs est qu'ils **partagent le noyau** :

- cela signifie qu'on ne peut pas faire tourner un OS non Linux dans un conteneur Linux (pas de Windows conteneur sur un Linux par ex., contrairement à une VM où on peut installer n'importe quel OS invité).
- De plus, la sécurité repose sur l'isolement du noyau ;
- si un attaquant brise l'isolement (via une vulnérabilité kernel), il peut sortir du conteneur.
- Néanmoins, en pratique, les conteneurs Linux offrent un isolement suffisant pour de nombreux usages et permettent une densité très supérieure aux VMs.
- C'est un peu un retour aux *chroot/jails* historiques, en bien plus abouti:
  - chaque conteneur a sa propre vue du système (montages privés, réseau virtuel, utilisateurs isolés) tout en étant beaucoup plus **agile** qu'une VM.
  - Docker a ajouté à cela un format d'images empilables et un écosystème qui a révolutionné le déploiement logiciel.

Architecture système Linux

La virtualisation sous Linux

En résumé

- **Linux est au cœur de ces deux technologies (Virtualisation lourde et legère) :**
  - en tant qu'hyperviseur KVM pour faire tourner des VM dans les cloud publics, et en tant que moteur de conteneurs (avec Docker/Containerd) pour isoler les applications.
  - Ainsi, la virtualisation sous Linux offre un **spectre complet** : de la VM lourde mais totalement flexible (n'importe quel OS invité) au conteneur léger mais même noyau, en passant par des compromis intermédiaires (LXC permet par ex. d'avoir un environnement proche d'une VM mais sans hyperviseur).
  - Cette polyvalence fait de Linux l'épine dorsale de la plupart des infrastructures virtualisées modernes.
  - Linux fournit dans sa documentation et son code des sous-systèmes complets pour la virtualisation (ex: KVM pour l'hyperviseur, cgroups/namespace pour les conteneurs, KVM API via `/dev/kvm`, interfaces /proc pour cgroups, etc.), et continue d'innover (ex : projets comme **Kata Containers** qui combinent VM légère et isolation de conteneur, ou **virtio-fs** pour partager un système de fichiers efficacement entre hôte et VM, etc.).

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Noyau Linux

Noyau Linux

Téléchargement des sources et des outils nécessaires

- Pour compiler le noyau Linux, il faut d'abord récupérer son **code source officiel** et installer les **outils de compilation** requis.
- Le code source du noyau Linux est disponible sur le site officiel **kernel.org**, qui répertorie les versions *mainline* (développement), *stables* et *LTS* (support long terme).
- On peut soit cloner le dépôt Git de Linus Torvalds, soit télécharger l'archive de la version stable souhaitée (fichiers `.tar.xz`) depuis kernel.org (Compilation et installation du noyau Linux | matteyeux's blog).
- Par exemple, pour télécharger la version 6.1.8, on utiliserait :

```
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.8.tar.xz
tar -xvf linux-6.1.8.tar.xz && cd linux-6.1.8
```

Noyau Linux

Téléchargement des sources et des outils nécessaires

- Une autre option sur Debian est d'**installer le paquet source** correspondant (`apt install linux-source-version`), qui place une archive du code source (ex: `/usr/src/linux-source-5.10.tar.xz`) qu'il suffit d'extraire dans un répertoire de travail utilisateur.
- Il est conseillé de *ne pas compiler en root* ni dans `/usr/src` directement, mais dans un dossier de son *HOME* (par ex. `~/kernel/`), pour éviter tout problème de permission ou de conflit avec le système.

Noyau Linux

Téléchargement des sources et des outils nécessaires

- Installez les **outils de build** nécessaires.
- Sur Debian/Ubuntu, le paquet meta `build-essential` fournit le compilateur C (GCC) et make.
- Il faut également les bibliothèques de développement Ncurses (pour l'interface menuconfig), et d'autres utilitaires. Par exemple :

```
sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev bc
```

- Cette commande installe GCC, Make, les bibliothèques Ncurses, Bison, Flex, OpenSSL (pour certaines options cryptographiques) et libelf (pour la génération de BPF, etc.).
- D'autres distributions ont des noms de paquets légèrement différents, mais ces composants sont généralement requis.
- Il peut être utile aussi d'installer `git` (si on souhaite cloner la dernière version du code source) et `fakeroot` (pour créer des paquets sans privilèges root, utile en méthode Debian).

Noyau Linux

Téléchargement des sources et des outils nécessaires

Une fois l'archive extraite, on obtient un répertoire (ex: `linux-6.1.8/`) contenant de nombreux sous-dossiers.

- **Les principaux sont :**
  - `arch/` – code spécifique à chaque architecture processeur (x86, ARM, etc.). Par exemple, `arch/x86` contient le code propre aux PC 32/64 bits.
  - `drivers/` – l'ensemble des pilotes de périphériques du noyau, classés par type de matériel (son, réseau, USB, etc.).
  - `fs/` – implémentations des systèmes de fichiers (ext4, NTFS, FAT, etc.), chaque sous-dossier correspondant à un FS supporté.
  - `kernel/` – le cœur du noyau (ordonnanceur, gestion du temps, synchronisation, etc.).
  - `mm/` – la gestion de la mémoire (avec des parties spécifiques par archi dans `arch/*/mm/`).
  - `net/` – la pile réseau du noyau (protocoles, sockets…).
  - `include/` – fichiers d'en-tête (.h) partagés.
  - `scripts/` – scripts utilitaires pour la configuration et la compilation.

Noyau Linux

Paramétrage du noyau avancé

- Avant de compiler, il faut **configurer le noyau** selon vos besoins matériels et fonctionnels.
- La configuration détermine quels pilotes et options seront compilés (intégrés ou en modules).
- La méthode classique est d'utiliser l'interface semi-graphique `make menuconfig`.

Exemple simple (classique) :

```
cd linux-6.8
make menuconfig
```

- Sélectionnez les modules/drivers nécessaires.
- Sauvegardez (`.config` généré automatiquement).

Noyau Linux

Paramétrage du noyau avancé

L'interface de menuconfig est organisée par **grandes sections** thématiques. Parmi les principales options de configuration, on retrouve par exemple :

- **General setup** – options générales du noyau (ex. définir une **version locale** personnalisée qui s'ajoutera au numéro de version du noyau, utile pour distinguer votre build)
- **Processor type and features** – options liées au processeur et à l'architecture (optimiser le noyau pour un type de CPU spécifique, activer/désactiver le support de SMP, de l'hyperthreading, etc.)
- **Power management and ACPI options** – gestion de l'énergie, suspendre/réveil, ACPI… (particulièrement important pour les portables)
- **Networking support** – prise en charge des protocoles réseau (IPv6, IPsec, Bluetooth, etc.) et des options de sécurité réseau
- **Device Drivers** – configuration des pilotes de périphériques : carte graphique, adaptateurs réseau, stockage, USB… On peut y activer ou non le support de certains matériels en fonction de son PC
- **File systems** – support des systèmes de fichiers (ext4, Btrfs, NFS, etc.) qu'on souhaite inclure
- *(Et ainsi de suite : options de sécurité (SELinux, AppArmor…), cryptographie, virtualisation, « Kernel hacking » pour le debug, etc.)*

Noyau Linux

Paramétrage du noyau avancé

Personnalisation avancée :

- Vous pouvez charger une base de configuration existante pour ne pas repartir de zéro.
- Par exemple, il est courant de **réutiliser la configuration du noyau actuellement installé** sur votre système comme point de départ.
- Sur Debian/Ubuntu, le fichier de config du noyau courant se trouve dans `/boot` (ex: `config-5.10.0-17-amd64`).
- Copiez-le dans le répertoire des sources sous le nom `.config` avant de lancer menuconfig :

```
cp /boot/config-$(uname -r) .config
```

- Ainsi, vous partez d'une config connue.
- Ensuite, exécutez `make menuconfig` et ne modifiez que ce qui vous intéresse.
- Si le noyau que vous compilez est d'une version différente de celui d'origine, utilisez la cible **`make oldconfig`** pour mettre à jour la config :
  - elle vous posera des questions seulement pour les nouvelles options apparues depuis (vous pouvez aussi faire `make olddefconfig` pour accepter les options par défaut sans interactivité).

Noyau Linux

Compilation et installation du noyau (méthode classique)

- Une fois la configuration prête, passons aux **étapes de compilation et d'installation** du noyau.
- La méthode « classique » consiste à utiliser directement Make pour construire le noyau et ses modules, puis à installer le tout manuellement.

Noyau Linux

Compilation et installation du noyau (méthode classique)

Les étapes détaillées :

Étape 1 : Compilation du noyau

- On lance la compilation à proprement parler via la commande `make`.
- Il est recommandé d'utiliser l'option `-j` pour paralléliser la compilation en fonction du nombre de cœurs CPU disponibles (par ex. `make -j$(nproc)` utilise tous les cœurs).
- Depuis le répertoire racine des sources du noyau :

```
make -j$(nproc)
```

- Cette commande va compiler l'image du noyau (généralement un fichier binaire compressé appelé **bzImage**) ainsi que tous les modules configurés.
- La durée peut varier de quelques minutes à plus d'une heure selon la taille du noyau et la puissance de la machine.
- À la fin, on doit voir un message indiquant que le **bzImage** du noyau est prêt (par exemple : *"Kernel: arch/x86/boot/bzImage is ready"*).
- Si des erreurs surviennent, il faudra les corriger (souvent il manque un paquet de développement requis, ou une option de config incompatible).

Noyau Linux

Compilation et installation du noyau (méthode classique)

Étape 2 : Installation des modules

- Une fois le noyau compilé, il faut installer les modules du noyau (les pilotes compilés en module “[M]”).
- Cette étape copie tous les fichiers `.ko` (kernel objects) générés vers le répertoire système approprié (`/lib/modules/<version-du-noyau>/`).
- On l’exécute avec les privilèges root :

```
sudo make modules_install
```

- Cette commande va installer chaque module au bon endroit et mettre à jour l’index des modules (via `depmod`) pour le noyau cible.
- Par exemple, on verra défiler la liste des modules installés (pilotes divers) et un message **DEPMOD** en fin de processus.
- Après cela, le répertoire `/lib/modules/$(make kernelrelease)` contient tous les modules du nouveau noyau.
- **À noter** : si vous recompilez un noyau avec *exactement la même version* qu’un noyau déjà installé, `make modules_install` risque d’écraser le répertoire de modules existant.
- Pour éviter tout conflit, assurez-vous que le *EXTRAVERSION* (ou LOCALVERSION) diffère, ou déplacez/renommez l’ancien dossier de modules avant d’installer (par exemple, sauvegarde en `.old`).

