Université de Cergy-Pontoise

Département Informatique

Réalisation d'un système de fichiers simulé en C

Meriem Bouazzaoui
Eliarisoa Andriantsitohaina
Mario Razafinony

Encadré par : Monsieur Jean-Luc Bourdon

Avril 2025

Table des matières

1 Introduction

Ce document présente un projet de mini-système de fichiers, développé en langage C. Le système repose sur un fichier binaire agissant comme une partition virtuelle, nommé `(filesystem.img).` dans lequel sont stockés l'ensemble des données et métadonnées simulant le fonctionnement d'un véritable système de fichiers. L'objectif de ce projet est pédagogique : il permet de mieux comprendre les concepts fondamentaux qui régissent la gestion de fichiers, de répertoires, d'inodes, de blocs de données, ainsi que les mécanismes de lecture/écriture et de navigation dans un système de type UNIX. Le projet inclut également la gestion des liens (durs et symboliques), ainsi qu'un mini-shell interactif permettant d'interagir dynamiquement avec le système simulé via des commandes (comme `ls`,`cd`,`mkdir`,`touch`, etc.).

2 Structures de données

2.1 Structure principale

```
typedef struct filesystem {
    FILE *file;
    Inode inodes[NUM_INODES];
    Directory root_dir;
    Directory directories[NUM_INODES];
    int free_blocks[NUM_BLOCKS];
    int current_dir;
    OpenFile opened_file[MAX_FILE_OPEN];
} Filesystem;
```

`Filesysem` est la structure centrale qui regroupe toutes les données du système du fichiers.

- `file` : Pointeur vers le fichier binaire principal `filesystem.img` simulant le stockage sur disque.
- `inodes` : Tableau de taille fixe où chaque élément décrit un fichier, répertoire ou lien.
- `root_dir` : Représentation du répertoire racine (inode 0).
- `directories` : Contient tous les répertoires, indexés par leur inode.
- `free_blocks` : Tableau d'entiers indiquant si un bloc est libre (0) ou alloué (1).
- `current_dir` : Inode actuel (répertoire courant).
- `opened_file` :Tableau représentant les fichiers actuellement ouverts, pour gérer la lecture/écriture avec des descripteurs.

2.2 Les inodes

```
typedef struct inode {
    int id;
    int type;
    int size;
    time_t creation_time;
    time_t modification_time;
    char permissions[3];
```

```
    int blocks[NUM_BLOCKS];
    int link_count;
    int inode_rep_parent;
} Inode;
```

- `id` : Identifiant unique de l'inode.
- `type` : 0 = répertoire, 1 = fichier, 2 = lien symbolique.
- `size` : Taille en octets (pour un fichier).
- `permissions` : Chaîne de 3 caractères (ex. `"rwx"`).
- `blocks` : Liste des blocs physiques (indices dans `filesystem.img`) où sont stockées les données.
- `link_count` : Nombre de liens pointant vers l'inode (pour les liens durs).
- `inode_rep_parent` : Inode du répertoire parent (utile pour `..`).

2.3 Répertoire et entrées

```
typedef struct directory_entry {
    char filename[MAX_FILE_NAME];
    int inode_index;
} DirectoryEntry;

typedef struct directory {
    DirectoryEntry entries[NUM_DIRECTORY_ENTRIES];
} Directory;
```

- Chaque `directory_entry` associe un nom de fichier (`filename`) à un numéro d'inode (`inode_index`).
- Un `Directory` est donc un tableau de `DirectoryEntry`, représentant le contenu du répertoire.
- La taille est fixe (`NUM_DIRECTORY_ENTRIES`), ce qui impose un nombre maximum d'éléments.

2.4 Gestion des fichiers ouverts

```
typedef struct {
    int inode;
    int tete_lecture;
} OpenFile;
```

- `inode` : Numéro d'inode du fichier ouvert.
- `tete_lecture` : Position (en octets) où la prochaine lecture/écriture se fera dans `filesystem.img`.

3 Fonctions et algorithmes principaux

Cette section décrit les fonctions principales du système de fichiers simulé. Toutes les opérations critiques sur les fichiers ou répertoires (création, suppression, copie, déplacement) intègrent une gestion explicite des **droits d'accès** (permissions) au niveau des inodes, en se basant sur un système simplifié de permissions Unix : `r` (lecture), `w` (écriture), `x` (exécution).

