**Rapport du projet MONCOMBLE\_OS**

*MONCOMBLE Jules – 4SI4*

**22/02/2025**

Table des matières

[2. Contexte initial et objectifs généraux : analyse et motivations techniques 1](#_Toc191149694)

[2.1. Un OS Rust no\_std 1](#_Toc191149695)

[2.1.1. Panic handler personnalisé 1](#_Toc191149696)

[2.1.2. Point d’entrée \_start() 1](#_Toc191149697)

[2.1.3. Cible JSON (i686-none.json ou x86\_64-unknown-none) et script de lien (linker.ld) 1](#_Toc191149698)

[2.1.4. Approche bare-metal 2](#_Toc191149699)

[2.2. Un boot sur une partition FAT32 2](#_Toc191149700)

[2.2.1. Conformité avec les méthodes de montage Linux 2](#_Toc191149701)

[2.2.2. Lecture BIOS de la partition 2](#_Toc191149702)

[2.2.3. Complexité par rapport à un superfloppy 2](#_Toc191149703)

[2.2.4. Chargement ultérieur du kernel 3](#_Toc191149704)

[2.3. Préserver la montabilité FAT 3](#_Toc191149705)

[2.3.1. La zone BPB (Bios Parameter Block) 3](#_Toc191149706)

[2.3.2. La signature 0x55AA 3](#_Toc191149707)

[2.3.3. La possibilité de montage sous Linux 3](#_Toc191149708)

[3. Avancement initial : compilation, bump et slab allocator 3](#_Toc191149709)

[3.1. Allocateurs (bump puis slab) 4](#_Toc191149710)

[3.1.1. Le bump allocator 4](#_Toc191149711)

[3.1.2. Le slab allocator 4](#_Toc191149712)

[3.2. Premier code no\_std Rust 5](#_Toc191149713)

[3.2.1. Le point d’entrée \_start() 5](#_Toc191149714)

[3.2.2. Le panic handler 5](#_Toc191149715)

[3.2.3. Le script de lien (linker.ld) et la cible JSON 5](#_Toc191149716)

[3.2.4. Test local vs. test sous QEMU 5](#_Toc191149717)

[4. Conception d’un boot sur “vrai” FAT32 et premiers écueils 6](#_Toc191149718)

[4.1. Création de l’image, parted et mkfs.fat 6](#_Toc191149719)

[4.1.1. Démarche habituelle 6](#_Toc191149720)

[4.1.2. Créer la partition FAT32 7](#_Toc191149721)

[4.1.3. Formater la partition en FAT32 7](#_Toc191149722)

[4.1.4. Problèmes fréquents 7](#_Toc191149723)

[4.1.5. Solutions adoptées 8](#_Toc191149724)

[4.2. Montage, signature FAT et “wrong fs type” 8](#_Toc191149725)

[4.2.1. Problèmes 8](#_Toc191149726)

[4.2.2. Conclusion 9](#_Toc191149727)

[5. Alternative avortée : superfloppy minimal 9](#_Toc191149728)

[5.1. Principe général : formater directement l’image en FAT32 9](#_Toc191149729)

[5.1.1. But recherché 9](#_Toc191149730)

[5.1.2. Commande mkfs.fat -F 32 -I 10](#_Toc191149731)

[5.2. Installation d’un boot sector minimal sur l’image superfloppy 10](#_Toc191149732)

[5.2.1. Extraction du boot sector et injection du code 10](#_Toc191149733)

[5.3. Lancement via QEMU : -fda / if=floppy 10](#_Toc191149734)

[6. Contraintes techniques et échecs rencontrés 11](#_Toc191149735)

[6.1. Expectatives du BIOS / SeaBIOS en mode “HDD” 11](#_Toc191149736)

[6.2. Problème de taille de “disquette” 11](#_Toc191149737)

[6.3. Espace réduit pour Stage1 11](#_Toc191149738)

[6.4. Problème de montage 11](#_Toc191149739)

[6.5. Tests infructueux 11](#_Toc191149740)

[6.6. Raisons de l’abandon 12](#_Toc191149741)

[6.7. La piste du superfloppy minimal a consisté à : 12](#_Toc191149742)

[7. Interférences parted/losetup/sfdisk 13](#_Toc191149743)

[7.1. Contexte et erreurs constatées 13](#_Toc191149744)

[7.2. Les erreurs : 13](#_Toc191149745)

[7.3. Origine technique des interférences 14](#_Toc191149746)

[7.3.1. Associations loop non libérées 14](#_Toc191149747)

[7.3.2. Partition déjà montée 14](#_Toc191149748)

[7.3.3. Mauvais ordre d’exécution 14](#_Toc191149749)

[7.3.4. Détection automatique 14](#_Toc191149750)

[7.4. Méthodes pour éviter / corriger ces conflits 15](#_Toc191149751)

[7.4.1. Vérifier et détacher tout loop device associé 15](#_Toc191149752)

[7.4.2. S’assurer que la partition n’est pas montée 15](#_Toc191149753)

[7.4.3. Automatiser la séquence dans un script 15](#_Toc191149754)

[7.4.4. Redémarrer la machine 16](#_Toc191149755)

[7.4.5. Exemple de commandes courantes 16](#_Toc191149756)

[8. Inclusion d’un code ASM plus complet dans le secteur 0 : détails, écueils et solution par patch 16](#_Toc191149757)

[8.1. Contexte : un boot sector de 512 octets 17](#_Toc191149758)

[8.1.1. Structure d’un boot sector FAT32 17](#_Toc191149759)

[8.2. Limites et erreurs rencontrées 17](#_Toc191149760)

[8.2.1. Dépassement de la zone 94..509 17](#_Toc191149761)

[8.2.2. incbin d’un fichier bpb.bin de 90 octets 17](#_Toc191149762)

[8.2.3. incbin "bootsector\_original.bin" + org 0x5E 18](#_Toc191149763)

[8.3. La solution par patch\_code.bin (injection) 18](#_Toc191149764)

[8.3.1. Principe du patch 18](#_Toc191149765)

[8.3.2. Avantages 19](#_Toc191149766)

[8.3.3. Limites 19](#_Toc191149767)

[9. QEMU “Booting from Hard Disk…” bloqué 19](#_Toc191149768)

[9.1. Éventuelle nécessité d’un MBR standard 20](#_Toc191149769)

[9.1.1. Contexte traditionnel du boot “Hard Disk” 20](#_Toc191149770)

[9.1.2. Raisons de l’absence de MBR 20](#_Toc191149771)

[9.2. Problème potentiel dans l’exécution réelle du code Stage1 20](#_Toc191149772)

[9.2.1. Possibilité d’un plantage immédiat 20](#_Toc191149773)

[9.2.2. Manque de debug visible 21](#_Toc191149774)

[9.3. Hypothèse d’absence d’affichage debug 21](#_Toc191149775)

[9.3.1. Raisons possibles de l’absence de trace 21](#_Toc191149776)

[9.3.2. Recours à un message “tout début” 21](#_Toc191149777)

[9.4. Solutions envisagées pour dépasser le blocage 21](#_Toc191149778)

[9.4.1. Installer un MBR minimal ou standard (Syslinux, GRUB, etc.) 22](#_Toc191149779)

[9.4.2. Forcer QEMU à traiter l’image comme “floppy” 22](#_Toc191149780)

[9.4.3. Corriger l’initialisation de Stage1 22](#_Toc191149781)

[9.5. Conclusion et perspectives 22](#_Toc191149782)

[10. “non-constant argument to TIMES” dans Stage2 23](#_Toc191149783)

[10.1. Contexte : assembler Stage2 pour un chargement à 0x8000 23](#_Toc191149784)

[10.1.1. Objectif initial 23](#_Toc191149785)

[10.1.2. Problème de la directive times 24](#_Toc191149786)

[10.2. Solution : assembler Stage2 en org 0, puis charger en 0x8000 24](#_Toc191149787)

[10.2.1. Méthode : [org 0] + times (4096 - ($)) 24](#_Toc191149788)

[10.2.2. Charger Stage2 à 0x8000 via Stage1 25](#_Toc191149789)

[10.3. Justification technique 25](#_Toc191149790)

[10.3.1. Org multiple, non-constant times 25](#_Toc191149791)

[10.3.2. Indépendance de l’assembleur et du chargeur 25](#_Toc191149792)

[10.4. Commandes et résumés 25](#_Toc191149793)

[11. Un script final build.sh 26](#_Toc191149794)

[11.1. But et structure du build\_full.sh 26](#_Toc191149795)

[11.1.1. Objectif 26](#_Toc191149796)

[11.1.2. Commandes et déroulé technique 27](#_Toc191149797)

[11.2. Extraction du secteur 0 (VBR) et injection de Stage1 27](#_Toc191149798)

[11.2.1. Extraction du boot sector 27](#_Toc191149799)

[11.2.2. Assemblage de Stage1 en patch\_code.bin 28](#_Toc191149800)

[11.2.3. Injection dans le secteur 0, offset 94 28](#_Toc191149801)

[11.2.4. Écriture du boot1.bin dans /dev/loop0p1 28](#_Toc191149802)

[11.2.5. Assemblage de Stage2 et compilation du kernel 28](#_Toc191149803)

[11.2.6. Lancement QEMU 29](#_Toc191149804)

[11.3. Blocage sur “Booting from Hard Disk…” et hypothèses 29](#_Toc191149805)

[11.3.1. Absence d’un code MBR standard 29](#_Toc191149806)

[11.3.2. Strict SeaBIOS 29](#_Toc191149807)

[11.3.3. Problèmes de patch ou offsets 29](#_Toc191149808)

[11.3.4. Approches alternatives 30](#_Toc191149809)

[12. Conclusion et perspectives : synthèse des réalisations et pistes pour la suite 31](#_Toc191149810)

[12.1. Réalisations effectives du projet 31](#_Toc191149811)

[12.1.1. Compilation Rust no\_std 31](#_Toc191149812)

[12.1.2. Code Stage1 + Stage2 31](#_Toc191149813)

[12.1.3. Expérimentations avec parted, sfdisk, mkfs.fat, incbin 31](#_Toc191149814)

[12.2. Points de blocage majeurs 31](#_Toc191149815)

[12.2.1. Chaînage BIOS 31](#_Toc191149816)

[12.2.2. Espace restreint dans le boot sector 32](#_Toc191149817)

[12.2.3. Nasm et “non-constant argument to TIMES” 32](#_Toc191149818)

[12.2.4. Maintien de la structure FAT32 pour Linux 32](#_Toc191149819)

[12.3. Pistes et propositions pour la suite 32](#_Toc191149820)

[12.3.1. Installer un MBR standard 32](#_Toc191149821)

[12.3.2. Minimiser Stage1 32](#_Toc191149822)

[12.3.3. Forcer QEMU en mode “floppy” 32](#_Toc191149823)

[12.3.4. S’appuyer sur un bootloader existant (Syslinux/GRUB) 33](#_Toc191149824)

[12.4. Conclusion finale 33](#_Toc191149825)

# Contexte initial et objectifs généraux : analyse et motivations techniques

L’orientation générale du projet visait à concevoir, puis à mettre en place, un système d’exploitation basique en Rust, sans dépendance à la bibliothèque standard.