Noyau Linux

Compilation et installation du noyau (méthode classique)

Étape 3 : Installation de l'image du noyau

- Ensuite, on installe le noyau lui-même ainsi que les fichiers associés dans **`/boot`**.
- Si votre distribution utilise un initramfs, il faut également générer cette image de démarrage.
- La manière la plus simple d'effectuer ces opérations est d'utiliser la cible Make **`make install`**, qui copie le noyau compilé et lance les hooks nécessaires :

```
sudo make install
```

- Cette commande copie le fichier du noyau (bzImage) vers `/boot/vmlinuz-<version>`, installe également le fichier **System.map** correspondant (table des symboles du noyau) et parfois le fichier de config utilisé (`/boot/config-<version>`).
- Sur les systèmes modernes (Debian/Ubuntu par exemple), `make install` va automatiquement déclencher la génération de l'**initramfs** et la mise à jour du chargeur de démarrage via des scripts post-install.
- En effet, on voit dans la sortie qu'il exécute des scripts comme **`/etc/kernel/postinst.d/initramfs-tools`** (qui lance `update-initramfs` pour créer l'initrd) et **`.../zz-update-grub`** (qui lance `update-grub`).
- L'**initrd** (ou **initramfs**) est un disque mémoire initial contenant les modules nécessaires au montage des systèmes de fichiers importants au démarrage (par ex. pilotes du contrôleur de disques, du système de fichiers racine, etc.).
- Il se retrouve généralement sous `/boot/initrd.img-<version>`.
- Le fichier **System.map** est la table des symboles du noyau, utile pour le débogage et pour les utilitaires comme klogd.

Noyau Linux

Compilation et installation du noyau (méthode classique)

- Après ces étapes, votre nouveau noyau est installé.
- On peut vérifier que tout est en place dans `/boot` : on doit y voir `vmlinuz-<version>` (l'image binaire du noyau), `initrd.img-<version>` (si votre config nécessite un initrd), et `System.map-<version>`.
- Par exemple :

```
ls -l /boot
# ... vmlinuz-5.15.8, initrd.img-5.15.8, System.map-5.15.8, config-5.15.8 ...
```

- Il ne reste plus qu'à **redémarrer** sur le nouveau noyau.
- Au menu de GRUB, choisissez la nouvelle entrée (généralement sélectionnée par défaut si c'est le noyau le plus récent).
- Une fois le système démarré, confirmez que c'est bien le bon noyau qui tourne avec `uname -r` (qui doit afficher la version que vous avez compilée).
- Par exemple, après l'installation réussie du noyau 4.11.8 dans un système, `uname -a` affiche bien la nouvelle version et la date de compilation correspondante.

Félicitations, vous avez compilé et démarré sur votre propre noyau Linux !

Noyau Linux

Compilation et installation du noyau (méthode classique)

Gestion des modules et firmwares :

- Les modules du noyau ont été installés dans `/lib/modules/<version>/`.
- Vous pouvez à tout moment activer un module (pilote optionnel) en utilisant `modprobe <nom_du_module>` ou le désactiver avec `rmmod`.
- Pensez à ajouter les modules importants à charger au démarrage (ex: via `/etc/modules` sur Debian) si vous avez compilé en module des éléments nécessaires dès l'amorçage.
- Concernant les **firmwares** (microcodes nécessaires à certains périphériques comme les cartes Wi-Fi, GPU, etc.), notez qu'ils ne sont généralement **pas inclus** dans le code source du noyau pour des raisons de licence.
- Beaucoup de pilotes prévoient le chargement d'un fichier firmware depuis `/lib/firmware` au moment de l'initialisation du matériel.
- Assurez-vous donc d'installer les paquets de firmware adéquats de votre distribution le cas échéant (ex: `firmware-linux-nonfree` sur Debian pour les firmwares propriétaires courants).
- Il est aussi possible d'inclure certains firmwares directement dans l'image du noyau en activant l'option **CONFIG_FIRMWARE_IN_KERNEL** et en spécifiant lesquels, mais la méthode la plus simple reste de copier les fichiers firmwares requis dans `/lib/firmware` (ou de passer par les paquets officiels).
- Si un firmware manque, le pilote concerné le signalera dans les logs (dmesg) lors du chargement du module : vous verrez une erreur du type "`firmware: failed to load ...`".

Noyau Linux

Intégration de drivers et outils spécifiques

- Dans certains cas, vous aurez besoin d'ajouter au noyau des éléments supplémentaires :
  - Par exemple un pilote matériel **externe** (qui n'est pas inclus dans les sources officielles) ou un **module tiers** (module externe au noyau, comme les pilotes NVIDIA propriétaires, VirtualBox, etc.)

Noyau Linux

Intégration de drivers et outils spécifiques

Compilation de modules externes avec DKMS :

- L'outil **DKMS (Dynamic Kernel Module Support)** facilite grandement la gestion des modules hors arbre (*out-of-tree*).
- Il permet de compiler automatiquement un module tiers pour chaque version de noyau installée, et de le recompiler en cas de mise à jour du noyau.
- Sur Debian, de nombreux pilotes externes sont fournis sous forme de paquets se terminant par `-dkms`.
- Par exemple, pour ajouter des modules iptables supplémentaires, il suffit d'installer `xtables-addons-dkms` : l'installation du paquet va automatiquement compiler et installer le module pour le noyau courant (à condition que les en-têtes du noyau correspondant soient présents).
- Concrètement, DKMS va récupérer le code source du module (souvent dans `/usr/src/<module>-<version>/`), puis l'intégrer au noyau en cours.
- On peut vérifier l'état avec `dkms status`
- **Avantage :** à chaque installation d'un nouveau noyau, DKMS recompilera automatiquement les modules externes enregistrés, ce qui évite de le faire manuellement. Si vous avez un pilote propriétaire (ex: NVIDIA), son installeur peut utiliser DKMS pour automatiser la recompilation à chaque changement de noyau.

Noyau Linux

Intégration de drivers et outils spécifiques

Ajout de pilotes non inclus par défaut :

- Si le pilote que vous souhaitez ajouter n'existe pas en paquet DKMS, vous avez deux approches.
  - **(1)** *Compilation externe simple:*
    - Si vous disposez des sources du module (par ex. un fichier driver.c + Makefile fournis par le fabricant), vous pouvez le compiler en utilisant les en-têtes du noyau que vous avez installés.
    - Il suffit d'installer le paquet `linux-headers-<version>` correspondant à votre noyau (si vous avez compilé manuellement, vous pouvez générer et installer ces en-têtes avec `make headers_install`).
    - Ensuite, depuis le dossier du module, une commande du type `make -C /usr/src/linux-headers-<version> M=$(pwd) modules` lancera la compilation du module avec le bon contexte.
    - Vous obtiendrez un fichier .ko que vous pourrez charger avec `insmod` ou installer dans `/lib/modules/<version>/extra` puis charger via modprobe.

Noyau Linux

Intégration de drivers et outils spécifiques

Ajout de pilotes non inclus par défaut :

- **(2)** *Recompiler un noyau patché :*
  - C'est la méthode à utiliser si le pilote nécessite des modifications profondes ou un patch du noyau. De nombreux projets fournissent des *patches* à appliquer au code source du noyau vanilla.
  - Par exemple, Debian fournit des paquets `linux-patch-*` contenant des ensembles de patches (sécurité GrSecurity, RT pour temps réel, etc.).
  - Pour appliquer un patch, on place généralement le fichier patch (.patch ou .diff) dans le répertoire des sources et on exécute la commande `patch -p1 < mon_patch.diff` (ou équivalent).
  - Sous Debian, les patches des paquets sont souvent compressés (fichiers `.gz` dans `/usr/src/kernel-patches/`), on peut les appliquer avec zcat comme dans l'exemple ci-dessous :

```
cd ~/kernel/linux-5.10.8
make clean            # nettoyage de l'ancienne compilation
zcat /usr/src/kernel-patches/diffs/monpatch/patchfile.gz | patch -p1
```

Noyau Linux

Intégration de drivers et outils spécifiques

Test et validation des nouveaux modules :

- Une fois un module externe compilé (que ce soit via DKMS ou manuellement), il convient de le tester.
- Assurez-vous que le module apparaît dans la liste avec `modinfo <module.ko>` ou `modprobe -n <nom>` (pour une simulation de chargement).
- Chargez-le avec `sudo modprobe <nom_du_module>` puis vérifiez avec `lsmod` qu'il est bien listé comme chargé.
- Surveillez `dmesg` ou `/var/log/kern.log` pour voir les messages du noyau relatifs au chargement du module : le pilote y logge souvent des informations, ou d'éventuelles erreurs (symboles inconnus, firmware manquant, etc.).
- Si le module remplace une version existante du noyau (par ex. vous testez une version plus récente d'un pilote réseau déjà présent dans le noyau), il peut être nécessaire de *blacklister* l'ancien module ou de démarrer avec l'option modprobe adéquate.
- En cas de succès, testez la fonctionnalité apportée par le module : par exemple, si c'est un pilote de périphérique, vérifiez que le matériel est bien reconnu (apparition d'une interface réseau, montage d'un système de fichiers, etc. selon le cas).

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Loadable Kernel Modules (LKM)

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

Un **Loadable Kernel Module (LKM)** est une extension dynamique du noyau Linux.
Il permet d'ajouter ou de retirer des fonctionnalités à chaud, sans nécessiter de recompiler ou de redémarrer le noyau.

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

- Un **module de noyau** Linux est un morceau de code pouvant être chargé ou déchargé dynamiquement dans le noyau en cours d'exécution.
- Cela permet d'ajouter ou retirer des fonctionnalités (souvent des pilotes de périphériques) sans recompiler ni redémarrer le noyau .
- À l'inverse des composants compilés *en dur* dans le noyau, les modules chargeables (LKM) peuvent être insérés à la demande et retirés une fois inutilisés, offrant un noyau plus modulable et léger.

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

Principe des Loadable Kernel Modules

Avantages :

- Modularité : charger uniquement les modules nécessaires.
- Flexibilité : ajouter ou retirer des fonctionnalités à chaud.
- Maintenance simplifiée : pas besoin de recompiler entièrement le noyau pour ajouter un pilote.

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

- **Un module comporte au minimum deux fonctions spéciales :**
  - Une fonction d'initialisation appelée lors du chargement et une fonction de nettoyage appelée juste avant sa désactivation .
- Historiquement, ces fonctions portent les noms par défaut `init_module` (exécutée par `insmod`) et `cleanup_module` (exécutée par `rmmod`).
- Depuis les noyaux 2.4, il est courant d'utiliser les macros `module_init()` et `module_exit()` pour associer des fonctions personnalisées à l'initialisation et la sortie du module (ce qui permet de nommer librement ces fonctions).
- La fonction d'init réalise généralement l'enregistrement de ressources ou de gestionnaires au sein du noyau (par ex. enregistrement d'un pilote de périphérique), tandis que la fonction de cleanup libère ces ressources afin que le module puisse être déchargé proprement.

