PROJET DE SYSTÈMES D'EXPLOITATION

Mini Système de Fichiers UNIX

Réalisé par :
Boukais Ines | Farez Samah Ikram | Gouarab Ali

Groupe A

Année universitaire : 2024/2025

Sommaire

1. Introduction

Ce projet consiste en la création d'un gestionnaire de fichiers simulant un mini-système de fichiers inspiré d'UNIX. Le système repose sur un fichier unique ("partition") de type UNIX, stocké sur le disque. Il permet d'effectuer les opérations de base telles que la création, la suppression, la copie et le déplacement de fichiers et de répertoires. De plus, une gestion des droits d'accès et la gestion des liens sont intégrées. Un aspect clé du projet concerne l'optimisation de la gestion de l'espace libre.

2. Objectifs

Les objectifs principaux du projet sont :

- Implémenter un système de fichiers fonctionnel basé sur un fichier UNIX.
- Permettre la gestion des fichiers et répertoires avec les opérations courantes.
- Assurer une gestion des droits d'accès et des liens.
- Optimiser l'utilisation de l'espace libre après suppression de fichiers.

3. Choix des structures de données

3.1 File

```c
typedef struct
{
    char filename[MAX_FILENAME];
    int size; // Will be calculated automatically
    PageTableEntry *page_table;
    int page_table_size;
    char owner[20];
    int permissions;
    time_t creation_time;
    time_t modification_time;
    int content_size;
    char *content;
    int file_position;
    int is_open;
    int open_count;
    int is_symlink;     // 1 if this is a symbolic link
    char *link_target; // Target path for symlinks
    int ref_count;      // For hard link reference counting
    ino_t inode;        // Unique inode number
} File;
```


Cette structure représente un fichier avec ses métadonnées : nom, taille, droits d'accès, propriétaire, contenu, etc. Elle contient une table de pages simulant la mémoire physique pour la gestion fine de l'espace mémoire.

3.2 Directory

```c
typedef struct
{
    char dirname[MAX_FILENAME];
    File files[MAX_FILES];
    int file_count;
    int parent_directory;
    time_t creation_time;
    ino_t inode; // Add this for directories
} Directory;
```

Chaque répertoire stocke le nom du répertoire, la date de création, contient un tableau de fichiers et un lien vers son parent. Cela permet de modéliser une hiérarchie de répertoires.

3.3 PageTableEntry

```c
typedef struct
{
    int physical_page; // Physical page number
    int is_allocated;  // Allocation status
} PageTableEntry;
```

Sert à gérer l'allocation mémoire des fichiers en simulant un système de pagination mémoire.
Permet de simuler une mémoire virtuelle fragmentée, où chaque fichier peut occuper plusieurs pages physiques.

3.4 User

```c
typedef struct
{
    char username[20];
    char password[20];
} User;
```

Pour la gestion des droits, chaque utilisateur est identifié par un nom et un mot de passe. Cela permet de restreindre certaines opérations selon les permissions définies.

3.5 FileSystemState

```c
typedef struct
{
    User users[MAX_USERS];
    Directory directories[MAX_DIRECTORIES];
    int current_directory;
} FileSystemState;
```

Cette structure représente la structure centrale du système de fichiers. Elle regroupe tous les utilisateurs, les répertoires, ainsi que l'état courant (répertoire actuel) du système.

3.1 Job

```c
typedef struct
{
    char *command;
} Job;
```

Utilisée pour représenter une commande à exécuter et la stocker dans la file d'attente, facilitant ainsi la gestion et l'exécution différée des commandes saisies par l'utilisateur.

4. Choix des algorithmes

4.1 Gestion de l’espace libre

L’espace de stockage est simulé par un fichier UNIX, découpé en pages. La gestion des blocs libres repose sur un bitmap, où chaque bit représente l’état d’un bloc :

- 0 pour un bloc libre
- 1 pour un bloc occupé

Algorithme d’allocation de blocs (First Fit) :

1. Parcourir le bitmap séquentiellement pour trouver le premier bloc libre.
2. Marquer ce bloc comme occupé.
3. Mettre à jour la table de pages du fichier concerné.

L'algorithme First Fit apparaît dans la fonction create_file (ainsi que dans allocate_pages). Pour chaque page nécessaire au fichier, le code parcourt linéairement le bitmap (page_bitmap) à la recherche du premier bloc libre. Dès qu'une page libre est trouvée (le bit correspondant est à 0), elle est marquée comme occupée et assignée à la page du fichier.

Algorithme de libération de blocs :

1. Identifier les blocs occupés par un fichier supprimé.
2. Marquer ces blocs comme libres dans le bitmap.

4.2 Gestion des fichiers et répertoires

Chaque répertoire contient une **table des fichiers** sous forme de tableau statique. La gestion des fichiers repose sur des algorithmes simples :

Création d’un fichier :

1. Vérifier si un fichier du même nom existe dans le répertoire.
2. Vérifier si de l’espace est disponible.
3. Créer une structure File et l’ajouter à la table du répertoire.

*Dans **create_file**, le système vérifie l'existence d'un fichier du même nom dans le répertoire courant, alloue les pages nécessaires via le bitmap et initialise les métadonnées du fichier.*

Suppression d’un fichier :

1. Vérifier que le fichier existe.
2. Libérer les blocs occupés via l’algorithme de libération.
3. Supprimer l’entrée du fichier dans la table et décaler les éléments restants pour combler l’espace vide.

*La fonction **delete_file** libère les pages occupées par le fichier dans le bitmap, supprime le fichier du tableau du répertoire et décale les entrées restantes pour combler l'espace.*

Déplacement d'un fichier :

1. Vérifier l'existence du fichier source.
2. Vérifier que le répertoire de destination a de la place.
3. Copier les métadonnées dans le répertoire cible et supprimer l'entrée dans le répertoire source.