Ce choix, inhabituel pour les projets en langage Rust, soulève plusieurs points de mise en œuvre, notamment la nécessité d’un code d’amorçage écrit en assembleur, d’un script de liaison personnalisé, ainsi que d’une configuration Rust ciblant un environnement bare-metal (i686-none.json ou x86\_64-unknown-none).

À cela s’ajoute l’exigence d’amorcer ce système depuis une partition FAT32, et de pouvoir continuer à monter cette partition sur un système Linux, ce qui contraint fortement la façon dont le secteur de boot doit être modifié ou préservé.

## Un OS Rust no\_std

La décision de travailler en mode no\_std implique de se passer de toute la standard library fournie par Rust dans un usage conventionnel. Cette configuration engendre plusieurs obligations :

### Panic handler personnalisé

Sans la bibliothèque standard, les routines de panique habituelles (générées automatiquement) sont inaccessibles. Il a donc fallu définir manuellement une fonction marquée #[panic\_handler], apte à gérer les situations critiques (par exemple, se contenter de boucler indéfiniment ou d’afficher un message en mode VGA si l’état de l’OS le permet).

### Point d’entrée \_start()

Les programmes Rust classiques intègrent la fonction main(), mais dans un contexte no\_std, cette fonction main() n’existe plus sous forme standard. Au lieu de cela, un point d’entrée “\_start()” devient nécessaire, orné de la mention #[no\_mangle] pour s’assurer que l’éditeur de liens le conserve en l’état et que le code démarre exactement à cet endroit. C’est ce point d’entrée qui se substitue au démarrage habituellement géré par la runtime standard.

### Cible JSON (i686-none.json ou x86\_64-unknown-none) et script de lien (linker.ld)

Pour gérer les spécificités d’un environnement sans système d’exploitation (bare-metal), le projet emploie un fichier de configuration JSON décrivant la cible (définissant l’architecture, l’endianness, la taille des pointeurs, etc.). De plus, un script de lien (linker.ld) permet de préciser les adresses de sections (texte, données, BSS) et d’indiquer où doit se situer l’entrée en mémoire.

L’utilisation combinée de cargo rustc -Z build-std=core,compiler\_builtins rend possible l’embarquement des éléments de base (core, compiler\_builtins) nécessaires à l’exécution minimaliste en Rust, alors qu’on ne dispose pas de la standard library.

### Approche bare-metal

La compilation en mode no\_std s’adresse typiquement à un environnement ne disposant pas d’un système d’exploitation sous-jacent. Le BIOS ou l’UEFI charge un secteur (512 octets) ou un ensemble de secteurs, qui contiennent le code d’amorçage (Stage1, Stage2). Ce code ASM ou Rust minimal vient ensuite initialiser les registres, préparer la transition vers du code plus complexe, puis exécuter le point d’entrée Rust.

Dans ce cas, l’OS no\_std en Rust oblige à redéfinir ou remplacer nombre de mécanismes habituels (panique, initialisation globale, points d’entrée, etc.) et à configurer la chaîne d’outils (cargo, rustc, nasm, scripts de lien), afin de générer un exécutable qui puisse tourner sans OS sous-jacent.

## Un boot sur une partition FAT32

Le second objectif clé consistait à rendre l’OS amorçable depuis une partition FAT32 réelle plutôt que depuis un superfloppy ou un ramdisk. Cette exigence soulève plusieurs dimensions :

### Conformité avec les méthodes de montage Linux

Avec un vrai système FAT32, il devient possible de monter la partition sous Linux par un simple mount -t vfat. Cette simplicité permet d’ajouter ou de mettre à jour des binaires (par exemple KERNEL.BIN, STAGE2.BIN, etc.) directement, sans passer par des outils particuliers de manipulation d’images.

### Lecture BIOS de la partition

Le BIOS standard s’attend généralement à un code MBR dans le secteur 0 du disque (512 octets), lequel code doit ensuite “chaîner” la partition active, c’est-à-dire aller lire le secteur 0 de la partition marquée bootable (son VBR). Dans le cas d’une partition FAT32, ce “VBR” (Volume Boot Record) contient la BPB (BIOS Parameter Block) et éventuellement un code d’amorçage minimal. Le projet devait exploiter ce mécanisme pour lancer le code Stage1.

### Complexité par rapport à un superfloppy

Contrairement à un superfloppy (où tout le disque est formaté comme s’il s’agissait d’une disquette, sans table de partitions), une approche “vraie partition FAT32” amène la question de l’espace alloué dans le VBR pour le code d’amorçage, la présence ou non d’un MBR standard, l’alignement, et la lecture du code par le BIOS. Il fallait donc négocier avec parted ou sfdisk pour créer la table de partitions et mkfs.fat -F 32 pour le formatage.

### Chargement ultérieur du kernel

Une fois le BIOS ayant transféré l’exécution au code du boot sector, ce code doit connaître suffisamment la structure FAT (ou recourir à un offset en dur) pour charger un noyau ou un second boot stage. Comme la partition est un FAT32 standard, le code d’amorçage ne peut pas écraser ou ignorer la BPB, sous peine de rendre la partition illisible.

## Préserver la montabilité FAT

Le troisième aspect fondamental était de ne pas rompre la structure FAT sur la partition. Il s’agissait de ménager :

### La zone BPB (Bios Parameter Block)

Typiquement, FAT32 place au moins 90 à 100 octets de données décrivant la géométrie (BytesPerSector, SecPerCluster, etc.). Or, si le projet “incbin bpb.bin” s’arrêtait à 90 octets, il risquait de ne pas recopier la totalité de la zone extended.

### La signature 0x55AA

Les deux derniers octets (offset 510..511) contiennent la signature “55 AA,” indispensable à la reconnaissance du boot sector. Un code d’assemblage trop volumineux ou mal paramétré pourrait empiéter sur ces octets, entraînant un échec de montage ou de boot.

### La possibilité de montage sous Linux

Si la zone 0..~90 du secteur 0 (VBR) est altérée, la partition n’est plus reconnue comme FAT. Les routines du noyau Linux refusent alors de la monter (“mount: wrong fs type”). De plus, mkfs.fat peut stocker des informations plus loin que l’offset 90, d’où la nécessité de ne toucher qu’un espace restreint (par exemple offset 94..509) pour insérer le code Stage1.

En somme, il fallait ruser pour incorporer un code d’amorçage occupant moins de 416 octets environ, afin de préserver la BPB et la signature. Toute écriture hors de ce sous-espace aurait corrompu la partition, contrecarrant l’objectif de conserver une FAT32 montable.

# Avancement initial : compilation, bump et slab allocator

La phase initiale du projet a porté sur deux volets intimement liés :

* Mettre en place un allocateur de mémoire fonctionnant en mode bare-metal, sans l’aide du noyau ou de la standard library.
* Valider la compilation d’un premier exécutable Rust en mode no\_std, de manière à pouvoir le charger ultérieurement sur une architecture de type x86 sans système d’exploitation.

## Allocateurs (bump puis slab)

Dans un contexte d’OS bare-metal, il n’existe pas d’allocateur mémoire fourni par un noyau hôte. Il a donc fallu coder un allocateur ex nihilo en langage Rust. Deux approches ont été testées :

### Le bump allocator

* Le principe : on maintient un “pointeur” ou un index qui progresse dans le tas à chaque fois qu’on alloue un bloc.
* Aucun mécanisme de libération fine n’est prévu dans la version la plus élémentaire : on incrémente la fin du tas jusqu’à saturation, puis on “remet à zéro” à un moment choisi (p. ex. au démarrage du système).
* Cette méthode est très simple à implémenter, mais elle n’est pas adaptée dès lors qu’on veut gérer de multiples allocations et libérations tout au long de la vie de l’OS.

### Le slab allocator

* Conçu pour segmenter la mémoire en plusieurs caches de tailles fixes (slabs). Chaque slab contient des blocs d’une taille spécifique, et on utilise un ensemble de slabs pour couvrir différentes tailles d’objets.
* Lorsqu’une allocation est demandée, on détermine le slab adapté (en fonction de la taille demandée). S’il reste un bloc libre dans ce slab, on le renvoie immédiatement ; sinon, on cherche un slab disponible ou on en crée un nouveau.
* À la libération, le bloc est replacé dans la liste des blocs libres du slab concerné, ce qui évite la fragmentation excessive.
* C’est un schéma plus complexe que le bump, car il exige des structures de métadonnées (bitmap, liste des slabs, etc.), mais il permet de libérer et de réutiliser la mémoire plus efficacement que le bump.

**Mises en pratique :**

Le code Rust de l’allocateur a été déclaré dans un module memory::slab\_allocator (par exemple). On y voyait souvent un static mut SLAB\_ALLOCATOR: SlabAllocator, ce qui a entraîné des avertissements :

* “creating a mutable reference to mutable static is discouraged”

en raison de la dangerosité potentielle (accès concurrent non géré). Cela restait toutefois un avertissement et non une erreur, la compilation aboutissait.

Le bump allocator, plus rudimentaire, était jugé suffisant pour un OS minimal, mais l’ajout d’un slab allocator plus élaboré a été préféré, car il constitue un meilleur socle pour un OS croissant en complexité.

**Outils de compilation :**

Pour compiler un allocateur (ou tout autre code Rust) en mode no\_std, il a fallu un fichier de configuration JSON (i686-none.json ou x86\_64-unknown-none) décrivant l’architecture, l’endianness, etc.

