GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING FACULTY

CSE 312 OPERATING SYSTEMS

PROJECT REPORT

ENES AYSU
1901042671

Program Description

This project presents the design and implementation of a simple file system, developed as part of the CSE 312 Operating Systems course. The file system comprises two main components: “create_fs.c” and “fs_operations.c”. The “create_fs.c” module is responsible for initializing a new file system with a specified block size, setting up critical structures such as the superblock, File Allocation Table (FAT), and root directory. The “fs_operations.c” module implements various file system operations including directory listing, file creation, file deletion, file reading and writing, permission management, and password protection. Through these components, the project aims to provide a comprehensive understanding of file system architecture and the underlying mechanisms that facilitate file management in an operating system.

PART 1: Design Report

1) Directory Table and Entries:

- Directory Table: The Directory Table in our file system implementation keeps track of directory entries. Each entry contains:

    - File name: Stored as a variable-length string, allowing for names up to 255 characters long.
    - Owner permissions: Represented by an unsigned integer with bitwise operations for read (R) and write (W) permissions.
    - Last modification date and time: Utilizing the time_t data type.
    - File creation date and time: Also using time_t.
    - Password protection: Indicated by a boolean flag (is_password_protected) and password hash (password_hash).

2) Free Blocks:

Our file system maintains a Free Block Management system using a File Allocation Table (FAT). Each block is represented by a bit in the FAT (0 for free, 1 for used), efficiently tracking used and free blocks.

3) Handling Arbitrary Length of File Names:

File names are stored in variable-length strings within directory entries, accommodating names longer than the block size. This design ensures flexibility in naming files within the system.

4) Handling Permissions:

Permissions are managed through attributes stored in directory entries. The permissions include read (R) and write (W) access, checked and set using bitwise operations for efficient permission handling.

5) Password Protection:

For password-protected files, we store password hashes rather than plain text passwords. The ”is_password_protected” flag indicates whether a file requires a password for access, enhancing security within the file system.

PART 2: Creating the File System

The purpose of “create_fs.c” is to create an empty file system with a specified block size and save it to a file. This file will contain all the metadata and structures required to manage the file system.

Key Components and Functions

- **Superblock Initialization:** The superblock contains metadata about the file system, such as the block size, total number of blocks, and other important information.
- **FAT Initialization:** The File Allocation Table (FAT) keeps track of the allocation status of each block in the file system.
- **Root Directory Initialization:** The root directory is the top-level directory of the file system. It is initialized with no entries.
- **Main Function:** The main function reads the command-line arguments to get the block size and the name of the file system file, then calls the initialization functions to create the file system.

PART 3: File System Operations

The purpose of “fs_operations.c” is to perform various operations on the file system created by “create_fs.c”. These operations include listing directories, creating/deleting directories, writing/reading files, and modifying file permissions.

Key Components and Functions

“loadFileSystem”:

Purpose: To load the file system from a file into memory.

Explanation:

- Input: The name of the file that contains the file system.
- Steps:
  - Opens the specified file in binary read mode.
  - Reads the file system data into the global FileSystem structure.
  - Closes the file.
- Output: Loads the file system into memory for further operations.

“saveFileSystem”:

Purpose: To save the current state of the file system from memory to a file.

Explanation:

- Input: The name of the file where the file system will be saved.
- Steps:
  - Opens the specified file in binary write mode.
  - Writes the file system data from the global FileSystem structure to the file.
  - Closes the file.
- Output: Saves the current file system state to disk.

“directoryExists”:

Purpose: To check if a directory exists.

Explanation:

- Input: The path of the directory to check.
- Steps:
  - Iterates through the directory entries to see if the path matches any existing directory.
- Output: Returns 1 if the directory exists, 0 otherwise.

“makeDirectory”:

Purpose: To create a new directory.

Explanation:

- Input: The path of the directory to create.
- Steps:
  - Tokenizes the path to create any intermediate directory as needed.
  - Checks if each part of the path exists, creating it if it does not.
  - Ensures that parent directories exist before creating subdirectories.
  - Adds a new directory entry in the file system structure.
- Output: Creates the directory if it does not exist and prints appropriate messages.

“removeDirectory”:

Purpose: To remove a directory.

Explanation:

- Input: The name of the directory to remove.
- Steps:
  - Searches for the directory entry by name.
  - Clears the entry if found.
- Output: Removes the directory and prints appropriate messages.

“findFreeBlock”:
Purpose: To find and allocate a free block within the file system using the FAT table.