3.1 Initialisation et sauvegarde

— `init_filesystem(filename)` : Initialise la structure `Filesystem`, met tous les inodes et blocs en « libre », et configure le répertoire racine (inode 0).
— `save_filesystem(filename)` et `load_filesystem(filename)` : Sérialisent la structure `fs` dans le fichier binaire `filesystem.img`.

3.2 Allocation et gestion des blocs (`allocate_block` / `free_block`)

— `allocate_block()` : Cherche un indice i dans `free_blocks` où `free_blocks[i] == 0` (bloc libre), puis le marque comme alloué (1).
— `free_block(i)` : Marque `free_blocks[i]` comme 0 (libre).

Gestion de la fragmentation

Dans le système de fichiers simulé, la gestion des blocs est assurée par un tableau `free_blocks[]` de taille fixe, où chaque case indique si un bloc est libre (valeur 0) ou occupé (valeur 1). Lorsqu'un nouveau fichier est créé ou qu'un fichier existant est agrandi, la fonction `allocate_block()` parcourt ce tableau séquentiellement pour trouver le premier bloc libre pour le marquer comme occupé.

```
int allocate_block() {
    for (int i = 0; i < NUM_BLOCKS; i++) {
        if (fs.free_blocks[i] == 0) {
            fs.free_blocks[i] = 1;
            return i;
        }
    }
    return -1;
}
```

Chaque inode qui représente chaque fichier possède un tableau `blocks[]` pour pouvoir indiquer à quelle position dans le fichier du filesystem il faudrait lire et écrire le contenu du fichier, et afin de pouvoir libérer les blocs alloués grâce à la fonction `free_block(int block_index)` lors de la suppression d'un fichier.

```
    void free_block(int block_index) {
    if (block_index >= 0 && block_index < NUM_BLOCKS) {
        fs.free_blocks[block_index] = 0;  // Marquer le bloc comme libre
        fprintf(fs.log,"\nLiberation du bloc %d\n",block_index);
    } else {
        printf("Erreur: tentative de liberation d'un bloc invalide (%d)
            .\n", block_index);
        fprintf(fs.log,"\nEchec de la liberation du bloc %d\n",
            block_index);
    }
}
```

Limites actuelles :
— Il n'existe pas de stratégie de réutilisation des espaces fragmentés entre des blocs alloués.
— L'algorithme utilise une politique dite *first fit*, ce qui peut entraîner une fragmentation externe au fil du temps.
— Aucun mécanisme de défragmentation ou de compactage des fichiers n'est prévu.

Proposition d'amélioration : Pour améliorer la gestion de la fragmentation, plusieurs solutions pourraient être envisagées :

- Implémenter un algorithme *best fit* ou *next fit* pour minimiser la fragmentation externe.
- Ajouter une table de mapping logique → physique pour autoriser les blocs discontinus tout en maintenant un suivi cohérent.
- Implémenter une fonction de *compactage périodique* du système de fichiers (défragmentation), en copiant les blocs de fichiers pour les rendre contigus, et en mettant à jour les métadonnées correspondantes.

Ce point représente une piste d'amélioration importante pour rendre le système de fichiers plus efficace, notamment dans le cas de nombreuses suppressions/créations successives de fichiers de tailles différentes.

3.3 Gestion des permissions

Avant chaque opération, une fonction dédiée `has_permission()` est appelée pour vérifier si l'inode visé possède la permission nécessaire :

- `w` est requise pour modifier un répertoire (ajout, suppression, déplacement d'éléments).
- `r` est nécessaire pour lire un fichier ou copier son contenu.
- `x` pourrait être utilisée dans des extensions futures pour exécuter un fichier ou autoriser l'accès à un répertoire.

```
int has_permission(int inode_index, char perm) {
    if (perm == 'r' && strchr(inode->permissions, 'r')) return 1;
    if (perm == 'w' && strchr(inode->permissions, 'w')) return 1;
    if (perm == 'x' && strchr(inode->permissions, 'x')) return 1;
    return 0;
}
```

3.4 Création de fichier et de répertoire

- **Fonction utilisée :** `create_file()`, `create_directory()`
- **Vérification de droits :** Permission `w` sur le répertoire parent requise.
- **Étapes principales :**
  1. Vérification que le nom n'existe pas déjà.
  2. Allocation d'un inode libre.
  3. Allocation d'un bloc (fichier uniquement).
  4. Mise à jour du répertoire parent.