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

- Lors de l'implémentation, un module doit inclure les en-têtes kernel nécessaires, en particulier `<linux/module.h>` (obligatoire pour tout module) et souvent `<linux/kernel.h>` (pour les macros de journalisation `KERN_INFO`, etc.).
- On utilise la fonction de log du noyau `printk()` pour émettre des messages (les fonctions d'E/S standard comme `printf` ne sont pas disponibles en espace noyau).
- Il est également recommandé de définir certaines macro-informations (licence, auteur, description...) qui seront visibles via `modinfo`.

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

Par exemple, voici la structure minimale d'un module simple qui affiche un message lors du chargement et du retrait :

```
#include <linux/module.h>
#include <linux/kernel.h>
static int __init hello_init(void) {
    printk(KERN_INFO "Hello, kernel module!\n");
    return 0;  // 0 = succès du chargement
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "Goodbye, kernel module!\n");
}
module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Exemple de module simple affichant des messages");
```

Loadable Kernel Modules (LKM)

Conception d'un module de noyau

Voici les commandes de base pour la gestion des modules du noyau :

Commande	Fonction
`lsmod`	Afficher les modules actuellement chargés
`modprobe`	Charger ou décharger des modules en gérant les dépendances
`insmod`	Charger un module spécifique (sans gestion automatique des dépendances)
`rmmod`	Retirer un module chargé (sans gestion automatique des dépendances)
`modinfo`	Afficher des informations détaillées sur un module

Loadable Kernel Modules (LKM)

Compilation et installation d'un module

Pour compiler un module externe sous Debian, il faut disposer des en-têtes du noyau correspondant à la version du noyau en cours, ainsi que des outils de build.

- Commencez par installer les paquets nécessaires :
  - **Headers du noyau et outils de compilation** :
    - Par exemple, pour le noyau courant :

```
sudo apt update && sudo apt install build-essential linux-headers-$(uname -r)
```

    - Le paquet `build-essential` fournit `make` et `gcc` entre autres, et `linux-headers-$(uname -r)` installe les en-têtes de développement du noyau actuellement exécuté. Assurez-vous d'utiliser la version correspondant au noyau cible.
  - **Code source du module** :
    - Placez le code du module (ex. fichier `hello.c` ci-dessus) dans un dossier dédié.
    - Écrivez ensuite un **Makefile** pour automatiser la compilation.
    - Un Makefile minimal pour un module s'appuie sur le système de build du noyau en spécifiant le répertoire des headers du noyau et le fichier objet du module.

Loadable Kernel Modules (LKM)

Compilation et installation d'un module

Par exemple :

```
# Makefile de compilation du module hello.ko
obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD  := $(shell pwd)

all:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) clean
```

- Ici, `obj-m := hello.o` indique que l'on construit un module à partir du code objet `hello.o`.
- La commande `$(MAKE) -C $(KDIR) M=$(PWD) modules` invoque la compilation des modules en utilisant les makefiles du noyau Linux situés dans `KDIR` (le lien vers les headers du noyau courant).
- Cette invocation va produire un fichier `hello.ko` (module compilé) si tout se passe bien.

Loadable Kernel Modules (LKM)

Compilation et installation d'un module

Compilation du module :

- Exécutez simplement `make` dans le répertoire contenant votre code et le Makefile.
- Le système de build du noyau va compiler le module en utilisant la configuration du noyau courant.
- À l'issue de la compilation, vous devriez obtenir un fichier `hello.ko` (ou similaire) correspondant au module.
- Vous pouvez vérifier les propriétés de ce module avec l'outil **`modinfo`** avant même de le charger : par exemple `modinfo hello.ko` affichera les informations incorporées (nom du module, version, licence, description, dépendances, auteur, etc.) Par exemple :

```
$ modinfo hello.ko
filename:       /home/user/hello/hello.ko
license:        GPL
description:    "Exemple de module simple affichant des messages"
depends:        <aucune>
```

Loadable Kernel Modules (LKM)

Compilation et installation d'un module

Installation du module (optionnel) :

- Pour que le module soit disponible via `modprobe` et chargé automatiquement si besoin, il faut l'installer dans le système.
- Cela consiste généralement à copier le fichier `.ko` dans un répertoire approprié sous `/lib/modules/<version_du_noyau>/` (par exemple dans `/lib/modules/$(uname -r)/extra/`) puis de mettre à jour l'index des modules avec `depmod`.
- Alternativement, si le module est destiné à être utilisé temporairement, on peut le charger directement depuis le répertoire courant sans l'installer, comme montré ci-après.

Loadable Kernel Modules (LKM)

Chargement / déchargement d'un module

Charger un module :

```
sudo insmod hello_module.ko
```

Vérifier qu'il est chargé :

```
lsmod | grep hello_module
```

Vérifier les messages du noyau :

```
dmesg | tail
```

Décharger le module :

```
sudo rmmod hello_module
```

Vérifier les logs du noyau après déchargement :

```
dmesg | tail
```

Loadable Kernel Modules (LKM)

Liste de tous les modules existants

Tous les modules disponibles pour le noyau actuel sont situés dans :

```
/lib/modules/$(uname -r)/
```

Lister tous les modules existants :

```
find /lib/modules/$(uname -r)/ -name '*.ko*'
```

Loadable Kernel Modules (LKM)

Liste de tous les modules existants

Pour voir la liste des modules **actuellement chargés** dans le noyau, on utilise la commande `lsmod`.

- Celle-ci lit simplement le contenu de `/proc/modules` et l'affiche dans un format tabulaire lisible.
- Chaque ligne correspond à un module actif et comporte généralement trois champs :
  - **Module** – le nom du module chargé.
  - **Size** – la taille en mémoire du module (en octets).
  - **Used by** – le compteur d'utilisation et la liste des autres modules qui l'utilisent.
- Un compteur à 0 indique que le module n'est pas requis actuellement et peut être déchargé sans risque. S'il est non nul, il liste les noms des modules dépendants.

Loadable Kernel Modules (LKM)

Liste de tous les modules existants

Par exemple, un extrait de `lsmod` peut donner :

```
$ lsmod | head -n5
Module                  Size  Used by
lp                     20480  0
ppdev                  24576  0
parport                53248  2 lp,ppdev
usbhid                 57344  0
```

- Ici, on voit que les modules *lp* (pilote d'imprimante parallèle) et *ppdev* sont chargés mais non utilisés (compteur 0).
- Le module *parport* est utilisé par 2 modules (lp et ppdev), ce qui est indiqué dans la colonne "Used by".
- De même *usbhid* (pilote générique HID USB) est chargé et pas utilisé par d'autres modules.

Loadable Kernel Modules (LKM)

Liste de tous les modules existants

- L'affichage `lsmod` permet de vérifier rapidement quels pilotes ou fonctionnalités sont actifs.
- C'est équivalent à lire le fichier texte `/proc/modules` qui liste les modules chargés avec les mêmes informations.
- Notons que cette liste ne comprend que les modules **dynamiques**; les fonctionnalités compilées en dur dans le noyau n'y apparaissent pas.

Loadable Kernel Modules (LKM)

Affichage des informations d'un module

Pour obtenir des détails sur un module du noyau (qu'il soit déjà chargé ou simplement disponible), on utilise la commande **`modinfo`**.

- Cet outil extrait les informations intégrées au module, soit à partir du fichier `.ko`, soit à partir de la base des modules installés (il cherche dans `/lib/modules/<version>/` si on lui donne un nom de module sans chemin).
- La syntaxe est simple : `modinfo <nom_module>` ou `modinfo <chemin/vers/module.ko>`.
- Par exemple, pour un module déjà présent sur le système comme **`usbcore`**, on peut exécuter :

```
$ modinfo usbcore
filename:       /lib/modules/5.10.0-19-amd64/kernel/drivers/usb/core/usbcore.ko
description:    USB core driver
author:         {See file}
license:        GPL
alias:          usb-host-class-device
srcversion:     5C6FF1234567890ABCDEF
depends:        usb-common
intree:         Y
name:           usbcore
vermagic:       5.10.0-19-amd64 SMP mod_unload modversions
```

Loadable Kernel Modules (LKM)

Affichage des informations d'un module

On obtient de nombreuses informations utiles :

- Le chemin exact du fichier du module (`filename`), une description textuelle, l'auteur, la licence, d'éventuels alias (noms alternatifs utilisés par le système, par exemple pour l'auto-chargement via udev), la liste des *dépendances* (`depends`) c'est-à-dire les autres modules qui doivent être chargés pour que celui-ci fonctionne, la version magique (`vermagic`) qui doit correspondre à la version du noyau, etc.
- La plupart de ces champs correspondent à des macros que le développeur du module a insérées (`MODULE_DESCRIPTION`, `MODULE_AUTHOR`, `MODULE_LICENSE`, `MODULE_ALIAS`, etc.).
- On peut utiliser `modinfo` aussi bien sur un module *déjà chargé* que sur un module simplement installé sur le disque.
- Si le module est chargé, les informations proviennent du fichier sur le disque (il n'interroge pas le noyau en direct).
- Par exemple, `modinfo hello.ko` (notre module d'exemple) ou `modinfo hello` (si installé dans `/lib/modules`) affichera la licence GPL et la description qu'on avait définies dans le code, confirmant que le module est bien conforme à nos attentes.

Loadable Kernel Modules (LKM)

Gestion des dépendances de modules

- Les modules du noyau peuvent dépendre les uns des autres.
- Par exemple, le module d'un périphérique réseau sans fil peut dépendre d'un module de pile crypto ou d'un module de bus PCI.
- Pour gérer automatiquement ces **dépendances**, Linux utilise un fichier d'index appelé `modules.dep` qui énumère, pour chaque module, la liste de ses éventuelles dépendances.

Loadable Kernel Modules (LKM)

Gestion des dépendances de modules

Génération de modules.dep (depmod) :

- Le fichier `/lib/modules/<version>/modules.dep` est généré par l'utilitaire `depmod`.
- À chaque installation ou mise à jour de modules (par ex. après l'installation d'un nouveau noyau ou d'un module tiers), on exécute `depmod -a` pour analyser tous les modules disponibles et recalculer leurs dépendances.
- `depmod` va lire chaque fichier `.ko` et déterminer de quels symboles ou autres modules il a besoin, puis écrire le résultat dans `modules.dep` (et un fichier binaire `modules.dep.bin`).
- Cela permet à d'autres outils de connaître instantanément les dépendances sans avoir à analyser les fichiers à chaque fois.
  Par exemple, si le module *abc.ko* a besoin des modules *def.ko* et *ghi.ko*, `modules.dep` contiendra une ligne du type :

```
/lib/modules/5.10.0-19-amd64/.../abc.ko: /lib/modules/5.10.0-19-amd64/.../def.ko /lib/modules/5.10.0-19-amd64/.../ghi.ko
```

  indiquant à modprobe qu'il faut charger *def* et *ghi* avant *abc*.