Explanation:

- Input: None.
- Steps:
  - Iterates through the FAT entries starting from index 2 (as FAT-12 reserves the first 2 entries).
  - Checks if the FAT entry for a block is 0, indicating it is free.
  - If a free block is found, marks it as used by setting its FAT entry to 0xFFF (end-of-chain marker) and decrements the count of free blocks.
- Output: Returns the index of the allocated free block or -1 if no free block is available.

“freeBlock”:
Purpose: To deallocate and free a block within the file system using the FAT table.

Explanation:

- Input: The index of the block to free.
- Steps:
  - Loops until reaching the end-of-chain marker (0xFFF) in the FAT entry for the given block index.
  - Clears the FAT entry corresponding to the block index, marking it as free.
  - Moves to the next block in the chain and increments the count of free blocks.
- Output: Marks the block and its chain as free for reallocation.

“writeFile”:

Purpose: To write a file from an external source to the file system.

Explanation:

- Input: The name of the file to create in the file system, the path to the source file, and an optional password.
- Steps:
  - Opens the source file and calculates its size.
  - Checks if the file can fit in the file system and if it already exists.
  - Finds free blocks to store the file data and writes the data in blocks.
  - Updates the FAT to link blocks and adds a directory entry for the new file.
- Output: Writes the file to the file system and prints appropriate messages.

“readFile”:

Purpose: To read a file from the file system and write its contents to an external destination.

Explanation:

- Input: The name of the file to read from the file system and the path to the destination file.
- Steps:
  - Searches for the file in the directory entries.
  - Checks if the file is password protected.
  - Opens the destination file and reads the data from the file system, block by block.
  - Writes the data to the destination file.
- Output: Reads the file from the file system and writes it to the specified location, printing appropriate messages.

“deleteFile”:

Purpose: To delete a file from the file system.

Explanation:

- Input: The name of the file to delete.
- Steps:
  - Searches for the file in the directory entries.
  - Checks if the file is password protected.
  - Frees the blocks used by the file.
  - Clears the directory entry.
- Output: Deletes the file from the file system and prints appropriate messages.

“changePermissions”:
Purpose: To change the permissions of a file.

Explanation:

- Input: The name of the file and the new permissions.
- Steps:
  - Searches for the file in the directory entries.
  - Updates the permissions.
- Output: Changes the file permissions and prints appropriate messages.

“setPassword”:
Purpose: To set a password for a file.

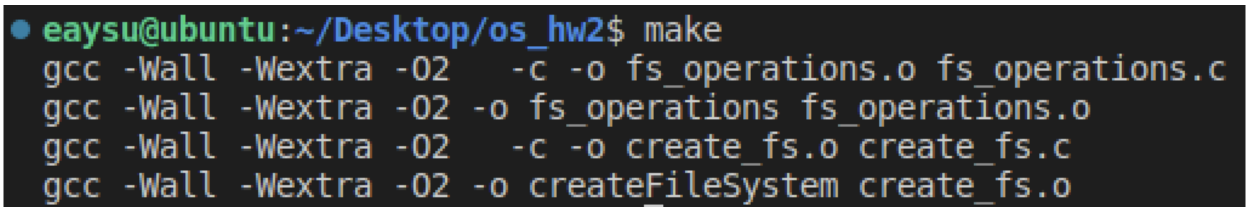
Explanation:

- Input: The name of the file and the password.
- Steps:
  - Searches for the file in the directory entries.
  - Sets the password protection flag and stores the hashed password.
- Output: Sets the password for the file and prints appropriate messages.