On recourait à la commande :

* cargo rustc -Z build-std=core,compiler\_builtins --target=i686-none.json --release

pour s’assurer que core et compiler\_builtins soient reconstruits pour la cible bare-metal.

Au final, cette première étape a permis d’avoir un module d’allocation en Rust qui se compilait, malgré quelques avertissements.

## Premier code no\_std Rust

Une fois l’allocation prête, on a mis en place un exécutable Rust minimal fonctionnant sans OS. Plusieurs points techniques se sont révélés déterminants :

### Le point d’entrée \_start()

Dans un binaire Rust standard, la fonction main() est invoquée par un code d’initialisation implicite. En no\_std, on n’a pas ce code d’amorçage : il faut définir nous-même la fonction \_start(), marquée #[no\_mangle] pour éviter la renommage par le compilateur.

\_start() devient l’équivalent du main, responsable de toute initialisation (par exemple, l’initialisation de l’allocateur global, ou de la table GDT si on le souhaite).

### Le panic handler

En no\_std, la routine de panique classique n’existe pas. On doit définir un #[panic\_handler] répondant à la signature attendue (fn panic\_handler(info: &PanicInfo) -> !). On peut soit boucler indéfiniment (loop {}), soit afficher un message sur un terminal ou un buffer VGA, dans la mesure du possible.

Sans ce panic handler, la compilation échoue avec une erreur indiquant qu’aucun “lang item” n’est présent pour la panique.

### Le script de lien (linker.ld) et la cible JSON

Pour un OS bare-metal, il faut spécifier où doivent se trouver les sections .text, .data, .bss, etc. Dans de nombreux projets, on place .text à une adresse comme 0x0010\_0000. On contrôle précisément l’adresse de chargement pour qu’un bootloader sache où charger le binaire.

Le fichier i686-none.json (ou x86\_64-unknown-none.json) signale à rustc que cette cible ne dispose d’aucun OS ni libc, que l’endianness est “little” et la width est “32”. On y précise relocation-model=static, disable-redzone=true, etc.

### Test local vs. test sous QEMU

Une fois le binaire no\_std compilé, on peut le charger dans QEMU à l’aide de commandes spéciales (par ex. -kernel si l’on simule un multiboot, ou un script custom si on le place dans une image).

Sans un code d’amorçage complet, on pouvait se contenter de vérifier la validité du binaire via file MONCOMBLE\_OS (devrait annoncer “data” ou “x86 bootable” suivant la configuration), ou exécuter un test minimal en assembleur simulant un chargement.

**Résultat :**

Le code no\_std Rust se compilait convenablement. L’allocateur Slab ou Bump était déclaré sous un static mut, initialisé dans \_start(), et on pouvait effectuer de petits tests d’allocation. Tout cela permettait de vérifier que l’infrastructure pour un OS Rust bare-metal était en place.

Conclusion de cette partie :

* La mise en route de l’allocation (bump ou slab) a été le premier jalon du projet, validé malgré des avertissements sur static mut.
* On a ensuite assuré que l’exécutable Rust no\_std se construise avec un point d’entrée \_start(), un panic handler, et un script de lien sur la cible i686-none.json.
* Ainsi, on disposait d’un micro-noyau Rust minimal, s’exécutant sans la standard library, prêt à être embarqué dans un code d’amorçage.

Cette base technique stable a servi de tremplin pour aborder la seconde grande difficulté : l’amorçage réel depuis une partition FAT32, intégrant la contrainte de ne pas endommager la structure du système de fichiers.

# Conception d’un boot sur “vrai” FAT32 et premiers écueils

Une fois la partie allocateur validée (bump ou slab) et le code Rust no\_std jugé opérationnel, l’étape décisive du projet a consisté à mettre en place un boot sur partition FAT32. L’idée était de créer et d’utiliser un fichier disque (disk.img) configuré avec une table MBR “msdos” et une unique partition FAT32, marquée active, censée être amorçable par un BIOS standard. Cette stratégie diffère du mode superfloppy (sans table de partitions) et demande davantage de précision pour que le BIOS reconnaisse la partition et que le montage Linux reste possible.

## Création de l’image, parted et mkfs.fat

### Démarche habituelle

La démarche standard pour configurer un disque.img de 64 Mo, en “vrai” FAT32 :

#### Créer un fichier vide de la taille souhaitée :

* dd if=/dev/zero of=disk.img bs=1M count=64

Cela génère un fichier de 64 Mo (environ 67 MB).

#### Définir une table MBR (msdos) à l’aide d’un outil comme parted

* parted -s disk.img mklabel msdos

Cette commande inscrit un label “msdos” (MBR) sur le secteur 0 du fichier, ce qui servira de base pour la création de partitions.

### Créer la partition FAT32

* parted -s disk.img mkpart primary fat32 1MiB 63MiB
* parted -s disk.img set 1 boot on

On réserve les secteurs initiaux (jusqu’à 1 MiB) pour laisser une marge et éviter certains problèmes d’alignement.

On arrête la partition à 63 MiB, ce qui lui alloue environ 62 Mo utiles, suffisamment grand pour mkfs.fat -F 32.

Le drapeau “boot on” marque la partition comme active, ce qui peut aider certains BIOS ou MBR minimaux à chaîner cette partition.

### Formater la partition en FAT32

Afin de pouvoir monter la partition, on doit alors associer l’image à un loop device et exécuter mkfs.fat :

* sudo losetup -fP disk.img

=> Suppose qu’on obtient /dev/loop0 et /dev/loop0p1

* sudo mkfs.fat -F 32 /dev/loop0p1

L’option -F 32 précise qu’on souhaite une FAT32, et /dev/loop0p1 est la partition unique du disque.img. On peut vérifier ensuite via fdisk -l disk.img ou parted disk.img print que la table de partitions inclut bien une partition de ~62 Mo en FAT32.

### Problèmes fréquents

#### Arrondi de taille

parted peut parfois arrondir la fin de la partition, ou si l’utilisateur saisit 100 Mo, parted le convertit en nombre de secteurs alignés. mkfs.fat peut alors répondre “Too few blocks for viable filesystem” si, par exemple, parted a réduit la partition à moins de 32 Mo effectifs (certains mkfs.fat exigent un minimum assez élevé pour FAT32).

#### Value out of range

En tapant “mkpart primary fat32 1MiB 63MiB,” parted renvoie parfois “Value out of range” si la taille d’image ne correspond pas exactement (p. ex. si l’image faisait 60.5 Mo utiles). L’utilisateur doit adapter le end sector ou la taille en question pour rester dans les limites.

#### Alignement

parted tente d’aligner la partition sur des frontiers de 2048 secteurs ou 1 MiB, ce qui peut déplacer légèrement le point de départ. On doit s’assurer que la partition fait bien au moins ~60 Mo après l’arrondi, pour que mkfs.fat -F 32 fonctionne.

### Solutions adoptées

* Forcer l’utilisation de 1 MiB comme start et ~63 MiB comme fin (par ex. “mkpart primary fat32 1MiB 63MiB”), garantissant environ 62 Mo de partition.
* Vérifier après parted qu’on a un partitionnement satisfaisant (via parted disk.img print).
* Installer un label msdos, marquer la partition en “boot on,” puis monter/umount /dev/loop0p1 pour s’assurer que tout se passe bien.

## Montage, signature FAT et “wrong fs type”

Après avoir partitionné et formaté, la partition est censée être fonctionnelle : on peut la monter sous Linux :

* sudo mount /dev/loop0p1 /mnt
* ls /mnt

La difficulté survient lorsqu’on désire installer un code d’amorçage (boot sector) dans le secteur 0 de la partition. Ce secteur 0, appelé VBR (Volume Boot Record), renferme :

* Le BPB (Bios Parameter Block) : un ensemble de champs décrivant la géométrie FAT (BytesPerSec, SecPerClus, etc.). Sur FAT32, cette structure peut dépasser 90 octets.
* Des champs étendus pour FAT32 (offset plus grand que 90).
* Une éventuelle zone de code minimal (quelques dizaines d’octets).
* La signature 0x55AA en offset 510..511.

### Problèmes

Si l’on écrit en assembleur un code occupant tout offset 0..509, on peut potentiellement détruire la signature FAT32 ou la signature 55AA.

mkfs.fat, lors du formatage, place parfois un jump, un OEM label, la table BPB, puis le label “FAT32,” etc. Plus loin que l’offset 90, on trouve parfois des champs pour la table FS info. Si on se contente d’incbin "bpb.bin" de 90 octets, on néglige la partie étendue. Le montage ultérieur (mount -t vfat /dev/loop0p1 /mnt) échoue alors avec :

* mount: wrong fs type, bad option, bad superblock on /dev/loop0p1, ...

parce que la structure FAT n’est plus reconnue.

### Conclusion

Il faut préserver strictement les octets critiques du VBR.

On ne peut occuper qu’une zone limitée, en général 94..509, pour insérer le code Stage1.

On doit laisser en place l’offset 0..~90 (où la BPB est enregistrée), ainsi que l’offset 510..511 (la signature 0x55AA).

Tout dépassement ou corruption de ces champs rend la partition illisible.

C’est à ce stade qu’une technique de “fusion” ou “patch” du bootsector\_original.bin a été envisagée. Elle consiste à :

* Extraire le secteur 0 de la partition après mkfs.fat (via dd if=/dev/loop0p1 of=bootsector\_original.bin bs=512 count=1).
* Assembler un patch code occupant ~200 octets max (org 0), qu’on insère en offset 94.. (via dd if=patch\_code.bin of=bootsector\_original.bin bs=1 seek=94 conv=notrunc).
* Écrire ce “boot1.bin” dans /dev/loop0p1 (secteur 0).

Malgré cette solution élégante, d’autres blocages sont apparus : QEMU n’exécutait pas toujours ce code patché, car il peut exiger un code MBR sur le secteur 0 du disque (et non pas seulement sur la partition). Néanmoins, c’est la base d’une configuration “vrai FAT32 partition” dite “boot sector hybride.”

# Alternative avortée : superfloppy minimal

Le projet a, à un moment, et à cause des problèmes de boot, exploré l’option de créer un “superfloppy minimal”, c’est-à-dire une image FAT32 sans MBR ni table de partitions, en formatant directement le fichier disk.img dans son intégralité.