*Les fonctions **copy_file_to_dir** et **move_file_to_dir** implémentent les algorithmes pour copier ou déplacer un fichier d'un répertoire à un autre en vérifiant l'existence du fichier source et la capacité du répertoire cible.*

4.3 Gestion des droits d'accès

Les fichiers et répertoires possèdent des permissions sous forme d'un **octet de bits** (similaire aux permissions UNIX).

Vérification des droits d'accès :

1. Récupérer les permissions du fichier.
2. Comparer avec les droits de l'utilisateur.
3. Autoriser ou refuser l'opération demandée.

Dans write_to_file, avant d'écrire dans un fichier, le code vérifie si les permissions du fichier autorisent l'écriture (vérification avec un masque bitwise sur 0222).

4.4 Gestion des liens

L'implémentation des liens (durs et symboliques) repose sur une duplication ou une référence partagée des métadonnées du fichier source.

Lien dur :

La création d'un lien dur se fait par une copie de la structure File, en gardant la même table de pages et le même contenu. Ainsi, les deux entrées pointent vers les mêmes blocs mémoire.

La fonction create_hard_link recherche le fichier source dans le répertoire actuel, copie ses métadonnées et ajoute une nouvelle entrée avec le nom du lien dans le répertoire.

Cas de suppression du fichier original

Un lien dur partage les mêmes blocs mémoire que le fichier original. Ainsi, si le fichier d'origine est supprimé, le contenu du fichier reste accessible tant qu'au moins un lien existe. Ce n'est que lorsque le dernier lien est supprimé que l'espace mémoire est libéré.

Lien symbolique :

Le lien symbolique est réalisé en stockant simplement le chemin vers le fichier cible. La résolution du lien se fait de manière récursive lors de l'ouverture ou de la lecture du fichier.

La fonction create_symbolic_link vérifie si le fichier cible existe, crée un nouveau fichier spécial contenant le chemin cible comme contenu et l'ajoute dans le répertoire courant.

Cas de suppression du fichier original

Contrairement au lien dur, un lien symbolique ne contient pas directement les données du fichier. Si le fichier original est supprimé, le lien symbolique devient alors danglant, ce qui signifie qu'il pointe vers une cible inexistante (NULL).

4.5 Gestion des commandes et de la concurrence

Pour assurer l'exécution différée et concurrente des commandes saisies par l'utilisateur, nous avons implémenté un ordonnanceur basé sur une file d'attente circulaire.

File d'attente :

Les commandes sont stockées dans un tableau circulaire. Chaque commande est traitée dans l'ordre d'arrivée, garantissant ainsi une exécution séquentielle au sein d'un contexte multi-threadé.

Synchronisation :

L'accès à la file d'attente et aux ressources partagées est protégé par des mutex et des variables conditionnelles. Cela permet d'éviter les problèmes d'accès concurrent et de garantir l'intégrité des données.

4. Makefile et Gestion de la Compilation

Un Makefile **est fourni pour** une compilation efficace du système de fichiers en respectant les bonnes pratiques de développement

Fonctionnalités principales

Compilation sécurisée

- -Wall et -Wextra activent les avertissements pour détecter les problèmes potentiels
- -pthread permet la gestion des threads

Cibles disponibles

- make ou make all : compilation complète du projet
- make clean : suppression des fichiers générés

Structure modulaire

- Séparation claire entre sources (src/) et headers (include/)
- Gestion automatique des dépendances

5. Extensions implémentées

5.1. Gestion des Accès Concurrents

Thread Scheduler

- Un thread dédié (scheduler_thread) traite les commandes en FIFO depuis une file d'attente circulaire (job_queue)
- Capacité de 10 commandes maximum (MAX_JOBS)
- Mécanisme de signalisation via pthread_cond_t lorsque des jobs sont disponibles

Protection des Accès Critiques

- Mutex global (pthread_mutex_t mutex) pour toutes les opérations sur le système de fichiers
- Verrouillage explicite dans chaque fonction manipulant l'état du FS
- Double verrouillage pour la file d'attente (queue_lock) et les données du FS

Flux d'Exécution

1. add_job() : Ajoute une commande à la file (avec vérification de capacité)
2. scheduler() : Thread worker qui dépile et exécute les jobs
3. execute_job() : Traite la commande avec verrouillage approprié

5.2 Sauvegarde / restauration

Sauvegarde (Backup)

- Copie binaire du fichier filesystem.dat vers <nom>.bak
- Utilisation de buffers de 4KB pour une copie efficace
- Vérification des erreurs d'I/O

Restauration

- Remplacement de filesystem.dat par le fichier backup
- Rechargement automatique du nouvel état
- Confirmation utilisateur avant écrasement

5.3 Formatage du système

Fonctionnement

- Réinitialisation complète du système de fichiers
- Recréation des structures de base : les répertoires root, home et les utilisateurs par défaut
- Réinitialisation du bitmap de pages

Sécurité

- Confirmation explicite de l'utilisateur
- Verrouillage complet pendant l'opération

Conclusion

Ce projet a permis de développer un système de fichiers minimaliste inspiré d'UNIX, combinant fonctionnalités de base et extensions avancées. L'implémentation en C a mis en œuvre :

- **Fonctions essentielles** :
  - Gestion complète des fichiers/répertoires
  - Système de permissions
  - Liens symboliques et physiques
- **Innovations techniques** :
  - Mécanisme de synchronisation basé sur une file d’attente FIFO
  - Système de pagination mémoire
  - Sauvegarde/restauration atomique

Ce projet valide notre compréhension des principes fondamentaux des systèmes d'exploitation tout en fournissant une base solide pour des développements ultérieurs.