Guide du Développeur

Implémentation d'une bibliothèque de fonctions de gestion de fichiers simulant celle d'UNIX

Projet Systèmes d'Exploitation
L3 Informatique

rédigé par

Yige YANG
Saran Kaba KOUYATÉ
Shalini KIRITHARAN

Mars 2024

Introduction

Ce guide du développeur offre un aperçu détaillé de l'architecture, des structures de données et des algorithmes que nous avons implémentés à notre bibliothèque de gestion de fichiers UNIX. Nous avons conçu cette bibliothèque pour fournir une interface conviviale aux utilisateurs tout en offrant une extensibilité et une robustesse aux développeurs. Nous expliquerons comment chaque composant de la bibliothèque interagit pour gérer les fichiers, en examinant les algorithmes implémentés.

Architecture Générale

La bibliothèque est conçue pour simuler les fonctionnalités de gestion de fichiers UNIX à l'intérieur d'un environnement isolé. Voici les composants principaux de l'architecture :

Fichier Source Principaux

main.c : Contient la fonction principale main() qui gère l'interface utilisateur et les interactions avec la bibliothèque.

test.c : Implémente les fonctions de la bibliothèque pour la gestion des fichiers ainsi que les commandes shell simulées.

Fichier d'En-tête

test.h : Déclare les structures de données et les prototypes de fonctions utilisées dans la bibliothèque et le programme principal. Il est inclus dans les fichiers qui utilisent ces fonctions de test afin que le compilateur sache qu'elles existent et qu'il puisse les utiliser correctement.

Makefile

Makefile : Fournit les règles de compilation pour compiler les fichiers source et générer l'exécutable.

1) Description du fichier test.h

C’est un fichier d’en-tête qui contient des déclarations de fonctions de la gestion d’un fichier simulant celle de l’UNIX.

Les constantes :

PARTITION_SIZE : Définit la taille de la partition simulée en octets. Dans notre cas du projet, la taille est de 1 Mo (1024 * 1024 octets).

MAX_LENGTH : Définit la longueur maximale des chaînes de caractères utilisées dans le programme. Nous avons défini la longueur maximale à 1024 caractères.

MAX_PARAMS : Définit le nombre maximum de paramètres pris en charge pour les commandes shell. Dans notre exemple, le nombre maximum de paramètres est de 20.

BLOCK_SIZE : Définit la taille de chaque bloc sur le disque en octets. Nous avons défini la taille de chaque bloc à 512 octets.

TOTAL_BLOCKS : Calcule le nombre total de blocs sur le disque en fonction de la taille de la partition et de la taille de chaque bloc.

Structures de Données

- Structure représentant l'état de chaque bloc sur le disque:

```
typedef struct {
   char block_usage[TOTAL_BLOCKS]; // '0' for free, '1' for used
} PartitionStatus;
```

Pour représenter l'état de chaque bloc sur le disque, nous avons défini une structure appelée *PartionStatus*. Cette structure contient un tableau de caractère *block_usage* qui stocke l'état de chaque bloc. Chaque élément du tableau est un caractère, où '0' représente un bloc libre et '1' représente un bloc utilisé.

La définition de cette structure est nécessaire pour suivre l'utilisation de l'espace disque et pour visualiser l'état de la partition.

Structure représentant les fichiers dans notre système simulé

La bibliothèque utilise une structure de données principale pour représenter un fichier ouvert dans le système. Voici la structure de données utilisée :

```
typedef struct {
char* name;
int size;
int current_position;
char* data;
int block_start;
int blocks_count;
} file;
```

name: Un pointeur vers une chaîne de caractères qui représente le nom du fichier qui permet d'identifier le fichier dans le système de fichiers.

size : C'est la taille du fichier en octets qui indique la quantité de données actuellement stockées dans le fichier.

current_position : La position actuelle du pointeur de lecture/écriture dans le fichier. Il indique l'endroit où se trouve le prochain octet à lire ou à écrire.

data : Un pointeur vers une zone mémoire qui contient les données réelles du fichier.

block_start : L'index du bloc de départ du fichier sur le disque simulé. Il indique où commence physiquement le fichier sur le disque.

blocks_count : Le nombre de blocs utilisés par le fichier sur le disque simulé. Il indique combien de blocs sont nécessaires pour stocker toutes les données du fichier.