L’idée était d’éliminer la complexité liée à la gestion du code MBR, à l’insertion d’une table de partitions et à la détection BIOS d’une partition active. Cette section détaille toutes les étapes et justifications techniques de cette approche, ainsi que les causes de son échec dans le contexte du projet.

## Principe général : formater directement l’image en FAT32

### But recherché

Il s’agissait de créer un fichier disk.img où :

* Le secteur 0 contient immédiatement la BPB (Bios Parameter Block) et la signature FAT32, plus un éventuel code d’amorçage minimal, au lieu de s’appuyer sur un MBR + partition table.
* L’image est reconnue comme un disque de type “disquette” (superfloppy) par le BIOS s’il est lancé dans un mode approprié, ce qui évite la notion de partition.

Cette structure « superfloppy » reproduit le fonctionnement d’un média amovible (floppy disk) d’autrefois : aucun MBR ni table de partitions, juste un volume FAT32 occupant tout l’espace, avec le premier secteur occupant le rôle de boot sector.

### Commande mkfs.fat -F 32 -I

Pour concrétiser cette idée, la commande suivante est utilisée :

* mkfs.fat -F 32 -I disk.img

L’option -F 32 force la création d’un système de fichiers FAT32.

L’option -I indique qu’on “ignore” l’absence de table de partitions, c’est-à-dire qu’on formate “l’ensemble du disque” comme un simple volume FAT.

Les octets 0..511 correspondent donc au boot sector + BPB + signature 0x55AA, et les clusters du FS commencent immédiatement après.

#### Avantage théorique

On n’a plus besoin d’exécuter parted ou sfdisk, ni de déclarer de table de partitions. Le BIOS, en configuration “disquette,” devrait pouvoir charger le secteur 0 et exécuter le code d’amorçage.

## Installation d’un boot sector minimal sur l’image superfloppy

### Extraction du boot sector et injection du code

Une fois l’image formatée, on récupère le secteur 0 dans un fichier local :

* dd if=disk.img of=bootsector\_original.bin bs=512 count=1

bootsector\_original.bin contient alors la BPB (les 90–100 octets initiaux, potentiellement plus sur FAT32), la signature "FAT32", un petit bout de code minimal inséré par mkfs.fat, et enfin la signature 0x55AA en offset 510..511.

Pour y insérer un code “Stage1” plus important, il faut :

* Assembler un patch ASM occupant ~200–300 octets, en prenant garde de laisser intacts les octets 0..~90 (BPB) et 510..511 (signature).
* Écrire ce patch dans offset ~94..509. Concrètement, on effectue :
  + dd if=patch\_code.bin of=bootsector\_original.bin bs=1 seek=94 conv=notrunc
  + Cela écrase uniquement la zone 94..(94 + taille du patch - 1), préservant la zone BPB et la signature 0x55AA.
* Mettre à jour l’image
  + dd if=bootsector\_original.bin of=disk.img bs=512 count=1 conv=notrunc

On obtient alors un “superfloppy.img” dont le secteur 0 combine la BPB FAT32, la signature 55AA, et un code ASM occupant ~416 octets au plus.

## Lancement via QEMU : -fda / if=floppy

Pour demander à QEMU de considérer disk.img comme une disquette (plutôt qu’un disque dur), on utilise :

* qemu-system-i386 -fda disk.img

ou

* qemu-system-i386 -drive if=floppy,format=raw,file=disk.img

Dans ce mode, le BIOS émulé se comporte comme s’il avait affaire à une disquette. Il lit donc le secteur 0, exécute le code s’il reconnaît la signature 0x55AA, et on peut espérer voir un message de debug ou un chargement de stage2.

# Contraintes techniques et échecs rencontrés

Malgré la simplicité apparente (pas de MBR, pas de parted...), la solution du superfloppy minimal a fait face à plusieurs difficultés :

## Expectatives du BIOS / SeaBIOS en mode “HDD”

De nombreux BIOS, y compris SeaBIOS de QEMU, traitent par défaut les images “drive” comme des disques durs, pas des disquettes.

Si on omet -fda ou if=floppy, QEMU fait un “Booting from Hard Disk…” et s’attend à un code MBR dans le secteur 0, ce qui n’existe pas dans un superfloppy. Résultat : le BIOS n’exécute pas le boot sector superfloppy et reste bloqué.

## Problème de taille de “disquette”

Les disquettes standard historiquement allaient de 360 Ko à 2.88 Mo. Ici, on parle de 32 Mo, 64 Mo, voire davantage. Certains BIOS émulés ne reconnaissent pas une disquette si sa taille dépasse 2.88 Mo, et peuvent refuser ou mal gérer la géométrie, menant à “Booting from Floppy…” qui n’aboutit pas.

## Espace réduit pour Stage1

Même en superfloppy, le boot sector ne fait que 512 octets, dont ~90 pour la BPB et 2 pour la signature = 510..511 = 0x55AA. Il reste ~416 octets (94..509) pour Stage1, ce qui se révèle parfois trop peu pour un code ASM un tant soit peu évolué (lecture depuis un offset fixe, messages de debug, etc.). D’où l’erreur “TIMES value -xxx is negative” lors de l’assemblage si Stage1 excède cet espace. On se retrouve alors à minifier le code ou à charger un stage2 en dur.

## Problème de montage

Certains environnements (par ex. un Linux standard) montent difficilement un volume FAT32 “partitionless.” On peut parfois employer mount -o loop disk.img /mnt, mais il se peut que la détection auto ne se fasse pas. S’il détecte un label MBR absent, le kernel peut exiger des solutions alternatives.

Mtools, via un fichier ~/.mtoolsrc contenant “mtools\_skip\_check=1,” peut manipuler l’image, mais la procédure sort du canevas “simplement monter la partition”.

## Tests infructueux

Même en forçant -fda disk.img, de nombreux tests se sont soldés par un blocage. Soit le BIOS (SeaBIOS) refusait de considérer la taille trop grande comme disquette valable, soit l’exécution plantait aussitôt sans afficher de message. On a inséré un “int 0x10” minimal tout au début du code ASM et on n’a rien vu s’afficher, confirmant une non-exécution du boot sector.

## Raisons de l’abandon

**Compte tenu de ces points :**

* Incompatibilité BIOS : QEMU, en configuration par défaut, traite un disque de plusieurs dizaines de Mo comme un “Hard Disk,” attend un MBR. Pour le booter en superfloppy, il faut -fda, mais la taille < 2.88 Mo n’est plus respectée.
* Limites du code ASM : 416 octets restants dans le boot sector pour Stage1 imposent un code extrêmement minimal. Tout parse FAT minimal ou routine de debug un tant soit peu verbeuse devient compliqué.
* Complexité du montage : si on voulait monter disk.img sous Linux, on devait recourir à un montage loop -o offset=..., ou à mtools, ce qui se montre moins trivial qu’une partition standard.
* Non-compatibilité avec l’objectif : L’utilisateur souhaitait explicitement un “vrai” disque partitionné, FAT32, montable comme n’importe quel volume. Le superfloppy va à l’encontre de cette finalité.

En conséquence, malgré la simplicité apparente (pas de parted, pas de MBR), la voie superfloppy n’a pas abouti à un système réellement bootable ni convenable pour la taille visée (>32 Mo). Les tests QEMU se sont majoritairement bloqués sur “Booting from Floppy…”, et l’étroitesse du boot sector s’avérait rapidement un obstacle.

## La piste du superfloppy minimal a consisté à :

* Formater l’intégralité d’un fichier disk.img en FAT32 grâce à mkfs.fat -F 32 -I.
* Extraire le boot sector, y patcher le code Stage1 dans offset 94..509, et préserver la signature 0x55AA.
* Lancer QEMU en mode floppy via -fda disk.img ou -drive if=floppy.

Cette approche ne s’est pas révélée fonctionnelle au regard de l’objectif de disposer d’un “vrai” FAT32 partitionné montable et d’un boot correct sous un BIOS standard.   
Les difficultés incluaient la reconnaissance BIOS d’une disquette de taille trop importante, l’espace très restreint pour Stage1, et l’aspect superfloppy lui-même, qui ne satisfaisait pas aux spécifications d’un “vrai” disque dur FAT32.

Par conséquent, la méthode a été délaissée au profit de la création d’une partition via parted ou sfdisk, l’installation d’un MBR minimal et la préservation du boot sector VBR, solution plus conforme aux attentes du projet.

# Interférences parted/losetup/sfdisk

Au cours de la création et de la manipulation d’une image disque (disk.img) pour y installer un système d’exploitation Rust, plusieurs problèmes se sont manifestés du côté de l’association loop device (losetup) et des outils de partitionnement (parted, sfdisk).

Cette partie détaille :

* Les types d’erreurs rencontrées (e.g. “failed to set up loop device: Device or resource busy”, “bad superblock”).
* Les origines techniques de ces interférences (p. ex. résidus de montages, associations loop non libérées).
* Les commandes précises et la séquence recommandée pour éviter ces blocages.
* Les justifications pour employer un script d’automatisation.

## Contexte et erreurs constatées

Dans un workflow standard pour construire un OS bare-metal, on suit généralement les étapes :

* Création d’un fichier vide, par exemple 64 Mo, via :
  + dd if=/dev/zero of=disk.img bs=1M count=64
* Configuration de ce fichier en tant que “disque virtuel” (loop device) :
  + sudo losetup -fP disk.img

Ce qui crée un périphérique /dev/loop0, et éventuellement /dev/loop0p1, si une table de partitions est détectée.

* Partitionnement (parted ou sfdisk).
* Formatage de la partition (/dev/loop0p1) en FAT32.
* Montage, copie de fichiers, etc.

## Les erreurs :

“losetup: disk.img: failed to set up loop device: Device or resource busy”

Se produit quand disk.img est déjà associé à un loop device (ou partiellement monté).

“mount: wrong fs type, bad superblock on /dev/loop0p1”

Souvent dû au fait que /dev/loop0p1 a été mal formaté, ou que parted a recréé la table de partitions alors qu’un montage /dev/loop0p1 subsistait.

D’autres messages comme “Value out of range” dans parted peuvent apparaître si parted / sfdisk n’arrive pas à écrire la table de partitions (verrou, offset incorrect, etc.).

