ESGI
école supérieure de génie informatique

RAPPORT : FAT32 FILESYSTEM EN RUST

Daniela Ceraku
23 Février 2025 | ESGI

Sommaire

Implémentation d'un Système de Fichiers Basé sur FAT32

1. Introduction

L'objectif de ce projet était d'implémenter un système de fichiers léger et efficace de type FAT32 pour un système d'exploitation personnalisé que j'a appelé My_OS. Le système de fichiers fournit des fonctionnalités de base telles que la gestion des processus, l'allocation de mémoire et les appels système, assurant une interaction fluide entre le noyau et les programmes en espace utilisateur. Le projet adhère à l'environnement no_std, ce qui le rend adapté aux systèmes embarqués et aux noyaux de systèmes d'exploitation.

Ce rapport explique la structure du projet, les choix techniques, le fonctionnement du code et les liens entre les différents composants.

2. Structure du Projet

Le projet suit une structure modulaire, où chaque dossier et fichier remplit un objectif distinct. Voici un aperçu de la hiérarchie du projet :

```
my_os/
├─ src/
│  ├─ directory/        // Modules du système de fichiers FAT32
│  │  ├─ attribute.rs    // Attributs des fichiers comme lecture seule, caché, système
│  │  ├─ cluster.rs      // Gestion des clusters pour FAT32
│  │  ├─ datetime.rs     // Gestion des dates et heures conforme à FAT
│  │  ├─ dir_entry.rs    // Opérations sur les entrées de répertoire
│  │  ├─ name.rs         // Prise en charge des noms de fichiers courts et longs
│  │  ├─ offset_iter.rs  // Itérateur pour la lecture des clusters
│  │  └─ table.rs        // Gestion de la table FAT
│  ├─ process/          // Gestion des processus
│  │  ├─ context.rs      // Changement de contexte pour les processus
│  │  ├─ mod.rs          // Déclarations des modules
│  │  └─ process.rs      // Création et gestion des processus
│  ├─ tests/            // Tests unitaires et d'intégration
│  │  ├─ tests.rs        // Tests des fonctionnalités de base
│  │  └─ filesystem.rs   // Tests spécifiques au système de fichiers
│  ├─ lib.rs            // Bibliothèque principale du noyau
│  ├─ main.rs           // Point d'entrée de l'OS (_start)
│  ├─ memory.rs         // Gestion de la mémoire
│  ├─ scheduler.rs      // Planification des processus
│  ├─ slab.rs           // Allocateur de mémoire (Slab allocator)
│  └─ syscall.rs        // Interface des appels système
├─ Cargo.toml           // Dépendances du projet et configuration de build
└─ Cargo.lock           // Fichier de verrouillage des dépendances
```

Chaque module encapsule une fonctionnalité spécifique, favorisant la réutilisation et la maintenabilité du code. Cette approche modulaire simplifie le débogage et améliore la clarté.

3. Choix Techniques

3.1 Environnement No-Std

J'ai choisi `**#![no_std]**` pour créer un noyau léger sans dépendance à la bibliothèque standard. Cela est essentiel pour la programmation bas-niveau et les environnements embarqués. Le crate `**alloc**` permet les allocations sur le tas.

3.2 Système de Fichiers de Type FAT32

Le système de fichiers FAT32 a été choisi pour sa simplicité et sa compatibilité avec les systèmes embarqués. Il prend en charge :

- Le stockage basé sur des clusters.
- Les noms de fichiers courts (8.3) et longs (LFN).
- Les attributs de fichiers (lecture seule, caché, système).

3.3 Slab Allocator

Un allocateur de type slab a été implémenté pour une gestion efficace de la mémoire. Il réduit la fragmentation et assure une allocation et une libération rapides des blocs de mémoire.

3.4 Spinlocks et Mutex

Pour le contrôle de la concurrence, nous avons utilisé le crate `spin`, fournissant des spinlocks et des mutex. Cela garantit un accès sécurisé aux ressources partagées sans nécessiter de threads au niveau du système d'exploitation.

4. Aperçu du Code

4.1 lib.rs (Point d'Entrée pour les Modules du Noyau)

```
// Allocateur global
use slab::GlobalAllocator;
#[global_allocator]
static ALLOCATOR: GlobalAllocator = GlobalAllocator;
```

```
// Déclarations des modules
pub mod directory;
pub mod process;
pub mod memory;
pub mod syscall;
```

Principales fonctionnalités :
- `#[global_allocator]` : Enregistre l'allocateur de type slab.
- `pub mod` : Expose les modules pour les autres parties de l'OS.
- `#[alloc_error_handler]` : Gère les échecs d'allocation.

4.2 main.rs (Point d'Entrée de l'OS)

```
#[no_mangle]
pub extern "C" fn _start() -> ! {
    println!("Welcome to My OS!");

    // Créer et exécuter un processus exemple
    let mut process = Process::new("Init");
    process.run();
    process.terminate();

    loop {}
}
```

Principales caractéristiques :
- `#[no_mangle]` garantit que `_start` ne soit pas renommé lors de la compilation.
- `Process::new("Init")` crée un nouveau processus.
- `loop {}` maintient l'OS en cours d'exécution indéfiniment.

5. Lien Entre les Modules

- **Création de Processus** : `main.rs` appelle `Process::new` de `process.rs`.
- **Allocation de Mémoire**: `memory.rs` repose sur `slab.rs` pour les allocations.
- **Opérations sur les Fichiers** : `directory/` interagit avec `filesystem.rs` pour la création et la lecture des fichiers.
- **Appels Système** : `syscall.rs` fait le pont entre l'espace utilisateur et les fonctionnalités du noyau.

Exemple de flux :
1. `main.rs` → Créer un processus → `process.rs`.
2. `process.rs` → Allouer de la mémoire → `memory.rs`.

3. `memory.rs` → Allocation de type slab → `slab.rs`.
4. `filesystem.rs` → Créer un fichier → `directory/`.

6. Tests et Débogage

Les tests unitaires sont situés dans `src/tests/tests.rs` :

```
#[test]
fn test_process_creation() {
    let process = Process::new("TestProcess");
    assert_eq!(process.name, "TestProcess");
    assert_eq!(process.state, ProcessState::Ready);
}
```

7. Instructions de Build et de Test

7.1 Build en mode no_std (Kernel mode)

Pour compiler le projet en mode no_std, utilisez la commande suivante :

cargo build --no-default-features --features "no_std"

7.2 Exécuter les Tests en mode std

Pour exécuter les tests en mode standard, utilisez cette commande :

cargo test --features std

7.3 Activer le Débogage

Pour activer le débogage, utilisez la commande suivante :

cargo build --features "debug global_alloc"

7.4 Démarrer l'OS

Si vous utilisez bootimage pour créer une image amorçable :

cargo bootimage

7.5 Exécuter avec QEMU :

qemu-system-x86_64 -drive format=raw,file=target/x86_64-my_os/debug/bootimage-my_os.bin

8. Conclusion

Ce projet démontre un système de fichiers fonctionnel et modulaire de type FAT32 intégré à un système d'exploitation personnalisé. La structure modulaire améliore la maintenabilité, tandis que l'environnement `no_std` assure la compatibilité avec les systèmes embarqués. La combinaison d'un slab allocator, d'une gestion efficace des processus et d'une gestion robuste des fichiers garantit un noyau fiable.

Les améliorations futures incluent l'ajout d'un cache en écriture, un journal pour la résilience en cas de crash et un planificateur de processus plus sophistiqué.

Ce projet constitue une étape fondamentale vers le développement d'un système d'exploitation complet avec prise en charge du système de fichiers.