Nous avons défini cette structure ‘file’ représentant un fichier dans le système de fichier. Elle contient les informations nécessaires pour définir un fichier telles que le nom, la taille, la position actuelle du curseur, les données, l’indice du premier bloc de données ainsi que le nombre de blocs utilisés par le fichier.

3) Les prototypes des fonctions utilisés pour la gestion du systèmes de fichier :

PartitionStatus* initializePartitionStatus() *:* C’est une fonction qui crée une nouvelle instance de la structure *PartitionStatus* et initialise l’état des blocs du disque. Elle fait une allocation mémoire pour le tableau *‘block_usage’* et initialise chaque élément à zéro. Cela permet d’avoir tous les blocs libres.

void visualizePartitionStatus(PartitionStatus* status) *:* Cette fonction prend en paramètre un pointeur vers la structure *PartitionStatus*. Elle affiche l’état de chaque bloc dans une format pour la lecture de l’utilisateur en parcourant le tableau *‘bloc_usage’*.

int allocateBlocks(file* f, int size) : Un fonction qui prend en paramètre un pointeur vers une structure ‘file’ et la taille en octets du fichier. C’est une fonction qui permet d’allouer sur le disque les blocs nécessaires pour faire le stockage des données du fichier ‘file’ avec la taille donnée en entrée. Elle utilise l’état des blocs du disque pour trouver des blocs libres et les marquer comme utilisés après stockage des données du fichier.

void freeBlocks(file* f) : Cette fonction prend en paramètre un pointeur vers une structure ‘file’, un fichier que l’on veut libérer. Elle libère donc les blocs utilisés par le fichier en marquant les blocs comme libre dans l'état des blocs du disque.

int getFileSize(const file* f) : C’est une fonction qui prend en entrée un pointeur vers une structure ‘file’ qui représente le fichier dont on veut sa taille. Elle retourne donc la taille du fichier donnée en paramètre en octets.

4) Les prototypes des fonctions des fonctionnalités du système de fichier :

int myFormat(char* partitionName) : Cette fonction est chargée de formater une partition, et de créer un fichier portant le nom partitionName que l’on donne en paramètre et de lui attribuer une taille ou un contenu spécifique. Elle retourne un entier qui indique si l'opération de formatage a réussi ou non.

file* myOpen(char* fileName) : Cette fonction est utilisée pour ouvrir un fichier dont le nom est spécifié en paramètre par fileName. Elle retourne un pointeur vers une structure file qui représente le fichier ouvert.

int myWrite(file* f, void* buffer, int nBytes) : Cette fonction permet d'écrire des données contenues dans buffer dans le fichier représenté par f. Elle écrit nBytes octets de données à partir du buffer. Elle retourne un entier qui indique le nombre d'octets réellement écrits dans le fichier.

int myRead(file* f, void* buffer, int nBytes) : Cette fonction est utilisée pour lire des données à partir du fichier représenté par f et les stocker dans buffer. Elle lit jusqu'à nBytes octets de données à partir du fichier. Elle retourne un entier qui indique le nombre d'octets réellement lus à partir du fichier.

void mySeek(file* f, int offset, int base) : Cette fonction permet de déplacer le curseur de lecture/écriture dans le fichier représenté par f selon les paramètres spécifiés. offset indique le nombre décalage d'octets par rapport à la position de départ spécifiée par base.

int myMove(const char* sourceName, const char* destName): une fonction appelée myMove qui prend deux paramètres de type pointeur vers const char. Ces paramètres sont sourceName, qui représente le chemin du fichier source à déplacer, et destName, qui représente le chemin de destination où le fichier doit être déplacé. Cette fonction est utilisée pour déplacer un fichier de l'emplacement spécifié par sourceName vers l'emplacement spécifié par destName. Elle retourne un entier qui indique le résultat de l'opération : 0 en cas de succès et -1 en cas d'échec.

void myDelete(file* f) : une fonction appelée myDelete qui prend un paramètre de type pointeur vers file. Ce paramètre f représente le fichier à supprimer. Cette fonction est utilisée pour supprimer un fichier du système de fichiers. Elle libère également les ressources associées à la structure de fichier passée en argument.