## Origine technique des interférences

Sous Linux, l’association loop device (via losetup) permet de voir un simple fichier (disk.img) comme s’il s’agissait d’un bloc de type /dev/loopX. Or, si un loop device est déjà en usage, de multiples conflits peuvent surgir :

### Associations loop non libérées

* Lorsque l’on exécute sudo losetup -fP disk.img, le système crée /dev/loop0 (et /dev/loop0p1 si parted a déjà été utilisé).
* Si l’on relance parted ou mkfs.fat sur /dev/loop0p1 sans détacher le loop device précédent, on peut se retrouver avec des modifications incongrues.
* Pire, si on tente d’associer à nouveau disk.img sans d’abord libérer /dev/loop0, on obtient “failed to set up loop device: Device or resource busy.”

### Partition déjà montée

* Une fois la partition /dev/loop0p1 formatée en FAT32 et montée (ex. sudo mount /dev/loop0p1 /mnt), si on essaye de reformater ou de re-partitionner l’image, le kernel juge la partition “en usage”.
* Toute reconfiguration avec parted (changement de label, agrandissement ou suppression de partition) pendant que la partition est active provoque un conflit.
* Cela se voit par des erreurs “bad superblock” ou “device is busy.”

### Mauvais ordre d’exécution

* Certaines commandes demandent l’exclusion totale du device. Si parted réécrit la table de partitions alors que mkfs.fat exécute un format, le superblock FAT peut être partiellement créé, corrompu, puis terminé ou annulé.
* On obtient des messages “bad option, wrong fs type,” car le système n’arrive pas à reconnaître une partition dont les méta-informations ont été partiellement détruites.

### Détection automatique

* Parfois, l’automount peut se déclencher (selon la configuration du système). L’utilisateur n’est pas toujours conscient qu’un montage s’est produit, ou qu’un démon (comme udisks2) tente de lire le partitionnement, générant un usage concurrent.
* Dans ces cas, la commande sudo umount /dev/loop0p1 ou sudo losetup -d /dev/loop0 s’avère indispensable pour réinitialiser la situation.

## Méthodes pour éviter / corriger ces conflits

### Vérifier et détacher tout loop device associé

Pour voir si disk.img est déjà pris en charge, on fait :

* sudo losetup -j disk.img

Si la sortie montre /dev/loop0: [some id]: (disk.img), cela signifie que l’image est déjà associée.

On peut alors libérer ce loop device :

* sudo losetup -d /dev/loop0

Pour détacher tous les loop devices d’un coup :

* sudo losetup -D

### S’assurer que la partition n’est pas montée

Même après avoir détaché le loop device, si la partition /dev/loop0p1 est restée montée, il faut faire :

* sudo umount /dev/loop0p1

ou n’importe quel point de montage l’ayant prise en charge (/mnt, /media/username/disk). Sinon, parted / mkfs.fat bloqueront avec “device is busy.”

### Automatiser la séquence dans un script

Pour éviter un ordre incorrect :

* Détacher tout (losetup -j, puis losetup -d, plus un umount /dev/loop0p1).
* Créer (ou recréer) disk.img si nécessaire (dd).
* Associer l’image (losetup -fP disk.img).
* Partitionner (parted ou sfdisk).
* Formater (mkfs.fat -F 32 /dev/loop0p1).
* Monter (mount /dev/loop0p1 /mnt).

Cette séquence doit être suivie rigoureusement. Les scriptes proposés par certains environnements effectuent souvent ce type de vérifications en amont, ce qui évite la confusion autour des messages “Device or resource busy.”

### Redémarrer la machine

Lorsque tout échoue, un redémarrage assure qu’aucun loop device n’est resté dans un état incertain et qu’aucun montage fantôme ne subsiste. C’est une solution de dernier recours, surtout en environnement de développement, lorsque de multiples scripts ont engendré un chaos dans la table de partitions ou la configuration loop.

### Exemple de commandes courantes

* losetup -j disk.img : détecter l’association éventuelle.
* sudo losetup -d /dev/loop0 : libérer un loop device spécifique.
* sudo losetup -D : libérer tous les loop devices.
* sudo umount /dev/loop0p1 : démonter la partition s’il y en a une montée.
* parted -s disk.img mklabel msdos : initialiser un label MBR sur le fichier.
* parted -s disk.img mkpart primary fat32 1MiB 63MiB : créer une partition unique FAT32.
* sudo mkfs.fat -F 32 /dev/loop0p1 : formater la partition.
* sudo mount /dev/loop0p1 /mnt : monter la partition localement.
* sudo umount /mnt : démonter pour la libérer.

Les erreurs “failed to set up loop device: Device or resource busy” et “wrong fs type, bad superblock” sont généralement le résultat d’un usage concurrent ou désordonné de parted, mkfs.fat, losetup et mount. Les solutions les plus efficaces consistent :

* À toujours détacher au préalable tout loop device (et démonter la partition) avant de repartir sur une re-création ou re-partition.
* À adopter un script qui exécute la séquence “losetup -d”, “parted …”, “mkfs.fat …” dans un ordre déterminé et “propre”.
* À recourir au redémarrage complet en dernier recours si le système semble conserver des associations ou montages fantômes impossibles à libérer autrement.

Ainsi, on évite de corrompre la structure FAT ou la table de partitions, et on garantit que parted / sfdisk / mkfs.fat puissent opérer sans entraves.

# Inclusion d’un code ASM plus complet dans le secteur 0 : détails, écueils et solution par patch

Dans le cadre de ce projet, il était impératif d’insérer un bootloader un peu plus riche (Stage1) au sein même du secteur 0 d’une partition FAT32. Cette section décrit :

* Les contraintes imposées par le format FAT32 concernant le boot sector.
* Les limites d’espace disponibles et les erreurs qui surviennent lorsqu’on les dépasse.
* Les tentatives d’utilisation de directives incbin, org multiples et directive TIMES.
* La solution finale par “patch” (injection de code préassemblé dans une zone précise du boot sector).

## Contexte : un boot sector de 512 octets

### Structure d’un boot sector FAT32

Le secteur 0 de la partition FAT32 contient plusieurs zones critiques :

#### Les premiers octets (0..~90) :

* La BPB (Bios Parameter Block) : variables telles que BytsPerSec, SecPerClus, RsvdSecCnt, NumFATs, FATSz32, etc.
* Des champs étendus (offset plus grand que 0x5A, etc.), spécifiques à FAT32, décrivant la géométrie plus en détail.

#### Les deux derniers octets (offset 510..511)

* La signature 0x55AA.
* Indispensable pour que le BIOS détecte un boot sector “valide.” Si elle manque ou est corrompue, le système refusera de booter et Linux rejettera également le volume.

Entre ces deux zones (0..~90 et 510..511), on ne dispose donc que d’environ 416 octets (94..509) pour y placer le code ASM qu’on appellera Stage1. Or, si l’on souhaite réaliser autre chose qu’un code minimal (p. ex. affichage de debug, chargement de Stage2, gestion plus avancée), ce code peut vite dépasser la taille de 416 octets et causer des erreurs d’assemblage ou de dépassement.

## Limites et erreurs rencontrées

### Dépassement de la zone 94..509

Dès que le code Stage1 excède ~416 octets, l’assembleur (NASM) tente de remplir le secteur (via directive times) jusqu’à l’offset 510. Si $, la position courante, est déjà supérieure à 510, la différence devient négative, d’où l’erreur :

* times value negative

ce qui signale que le code occupant la zone est trop volumineux. Cela entraîne l’échec de l’assemblage.

### incbin d’un fichier bpb.bin de 90 octets

Une première piste a consisté à :

* Inclure (incbin) un bpb.bin de ~90 octets (la BPB),
* Puis enchaîner directement un code ASM minimal dans le même fichier.

Malheureusement, des soucis d’alignement et d’offset exact se sont manifestés :

* La BPB s’étend parfois au-delà de 90 octets, car FAT32 peut stocker des champs “extended.”
* L’ajout d’un code minimal mal positionné risquait d’écraser des valeurs critiques.

**Résultat :** “mount: wrong fs type” au moment du montage, car la zone étendue était corrompue.

### incbin "bootsector\_original.bin" + org 0x5E

Une autre approche :

* Extraire le secteur 0 complet après formatage par mkfs.fat (512 octets), le fichier bootsector\_original.bin.
* Dans le code ASM, faire incbin "bootsector\_original.bin", puis org 0x5E pour positionner le code Stage1 en offset 94.

NASM a refusé divers aspects :

* “program origin redefined,” car on ne peut pas avoir [org 0] puis [org 0x5E] dans le même fichier.
* “attempt to assemble code in [ABSOLUTE] space,” error : NASM n’autorise pas l’assemblage d’instructions hors d’un bloc data pur en “[absolute offset].”
* “non-constant argument to times,” car (510 - ($-0x5E)) n’est pas jugé constant par l’assembleur.

En clair, l’idée “org 0x5E” a buté sur les limitations de NASM concernant la directive org multiple et la syntaxe absolute.

## La solution par patch\_code.bin (injection)

### Principe du patch

Pour contourner ces difficultés, la méthode la plus fiable a été :

* Assembler un code ASM minimal (Stage1) en un fichier séparé, qu’on appellera patch\_code.bin, dont la taille fait ~200 octets.
* Dans ce patch ASM :
  + On utilise [org 0], ce qui rend $ (la position courante) un offset constant depuis 0.
  + On peut employer times (...) db 0 sans erreur “non-constant,” du moment qu’on ne dépasse pas 416 octets.
  + On place nos instructions, affichage debug éventuel, etc.

Injecter ce binaire dans le secteur 0, offset 94, du vrai boot sector FAT32 généré par mkfs.fat :

* dd if=patch\_code.bin of=bootsector\_original.bin bs=1 seek=94 conv=notrunc

Autrement dit, on remplace octet par octet, dans bootsector\_original.bin, l’espace 94..(94 + taille\_patch - 1) par le code ASM.

Enfin, bootsector\_original.bin, modifié, redevient le boot sector final qu’on écrit sur la partition :

* dd if=bootsector\_original.bin of=/dev/loop0p1 bs=512 count=1 conv=notrunc

*(ou tout autre moyen d’inscrire en LBA=0 de la partition)*

### Avantages

* On préserve la zone 0..93 (BPB) + extensions, ainsi que 510..511 = 0x55AA.
* L’assembleur n’a pas à gérer un second org. On compile un code ASM “org 0” classique.
* On évite les erreurs “times value negative,” car la taille du patch est maîtrisée.
* On obtient un boot sector complet de 512 octets, convenant au BIOS et au montage Linux.