Loadable Kernel Modules (LKM)

Gestion des dépendances de modules

Utilisation de `modprobe` vs `insmod`:

- Grâce à `modules.dep`, la commande `modprobe` peut automatiquement charger tous les modules dont dépend un module cible.
- Par exemple, si *foo* dépend de *bar* et *baz*, un simple `modprobe foo` insérera *bar*, *baz* puis *foo* dans le bon ordre.
- C'est pourquoi il est **recommandé d'utiliser `modprobe`** pour charger les modules plutôt que `insmod`.
- `insmod` ne fait aucune résolution :
  - si vous insérez *foo.ko* sans avoir préalablement inséré ses dépendances, l'appel échouera (ou le module foo plantera car il ne trouve pas ce dont il a besoin).
  - En pratique, on réserve `insmod` aux tests rapides en connaissant exactement l'ordre de chargement, ou pour des cas très spécifiques.
  - Pour une utilisation courante, `modprobe` gère tout automatiquement en s'appuyant sur `modules.dep`.

Loadable Kernel Modules (LKM)

Gestion des dépendances de modules

Mise à jour des dépendances:

- Sur Debian, lors de l'installation d'un nouveau noyau ou module, le script d'installation exécute généralement `depmod -a` automatiquement.
- Cependant, si vous ajoutez manuellement un fichier `.ko` dans `/lib/modules/...`, pensez à lancer `sudo depmod -a` avant de tenter un `modprobe` ou un `modinfo` sur ce module.
- Sans cela, `modinfo` ou `modprobe` risquent de ne pas le trouver ou de ne pas connaître ses dépendances, puisque `modules.dep` n'aura pas été mis à jour (d'où des erreurs *"Module not found"* possibles).

Loadable Kernel Modules (LKM)

Blocage d'un module

- Il peut arriver que l'on veuille empêcher le chargement d'un module particulier, par exemple pour désactiver un pilote gênant ou qui entre en conflit avec un autre
- Deux mécanismes principaux existent pour **blacklister** un module sous Debian : via la configuration de modprobe, ou via les options du noyau au démarrage.

Loadable Kernel Modules (LKM)

Blocage d'un module

Blacklister via modprobe (fichier de config) :

- On peut interdire le chargement automatique d'un module en le listant dans un fichier de configuration de modprobe.
- Le fichier commun est `/etc/modprobe.d/blacklist.conf` (ou on peut créer un fichier séparé `.conf`).
- Il suffit d'y ajouter une ligne :

```
blacklist <nom_du_module>
```

- Par exemple, pour blacklister le module `nouveau` (pilote libre Nvidia), on ajouterait `blacklist nouveau`. Après cela, il est conseillé d'exécuter `depmod -ae` puis de régénérer l'initramfs : `sudo update-initramfs -u`.
- Ceci assure que lors du prochain démarrage, le module ne sera pas chargé automatiquement, y compris pendant la phase d'initramfs.
- **Attention** : la blacklist modprobe empêche le chargement *automatique* du module (par udev ou autres mécanismes), mais n'empêche pas un administrateur de le charger manuellement par la suite.
- En effet, un `sudo modprobe <module>` explicitera forcera le chargement même s'il est blacklisté dans modprobe.d.
- Pour réellement bloquer toute insertion, y compris manuelle, on peut utiliser une directive *install* qui redirige l'action vers `/bin/false`. Par exemple, dans `/etc/modprobe.d/blacklist.conf` :

```
install <nom_du_module> /bin/false
```

Loadable Kernel Modules (LKM)

Blocage d'un module

Blacklister via une option du noyau (GRUB):

- L'autre méthode consiste à passer une option au noyau Linux pour qu'il ignore un module. On utilise la syntaxe `modprobe.blacklist=<nom_module>` dans la ligne de commande du noyau.
- Sous Debian, on peut éditer le fichier `/etc/default/grub` et ajouter le ou les modules à blacklister dans la variable `GRUB_CMDLINE_LINUX`.
- Par exemple :

```
GRUB_CMDLINE_LINUX="modprobe.blacklist=nouveau"
```

  - On peut l'ajouter aux autres paramètres existants, séparé par des espaces.
  - Pour plusieurs modules, on peut les lister séparés par des virgules : ex. `modprobe.blacklist=nouveau,firewire_ohci`.
  - Ensuite, exécutez `sudo update-grub`.
  - Au prochain redémarrage, le noyau ne chargera pas ces modules blacklistés au démarrage.
  - Cela est équivalent à la méthode précédente, mais agit dès les premiers instants du boot, avant même que l'initramfs ne charge quoi que ce soit.
  - Cette technique est souvent utilisée pour désactiver *nouveau* en vue d'installer le pilote propriétaire Nvidia.

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

- Dans certaines situations, on peut souhaiter compiler un noyau personnalisé où certains modules sont directement **intégrés en dur** dans l'image du noyau (plutôt qu'en modules séparés).
- Sous Debian, il est tout à fait possible de compiler son propre noyau tout en conservant un emballage `.deb` propre pour l'installer.

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

1) Préparer l'environnement de compilation – Installez les outils nécessaires :

```
sudo apt install build-essential kernel-package fakeroot libncurses-dev
```

- *build-essential* (gcc, make, etc.) et *libncurses-dev* (pour l'interface textuelle de configuration du noyau) sont indispensables.
- *kernel-package* fournit l'outil `make-kpkg` pour construire des noyaux Debian.
- *fakeroot* permettra de construire le paquet sans droits root.

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

2) Obtenir les sources du noyau – Deux approches :

- soit récupérer les sources fournies par Debian (paquet `linux-source-<version>`),
- soit télécharger les sources kernel.org de la version souhaitée.
- Pour un noyau Debian, il est recommandé d'utiliser `apt` : par ex. `sudo apt install linux-source-5.10`.
- Ceci placera une archive (typiquement dans `/usr/src/`) que vous devrez décompresser :

```
cd /usr/src/
tar xf linux-source-5.10.tar.xz
cd linux-source-5.10
```

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

3) Configurer le noyau (make menuconfig) –

- Avant de compiler, il faut configurer quels composants seront inclus.
- Vous pouvez partir de la config actuelle du noyau Debian (fichier `/boot/config-$(uname -r)`) en le copiant dans `.config` puis en lançant `make oldconfig` ou `make menuconfig`.
- La commande `make menuconfig` ouvre une interface en mode texte qui permet de parcourir toutes les options du noyau.
- C'est ici que vous pouvez choisir d'intégrer un module en dur.
- Naviguez jusqu'à l'option correspondant à la fonctionnalité/module désiré et changez son état :
- `[*]` (Y) = compilé en dur dans le noyau (builtin)
- `[M]` = compilé en module chargeable
- `[ ]` (N) = non inclus du tout

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

4) Compilation du noyau et des modules –

- Utilisez maintenant `make-kpkg` pour compiler et empaqueter le noyau.
- Cet outil va automatiser la compilation (`make`) puis construire un paquet .deb installable.
- Par exemple :

```
export CONCURRENCY_LEVEL=4    # pour accélérer la compilation sur multi-cœurs (facultatif)
fakeroot make-kpkg --initrd --revision=1.0-custom kernel_image kernel_headers
```

- Cette commande compile le noyau avec génération d'une image initrd (--initrd) et assigne un numéro de révision personnalisé.
- Elle produira deux fichiers .deb dans le répertoire parent : l'image du noyau (`linux-image-...custom_amd64.deb`) et les en-têtes (`linux-headers-...custom_amd64.deb`).
- Vous pouvez ajuster `--revision` pour identifier votre build.
- La compilation peut prendre un certain temps selon la puissance de votre machine et le nombre de modules/niveaux d'optimisation du noyau.

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

5) Installation du noyau personnalisé –

- Une fois les paquets `.deb` générés, installez-les via dpkg :

```
sudo dpkg -i ../linux-image-5.10.0-custom_amd64.deb ../linux-headers-5.10.0-custom_amd64.deb
```

- Le paquet `linux-image` va placer le fichier du noyau (`vmlinuz-5.10.0-custom`) dans `/boot` ainsi que les modules compilés dans `/lib/modules/5.10.0-custom/`.
- Il va également générer un initrd incluant vos modules nécessaires (puisque on avait passé --initrd) et mettre à jour le chargeur **GRUB** automatiquement pour ajouter une entrée pour ce noyau (grâce aux scripts *post-install* Debian).
- Le paquet `linux-headers` n'est pas strictement nécessaire pour le fonctionnement du noyau, mais utile si vous voulez compiler d'autres modules externes pour ce noyau.

Loadable Kernel Modules (LKM)

Création d'un noyau personnalisé

6) Redémarrage sur le nouveau noyau

- Mettez à jour GRUB le cas échéant (`sudo update-grub`) – normalement fait automatiquement – puis redémarrez en choisissant votre noyau personnalisé.
- Une fois démarré, vous pouvez vérifier avec `uname -r` que c'est bien la version custom.
- Vos **modules intégrés** apparaîtront comme faisant partie du noyau : ils **ne figureront pas dans** `lsmod` (puisqu'ils ne sont pas dynamiques).
- Par exemple, si vous aviez intégré un pilote auparavant modulaire, la commande `lsmod` ne le listera plus, mais la fonctionnalité sera bien active (et visible éventuellement via `/proc/devices` ou autres).
- D'autre part, ces modules intégrés ne pourront pas être retirés à chaud (pas de `rmmod` possible) – il faudrait recompiler un noyau pour les enlever.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

"/proc" et "/sys"

"/proc" et "/sys"

Présentation du pseudo-système de fichiers /proc

- Le répertoire `/proc` est un **pseudo-système de fichiers** exposant des informations du noyau.
- On parle de *pseudo*-système car son contenu est généré dynamiquement en mémoire par le noyau, et non stocké de façon permanente sur le disque.
- En pratique, cela signifie que lire un fichier dans `/proc` interroge directement le noyau pour obtenir la donnée demandée.
- Ces fichiers apparaissent vides (`0` octet) si on liste leur taille avec `ls`, ce qui illustre qu'ils n'occupent pas d'espace disque réel (seule une petite quantité de mémoire vive est utilisée).
- Le pseudo-filesystem `/proc` joue un rôle d'**interface entre l'utilisateur et les structures de données du noyau** .
- Il a initialement été conçu pour fournir des infos sur les processus en cours d'exécution, d'où son nom *proc* (processus), mais il englobe aujourd'hui bien d'autres informations système.
- Sur Debian (comme sur la plupart des distributions Linux), `/proc` est **monté automatiquement** au démarrage par le système (généralement par le noyau ou le programme d'initialisation).
- Il est normalement toujours disponible, sans intervention manuelle.
- À titre indicatif, on pourrait le monter soi-même via `mount -t proc proc /proc`, mais cela n'est généralement pas nécessaire puisque le boot s'en occupe.