int myCopy(const char* sourceName, const char* destName) : une fonction appelée myCopy qui prend deux paramètres de type pointeur vers const char. Ces paramètres sont sourceName, qui représente le chemin du fichier source à copier, et destName, qui représente le chemin de destination où le fichier doit être copié. Cette fonction est utilisée pour copier le contenu du fichier spécifié par sourceName vers un nouveau fichier à l'emplacement spécifié par destName.

int myRename(const char* oldName, const char* newName): La fonction prend deux paramètres de type pointeur vers const char. Le premier paramètre, oldName, représente le nom actuel du fichier ou du répertoire à renommer. Le deuxième paramètre, newName, représente le nouveau nom que le fichier ou le répertoire doit avoir après le renommage. Cette fonction est utilisée pour renommer un fichier ou un répertoire du nom spécifié par oldName au nouveau nom spécifié par newName. Elle retourne un entier qui indique le résultat de l'opération : 0 en cas de succès et -1 en cas d'échec.

Les prototypes de fonctions de simulation de commandes de shell:

void myEcho(char **params): est une fonction qui implémente la commande "echo" du shell. Elle affiche simplement sur la sortie standard les paramètres qui lui sont passés. Les paramètres sont stockés dans un tableau de chaînes de caractères.

int myCd(char **params): est une fonction qui implémente la commande "cd" du shell. Elle est utilisée pour changer de répertoire de travail. Elle prend en paramètre le chemin du répertoire vers lequel l'utilisateur souhaite se déplacer. Si le changement de répertoire est réussi, la fonction retourne 0 ; sinon, elle retourne -1.

int myExit(char **params):Cette fonction implémente la commande "exit" du shell. Elle est utilisée pour quitter l'application. Cette fonction prend généralement en charge les paramètres optionnels qui peuvent être fournis avec la commande "exit", mais dans votre cas, elle ne semble pas utiliser les paramètres. Elle affiche simplement un message indiquant que le programme se termine et appelle la fonction exit(0) pour terminer l'exécution du programme.

2) Description du fichier test.c :

Implémentation de la fonction d'initialisation de l'état de Partition

La fonction *initializePartitionStatus* initialise l'état de la partition en marquant tous les blocs comme

libres. Elle ne prend aucun paramètre et retourne un pointeur vers la structure *PartitionStatus* représentant l'état de la partition.

→ L'explication détaillée de son implémentation :

La fonction memset est utilisée pour remplir les premiers octets de la mémoire avec la valeur '0', qui représente un bloc libre. Cela garantit que tous les blocs de la partition sont initialisés comme libres.La fonction retourne un pointeur vers la structure représentant l'état de la partition. Ce pointeur pointe vers la variable globale qui contient l'état initialisé de la partition.

Cette fonction est finalement responsable de préparer l'état initial de la partition en marquant tous les blocs comme libres. Cela permet d'assurer que lors de la création d'une nouvelle partition, tous les blocs sont disponibles pour être utilisés.

Implémentation de la fonction pour rechercher les blocs libres

La fonction ***findFreeBlocks*** recherche des blocs libres consécutifs dans la partition pour allouer de l'espace à un fichier.

→ Explication détaillée de son implémentation :

La fonction commence par initialiser un compteur à zéro. Ce compteur est utilisé pour suivre le nombre de blocs libres consécutifs trouvés.

La fonction parcourt tous les blocs de la partition en utilisant une boucle for pour i allant de 1 au nombre total de blocs. À chaque itération, elle vérifie si le bloc est libre:

- si le bloc à l'index i est libre, elle incrémente le compteur pour indiquer la présence d'un bloc libre consécutif.

- Si un bloc n'est pas libre, la fonction réinitialise le compteur à zéro, car la séquence de blocs libres consécutifs est interrompue.

Ainsi lorsque le compteur atteint le nombre de blocs nécessaires, cela signifie que suffisamment de blocs libres consécutifs ont été trouvés pour répondre aux besoins de l'allocation. Dans ce cas, la fonction retourne l'indice du premier bloc libre trouvé, ce qui correspond à l'indice du début de la séquence de blocs libres consécutifs.

Si la fonction parcourt toute la partition sans trouver suffisamment de blocs libres consécutifs, elle retourne -1 pour indiquer qu'il n'y a pas assez de blocs libres pour l'allocation demandée.

En résumé, cette fonction est chargée de trouver et de renvoyer l'indice du premier bloc libre dans la partition où une séquence de blocs libres consécutifs d'une taille spécifiée peut être allouée.