### Limites

* Le code Stage1 demeure limité à ~416 octets au total. Au-delà, on écrase 510..511, ou on a un “times value negative.”
* Il faut gérer la maintenance de patch\_code.bin séparément, puis penser à l’injection via dd. Un script build.sh s’y prête bien.
* Si l’on a besoin d’un Stage1 plus gros (p. ex. parse FAT complet), il faudra charger un Stage2 en LBA fixe, car ce n’est pas faisable entièrement dans les 416 octets restants.

L’inclusion d’un code ASM plus complet dans le boot sector FAT32 exige de préserver :

* La BPB (offset 0..~90) et champs étendus, qui décrivent la géométrie FAT32.
* La signature 0x55AA en offset 510..511, sans laquelle ni BIOS ni Linux n’accepteront le boot sector.
* La limite de ~416 octets de code maximum.

Les premières tentatives (incbin d’un bpb.bin ou usage d’un second org 0x5E) ont conduit à des erreurs multiples. La méthode “patch” consiste à assembler un petit code dans patch\_code.bin (org 0), puis à l’injecter en offset 94 via dd. Ce procédé :

* Évite de corrompre la structure FAT32,
* Circule autour des limites de nasm concernant org multiples et non-constant times,
* Permet d’inscrire un code d’amorçage un peu plus élaboré (jusqu’à 200–300 octets de logique) dans le secteur 0.

Ainsi, le boot sector final, une fois écrit sur la partition, reste montable et bootable, offrant un Stage1 capable de charger un Stage2 ou d’afficher un message de debug avant de poursuivre.

# QEMU “Booting from Hard Disk…” bloqué

Ici, la situation observée était qu’après avoir soigneusement préservé la structure FAT32 au niveau du secteur 0 de la partition (en veillant notamment à ce que la signature 0x55AA demeure présente aux offsets 510–511), QEMU persistait à afficher le message “Booting from Hard Disk…” sans aller plus loin. Aucune instruction du code Stage1 n’apparaissait exécutée (en particulier, aucun appel à int 0x10, AH=0x0E n’émettait de caractère), et il n’y avait aucune trace de progression ou de message de débogage. Plusieurs hypothèses ont alors été étudiées pour cerner pourquoi le BIOS de QEMU ne transférait pas la main au code d’amorçage.

## Éventuelle nécessité d’un MBR standard

### Contexte traditionnel du boot “Hard Disk”

Sur un disque dur, le BIOS (et plus particulièrement SeaBIOS dans QEMU) s’attend classiquement à trouver :

* Au secteur 0 du disque : un Master Boot Record (MBR), contenant :
  + Une section de code d’amorçage (les 446 premiers octets environ).
  + La table de partitions (16 octets par partition, jusqu’à 4 partitions primaires).
  + La signature 0x55AA en fin (offset 510..511).
* Sur la partition active : le boot sector (souvent appelé VBR), qui, lui, inclut la BPB et la signature 0x55AA, plus un code minimal.

Le BIOS lit d’abord le secteur 0 du disque (MBR), exécute le code qu’il contient. Ce code MBR se charge de localiser la partition active et de lui passer la main en lisant son secteur 0 (VBR). Si, au contraire, l’utilisateur a directement installé un boot sector FAT32 complet dans le secteur 0 du disque sans la structure MBR, SeaBIOS risque de ne pas détecter ce disque comme amorçable ou de ne pas effectuer le chaînage approprié. C’est ainsi que le BIOS peut rester figé sur “Booting from Hard Disk…” sans franchir l’étape de lancer le code du VBR.

### Raisons de l’absence de MBR

Dans la configuration actuelle, l’idée était de placer immédiatement le boot sector FAT32 (comportant la BPB, etc.) en offset 0 du disque. Cela peut convenir à un environnement de type superfloppy ou à un BIOS plus permissif, mais sur QEMU, quand il s’agit d’un “disque dur,” SeaBIOS recherche un code MBR standard. En l’absence de ce code, le BIOS juge le disque non amorçable et se bloque.

Conclusion partielle : L’une des principales causes envisagées du blocage est que SeaBIOS, en mode HDD, ne reconnaît pas le boot sector FAT32 “tout seul” comme code d’amorçage valable. D’où l’idée d’installer un MBR minimal qui effectuerait la transition vers ce secteur 0 de la partition (VBR).

## Problème potentiel dans l’exécution réelle du code Stage1

### Possibilité d’un plantage immédiat

Même s’il arrivait que le BIOS exécute le secteur 0, il se peut que le code Stage1 meure instantanément, par exemple à cause :

* D’une initialisation incorrecte de registres, dont le registre DL, qui indique quel disque est en usage (0x80 pour le premier disque dur). Si le code Stage1 part du principe que DL=0x80 alors que le BIOS a mis une autre valeur, toute interruption 0x13 (lecture de secteurs) échouerait brutalement.
* D’un saut invalide, d’une adresse en dehors de la zone loaded, ou d’un DAP (Disk Address Packet) mal configuré pour la lecture de Stage2, causant un lock ou un reset silencieux avant d’afficher le moindre message.

### Manque de debug visible

Si Stage1 plante très tôt ou exécute un code erroné, aucun message via int 0x10 ne se produit. On perçoit alors le même symptôme : “Booting from Hard Disk…” qui ne progresse pas. Dans les journaux QEMU, aucun caractère n’apparaît. L’utilisateur croit alors que le BIOS n’a pas exécuté le code, alors qu’il l’a peut-être fait, mais le code s’est effondré en silence. Faire un debug plus poussé (ex. activer GDB, ou -d int,cpu\_reset) aiderait à confirmer si le code ASM est lu ou non.

## Hypothèse d’absence d’affichage debug

### Raisons possibles de l’absence de trace

* Le code Stage1 n’est jamais appelé parce que le BIOS ne reconnaît pas le secteur 0 comme un code MBR.
* Le code Stage1 est appelé mais meurt immédiatement, faute de segments initialisés, d’un saut invalide, ou d’instructions non reconnues en mode réel.
* L’int 0x10 n’est pas invoqué correctement (ex. AH = 0x0E, BH=0, BL=0x07 pour du texte blanc sur fond noir), ou l’assembleur a dépassé la limite d’octets autorisée dans le boot sector (416 octets), menant à un patch incohérent.

### Recours à un message “tout début”

Pour lever le doute, un code Stage1 minimal insérant dès la première instruction un :

* mov ah, 0x0E
* mov al, 'A'
* int 0x10
* jmp $

permet de voir si on obtient un ‘A’ à l’écran. Sans ce test, on demeure incertain. Les logs QEMU ou un usage de GDB peuvent également éclairer ce moment critique.

## Solutions envisagées pour dépasser le blocage

### Installer un MBR minimal ou standard (Syslinux, GRUB, etc.)

La méthode la plus universelle :

* Le secteur 0 du disque contient un code MBR standard (446 octets de code + 64 octets de partition table + 2 octets signature).
* Ce code MBR repère la partition active, lit son secteur 0 (offset LBA correspondant), et y saute.
* Le boot sector de la partition (FAT32) agit ensuite comme VBR, lisant Stage2 ou KERNEL.BIN.

C’est la stratégie conforme aux BIOS traditionnels pour un disque dur. De nombreux projets utilisent un “mbr.bin” standard (tel que Syslinux, GRUB, etc.) : on écrit ce mbr.bin sur les 446 premiers octets, on préserve la table de partitions (464..509?), et on garde la signature 0x55AA en 510..511. QEMU reconnaît alors la structure et exécute le code MBR.

**Inconvénient :** Cela oblige à employer un code MBR externe (Syslinux) ou à écrire un MBR minimal soi-même. Si l’objectif était d’éviter tout composant externe, c’est plus délicat.

### Forcer QEMU à traiter l’image comme “floppy”

En spécifiant -fda disk.img ou -drive if=floppy,format=raw,file=disk.img, QEMU se comportera différemment :

* Il lit directement le secteur 0 comme un boot sector de disquette.
* Il n’exige plus de code MBR.
* Cependant, la taille d’une “disquette” dépassant 2.88 Mo peut poser d’autres soucis (SeaBIOS évalue la géométrie ou rejette un volume trop volumineux).

Cette solution s’écarte de la volonté d’avoir un disque dur FAT32 standard, mais elle peut permettre de déverrouiller la situation et valider que le code Stage1 (VBR) s’exécute réellement.

### Corriger l’initialisation de Stage1

Si on suppose que le BIOS appelle bien le boot sector, alors vérifier :

* La première instruction : un jump valide ou un short jump + NOP ?
* L’initialisation du registre DL (0x80 pour le premier disque dur).
* La place occupée par Stage1 ne dépasse pas ~416 octets.
* Les interruptions (int 0x13, int 0x10) y sont correctement paramétrées.
* Les instructions de lecture de secteurs ou le DAP (Disk Address Packet) ne sont pas erronées.

## Conclusion et perspectives

Plusieurs indices suggèrent que QEMU, en mode “Hard Disk,” ne chaîne pas directement le boot sector FAT32 si aucun code MBR complet n’est présent dans le secteur 0 :

* Le BIOS de QEMU (SeaBIOS) attend un code MBR identifiant la partition active. Sans ce code, il reste bloqué sur “Booting from Hard Disk…”
* L’exécution de Stage1 peut néanmoins être effectif si on lui fournit un MBR minimal (ou syslinux mbr.bin).
* À défaut, on peut forcer l’émulation “floppy,” au prix de s’éloigner de la configuration “disque dur FAT32.”

Les tentatives de contournement ont inclus la vérification de la signature 55AA, la préservation du BPB, l’écriture d’un patch code Stage1 dans la zone 94..509, et l’option d’afficher un message de debug via int 0x10. Aucune n’a résolu la situation tant que QEMU demeurait en mode HDD, indiquant la forte probabilité que le BIOS n’exécute pas le VBR dans ce cas.

#### Marche à suivre :

* Intégrer un MBR minimal : en écrivant un code occupant les 446 premiers octets + la table de partitions + la signature. On se rapproche alors du flux normal d’un PC : MBR → partition active → boot sector FAT32.
* Ou passer en superfloppy en forçant -fda, tout en acceptant de ne plus émuler un disque dur.
* Dans tous les cas, on doit s’assurer que Stage1 ne plante pas dès l’exécution (DL correct, code occupant ≤416 octets, instructions int 0x13 valides).