Tests and Screenshots

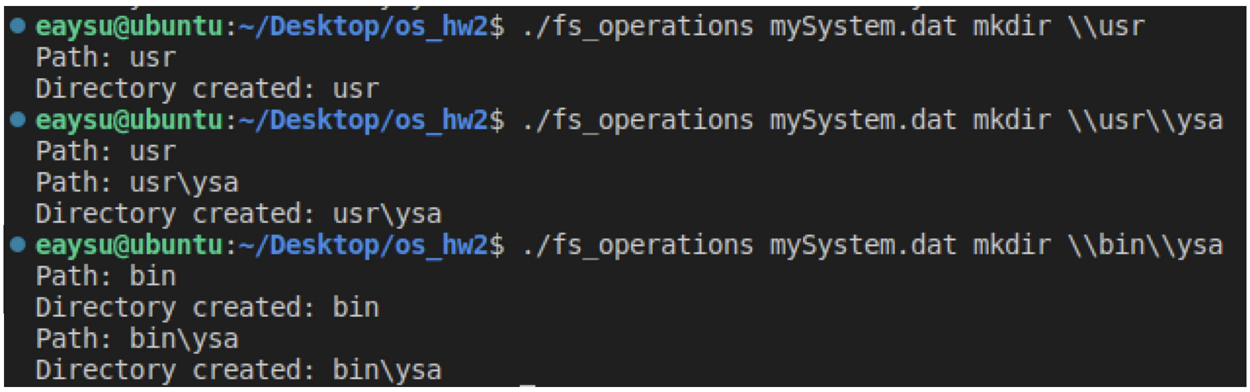
For Compilation

$ make



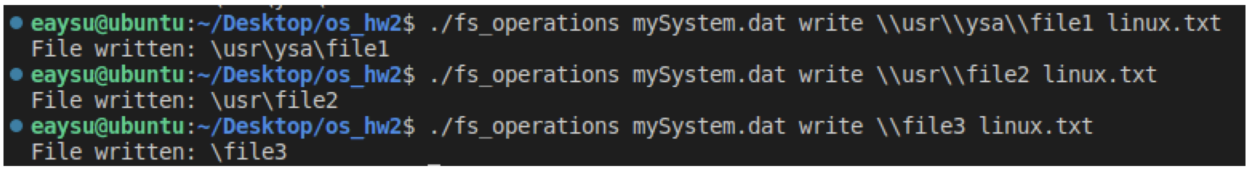
```
eaysu@ubuntu:~/Desktop/os_hw2$ make
gcc -Wall -Wextra -O2    -c -o fs_operations.o fs_operations.c
gcc -Wall -Wextra -O2 -o fs_operations fs_operations.o
gcc -Wall -Wextra -O2    -c -o create_fs.o create_fs.c
gcc -Wall -Wextra -O2 -o createFileSystem create_fs.o
```

Path Creation

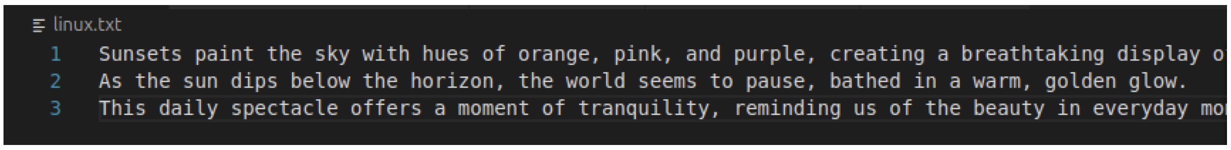


```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat mkdir \\usr
Path: usr
Directory created: usr
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat mkdir \\usr\\ysa
Path: usr
Path: usr\ysa
Directory created: usr\ysa
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat mkdir \\bin\\ysa
Path: bin
Directory created: bin
Path: bin\ysa
Directory created: bin\ysa
```

Write Operation

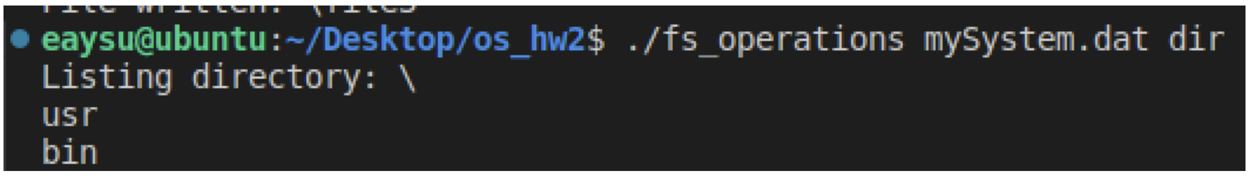


```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat write \\usr\\ysa\\file1 linux.txt
File written: \usr\ysa\file1
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat write \\usr\\file2 linux.txt
File written: \usr\file2
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat write \\file3 linux.txt
File written: \file3
```



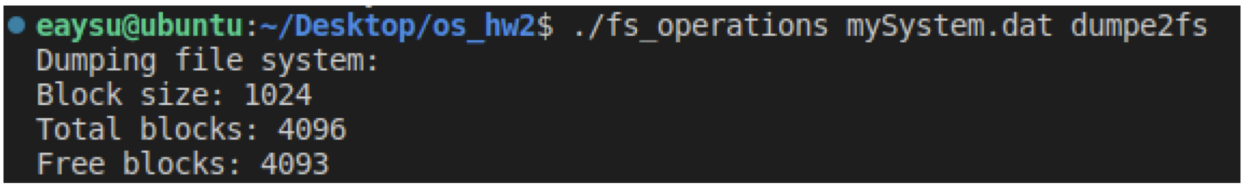
```
linux.txt
Sunsets paint the sky with hues of orange, pink, and purple, creating a breathtaking display o
As the sun dips below the horizon, the world seems to pause, bathed in a warm, golden glow.
This daily spectacle offers a moment of tranquility, reminding us of the beauty in everyday mo
```