Implémentation de la fonction d'allocation des blocs dans la partition:

La fonction **allocateBlocks** permet d'allouer des blocs dans la partition pour stocker un fichier.

→ Explication détaillée :

La fonction commence par calculer le nombre de blocs nécessaires pour stocker le fichier en divisant la taille du fichier par la taille de chaque bloc et en arrondissant le résultat à l'entier supérieur. Cela garantit que suffisamment de blocs sont alloués pour stocker toutes les données du fichier, en tenant compte de la taille des blocs.

En utilisant la fonction ***findFreeBlocks*** que nous avons défini auparavant, elle recherche dans la partition un emplacement pour allouer une séquence de blocs libres consécutifs de la taille requise. Si cette recherche réussit, elle obtient l'indice du premier bloc libre trouvé, qui servira de point de départ pour l'allocation des blocs.

Si la recherche ne parvient pas à trouver suffisamment d'espace libre dans la partition, la fonction retourne -1 pour indiquer qu'il n'y a pas assez d'espace pour allouer les blocs nécessaires au fichier.

Une fois les blocs libres trouvés, la fonction les marque comme utilisés en modifiant les valeurs correspondantes dans le tableau de blocs de la structure qui contient l ‘état de la partition. Elle parcourt la séquence de blocs libres et met à jour chaque entrée du tableau en lui attribuant la valeur '1'.

Elle enregistre ensuite l'indice du premier bloc alloué et le nombre total de blocs alloués dans la structure du fichier f. Cela permettra de suivre les blocs alloués pour ce fichier.

Enfin, la fonction retourne le nombre de blocs alloués, pour montrer le succès de l'allocation.

En résumé, la fonction ***allocateBlocks*** trouve et alloue des blocs libres dans la partition pour stocker un fichier, tout en maintenant un suivi de ces blocs dans la structure du fichier. Si l'espace disponible dans la partition est insuffisant, elle signale l'échec de l'allocation en retournant -1.

Implémentation de la fonction de la libération des blocs alloués à un fichier:

La fonction ***freeBlocks*** est chargée de libérer les blocs qui ont été précédemment alloués à un fichier.

→ Explication détaillée de la fonction

La fonction commence par parcourir tous les blocs qui ont été alloués au fichier, en commençant par le début dans la partition et en s'arrêtant à la fin. Pour chaque bloc alloué au fichier, la fonction met à jour la valeur correspondante dans le tableau la structure PartitionStatus qui stocke l’état de chaque bloc. Elle modifie la valeur de l'élément du tableau à la valeur 0 pour le marquer comme étant libre.

En effectuant cette opération pour chaque bloc alloué au fichier, la fonction libère efficacement tous les blocs associés à ce fichier en les marquant comme libres dans la partition.

En résumé, la fonction freeBlocks s'assure de libérer tous les blocs qui ont été précédemment alloués à un fichier, en mettant à jour le tableau block_usage de la structure PartitionStatus pour indiquer que ces blocs sont à nouveau disponibles pour une allocation ultérieure.

Implémentation de la fonction pour l’ouverture d’un fichier :

La fonction myOpen est utilisée pour ouvrir un fichier spécifié par son nom et initialiser une structure file associée à ce fichier.

→ Explication détaillée de son implémentation :

La fonction commence par allouer dynamiquement de la mémoire pour une structure d'un fichier. Si l'allocation échoue, elle affiche un message d'erreur à l'aide de perror et retourne NULL pour signaler une erreur.

Elle initialise ensuite les champs de la structure file tels que son nom, sa taille, le début du fichier et le contenu du fichier. Puis elle essaye d'ouvrir le fichier spécifié en mode lecture binaire ("rb"). Si l'ouverture échoue, cela signifie que le fichier n'existe pas. Dans ce cas, la fonction demande à l'utilisateur s'il souhaite créer le fichier. Si la réponse est positive, elle tente de créer le fichier en mode lecture/écriture binaire ("wb+"). Si la création échoue, la fonction affiche un message d'erreur avec perror, libère la mémoire allouée pour le nom du fichier et la structure file, puis retourne NULL pour signaler une erreur.