"/proc" et "/sys"

Informations contenues dans `/proc`

- Le contenu de `/proc` est organisé sous forme de fichiers et de répertoires virtuels reflétant l'état du système.
- On y trouve à la fois des **fichiers globaux** décrivant le système et des **répertoires par processus**.
- La plupart de ces fichiers sont en lecture seule (pour consulter des informations), mais certains sont modifiables pour ajuster des paramètres du noyau (voir section sysctl).

"/proc" et "/sys"

Informations contenues dans `/proc`

- **`/proc/cpuinfo`** – Détails sur les processeurs : modèle, fréquence, nombre de cœurs, fonctionnalités, etc. En affichant ce fichier (`cat /proc/cpuinfo`), on peut identifier le CPU présent et ses caractéristiques.
- **`/proc/meminfo`** – Informations sur la mémoire vive et le swap : quantités totales, libres, en cache, tampon, etc. Ce fichier est utilisé par des outils comme la commande `free` pour résumer l'utilisation mémoire du système .
- **`/proc/uptime`** – Données sur le temps d'activité du système. Les deux nombres qu'il contient correspondent respectivement au nombre de secondes écoulées depuis le dernier démarrage et au temps total d'inactivité des processeurs (somme des temps idle).
- **`/proc/version`** – Version du noyau et informations sur la compilation de celui-ci. Par exemple, il indique la version exacte de Linux en cours d'exécution.
- **`/proc/filesystems`** – Liste des systèmes de fichiers pris en charge par le noyau.
- **`/proc/interrupts`** – Compteurs d'interruptions matériel par processeur (permet de voir l'utilisation des IRQ par les périphériques).
- **`/proc/swaps`** – Liste des espaces de swap utilisés et leur utilisation.
- **`/proc/loadavg`** – Charge moyenne du système (moyennes sur 1, 5 et 15 minutes, nombre de processus en cours, PID récent).

"/proc" et "/sys"

Informations contenues dans `/proc`

- Pour **consulter ces fichiers**, on utilise les commandes shell classiques.
- Par exemple, `cat /proc/cpuinfo` affichera le détail des processeurs.
- On peut combiner avec `grep` pour filtrer une information précise :

```
$ grep "model name" /proc/cpuinfo
```

- Cette commande retournera la ligne du modèle de chaque CPU (pratique pour repérer le modèle du processeur).
- De même, `grep MemTotal /proc/meminfo` donnera la quantité de RAM totale.
- On peut aussi utiliser `find` pour parcourir l'arborescence de `/proc`.
- Par exemple, `find /proc -maxdepth 1 -name "*meminfo*"` permettrait de vérifier l'existence du fichier `meminfo` (ici au niveau 1).

"/proc" et "/sys"

Informations contenues dans `/proc`

Fichiers par processus (`/proc/<PID>/` et `/proc/self/`) :

- Chaque processus actif dans le système possède un répertoire dédié sous `/proc`, nommé d'après son identifiant de processus (PID).
- Par exemple, le processus ayant PID 1234 aura un répertoire `/proc/1234`.
- Ce répertoire contient de nombreux fichiers fournissant des renseignements sur le processus en question :
  - **`/proc/<PID>/cmdline`** – La ligne de commande ayant lancé le processus (y compris les arguments).
  - **`/proc/<PID>/status`** – Un résumé en texte lisible de l'état du processus : utilisateur propriétaire, utilisation mémoire, état (actif, suspendu…), PID parent, etc.
  - **`/proc/<PID>/cwd`** – Lien symbolique vers le *current working directory*. La commande utilitaire `pwdx <PID>` exploite ce lien pour afficher le répertoire courant d'un processus (procfs — Wikipédia).
  - **`/proc/<PID>/exe`** – Lien vers l'exécutable en cours d'utilisation pour ce processus.
  - **`/proc/<PID>/fd/`** – Répertoire contenant les descripteurs de fichiers ouverts par le processus (chaque entrée est un lien vers les fichiers ou ressources que le processus a ouverts).
  - **`/proc/<PID>/maps`**, **`/proc/<PID>/smaps`** – Détails sur la mémoire virtuelle allouée (segments mémoire).
  - **`/proc/<PID>/task/`** – Sous-répertoire listant les threads du processus (chaque thread ayant un sous-répertoire par TID).

"/proc" et "/sys"

Informations contenues dans `/proc`

Droits d'accès :

- La plupart des informations de `/proc` sont lisibles par tous les utilisateurs, mais certaines sont restreintes.
- Par exemple, un utilisateur ne peut normalement pas lire les détails d'un processus appartenant à un autre utilisateur (surtout sur les systèmes configurés avec l'option de montage `hidepid` pour `/proc`).
- De même, la modification des fichiers (dans `/proc/sys` notamment) requiert les privilèges super-utilisateur.

"/proc" et "/sys"

Modification des paramètres du noyau avec `sysctl`

L'utilitaire `sysctl` permet de consulter et de modifier à chaud des paramètres du noyau Linux.

- Ces paramètres affectent directement le comportement du système d'exploitation dans plusieurs domaines, notamment :
  - **Réseau** (ex : TCP/IP, routage, sécurité réseau)
  - **Gestion de mémoire**
  - **Sécurité du système**
  - Performances globales du système

"/proc" et "/sys"

Modification des paramètres du noyau avec `sysctl`

- Le noyau Linux expose un certain nombre de paramètres modifiables à chaud via l'espace `/proc/sys/`.
- Ces paramètres, aussi appelés *variables sysctl*, peuvent être consultés et ajustés au vol grâce à la commande **`sysctl`**.
- L'utilitaire `sysctl` fournit une interface en ligne de commande plus conviviale que la manipulation manuelle des fichiers dans `/proc/sys`.
- En arrière-plan, `sysctl` lit et écrit justement dans ces fichiers du pseudo-fichier système procfs.

"/proc" et "/sys"

Modification des paramètres du noyau avec `sysctl`

On peut lister **tous les paramètres disponibles** avec :

```
$ sysctl -a
```

- Cette commande affiche la totalité des clés sysctl et leurs valeurs actuelles.
- La liste est longue (des centaines de paramètres) couvrant divers sous-systèmes :
  - noyau pur (`kernel.*`),
  - réseau (`net.*`),
  - mémoire virtuelle (`vm.*`),
  - sécurité (`fs.*` pour filesystems, etc.).
- On peut filtrer l'affichage, par exemple ne montrer que les paramètres réseau : `sysctl -a | grep '^net.'` ou utiliser l'option intégrée `sysctl --pattern '^net.'`.

"/proc" et "/sys"

Modification des paramètres du noyau avec `sysctl`

Pour **consulter une clé spécifique**, on utilise son nom complet, par exemple :

```
$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
```

- Ce paramètre `net.ipv4.ip_forward` indique si le routage IP est activé (0 = désactivé, 1 = activé).
- L'exemple ci-dessus montre une sortie typique :
  - par défaut la plupart des distributions mettent `ip_forward` à 0 (la machine ne fait pas office de routeur).
  - À noter qu'on obtiendrait le même résultat en affichant directement le fichier correspondant : `cat /proc/sys/net/ipv4/ip_forward` – les deux méthodes sont interchangeables.

"/proc" et "/sys"

Modification des paramètres du noyau avec `sysctl`

- **Modifier une valeur temporairement :**

```
sudo sysctl -w net.ipv4.ip_forward=1
```

  - Cette modification est immédiate mais non persistante après redémarrage.

- **Rendre les modifications permanentes :**

  Éditez le fichier `/etc/sysctl.conf` :

```
sudo nano /etc/sysctl.conf
```

  Ajouter la ligne suivante :

```
net.ipv4.ip_forward = 1
```

- **Appliquer les modifications immédiatement :**

```
sudo sysctl -p
```

"/proc" et "/sys"

Présentation du pseudo-système de fichiers `/sys`

- Le répertoire `/sys` correspond au pseudo-système de fichiers appelé **sysfs**.
- Introduit avec le noyau Linux 2.6, il a pour objectif d'exposer à l'espace utilisateur une vue unifiée des **périphériques matériels et de leurs pilotes**.
- Comme `/proc`, il s'agit d'un filesystem virtuel maintenu en mémoire (basé à l'origine sur *ramfs*).
- Toutefois, son contenu et sa finalité diffèrent de `/proc` : `/sys` est organisé selon la structure interne du noyau (le *Device Tree* ou arbre des périphériques) et vise principalement à représenter la configuration matérielle du système.
- Historiquement, avant Linux 2.6, le répertoire `/proc` avait commencé à accumuler des informations ne concernant pas directement les processus (par exemple des données sur le matériel, les pilotes, etc.), ce qui le rendait moins lisible.
- Sysfs a été conçu pour **désengorger `/proc` en déplaçant dans `/sys` toutes les informations relatives aux périphériques** et aux sous-systèmes du noyau.
- Concrètement, `/sys` offre une vision arborescente de tous les composants matériels (bus, périphériques, drivers…) du système, séparée des informations purement liées aux processus que l'on trouve dans `/proc`.

"/proc" et "/sys"

Présentation du pseudo-système de fichiers `/sys`

- Comme `/proc`, le pseudo-fichier système `/sys` est monté automatiquement au démarrage.
- Sur les systèmes à init **systemd**, celui-ci monte `sysfs` très tôt dans la séquence de boot.
- Sur Debian/Ubuntu classiques (SysVinit ou systemd), on trouve également souvent une entrée dans `/etc/fstab` du type: `sysfs /sys sysfs defaults 0 0`, indiquant de monter sysfs sur `/sys`.
- Dans la pratique, quelle que soit la distribution (Debian, Arch, Fedora...), `/sys` **est toujours monté par le système au boot** car des composants critiques comme udev en dépendent pour détecter le matériel.
- Il n'est donc pas nécessaire de le monter manuellement (sauf en environnement minimaliste ou en chroot, le cas échéant).

"/proc" et "/sys"

Informations contenues dans `/sys`

La structure de `/sys` est hiérarchique et reflète l'architecture interne du noyau. À la racine de `/sys`, on trouve notamment :

- **`/sys/devices/`** – Regroupe les périphériques physiques par hiérarchie matérielle.
  - On y voit l'arborescence réelle : par exemple, sous `/sys/devices/system/cpu/` se trouvent les CPU, sous `/sys/devices/pci0000:00/` les appareils sur le bus PCI racine, etc.
  - C'est ici qu'apparaissent concrètement tous les dispositifs détectés dans le système (disques, interfaces réseau, USB, etc.), organisés par bus et connexions.
- **`/sys/class/`** – Vue logique par *classe* de périphériques.
  - Les *classes* regroupent des périphériques ayant des fonctions similaires, indépendamment de leur localisation sur un bus.
  - Par exemple, la classe `net` contient toutes les interfaces réseau du système (eth0, wlan0, lo, etc.), la classe `block` contient tous les périphériques de stockage en bloc (sda, sdb, loop0, ...), la classe `tty` les terminaux, etc.
  - Parcourir `/sys/class` permet de trouver rapidement un type de périphérique sans connaître sa position exacte dans `/sys/devices`.