Directory Check



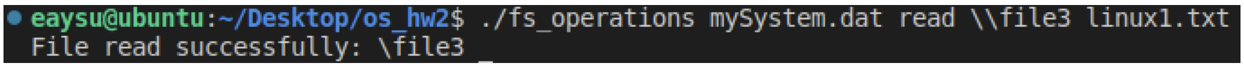
```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat dir
Listing directory: \
usr
bin
```

Dump2fs

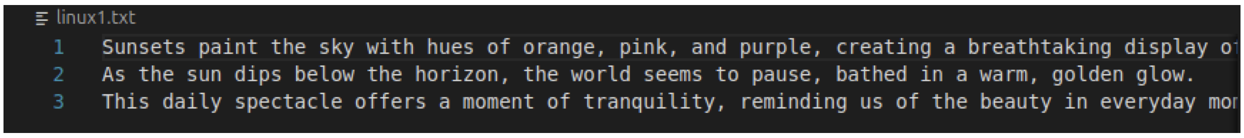


```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat dumpe2fs
Dumping file system:
Block size: 1024
Total blocks: 4096
Free blocks: 4093
```

Read Operation

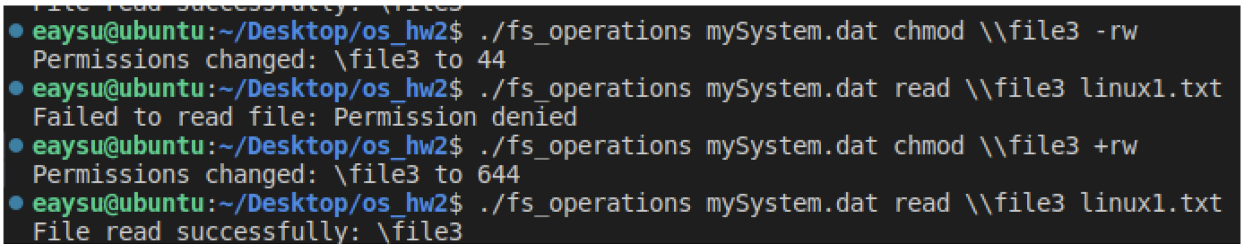


```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat read \\file3 linux1.txt
File read successfully: \file3
```



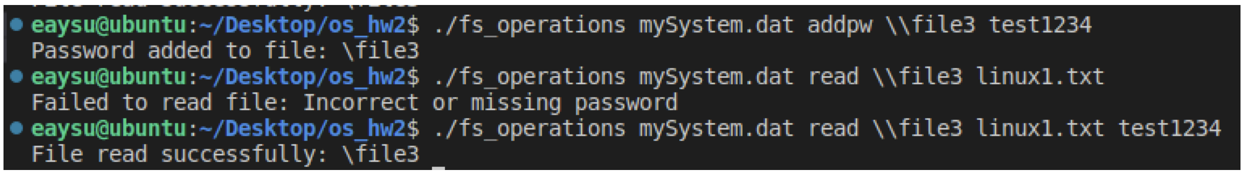
```
Sunsets paint the sky with hues of orange, pink, and purple, creating a breathtaking display o
As the sun dips below the horizon, the world seems to pause, bathed in a warm, golden glow.
This daily spectacle offers a moment of tranquility, reminding us of the beauty in everyday mor
```

Permission Control



```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat chmod \\file3 -rw
Permissions changed: \file3 to 44
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat read \\file3 linux1.txt
Failed to read file: Permission denied
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat chmod \\file3 +rw
Permissions changed: \file3 to 644
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat read \\file3 linux1.txt
File read successfully: \file3
```

Password Control



```
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat addpw \\file3 test1234
Password added to file: \file3
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat read \\file3 linux1.txt
Failed to read file: Incorrect or missing password
eaysu@ubuntu:~/Desktop/os_hw2$ ./fs_operations mySystem.dat read \\file3 linux1.txt test1234
File read successfully: \file3
```

Summary

The implementation of this file system project provides valuable insights into the fundamental workings of file system management within an operating system. By dividing the project into two distinct modules, we successfully encapsulate the process of file system creation and the execution of various file operations. The “create_fs.c” module efficiently sets up the necessary structures to manage the file system, ensuring proper initialization of the superblock, FAT, and root directory. The “fs_operations.c” module then extends this foundation by offering robust functionality to manipulate files and directories, including essential operations such as reading, writing, and permission management. Overall, this project not only demonstrates the intricacies of file system design but also highlights the importance of organized data management and access control in operating systems.