Si l'utilisateur ne souhaite pas créer le fichier ou si la réponse n'est pas valide, la fonction libère également la mémoire allouée pour le nom du fichier et la structure file, puis retourne NULL.

Si le fichier est ouvert avec succès, la fonction le referme immédiatement pour récupérer sa taille. La fonction utilise ensuite fseek pour déplacer le pointeur de fichier à la fin du fichier, puis ftell pour obtenir la taille totale du fichier en octets. Ensuite, elle utilise rewind pour ramener le pointeur de fichier au début du fichier.

L'étape suivante est que la fonction alloue dynamiquement de la mémoire pour stocker le contenu du fichier. Une fois que le contenu du fichier est chargé en mémoire, la fonction ferme le fichier en utilisant fclose(fp) pour libérer les ressources associées au fichier.

Si toutes les étapes précédentes se déroulent sans erreur, la fonction retourne un pointeur vers la structure file initialisée, qui contient des informations sur le fichier ouvert.

En résumé, la fonction myOpen gère l'ouverture d'un fichier, la création du fichier si nécessaire, la récupération de sa taille et le chargement de son contenu en mémoire. Elle prend en charge les cas où l'allocation de mémoire échoue, où l'ouverture ou la création du fichier échoue, et où la lecture du contenu du fichier échoue. Elle retourne une structure file qui contient des informations sur le fichier ouvert, ou NULL en cas d'erreur.

Implémentation de la fonction pour l'écriture dans un fichier :

La fonction myRead est utilisée pour lire des données à partir d'un fichier spécifié par un pointeur vers la structure file et les stocker dans un tampon de données (buffer).

→ Explication détaillée de son implémentation :

La fonction utilise fseek pour déplacer le curseur de fichier à la position actuelle spécifiée par le champ current_position de la structure file. Si le déplacement échoue, elle affiche un message d'erreur à l'aide de perror indiquant l'échec du déplacement du curseur de fichier et ferme le fichier en utilisant fclose(fp) avant de retourner -1 pour signaler une erreur.

La fonction utilise fread pour lire jusqu'à nBytes octets de données à partir du fichier dans le tampon de données (buffer). Le nombre d'octets lus est stocké dans une variable.

Si le nombre d'octets lus est inférieur ou égal à zéro, cela peut indiquer soit la fin du fichier (feof(fp)), soit une erreur lors de la lecture. Dans ce cas :

Si la fin du fichier est atteinte, la fonction affiche un message indiquant que la fin du fichier a été atteinte Sinon, elle affiche un message d'erreur à l'aide de perror indiquant l'échec de la lecture à partir du fichier.

Implémentation de la fonction de décalage du curseur lecture/écriture dans un fichier

La fonction mySeek est utilisée pour déplacer le curseur de lecture/écriture dans un fichier spécifié par un pointeur vers la structure file.

→ Explication détaillée de son implémentation :

La fonction commence par vérifier si le pointeur vers la structure file (f) est nul. Si c'est le cas, elle affiche un message d'erreur à l'aide de perror, indiquant une structure de fichier invalide, puis elle quitte la fonction. En fonction de la valeur de l'argument base, la fonction calcule la nouvelle position de recherche dans le fichier. Une fois que la nouvelle position a été déterminée et validée, la fonction met à jour le champ current_position de la structure file avec la nouvelle position calculée.

Implémentation de la fonction d'affichage de message:

La fonction myEcho est conçue pour afficher un message sur la console, en prenant comme argument un tableau de chaînes de caractères (params).

→ Explication détaillée de son implémentation :

La fonction commence par vérifier si le premier élément du tableau params (index 0) est non nul, c'est-à-dire s'il contient une chaîne de caractères. Si params[0] n'est pas nul, cela signifie qu'un message a été fourni en argument. La fonction passe alors à l'étape suivante. Sinon, si params[0] est nul, cela indique qu'aucun message n'a été fourni en argument, et la fonction affiche un message par défaut indiquant qu'aucun message n'a été fourni. Si un message est présent dans params[0], la fonction utilise printf pour afficher ce message suivi d'un saut de ligne (\n). Si aucun message n'a été fourni et que la fonction affiche le message par défaut, elle se contente d'afficher "No message provided." suivi d'un saut de ligne.

Implémentation de la fonction changement de répertoire

La fonction myCd est chargée de changer le répertoire de travail courant du processus.