"/proc" et "/sys"

Informations contenues dans `/sys`

- **`/sys/bus/`** – Vue par bus systèmes.
  - Ce répertoire contient une entrée pour chaque type de bus présent dans le noyau (pci, usb, spi, i2c, etc.).
  - À l'intérieur, on peut voir les périphériques attachés à chaque bus (`/sys/bus/pci/devices/...` par ex.) ainsi que les drivers associés (`/sys/bus/pci/drivers/...`).
  - C'est une autre manière de naviguer vers les mêmes informations, mais classées par bus de communication.
- **`/sys/modules/`** – Informations sur les modules du noyau chargés.
  - Chaque module (pilote ou composant du noyau chargé dynamiquement) a un dossier ici, contenant notamment un fichier `parameters/` qui expose les paramètres modulables de ce module.
  - Par exemple, si un module pilote accepte des options (comme la taille de buffer, etc.), elles seront visibles et modifiables via des fichiers dans `/sys/modules/<nom_module>/parameters/`.
- **`/sys/kernel/`** – Données spécifiques au noyau lui-même.
  - On y trouve par exemple des informations de debug (`/sys/kernel/debug/` si monté), des réglages de sécurité (`/sys/kernel/security/`), ou la configuration à chaud du noyau (`/sys/kernel/mm/` pour la mémoire, etc.).
  - Ce répertoire peut également accueillir d'autres pseudo-systèmes de fichiers comme *configfs* (généralement monté sur `/sys/kernel/config`).

"/proc" et "/sys"

Utilitaire `sysTool` (systool)

- **`systool`** (à ne pas confondre avec *sysctl*) est un utilitaire en ligne de commande permettant d'interroger l'arborescence de sysfs plus aisément.
- Fourni par le paquet `sysfsutils` sur Debian/Ubuntu, il sert à lister les périphériques et leurs attributs par bus, classe ou module kernel, en utilisant l'API de libsysfs.
- En d'autres termes, `systool` offre une vue formatée de ce qui se trouve sous `/sys`, ce qui peut être plus pratique que de parcourir manuellement les répertoires.

"/proc" et "/sys"

Utilitaire `sysTool` (systool)

- **Présentation et installation :**
  - Sur Debian, pour disposer de `systool`, il faut installer le paquet **`sysfsutils`** (s'il ne l'est pas déjà).
  - Ce paquet contient également un fichier de configuration `/etc/sysfs.conf` permettant de définir des valeurs à écrire dans certains nœuds de sysfs au démarrage du système.
  - Sur Arch Linux et Fedora, `systool` n'est pas installé par défaut non plus, mais on le trouve respectivement dans le paquet `sysfsutils` (installation via pacman) et dans les dépôts officiels (installation via dnf/yum).

"/proc" et "/sys"

Utilitaire `sysTool` (systool)

Usage courant :

- Sans argument, la commande `systool` affiche tous les types de bus, toutes les classes de périphériques et tous les périphériques racine disponibles sur le système. Cela donne une vision d'ensemble de la hiérarchie du matériel.
- On peut affiner la requête avec des options :
- `systool -b <bus>` – Liste les périphériques et informations pour un bus donné. Par exemple `systool -b pci` affichera la liste des périphériques PCI et leurs attributs (identifiants, ressources, driver associé, etc.).
- `systool -c <classe>` – Affiche les périphériques d'une **classe** spécifique.
- `systool -m <module>` – Donne des informations sur un **module du noyau** chargé. C'est très utile pour voir les paramètres d'un pilote.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Dépannage matériel

Dépannage matériel

Types de problèmes matériels

Les serveurs peuvent rencontrer divers problèmes matériels courants affectant le **CPU**, la **mémoire RAM**, le **stockage**, l'**alimentation** ou le **réseau**.

- Par exemple, une surchauffe ou un défaut du CPU peut provoquer des ralentissements, des **kernel panic** ou des arrêts subits du système.
- Une panne de **RAM** (barrette défectueuse) peut causer des erreurs aléatoires, comme des processus qui se terminent de façon inattendue ou des **segmentation faults**, voire des crashs du noyau qui semblent logiciels mais résultent en fait d'une défaillance matérielle.
- Un disque de **stockage** (HDD/SSD) en fin de vie peut générer des erreurs d'**E/S** (entrées/sorties) dans les journaux, des secteurs illisibles ou des performances très dégradées. Une alimentation électrique défaillante peut entraîner des redémarrages intempestifs ou l'impossibilité de démarrer le serveur.
- Une carte **réseau** défectueuse peut provoquer des coupures réseau, une perte de connectivité ou des paquets erronés.

Dépannage matériel

Analyse du matériel

L'analyse du matériel sur Linux repose sur plusieurs outils et techniques :

a) Outils d'inventaire et d'affichage des caractéristiques matérielles

- **`lspci`**
  Affiche les périphériques PCI connectés.
  Exemple :

```
lspci -v
```

  La commande fournit une description détaillée des contrôleurs, cartes graphiques, interfaces réseau, etc.

- **`lsusb`**
  Affiche les périphériques USB connectés.
  Exemple :

```
lsusb -v
```

  Ceci permet de vérifier si les périphériques USB sont bien détectés et quels drivers y sont associés.

Dépannage matériel

Analyse du matériel

- **`lshw`**
  Donne un inventaire complet du matériel, avec des détails sur la configuration et les capacités.
  Exemple :

```
sudo lshw -short
```

  Cette commande offre un résumé facile à lire et peut être utilisée pour générer un rapport complet.

- **`inxi`**
  Un outil d'informations système très complet (souvent installé manuellement sur Debian/Ubuntu) qui fournit une vue globale sur le matériel et le système.
  Exemple :

```
inxi -F
```

Dépannage matériel

Analyse du matériel

- **`dmidecode`**
  Permet de lire les informations du BIOS et de la DMI (Desktop Management Interface).
  Exemple :

```
sudo dmidecode
```

  Cela permet d'obtenir des informations sur le fabricant, le modèle de la carte mère, la version du BIOS, la configuration de la mémoire, etc.

Dépannage matériel

Analyse du matériel

b) Analyse des incidents matériels dans les logs

- `dmesg`
  Affiche le journal du noyau, qui contient de nombreux messages sur le matériel et les pilotes.
  Exemple :

```
dmesg | less
```

  Recherchez des mots-clés comme *error*, *fail*, *timeout* ou *warn* pour identifier des incidents.
  Par exemple,

```
dmesg | grep -i error
```

  permet de filtrer les erreurs signalées par le noyau.

Dépannage matériel

Analyse du matériel

b) Analyse des incidents matériels dans les logs

- **`journalctl`** (pour les systèmes utilisant systemd)
  Permet de consulter les logs système et du noyau.
  Exemple :

```
sudo journalctl -k | grep -i usb
```

  Cela permet d'identifier des incidents liés aux périphériques USB ou à d'autres composants matériels.

- **Fichiers de logs spécifiques**
  Certains problèmes matériels peuvent aussi être consignés dans `/var/log/syslog`, `/var/log/messages` ou `/var/log/kern.log` (selon la distribution). Vous pouvez y chercher des indices sur des dysfonctionnements matériels.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Logicial Volume Manager (LVM)

Logicial Volume Manager (LVM)

Rappel des principaux systèmes de fichiers

- **ext2** : Ancien système sans journalisation, peu fiable en cas de crash, rarement utilisé aujourd'hui.
- **ext3** : Ajoute la journalisation à ext2, permettant une récupération plus rapide après un crash.
- **ext4** : Système moderne, rapide, fiable, supportant de grandes tailles de fichiers (jusqu'à 1 EiB).
- **XFS** : Optimisé pour la gestion des grands fichiers et systèmes de stockage massifs, performant pour les serveurs.
- **ZFS** : Système avancé avec intégrité des données, gestion RAID intégrée, et snapshots intégrés.

Logicial Volume Manager (LVM)

Description détaillée de LVM et Device Mapper

- Le *Logical Volume Manager (LVM)* est un système de gestion de volumes logiques présent sur les systèmes Linux.
- Au lieu d'utiliser directement des partitions fixes sur un disque dur (ou des ensembles de disques), LVM permet de créer et d'administrer des volumes logiques plus flexibles, abstraits du matériel sous-jacent.
- Ainsi, on peut redimensionner, déplacer ou fusionner des volumes avec plus de souplesse qu'avec des partitions classiques.

Logicial Volume Manager (LVM)

Description détaillée de LVM et Device Mapper

- Le *Device Mapper* est un composant du noyau Linux (une couche d'abstraction) qui sert de fondation pour créer et gérer divers dispositifs de stockage virtuels.
- Il fournit un mécanisme générique de "mapping" entre des périphériques bloc virtuels et des périphériques bloc physiques réels
- Les outils de LVM s'appuient sur ce Device Mapper pour créer leurs volumes logiques.

Logicial Volume Manager (LVM)

Description détaillée de LVM et Device Mapper

- **Le Device Mapper** est la brique de base au niveau du noyau qui permet de :
  1. Définir des tables de mapping pour répartir des blocs (secteurs, clusters) réels sur un ou plusieurs périphériques virtuels (ex. un volume logique).
  2. Mettre en place des fonctionnalités comme la création de *snapshots*, le chiffrement au niveau bloc (via dm-crypt), la mise en miroir (*mirroring*), etc.

Logicial Volume Manager (LVM)

Gestion des Volume Groups (VG), des Physical Volumes (PV) et des Logical Volumes (LV)

LVM introduit une couche d'abstraction qui repose sur trois concepts principaux :

1. **Physical Volumes (PV)**
   - Correspondent aux disques ou partitions physiques (par exemple, `/dev/sda1`, `/dev/sdb`, etc.) marqués pour être gérés par LVM.
   - Chaque PV est découpé en "Physical Extents" (PE), unités de stockage de taille fixe (par défaut 4 Mo).

Logicial Volume Manager (LVM)

Gestion des Volume Groups (VG), des Physical Volumes (PV) et des Logical Volumes (LV)

LVM introduit une couche d'abstraction qui repose sur trois concepts principaux :

2. **Volume Groups (VG)**
   - Un Volume Group est un regroupement de un ou plusieurs PV.
   - L'ensemble des Physical Extents (PE) de ces PV constitue une "pool" de stockage commune.
   - Une fois un VG créé, on peut y créer autant de volumes logiques que l'espace le permet.

Logicial Volume Manager (LVM)

Gestion des Volume Groups (VG), des Physical Volumes (PV) et des Logical Volumes (LV)

LVM introduit une couche d'abstraction qui repose sur trois concepts principaux :

3. **Logical Volumes (LV)**
    - Ce sont les volumes logiques à proprement parler. On peut les considérer comme l'équivalent "virtuel" d'une partition.
    - Les LV sont eux-mêmes découpés en "Logical Extents" (LE), qui correspondent en pratique aux Physical Extents sous-jacents.
    - Sur un LV, on installe un système de fichiers (ext4, xfs, btrfs, etc.) ou un autre type de service (swap, etc.).
    - Les LV bénéficient de la flexibilité d'être redimensionnables, déplacés, clonés, etc.