Ainsi, le problème “QEMU ‘Booting from Hard Disk…’ bloqué” se justifie principalement par l’absence d’un MBR standard dans le secteur 0, et potentiellement par un bug interne dans Stage1, l’empêchant d’afficher le moindre message. Le plus fiable demeure de mettre en place un code MBR minimal qui chain-load la partition FAT32, suivant la configuration usuelle d’un BIOS cherchant à booter un disque dur.

# “non-constant argument to TIMES” dans Stage2

Ici, il s’agissait de produire un binaire Stage2 d’une taille précise à destination du code Stage1. Cependant, l’emploi d’une directive NASM de type times (4096 - ($ - 0x8000)) db 0 a provoqué l’erreur “non-constant argument supplied to TIMES”. Ce rapport explique :

Pourquoi l’expression ($ - 0x8000) n’est pas considérée comme constante par NASM,

Comment on a modifié l’assemblage pour contourner cette limite,

Les implications techniques du fait de charger Stage2 à 0x8000 en RAM alors qu’on l’assemble en “org 0.”

## Contexte : assembler Stage2 pour un chargement à 0x8000

### Objectif initial

Le but était de concevoir un binaire Stage2 :

* Dont la taille totale ferait 4096 octets (ou tout autre multiple de la taille d’un certain nombre de secteurs).
* Qui serait chargé en RAM à l’adresse segment:offset équivalente à 0x0000:0x8000 (soit l’adresse linéaire 0x8000) par le code Stage1.
* Qui profiterait d’un “org 0x8000” pour être cohérent : le code Stage2 se verrait comme étant placé à 0x8000 en mémoire, permettant d’adresser, par exemple, des labels en tenant compte de ce décalage.

### Problème de la directive times

Pour remplir les octets restants jusqu’à 4096, il a été tenté :

* [org 0x8000]
* bits 16
* ; ...
* times (4096 - ($ - 0x8000)) db 0

L’expression ($ - 0x8000) est censée représenter “le nombre d’octets déjà utilisés dans ce fichier (=$) moins 0x8000.” Or, NASM n’évalue pas cela comme une constante. Le compilateur d’assemblage refuse donc de calculer times (someNonConstantValue).

**Raison :**

[org 0x8000] ne modifie pas réellement la taille du fichier en cours d’assemblage ; cela indique plutôt l’adresse en mémoire où le code se retrouvera.

À l’intérieur de times, $ représente la position courante dans le flux binaire, et 0x8000 ne peut s’y soustraire de façon purement statique, selon les règles de NASM. On obtient donc l’erreur “non-constant argument supplied to TIMES.”

## Solution : assembler Stage2 en org 0, puis charger en 0x8000

### Méthode : [org 0] + times (4096 - ($))

Pour contourner l’impossibilité d’utiliser ($ - 0x8000), on s’est orienté vers :

* Assembler Stage2 comme si son point de départ (offset) valait 0.
  + [org 0]
  + bits 16
  + ; ...
  + times (4096 - ($)) db 0
* Ainsi, $ devient un compteur depuis le début du fichier binaire (0..4095), ce qui est un calcul strictement constant aux yeux de NASM.
* Créer un binaire “stage2.bin” de 4096 octets exactement. La ligne times (4096 - ($)) db 0 veille à remplir le fichier jusqu’à 4096, quelles que soient la longueur de code et de data déjà insérés.

### Charger Stage2 à 0x8000 via Stage1

Par ailleurs, on sait que Stage1, fonctionnant en mode réel, va lire (par int 0x13 ou autre mécanisme) les 4096 octets de “stage2.bin” depuis le disque, puis les copier en RAM à l’offset 0x8000. De la sorte, l’assembleur ignore l’adresse de chargement réelle, la compilation se fait “org 0”, et la relocalisation “physique” incombe au code Stage1.

**Avantages :**

* Le calcul times (4096 - ($)) devient trivial et constant : $ est le compteur local, partant de 0, facile à évaluer par NASM.
* Plus de conflit avec “non-constant argument.”
* La taille finale du binaire stage2.bin est garantie d’être 4096 octets. On peut alors l’écrire sur la partition à un LBA fixe (par exemple LBA=1..8) ou un fichier sur FAT32, selon le choix.

**Inconvénients :**

Le code Stage2 doit assumer qu’il sera “manuellement” chargé en 0x8000, et que les adresses absolues (p. ex. jumps) doivent être écrites de manière relative ou traitées au runtime. En pratique, on privilégie le code relatif ou on fait du 16-bit “org 0” sans recourir à des adresses absolues, pour éviter les soucis de relocation.

## Justification technique

### Org multiple, non-constant times

Si l’on insistait pour faire [org 0x8000], $ dans NASM représenterait la position dans le fichier binaire, mais 0x8000 n’est pas un offset “fichier,” c’est une adresse de chargement en RAM. Le calcul $ - 0x8000 n’est donc pas un littéral constant que l’assembleur peut résoudre à la compilation. NASM exige que times reçoive un entier constant, ce qui ne se vérifie pas ici.

### Indépendance de l’assembleur et du chargeur

En adoptant [org 0], on rend la taille du fichier binaire purement déterminée par $. L’endroit mémoire (0x8000) demeure un concept de runtime. On laisse la charge (Stage1 ou tout autre loader) lire stage2.bin et le poser à 0x8000. De fait, on gagne en souplesse : si, plus tard, on souhaitait charger Stage2 ailleurs (p. ex. 0x9000), il suffirait de changer le chargeur, sans recompiler Stage2.

## Commandes et résumés

Pour assembler Stage2 avec la méthode aboutie :

* [org 0]
* bits 16
* ; ... instructions Stage2 ...
* ; ...
* times (4096 - ($)) db 0

Puis, dans un script :

* nasm -f bin boot2.asm -o stage2.bin
* ls -l stage2.bin
* => 4096 octets

Stage1 charge “stage2.bin” (4096 octets) en RAM à 0x8000. On n’utilise plus l’expression (4096 - ($ - 0x8000)).

L’erreur “*non-constant argument to TIMES*” apparaît parce qu’une expression comme ($ - 0x8000) n’est pas considérée comme une constante par NASM, compte tenu de la directive [org 0x8000] et du fait que $ se réfère à la position courante dans le fichier binaire. Pour résoudre cela, il suffit :

* D’assembler Stage2 en org 0,
* De calculer la taille de remplissage via times (4096 - ($)),
* De laisser le code Stage1 effectuer la charge en mémoire à l’adresse 0x8000, plutôt que de l’imposer directement à l’assembleur.

Ainsi, on obtient un binaire Stage2 de 4096 octets fixes, sans erreur d’assemblage, et la relocalisation se fait “manuellement” : Stage1 lit stage2.bin et le recopie à 0x8000. Ce mécanisme contourne la limitation technique de NASM tout en assurant que Stage2 respecte la taille visée.

# Un script final build.sh

Dans l’optique d’obtenir un workflow fiable et reproductible pour construire l’OS bare-metal en Rust et le booter sur une partition FAT32, un script complet a été mis au point. Cette section expose :

* Le rôle de ce script : l’automatisation, la création d’une image disque, son partitionnement, le formatage FAT32, l’injection du code ASM (Stage1, Stage2), la compilation du kernel, et enfin le lancement sous QEMU.
* Les commandes concrètes utilisées étape par étape, ainsi que les justifications techniques.
* Les problèmes (QEMU bloqué sur “Booting from Hard Disk…”) et hypothèses explicatives.
* Les pistes pour corriger ou améliorer la compatibilité BIOS / MBR.

## But et structure du build\_full.sh

### Objectif

Le script build.sh vise à :

* Créer une table de partitions MBR (msdos) sur disk.img.
* Créer une unique partition FAT32 (typiquement 1 MiB → 63 MiB).
* Formater cette partition en FAT32 via mkfs.fat.
* Extraire le secteur 0 de la partition (boot sector), y injecter un code ASM Stage1 sans casser la BPB et la signature 0x55AA.
* Assembler Stage2, l’écrire dans quelques secteurs (ex. LBA=1..8).
* Compiler le kernel Rust via cargo rustc -Z build-std=core,compiler\_builtins.
* Copier KERNEL.BIN sur la partition FAT32.
* Lancer QEMU (-drive file=disk.img,format=raw) pour tester le boot.

Rendre l’ensemble automatique permet d’éviter les erreurs manuelles de manipulation (loop devices mal détachés, partition non montée/démontée, etc.).

### Commandes et déroulé technique

#### Création et partitionnement

Création du fichier :

* dd if=/dev/zero of=disk.img bs=1M count=64

On obtient un disk.img de 64 Mo.

parted pour label msdos + partition unique FAT32 :

* parted -s disk.img mklabel msdos
* parted -s disk.img mkpart primary fat32 1MiB 63MiB
* parted -s disk.img set 1 boot on

On réserve ~1 MiB au début, ~63 MiB à la fin, ce qui donne ~62 Mo pour FAT32. Mark “boot on” rend la partition active (en cas de MBR usage standard).

Formatage en FAT32 :

Plusieurs possibilités. Soit on associe l’image à un loop device, puis :

* sudo losetup -fP disk.img
* sudo mkfs.fat -F 32 /dev/loop0p1
* sudo losetup -d /dev/loop0

Ou on peut user de mkfs.fat -F 32 disk.img si partitionless, mais ici on se concentre sur la partition.

## Extraction du secteur 0 (VBR) et injection de Stage1

### Extraction du boot sector

* sudo losetup -fP disk.img
* dd if=/dev/loop0p1 of=bootsector\_original.bin bs=512 count=1
* sudo losetup -d /dev/loop0

On obtient un fichier bootsector\_original.bin (512 octets) contenant la BPB, un code minimal, la signature 0x55AA, etc. mis par mkfs.fat.

### Assemblage de Stage1 en patch\_code.bin

* nasm -f bin patch\_code.asm -o patch\_code.bin

Ce code, de ~200 octets, s’assemble avec [org 0], occupant offset 0..(taille-1).

### Injection dans le secteur 0, offset 94

* dd if=patch\_code.bin of=bootsector\_original.bin bs=1 seek=94 conv=notrunc

Préserve offset 0..93 (BPB) et 510..511 (signature).