→ Explication détaillée de son implémentation :

La fonction commence par vérifier si le deuxième élément du tableau params (index 1) est nul. Si params[1] est nul, cela signifie qu'aucun chemin de répertoire n'a été fourni en argument. Dans ce cas,

la fonction affiche un message d'erreur approprié sur la sortie d'erreur standard à l'aide de fprintf et retourne -1 pour indiquer une erreur. Si un chemin de répertoire est fourni dans params[1], la fonction tente de changer le répertoire de travail courant en utilisant la fonction système chdir. Si l'appel à chdir réussit (retourne 0), cela signifie que le répertoire a été changé avec succès, et la fonction retourne 0 pour indiquer une opération réussie. Si l'appel à chdir échoue, la fonction affiche un message d'erreur approprié à l'aide de perror pour indiquer la cause de l'échec, puis retourne -1 pour indiquer une erreur.

Implémentation de la fonction suppression d'un fichier :

La fonction myDelete est responsable de supprimer un fichier du système de fichiers.

→ Explication détaillée de son implémentation :

La fonction commence par vérifier si le pointeur de fichier f est nul. Si c'est le cas, elle affiche "Invalid file." sur la sortie standard et retourne immédiatement. Ensuite, elle appelle la fonction freeBlocks(f) pour libérer les blocs du disque occupés par le fichier. Cela garantit que les blocs de données alloués au fichier sont marqués comme libres dans la structure de statut de partition.

Elle utilise la fonction remove(f->name) pour supprimer le fichier du système de fichiers. Si la suppression est réussie (c'est-à-dire que la fonction remove renvoie 0), elle affiche "File deleted successfully." sur la sortie standard. Sinon, elle affiche un message d'erreur approprié à l'aide de perror. Enfin, elle libère la mémoire occupée par les champs name et data de la structure de fichier f, ainsi que la mémoire occupée par la structure f elle-même, en appelant les fonctions free.

Implémentation de la fonction pour la copie d'un fichier:

La fonction myCopy est utilisée pour copier le contenu d'un fichier source dans un fichier de destination.

→ Explication détaillée de son implémentation

La fonction commence par ouvrir le fichier source spécifié par sourceName en mode lecture binaire à l'aide de la fonction fopen. Si l'ouverture échoue (c'est-à-dire que fopen renvoie NULL), elle affiche un message d'erreur approprié à l'aide de perror et retourne -1 pour indiquer une erreur. Ensuite, elle ouvre le fichier de destination spécifié par destName en mode écriture binaire. Si l'ouverture échoue, elle affiche un message d'erreur approprié à l'aide de perror, ferme le fichier source précédemment ouvert avec fclose, puis retourne -1 pour indiquer une erreur.

Elle utilise une boucle while pour lire des blocs de données du fichier source à l'aide de fread. Elle lit jusqu'à 1024 octets (ou moins si le fichier est plus petit) dans le tampon buffer à chaque itération. La taille des données lues est stockée dans bytesRead. Ensuite, elle écrit les données lues dans le fichier de destination à l'aide de fwrite. La fonction fwrite écrit bytesRead octets depuis le tampon buffer dans le fichier de destination.

Ce processus de lecture et d'écriture se poursuit tant que des données peuvent être lues depuis le fichier source. Une fois la copie terminée, la fonction ferme les fichiers source et de destination à l'aide de fclose.

Implémentation de la fonction pour renommer un fichier :
La fonction myRename est utilisée pour renommer un fichier existant.

→ Voici une explication détaillée de son implémentation :

La fonction prend deux arguments : oldName, le nom actuel du fichier, et newName, le nouveau nom que nous voulons attribuer au fichier.

Elle utilise la fonction rename pour renommer le fichier. Si rename renvoie une valeur différente de zéro, cela signifie qu'une erreur s'est produite lors du renommage du fichier. Dans ce cas, la fonction affiche un message d'erreur approprié à l'aide de perror, indiquant l'échec du renommage, et retourne -1 pour signaler une erreur.

Si la fonction rename réussit, elle renomme le fichier spécifié par oldName avec le nouveau nom newName.

Implémentation de la fonction pour déplacer un fichier :
La fonction myMove est utilisée pour déplacer un fichier d'un emplacement source vers une destination.