Logicial Volume Manager (LVM)

Gestion des Volume Groups (VG), des Physical Volumes (PV) et des Logical Volumes (LV)

LVM permet de :

- Ajouter à chaud un nouveau disque dans un Volume Group, augmentant ainsi la capacité totale du “pool” de stockage.
- Redimensionner un volume logique (l’agrandir ou le réduire, selon les contraintes du système de fichiers) sans avoir besoin de restructurer physiquement toutes les partitions sur le disque.
- Déplacer des extents d’un disque vers un autre, par exemple pour retirer un disque physique d’un groupe, ou pour équilibrer la charge de stockage.
- Créer des *snapshots* de volumes (copies instantanées *read-only* ou *read-write*), notamment utiles pour la sauvegarde ou la duplication.

Logicial Volume Manager (LVM)

Gestion des Volume Groups (VG), des Physical Volumes (PV) et des Logical Volumes (LV)

Comment LVM utilise Device Mapper

- Au niveau du noyau, chaque Logical Volume géré par LVM est exposé comme un périphérique bloc virtuel dans `/dev/mapper/…` ou parfois `/dev/VGName/LVName`.
- Sous le capot, LVM transmet au *Device Mapper* une table de correspondances qui précise : “tel segment de ce volume logique correspond à telle zone sur le disque `/dev/sdb1`, tel autre segment correspond à `/dev/sdc1`, etc.”.
- Le Device Mapper se charge ensuite d’agréger ces blocs physiques en un seul périphérique bloc virtuel cohérent.

Logicial Volume Manager (LVM)

Extensions Physiques (PE) et Extensions Logiques (LE)

1. Physical Extents (PE)

1. **Définition :**
   Les *Physical Extents* sont les plus petites unités de stockage allouables sur un *Physical Volume (PV)*.
2. **Taille fixe :**
   - Au moment de la création du Volume Group (VG), on définit la taille d'un extent (par défaut souvent 4 Mo).
   - Tous les PV inclus dans ce Volume Group seront découpés en PEs de cette même taille.
3. **Rôle :**
   - Les PEs constituent la base sur laquelle les *Logical Volumes (LV)* vont puiser leur espace.
   - En gérant le stockage par blocs de quelques mégaoctets (plutôt qu'au niveau du secteur ou bloc disque), LVM facilite l'agrandissement, la réduction et le déplacement des données.

Logicial Volume Manager (LVM)

Extensions Physiques (PE) et Extensions Logiques (LE)

2. Logical Extents (LE)

1. **Définition :**
   Les *Logical Extents* sont les blocs logiques qui composent un *Logical Volume (LV)*.
2. **Correspondance 1:1 :**
   - La taille d'un Logical Extent est la même que celle d'un Physical Extent (définie au niveau du Volume Group).
   - Chaque LE pointe vers un PE précis.
   - Ainsi, 1 LE ↔️ 1 PE.
3. **Rôle :**
   - Les LE permettent de représenter le volume logique de façon continue et uniforme, même si derrière, les PEs peuvent être dispersés sur différents disques physiques.
   - Quand on "agrandit" ou "rétrécit" un Logical Volume, on ajoute ou on enlève un certain nombre de LEs (qui correspondent à des PEs disponibles ou libérés sur les PV).

Logicial Volume Manager (LVM)

Extensions Physiques (PE) et Extensions Logiques (LE)

3. Comment ça fonctionne en pratique

1. **Création d'un Physical Volume (PV)**
   - On initialisera par exemple un disque ou une partition :

     ```
     pvcreate /dev/sdb1
     ```

   - Cela permet à LVM de le considérer comme un "réservoir" de PEs (une fois intégré dans un Volume Group).
2. **Création d'un Volume Group (VG)**
   - On agrège un ou plusieurs PV dans un VG :

     ```
     vgcreate VG_DATA /dev/sdb1
     ```

   - Durant cette étape, on spécifie (ou LVM choisit par défaut) la taille de l'extent (ex. `--extent-size 4M`).
   - Le VG peut alors contenir un grand nombre de PEs de 4 Mo chacun.

Logicial Volume Manager (LVM)

Extensions Physiques (PE) et Extensions Logiques (LE)

3. Comment ça fonctionne en pratique

3. **Création d'un Logical Volume (LV)**
   - On réserve une partie de l'espace du VG pour un LV :

```
lvcreate -n LV_home -L 10G VG_DATA
```

   - LVM associe alors un nombre de LEs correspondant à 10 Go (p. ex. si 1 LE = 4 Mo, on aura 2560 LEs)
   - Chaque LE de ce LV pointe vers un PE sur l'un des PV du VG.
4. **Évolutions et flexibilité**
   - Pour agrandir un LV, on ajoute des LEs (et donc on consomme plus de PEs).
   - Pour déplacer un LV, LVM peut rediriger les LEs vers d'autres PEs disponibles.

Logicial Volume Manager (LVM)

Extensions Physiques (PE) et Extensions Logiques (LE)

4. Avantages de cette approche

1. **Gestion fine :**
   En travaillant par blocs de plusieurs Mo (au lieu de partitions fixes), LVM facilite les réallocations de stockage sans nécessiter de repartitionner.
2. **Souplesse :**
   Vous pouvez agrandir un LV à partir de l'espace disponible dans le VG, même si cet espace est réparti sur plusieurs disques physiques.
3. **Performances et maintenance :**
   Le découpage en extents simplifie les opérations de migration de données ou de maintenance (déplacement d'extents d'un disque à un autre, retrait/ajout de disques, etc.).

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

- Dans le contexte de LVM (Logical Volume Manager), on distingue plusieurs zones de métadonnées sur chaque *Physical Volume* (PV)
- La documentation et certains supports de formation les désignent souvent par des noms du type **PVRA**, **VGRA**, voire **BBRA**.
- Ces acronymes peuvent varier selon les versions ou les éditeurs (certaines viennent de l'historique d'HP-UX LVM ou d'autres implémentations), mais l'idée générale reste la même :
  - il s'agit d'espaces réservés sur le disque (ou la partition) pour stocker les informations critiques sur le PV lui-même, le Volume Group,
  - et éventuellement d'autres données (comme un bloc de boot ou la gestion des blocs défectueux)

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

1. PVRA (Physical Volume Reserved Area)

1. **Rôle principal**
   - La *Physical Volume Reserved Area* correspond à la zone de métadonnées réservée au tout début (ou parfois la fin) d'un disque ou d'une partition lorsqu'on en fait un Physical Volume.
   - Elle contient notamment :
     - Le *label* LVM (signature qui identifie ce disque comme un PV).
     - Les informations de base sur le PV : son UUID, la taille de ses extents, etc.
2. **Localisation et taille**
   - Généralement, il s'agit de quelques secteurs ou blocs réservés, typiquement au début du PV (lorsque vous faites un `pvcreate`, LVM y inscrit sa signature et des informations de base).
   - Cette zone est minuscule par rapport au disque (quelques kilo-octets), mais critique car elle permet de reconnaître le disque comme un PV et d'y accéder correctement.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

1. PVRA (Physical Volume Reserved Area)

3. **Historique et nommage**
   - On parle parfois de “PV label” ou “entête LVM” (LVM header) dans la documentation plus récente.
   - Sur des systèmes historiques (HP-UX LVM, etc.), on utilisait le terme “Physical Volume Reserved Area” pour englober toutes les métadonnées du disque en lien avec le statut de PV.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

2. VGRA (Volume Group Reserved Area)

1. **Rôle principal**
   - La *Volume Group Reserved Area* est la zone de métadonnées où sont stockées les informations relatives à la configuration du *Volume Group* (VG).
   - Concrètement, c'est là que LVM enregistre la liste des Logical Volumes, leur taille, leur mapping vers les Physical Extents, etc.
   - Elle peut également contenir des informations de "journalisation" pour garder une cohérence en cas de panne.
2. **Copie redondante**
   - Quand vous avez plusieurs disques dans un même VG, LVM peut conserver plusieurs copies des métadonnées du VG (une sur chaque PV). Cela permet de survivre à la perte d'un disque unique si ces métadonnées sont dupliquées.
   - Le nombre de copies dépend de la configuration (paramètre `metadata copies`).

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

2. VGRA (Volume Group Reserved Area)

3. **Flexibilité d'emplacement**
   - Selon la version de LVM et la configuration, cette zone peut être positionnée au début ou à la fin du PV.
   - Par défaut, LVM place souvent la métadonnée (VG metadata) vers le début, mais on peut spécifier `--metadataarea` ou `--metadataignore` pour changer cette organisation.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

3. BBRA (Boot Block ou Bad Block Reserved Area)

- Cet acronyme est moins standard dans la documentation LVM officielle, mais on peut le rencontrer dans des documentations plus anciennes ou spécifiques à certains OS (HP-UX, AIX, etc.) ou formations.
- Deux interprétations principales reviennent :

1. **Boot Block Reserved Area**
    - Sur certaines implémentations, c'est une zone réservée pour stocker des informations de boot ou un chargeur d'amorçage (en conjonction avec LVM).
    - Dans le cas de boot direct sur un volume LVM (peu fréquent sans /boot séparé), il peut exister un espace réservé pour le chargeur ou pour des informations nécessaires à l'amorçage.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

3. BBRA (Boot Block ou Bad Block Reserved Area)

2. **Bad Block Relocation Area**
    - Sur de vieux systèmes ou anciens disques, on réservait parfois une zone pour “relocaliser” les blocs défectueux (bad blocks).
    - LVM ou l’OS pouvaient marquer ces blocs comme inutilisables et utiliser cette zone pour remapper ces blocs ou stocker la table des blocs défectueux.
    - Aujourd’hui, les disques modernes (SSD ou HDD avec firmware interne) gèrent souvent eux-mêmes la remap de secteurs défaillants, rendant cette fonctionnalité moins cruciale côté LVM.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

4. Comment tout cela se traduit concrètement

- Quand vous exécutez un `pvcreate /dev/sdb` :
  1. LVM écrit le PV label + la PVRA en début (ou fin) de `/dev/sdb`.
  2. Il réserve également un espace pour stocker la VGRA (même si vous n'avez pas encore créé le Volume Group).
- Quand vous exécutez un `vgcreate myVG /dev/sdb` :
  1. LVM initialise la partie "Volume Group metadata" (VGRA) dans l'espace réservé.
  2. Il y place des informations sur le nouveau VG (nom, UUID, etc.).
- Si la notion de **BBRA** est gérée sur votre OS ou distribution (peu fréquent désormais), elle sera aussi mise en place lors du `pvcreate` ou via des outils spécifiques.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

4. Comment tout cela se traduit concrètement

En pratique, la plupart des utilisateurs LVM n'ont pas à manipuler directement ces trois zones :

- On sait qu'un **PV** contient un label + métadonnées au début (et parfois en fin).
- On sait que la **VG metadata** (qui décrit LV, PV, etc.) est stockée dans une zone spéciale.
- Le reste du disque est découpé en *Physical Extents* (PE) pour y placer les données utilisateur.