Recherche d'une entrée vide dans un répertoire

La fonction `rechEntree()` permet de trouver la première entrée disponible dans un répertoire, nécessaire lors de l'ajout de nouveaux fichiers ou dossiers.

```
int rechEntree(int dir_inode) {
    Directory dir = fs.directories[dir_inode];
    for (int i = 0; i < NUM_DIRECTORY_ENTRIES; i++) {
        if (dir.entries[i].inode_index == -1) return i;
    }
```

```
    return -1;
}
```

3.5 Suppression de fichier ou de répertoire

- **Fonction utilisée :** `delete_file()`, `delete_directory()`
- **Vérification de droits :** Permission `w` sur l'inode visé.
- **Particularité :** Suppression récursive d'un répertoire et de tout son contenu si non vide.
- **Libération :** Les blocs de données et les inodes sont réinitialisés.

4.4 Suppression récursive d'un répertoire

L'algorithme `delete_directory()` supprime de manière récursive un répertoire et tout son contenu (fichiers, sous-répertoires, liens).

- Parcourt les entrées du répertoire.
- Pour chaque entrée :
  - Si c'est un fichier ou un lien : appel à `delete_file()`.
  - Si c'est un répertoire : appel récursif à `delete_directory()`.
- Une fois vide, supprime l'inode du répertoire lui-même.

3.6 Copie de fichier ou de répertoire

- **Fonction utilisée :** `copy_file()`, `copy_directory()`
- **Vérification de droits :**
  - `r` sur la source.
  - `w` sur le répertoire de destination.
- **Répertoire :** copie récursive du contenu (appel de `copy_directory()` sur les sous-répertoires).
- **Fichier :** lecture complète du contenu + écriture bloc par bloc dans une nouvelle structure.

Copie récursive d'un répertoire

L'algorithme `copy_directory()` explore récursivement le contenu d'un répertoire pour en créer une réplique exacte (sous-répertoires inclus) dans un autre emplacement.

- Vérifie les permissions sur la source et la cible.
- Crée un nouveau répertoire avec le même nom.
- Parcourt les entrées du répertoire source.
  - S'il s'agit d'un fichier, appel de `copy_file()`.
  - S'il s'agit d'un sous-répertoire, appel récursif de `copy_directory()`.

3.7 Déplacement de fichier ou de répertoire

- **Fonction utilisée :** `move_file()`, `move_directory()`
- **Vérification de droits :**
  - `w` sur le répertoire source (pour supprimer l'entrée).
  - `w` sur le répertoire cible (pour ajouter l'entrée).
- **Étapes :**

1. Retrait de l'entrée du répertoire source.
2. Ajout dans le répertoire cible.
3. Mise à jour de l'inode (répertoire parent, timestamp).

3.8 Navigation et chemins

— `changerRep(...)` : Equivalent d'un `cd`, met à jour le `current_dir`.
— `get_inode_from_path(...)` : Découpe le chemin par `/` et parcourt chaque composant en cherchant l'inode correspondant via `rechInode`.

Recherche d'un inode dans un répertoire

La fonction `rechInode()` permet d'identifier l'inode associé à un fichier ou répertoire à partir de son nom. Elle parcourt les entrées d'un répertoire jusqu'à trouver une correspondance.

```
int rechInode(const char *filename, Directory dir) {
    int inode = -1;
    for (int i = 0; i < NUM_DIRECTORY_ENTRIES && inode == -1; i++) {
        if (strcmp(filename, dir.entries[i].filename) == 0) {
            inode = dir.entries[i].inode_index;
        }
    }
    return inode;
}
```

3.9 Liens durs et symboliques

— `create_hard_link(...)` : Crée une nouvelle entrée pointant vers le **même** inode, incrémente `link_count`.
— `create_symbolic_link(...)` : Alloue un inode de type 2, stocke la chaîne cible dans un bloc, et ajoute l'entrée au répertoire parent.

3.10 Ouverture, lecture et écritures des fichiers

— `open_file(...)` : Associe un fichier (inode) à un descripteur (index dans `opened_file`).
— `read_file(...)` / `write_file(...)` : Se positionnent dans `filesystem.img` via `fseek`, lisent/écrivent, et mettent à jour la `tete_lecture`.
— `close_file(...)` : Libère le descripteur de fichier.

3.11 Shell interactif

Le système est conçu pour être manipulé via un **mini-shell interactif** (à implémenter ou déjà partiellement fait), qui permet d'exécuter dynamiquement des commandes similaires à un shell Unix :
— `ls` : liste les fichiers du répertoire courant.
— `cd` : change le répertoire courant.
— `mkdir`, `touch` : crée un répertoire ou fichier.
— `rm`, `rmdir` : supprime un fichier ou un répertoire.
— `cp`, `mv` : copie ou déplace des fichiers/répertoires.