→ explication détaillée de son implémentation :

La fonction essaie d'utiliser la fonction rename pour renommer le fichier sourceName avec le nom destName. Si le renommage réussit (c'est-à-dire si rename renvoie 0), cela signifie que le fichier a été déplacé avec succès. Dans ce cas, la fonction affiche un message indiquant le succès du déplacement et retourne 0 pour indiquer une opération réussie. Si le renommage direct échoue (c'est-à-dire si rename renvoie une valeur différente de zéro), cela peut être dû à diverses raisons telles que l'existence du fichier de destination, des permissions insuffisantes, etc. Dans ce cas, la fonction affiche un message d'erreur à l'aide de perror pour indiquer que le renommage direct a échoué et qu'elle va tenter de copier le fichier.

La fonction copie ensuite le fichier depuis le fichier source vers le fichier destination en utilisant la fonction myCopy et supprime le fichier source en utilisant la fonction remove.

Implémentation de la fonction sortie du programme :

La fonction myExit est chargée de terminer le programme de manière propre.

→ Explication de son implémentation

La fonction commence par afficher le message "Exiting program." sur la sortie standard à l'aide de printf. Ce message informe l'utilisateur que le programme se termine.

Ensuite, la fonction appelle la fonction system exit(0). Cette fonction termine immédiatement le programme avec un code de sortie de 0, indiquant une terminaison réussie sans erreur.

3) Description du fichier main. c :

Ce fichier contient la fonction principale main() qui gère l'interface utilisateur et les interactions avec la bibliothèque

Les fonctionnalités Principales :

1. Affichage du Menu (`displayMenu`) : affiche les 14 options disponibles pour l'utilisateur, incluant le formatage de partition, l'ouverture, la lecture, l'écriture, et la suppression de fichiers, entre autres. Cela permet une interaction intuitive avec le système de gestion de fichiers.

2. Boucle Principale (`main`): Maintient l'exécution du programme, permettant à l'utilisateur de sélectionner et d'exécuter différentes opérations à partir du menu. Elle gère la logique de sélection des options et appelle les fonctions correspondantes pour chaque action.

Description des Opérations

- **Format Partition**: demande le nom de la partition à formater et exécute la fonction `myFormat` avec le nom fourni. Elle confirme le succès ou l'échec du formatage de la partition.

- **Open File**: permet d'ouvrir un fichier existant ou d'en créer un nouveau et utilise `myOpen` pour accomplir cette tâche.

- **Write to File**: écrit dans un fichier ouvert et demande à l'utilisateur d'entrer le texte à écrire et exécute `myWrite`.

- **Read from File**: lit les données d'un fichier ouvert et affiche le contenu lu à l'utilisateur.

- **Seek in File**: Modifie la position du pointeur dans un fichier ouvert, permettant à l'utilisateur de lire ou écrire à partir d'un certain offset.

- **Get File Size**: Affiche la taille d'un fichier ouvert, en utilisant une fonction hypothétique `getFileSize`.

- **Delete File**: Supprime un fichier ouvert et libère les ressources associées.

- **Copy, Rename, Move File**: Fournissent des fonctionnalités pour copier, renommer et déplacer des fichiers au sein de la partition.

- **Change Directory (cd)**: Change le répertoire courant de l'application.

- **Echo Message**: Affiche un message saisi par l'utilisateur, illustrant une simple interaction d'entrée/sortie.

- **Visualize Disk Space**: Affiche une vue d'ensemble de l'espace disque utilisé et disponible.

- **Exit**: Termine l'exécution du programme proprement, en libérant toutes les ressources allouées.

Organisation au sein de l'équipe : Répartition des tâches

	Yige	Saran	Shalini
Gestion de l'espace disque	X		
Développement des fonctions du système de fichier	X	X	X
Développement des fonctions simulant les commandes shell			X
Programme main: interface utilisateur	X	X	X
Guide utilisateur	X		
Guide développeur		X	X
Doxygen		X	

Conclusion

En conclusion, nous avons réussi à concevoir une bibliothèque de gestion de fichiers UNIX fonctionnelle. Grâce à une collaboration efficace, nous avons développé des fonctionnalités de base ainsi que quelques extensions. Cette documentation détaillée décrit les différentes fonctionnalités de la bibliothèque. Ce projet a été une grande opportunité pour renforcer nos compétences en programmation ainsi que les différentes notions en système d'exploitation.