Logicial Volume Manager (LVM)

Métadonnées (PVRA, VGRA, BBRA)

5. En résumé

- **PVRA (Physical Volume Reserved Area)**
  Désigne le bloc de métadonnées marquant un disque/partition comme Physical Volume et contenant les infos d'identification de base (signature LVM, UUID, taille d'extent, etc.).
- **VGRA (Volume Group Reserved Area)**
  Zone de métadonnées qui décrit la configuration complète du Volume Group : liste des LV, attributs, mapping, journal, etc.
  Peut être répliquée sur plusieurs PV du même VG pour la résilience.
- **BBRA (Boot Block/Bad Block Reserved Area)**
  Un espace qui *pouvait* être dédié à l'amorçage ou à la relocalisation de blocs défectueux, surtout sur des implémentations LVM plus anciennes ou propres à certains UNIX. Sous Linux moderne, cette notion est rarement mise en avant ou utilisée.

Logicial Volume Manager (LVM)

Sécurisation des volumes

- La sécurisation des volumes LVM repose sur plusieurs stratégies complémentaires qui visent à protéger vos données tant au niveau de leur intégrité que de leur confidentialité.

Logicial Volume Manager (LVM)

Sécurisation des volumes

1. Chiffrement des volumes

- **Utiliser LUKS et dm-crypt :**
  LUKS (Linux Unified Key Setup) est une solution de chiffrement de disque largement utilisée. En associant LUKS à dm-crypt, vous pouvez chiffrer vos volumes LVM pour garantir que les données restent protégées même en cas de vol ou d'accès physique non autorisé.
- **Procédure de chiffrement :**
  Avant de créer un volume logique, initialisez le chiffrement sur la partition ou le volume physique. Vous pouvez ensuite créer des volumes logiques à l'intérieur d'un conteneur chiffré. Cela permet d'assurer que toutes les données stockées sont cryptées par défaut.

Logicial Volume Manager (LVM)

Sécurisation des volumes

2. Gestion des accès et permissions

- **Contrôle des accès utilisateurs :**
  Veillez à ce que seuls les utilisateurs autorisés aient les droits d'accès aux volumes LVM. Pour cela, gérez correctement les permissions sur les périphériques de blocs ainsi que les fichiers de configuration (par exemple dans `/etc/lvm/`).
- **Utiliser sudo et limiter les privilèges :**
  N'accordez des droits administratifs qu'aux utilisateurs strictement nécessaires, en utilisant des mécanismes comme sudo afin de limiter l'exposition en cas de compromission.

Logicial Volume Manager (LVM)

Sécurisation des volumes

3. Sauvegarde et snapshots

- **Mise en place de snapshots sécurisés :**
  LVM permet de créer des snapshots pour effectuer des sauvegardes cohérentes. Assurez-vous que ces snapshots soient eux aussi protégés et, si nécessaire, chiffrés pour éviter qu'un accès non autorisé ne compromette vos données de sauvegarde.
- **Plan de sauvegarde :**
  Établissez une stratégie de sauvegarde régulière et testez vos procédures de restauration. La sauvegarde hors site ou sur des supports cryptés renforce la sécurité globale.

Logicial Volume Manager (LVM)

Sécurisation des volumes

4. Mise à jour et surveillance

- **Maintenance régulière :**
  Assurez-vous que le système d'exploitation et les outils LVM soient maintenus à jour pour bénéficier des correctifs de sécurité.
- **Audit et surveillance :**
  Mettez en place des outils de monitoring et de journalisation (logs) pour détecter toute activité anormale. Des audits réguliers des accès aux volumes permettent d'identifier rapidement une éventuelle tentative d'intrusion.

Logicial Volume Manager (LVM)

Sécurisation des volumes

5. Sécurisation physique et réseau

- **Sécurisation du matériel :**
  Le chiffrement protège les données, mais la sécurité physique des serveurs est également cruciale. Assurez-vous que le matériel est dans un environnement sécurisé.
- **Accès réseau sécurisé :**
  Si vos volumes LVM sont accessibles via des services réseau (par exemple, pour une solution de stockage en réseau), veillez à ce que ces services utilisent des protocoles sécurisés et des connexions chiffrées.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

BTRFS

BTRFS

Definition

- Btrfs (B-tree File System) est un système de fichiers moderne conçu pour offrir une grande flexibilité, de nombreuses fonctionnalités avancées et une gestion intégrée des volumes.
- Conçu dès 2007 et intégré au noyau Linux en 2009, Btrfs se positionne comme une alternative aux systèmes de fichiers traditionnels comme ext4 ou XFS.

BTRFS

Caractéristiques Principales de Btrfs

1. Gestion intégrée des volumes

- **Ajout et retrait de périphériques :**
  Btrfs permet d'ajouter ou de retirer des disques dans un pool de stockage à chaud, ce qui est très utile pour évoluer en fonction de vos besoins. Par exemple, avec la commande suivante, vous pouvez ajouter un nouveau périphérique à un système de fichiers déjà monté :

```
sudo btrfs device add /dev/sdc /mnt
sudo btrfs balance start /mnt
```

BTRFS

Caractéristiques Principales de Btrfs

1. Gestion intégrée des volumes

- **RAID logiciel intégré :**
  - Btrfs supporte différentes configurations RAID (comme RAID 0, RAID 1, RAID 10 et des modes expérimentaux pour RAID 5/6).
  - Cela signifie que vous pouvez configurer la redondance et la répartition des données sans outils supplémentaires comme mdadm ou LVM.
  - La commande suivante, par exemple, rééquilibre les données sur tous les périphériques du pool en appliquant une stratégie RAID 1 pour la redondance :

```
sudo btrfs balance start -dconvert=raid1 -mconvert=raid1 /mnt
```

BTRFS

Caractéristiques Principales de Btrfs

2. Subvolumes

- **Nature et fonctionnement :**
  - Un subvolume est une entité logique qui fonctionne comme un sous-système de fichiers à part entière.
  - Il permet de structurer vos données de manière plus granulaire sans nécessiter de partitionnement physique séparé.
  - Chaque subvolume a son propre inode racine, ce qui le rend indépendant en termes de gestion, bien qu'il partage l'espace global.
- **Cas d'usage :**
  - Par exemple, vous pouvez isoler vos données utilisateur dans un subvolume `/home` et vos données système dans un autre subvolume.
  - Cela facilite les sauvegardes et les restaurations, car vous pouvez créer des snapshots spécifiques pour chaque subvolume sans impacter l'ensemble du système.

BTRFS

Caractéristiques Principales de Btrfs

2. Subvolumes

- **Création et montage :**
  La création se fait facilement :

```
sudo btrfs subvolume create /mnt/home
```

  Et pour monter un subvolume spécifique, vous pouvez utiliser l'option `subvol` dans la commande mount :

```
sudo mount -o subvol=home /dev/sdb1 /home
```

BTRFS

Caractéristiques Principales de Btrfs

3. Snapshots

- **Création rapide grâce au CoW :**
  Le mécanisme de Copy-on-Write permet de créer des snapshots en quelques secondes, car il n'est pas nécessaire de copier physiquement toutes les données. Le snapshot se contente de marquer les blocs de données existants, puis ne copie que les blocs modifiés par la suite.
- **Snapshots en lecture seule vs. lecture-écriture :**
  - *Lecture seule* : Ces snapshots garantissent que l'état enregistré reste intact et non modifié, idéal pour les sauvegardes ou les points de restauration.
  - *Lecture-écriture* : Vous pouvez également créer des snapshots qui permettent des modifications. Ils offrent une flexibilité pour tester des mises à jour ou des configurations sans affecter l'original.

BTRFS

Caractéristiques Principales de Btrfs

3. Snapshots

- **Exemple de création d'un snapshot :**

```
sudo btrfs subvolume snapshot -r /mnt/home /mnt/home_snapshot
```

Ici, le flag `-r` crée un snapshot en lecture seule. Pour revenir en arrière, vous pouvez monter ce snapshot ou même le convertir en subvolume actif.

BTRFS

Caractéristiques Principales de Btrfs

4. Copy-on-Write (CoW)

- **Principe détaillé :**
  - Plutôt que d'écrire les modifications directement sur le bloc existant, Btrfs réserve de nouveaux blocs pour y écrire les modifications.
  - Cela signifie que tant que le CoW n'est pas activé pour une écriture (par exemple, lors de l'actualisation d'un fichier modifié), l'ancien contenu reste intact.
  - Ce mécanisme permet :
    - De réduire les risques de corruption en cas de coupure inopinée,
    - De faciliter la création de snapshots puisque le contenu initial n'est pas écrasé.

BTRFS

Caractéristiques Principales de Btrfs

4. Copy-on-Write (CoW)

- **Impact sur les performances :**
  - Le CoW peut, dans certains cas, entraîner une fragmentation accrue, surtout sur des fichiers très fréquemment modifiés.
  - Des options comme `nodatacow` existent pour désactiver ce mécanisme sur des fichiers particuliers si nécessaire (par exemple pour des bases de données).

BTRFS

Caractéristiques Principales de Btrfs

5. Compression

- **Compression à la volée :**
  La compression dans Btrfs s'effectue automatiquement lors de l'écriture des données, sans intervention manuelle sur les fichiers. Vous pouvez choisir parmi plusieurs algorithmes :
  - **zlib** : Bonne compression, mais potentiellement plus lourde en termes de CPU.
  - **lzo** : Moins gourmand en ressources mais avec un taux de compression souvent inférieur.
  - **zstd** : Offre un bon compromis entre vitesse et taux de compression.
- **Utilisation lors du montage :**
  Pour activer la compression, vous montez le système de fichiers avec l'option correspondante. Par exemple :

```
sudo mount -o compress=zstd /dev/sdb1 /mnt
```

  Une fois monté avec cette option, les fichiers nouvellement écrits seront compressés automatiquement.

BTRFS

Caractéristiques Principales de Btrfs

6. En résumé

- **Volumes et gestion dynamique** avec la possibilité d'ajouter ou retirer des périphériques et de gérer le RAID de manière native.
- **Subvolumes** pour une structuration fine des données, facilitant ainsi la gestion et la sauvegarde.
- **Snapshots rapides** grâce au mécanisme CoW, offrant des points de restauration efficaces en cas de besoin.
- **Copy-on-Write** qui garantit une meilleure intégrité des données et facilite la création de copies instantanées.
- **Compression en temps réel** pour optimiser l'utilisation de l'espace disque tout en maintenant de bonnes performances.