### Écriture du boot1.bin dans /dev/loop0p1

* cp bootsector\_original.bin boot1.bin
* dd if=boot1.bin of=/dev/loop0p1 bs=512 count=1 conv=notrunc

Désormais, la partition contient un boot sector “hybride” : la BPB est intacte, signature 0x55AA aussi, et la zone 94..509 renferme Stage1 ASM.

### Assemblage de Stage2 et compilation du kernel

#### Stage2

* nasm -f bin boot2.asm -o stage2.bin
* dd if=stage2.bin of=disk.img bs=512 seek=1 count=8 conv=notrunc
* LBA=1..8 : 4096 octets. Stage1 lit ces secteurs en RAM (par ex. 0x8000).

#### Compilation du kernel Rust

* cargo rustc -Z build-std=core,compiler\_builtins --target=i686-none.json –release
* mv target/i686-none/release/MONCOMBLE\_OS KERNEL.BIN

On obtient un binaire no\_std, qui sera copié sur la partition FAT32 :

* sudo losetup -fP disk.img
* sudo mount /dev/loop0p1 mnt
* sudo cp KERNEL.BIN mnt/
* sync
* sudo umount mnt
* sudo losetup -d /dev/loop0

### Lancement QEMU

Enfin, on lance :

* qemu-system-i386 -drive file=disk.img,format=raw -m 256

Le BIOS QEMU est censé charger le MBR (secteur 0 du disque), puis chaîner la partition. Si tout fonctionnait, on verrait Stage1 => Stage2 => kernel – *ce qui n’est pas le cas.*

## Blocage sur “Booting from Hard Disk…” et hypothèses

Malgré le script final, QEMU restait figé sur “Booting from Hard Disk…,” suggérant que le BIOS n’exécute pas le VBR patché. Parmi les hypothèses :

### Absence d’un code MBR standard

Dans un usage normal, le secteur 0 du disque contient un MBR qui localise la partition active. Or, l’approche décrite place directement le code FAT32 boot sector dans la partition, sans MBR occupant le secteur 0 du disque. Le BIOS SeaBIOS cherche un MBR pour un disque dur, ne le trouve pas, et ne chaîne pas le VBR => blocage.

### Strict SeaBIOS

SeaBIOS peut exiger une structure standard de disque dur (MBR, table de partitions). S’il voit juste un label msdos + partition, mais pas de code MBR en offset 0, il ne remet jamais la main à la partition active.

### Problèmes de patch ou offsets

Si le script s’exécute dans le désordre, ou s’il y a un reliquat de loop device monté, on pourrait avoir une corruption du boot sector ou de la FAT. Toutefois, dans la majorité des tests, c’est l’absence de MBR qui semble la cause principale.

### Approches alternatives

#### Installer un MBR minimal

* On peut copier un petit code MBR occupant les 446 premiers octets, suivi de la table de partitions (64 octets), puis la signature 0x55AA. Ce code MBR standard repérera la partition bootable, lira son secteur 0, et exécutera Stage1 patché.
* Par exemple, en utilisant un mbr.bin issu de Syslinux ou GRUB.

#### Forcer QEMU à considérer l’image comme une disquette

* qemu-system-i386 -fda disk.img -boot a

Alors le BIOS lit direct le secteur 0 comme un boot sector de floppy, sans exiger de MBR. Mais ce n’est plus un “disque dur” FAT32, c’est un superfloppy, ce qui s’éloigne de l’objectif initial.

#### Vérifier si on veut un “partitionless” superfloppy (pas d’MBR ni parted), ou un “disque dur”

Le script build.sh actuel simule un disque dur avec parted, mais la structure d’amorçage n’est pas conforme aux attentes du BIOS, car on n’a pas de code MBR.

Le script build.sh, dans sa forme actuelle, enchaîne correctement :

* Création d’un disk.img de 64 Mo,
* Partition unique FAT32 via parted,
* Formatage mkfs.fat,
* Extraction boot sector, injection Stage1 offset 94,
* Assemblage + écriture Stage2,
* Compilation + copie du kernel,
* Lancement QEMU.

Toutefois, QEMU reste bloqué sur “Booting from Hard Disk…,” faute d’un code MBR dans le secteur 0 du disque. Le BIOS n’exécute donc pas le VBR patché. Pour aller plus loin, la solution la plus “standard” consiste à introduire un MBR minimal qui, lui, saura chaîner la partition active où se trouve le VBR. Alternativement, on peut forcer “-fda disk.img” et accepter le mode disquette.

En résumé, build\_full.sh réalise une automatisation complète, mais l’absence de code MBR demeure l’obstacle principal au boot effectif.

Pour atteindre l’objectif d’un “disque dur FAT32” bootable sous QEMU, il est nécessaire d’ajouter un MBR standard ou de choisir l’option disquette. Cela constituera le prochain pas pour finaliser un boot fonctionnel.

# Conclusion et perspectives : synthèse des réalisations et pistes pour la suite

Tout au long de ce projet, l’objectif a été de créer un système d’exploitation écrit en Rust no\_std, avec un code d’amorçage (Stage1, Stage2) chargé sur une partition FAT32 tout en préservant la montabilité du volume sous Linux. Les résultats finaux montrent un blocage persistant sur “Booting from Hard Disk…” sous QEMU, mais de nombreux progrès ont été réalisés et d’importants enseignements techniques ont été tirés.

## Réalisations effectives du projet

### Compilation Rust no\_std

Le kernel Rust s’est avéré compilable en mode no\_std, avec un slab allocator fonctionnel et un \_start() en lieu et place de la fonction main(). L’infrastructure “i686-none.json” ou “x86\_64-unknown-none,” la configuration de la toolchain nightly, et l’option -Z build-std=core,compiler\_builtins ont été validées.

### Code Stage1 + Stage2

Un code assembleur minimal (Stage1) a été écrit pour occuper ~416 octets dans le secteur 0 (offset 94..509), préservant la BPB + signature 0x55AA. Il se chargeait d’appeler un Stage2 plus volumineux, censé charger KERNEL.BIN depuis la FAT, en théorie. Les assemblages ont été gérés via NASM, avec un usage de directives incbin ou patch, et la résolution de problèmes de directive org, times, etc.

### Expérimentations avec parted, sfdisk, mkfs.fat, incbin

La mise en place d’un script build.sh a englobé la création et le partitionnement d’un disk.img (64 Mo), la configuration FAT32, l’extraction du boot sector original, l’injection d’un patch code ASM, l’écriture de Stage2, la compilation du kernel Rust, et le lancement de QEMU.

## Points de blocage majeurs

### Chaînage BIOS

Sans un code MBR dans le secteur 0 du disque (et non simplement de la partition), le BIOS SeaBIOS de QEMU ne semble pas appeler le VBR patché, aboutissant à “Booting from Hard Disk…” bloqué. Traditionnellement, un code MBR identifie la partition active, puis y lit le secteur 0 (le boot sector). L’absence d’un MBR standard rompt ce chaînage.

### Espace restreint dans le boot sector

On ne dispose que de ~416 octets (offset 94..509) pour Stage1, car on doit laisser intacts la BPB (offset ~0..93) et la signature (510..511 = 0xAA55). Lorsque Stage1 dépasse cette taille, on obtient des erreurs du type “times value negative.”

### Nasm et “non-constant argument to TIMES”

Les expressions comme (4096 - ($ - 0x8000)) sont refusées par NASM en raison de son évaluation stricte des constantes. On a contourné cela en fixant [org 0] et en laissant $ partir de 0. La relocation réelle (chargement à 0x8000) est gérée par le code Stage1, sans impliquer l’assembleur.

### Maintien de la structure FAT32 pour Linux

Pour monter la partition, il est impératif de ne pas écraser la BPB (ou des zones étendues) ni la signature 0x55AA. Les techniques de “patch” (injection d’un code ASM en offset 94) garantissent la conservation de la zone FAT, mais imposent des limites de taille et de positionnement.

## Pistes et propositions pour la suite

### Installer un MBR standard

La plus “orthodoxe” solution pour booter un disque dur est de placer dans le secteur 0 du disque (et non de la partition) un Master Boot Record complet, qu’il soit minimal (quelques dizaines d’octets + table) ou provenant d’outils existants (p. ex. Syslinux mbr.bin). Ce code MBR sait détecter la partition active et exécuter son secteur 0. C’est la pratique normalisée pour un BIOS.

### Minimiser Stage1

Si Stage1 doit résider dans ~416 octets, on peut en faire un code minimaliste (un simple jump + routine d’accès disque) qui lit un Stage2 plus grand en LBA fixe. Stage2, lui, aura davantage de place pour parser la FAT et charger KERNEL.BIN.

### Forcer QEMU en mode “floppy”

Si on souhaite absolument éviter le MBR, on peut recourir à -fda disk.img ou -drive if=floppy,format=raw,file=disk.img. Cela transforme disk.img en superfloppy. On contourne la problématique MBR, mais on s’éloigne de l’objectif “disque dur FAT32.”

### S’appuyer sur un bootloader existant (Syslinux/GRUB)

Ces outils prennent en charge le code MBR, la détection de partitions, et le chaînage vers un kernel ELF ou un binaire raw. On perd un peu de contrôle bas niveau, mais on simplifie énormément l’amorçage, s’épargnant la maintenance d’un code MBR/Stage1 propre.

## Conclusion finale

L’absence d’un résultat concret — QEMU se figeant sur “Booting from Hard Disk…” — ne remet pas en cause les multiples avancées :

* Un kernel Rust no\_std compilable et fonctionnel, avec un slab allocator.
* Un code ASM (Stage1, Stage2) pour lire la partition FAT32 en théorie.
* Un script build.sh rassemblant parted, mkfs.fat, injection ASM, compilation Rust, et QEMU.

Le point manquant reste l’enchaînement BIOS : un BIOS standard, pour un disque dur, cherche un MBR. Ne le trouvant pas, il n’exécute pas le VBR patché. D’autres pistes (mode superfloppy ou MBR minimal) sont à envisager pour aboutir à un OS complet boottable en partition FAT32, tout en maintenant la compatibilité avec le montage Linux. Malgré ces obstacles, la base technique (Rust no\_std, Stage1→Stage2→kernel) est clairement définie et n’attend plus qu’une adaptation du chaînage BIOS pour être pleinement opérationnelle.