🔁 CHAÎNE COMPLÈTE : DU BIOS AU FAT32

1 - [BIOS]

2 - [Bootloader]

3 - [Kernel]

4 - [Drivers disque]

5 - [FAT32]

Au bouton power : microcode processeurs qui s'activent jusqu'au BIOS.

🟦 1. BIOS (ou UEFI en mode BIOS)

📌 Ce que fait le BIOS

- **Initialise le matériel minimal**
- **Cherche un périphérique bootable**
- **Charge le premier secteur (512 octets) du disque à 0x7C00**
- **Passe l'exécution à ce code**

📌 Le BIOS ne connaît PAS FAT32, ni Rust, ni ELF.

👉 Il ne fait qu'un saut vers du code brut.

BIOS = Basic Input Output

UEFI = Unified Extensible Firmware Interface

L'UEFI offre de nombreux avantages sur le BIOS : fonctionnalités réseau en standard, interface graphique de bonne résolution, gestion intégrée d'installations multiples de systèmes d'exploitation et affranchissement de la limite des disques à 2,2 To.

Le BIOS, écrit en assembleur, limitait les modifications et/ou remplacements, gage de sûreté de fonctionnement et de sécurité.

L'UEFI est écrit en C, ce qui rend sa maintenance plus souple et reste acceptable en raison des coûts décroissants de la mémoire. Développé pour assurer l'indépendance entre système d'exploitation et plate-forme matérielle sur laquelle il fonctionne, l'UEFI est disponible sur les plates-formes Itanium (IA-64), x86 (32 et 64 bits) et ARM.

🟦 2. Bootloader (ton cas : bootloader en Rust)

🎯 Rôle du bootloader

Il fait tout ce que le BIOS ne sait pas faire :

- **Passe du real mode → protected → long mode (64 bits)**
- **Active :**
  - **Paging**
  - **GDT minimale**
- **Charge ton kernel ELF en mémoire**
- **Fournit :**
  - **Une pile**
  - **Un mapping mémoire**
- **Appelle « _start »**

📌 **Le bootloader n'est PAS ton kernel**
📌 **Il est jetable après le boot**

🟦 3. Entrée dans ton kernel (_start) – « entry point »

À ce moment-là :

- **CPU en 64 bits (16 bits real address mode, 32 bits protected mode puis 64 bits long mode)**
- **Paging actif**
- **Interruptions désactivées**
- **Aucun service OS**

📌 **Il n'y a rien :**

- **Pas de heap**
- **Pas de filesystem**
- **Pas d'allocateur**
- **Pas de drivers**

🟦 4. Initialisation du kernel (OBLIGATOIRE)

🔧 Étapes critiques (dans cet ordre)

1. **Désactiver / configurer interruptions**
2. **Initialiser GDT / IDT**
3. **Initialiser le PIC / APIC**
4. **Initialiser le timer**
5. **Initialiser la heap**
6. **Initialiser les drivers matériels**

📌 **Tant que le disque n'est pas accessible, FAT32 est impossible.**

🟦 5. Accès au disque (PRÉREQUIS FAT32)

🧱 Le kernel ne voit pas "des fichiers"

Il voit :

- **Des secteurs (512 octets)**
- **Des LBA**

👉 **Il te faut un driver de stockage.**

🟦 6. Lecture du disque brut

Une fois le driver prêt :

LBA 0 → MBR ou GPT

LBA X → Partition FAT32

👉 **Tu travailles au niveau octet, pas fichier.**

🟦 7. Structures FAT32 (LE CŒUR DU FS)

FAT32 est 100 % logiciel, aucune magie.

📦 Structures clés

Structure	Description
Boot Sector	Infos générales
FAT	Table de clusters
Root Directory	Dossiers / fichiers
Data Region	Contenu réel

Paging : mémoire virtuelle vers mémoire physique (protection de la mémoire)

GDT / IDT : Global Descriptor Table -> segment mémoire.

Interrupt Descriptor Table -> Indique au CPU quoi faire en cas d'interruption.

PIC / APIC : Advanced Programmable Interrupt Controller -> puce qui reçoit les interruptions matérielles et les envoie au CPU. Support le multicœurs pas comme PIC (obsolète). Indispensable en SMP.

Timer : Un timer génère des interruptions périodiques (100 Hz ou 1000 pas plus). Multitâche, gestion du temps, scheduler, sleep(). On sait que 1 tick = 1 ms donc 100 Hz = 1 sec, ce qui permet au CPU de gérer le temps.

Heap : Mémoire dynamique. Gestion de grandes structures, durée de vie variable (=!stack)

Drivers matériels : Code permettant de parler à un périphérique.

LBA : Logical Block Addressing -> Le LBA est une façon de numéroter les secteurs d'un disque (qui sont chacun de 512 octets). Les secteurs sont obligatoires car un disque ne peut pas lire par octet donc on créer des blocs.

SMM : System Management Mode -> Anneau (ring) -2 = mode CPU ultra-privilégié, au-dessus du kernel. Gestion de l'énergie, sécurité firmware. Dangereux car invisible pour l'OS (source de backdoors).

SMP : Symetric Multiprocessing -> Plusieurs CPU/cœurs qui se partagent la mémoire et exécutent le même kernel. Conséquences kernel = Interruptions par cœur, stack par cœur.

Stack (pile) : Est créée au **démarrage du thread.** Taille fixe, gère les variables locales, appels de fonctions, adresses de retour etc.

Le kernel choisi quel process tourne à quel moment (1 thread par cœur – le timer choisi le thread).

Syscall permet de rendre la main au kernel. Le timer évite que le syscall tourne en boucle grâce aux ticks.

Exécuter un programme : prendre quelque-chose et le mettre en mémoire.

Le kernel contient l'interpréteur.

Interruptions et Exceptions.

Double Fault = OK

Triple Fault = fatal = reset

Deadlocks = plusieurs codes veulent accéder à la même ressource, ils s'attendent et se managent.