Le shell s'appuie sur une boucle `while` qui :

1. Lit une commande tapée par l'utilisateur.
2. Analyse les arguments.
3. Appelle les fonctions C correspondantes.
4. Sauvegarde l'état du système après chaque opération via `save_filesystem()`.

4 Choix techniques et motivations

Cette section présente les principaux choix d'implémentation effectués dans le cadre du projet, en expliquant les raisons pédagogiques ou techniques qui les motivent.

4.1 Structures statiques

— **Choix :** Utilisation de tableaux statiques pour les inodes, les répertoires, les blocs libres, etc.
— **Motivation :** Simplicité de manipulation en C et sérialisation directe dans le fichier binaire.
— **Limite :** Taille fixe → nombre maximum de fichiers, répertoires et blocs limité.

4.2 Bloc de taille fixe

— **Choix :** Blocs de taille constante (512 octets).
— **Motivation :** Reproduit les systèmes réels (disques durs, FAT, etc.).
— **Limite :** Fragmentation potentielle, non gérée actuellement.

4.3 Sérialisation directe

— **Choix :** Sauvegarde complète de la structure `Filesystem` via `fwrite()`.
— **Motivation :** Implémentation rapide et facile à comprendre.
— **Limite :** Peu portable (dépend de l'architecture mémoire), pas de journaling ni de tolérance aux pannes.

4.4 Gestion simplifiée des permissions

— **Choix :** Permissions sous forme de chaîne de 3 caractères `"rwx"`.
— **Motivation :** Faciliter la vérification des droits sans entrer dans la gestion des utilisateurs/groupes.
— **Limite :** Pas de notion d'utilisateur ni de sécurité multi-usagers.

4.5 Liens durs et symboliques

— **Choix :** Implémentation de deux types de liens :
  — Liens durs : nouvelle entrée pointant vers le même inode.
  — Liens symboliques : inode de type 2 contenant une chaîne de redirection.
— **Motivation :** Reproduire des mécanismes fondamentaux de UNIX.
— **Limite :** Pas de gestion des liens circulaires ou cassés.

4.6 Verrouillage global (optionnel)

— **Choix :** Utilisation de fonctions `lock_filesystem()` et `unlock_filesystem()` pour éviter les conflits d'accès.
— **Motivation :** Simulation simple d'un verrou exclusif.
— **Limite :** Pas de gestion multi-thread ou concurrente réelle.

4.7 Approche récursive pour les opérations sur répertoires

— **Choix :** Utilisation de la récursivité pour la suppression et la copie de répertoires.
— **Motivation :** Clarté du code et gestion intuitive des structures arborescentes.
— **Limite :** Risque de dépassement de pile si très grande profondeur (peu probable dans notre cas).

5 Conclusion

Ce mini-projet nous a permis de concevoir et d'implémenter un système de fichiers simulé en langage C, en reproduisant les grandes fonctionnalités des systèmes UNIX : gestion des inodes, allocation de blocs, hiérarchie de répertoires, permissions, liens symboliques et durs, ainsi qu'un shell interactif pour manipuler l'ensemble de manière intuitive.

Le projet nous a amenés à manipuler directement des structures mémoire, à implémenter des algorithmes de navigation, de copie et de suppression récursives, et à réfléchir à la sérialisation de données dans un fichier binaire simulant une partition virtuelle (`filesystem.img`). Ces notions sont au cœur de nombreux systèmes informatiques et constituent une excellente introduction aux systèmes d'exploitation et à la gestion bas-niveau.

Bien que fonctionnel, notre système présente certaines limitations :
— nombre fixe de fichiers, répertoires et blocs (structures statiques) ;
— permissions simplifiées sans gestion multi-utilisateur.

Perspectives d'amélioration :
— ajouter une gestion dynamique des structures (listes chaînées, malloc) ;
— implémenter une table d'allocation plus flexible et un mécanisme de défragmentation ;
— renforcer la compatibilité multi-utilisateur avec gestion des UID/GID ;
— intégrer un système de logs ou de journaling pour améliorer la tolérance aux erreurs ;
— enrichir le shell avec des commandes supplémentaires et une interface plus conviviale.

Ce projet constitue une base solide pour approfondir les concepts liés aux systèmes de fichiers et à la programmation système.