

\_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp \_tmp

**Gebze Technical University**  
**Computer Engineering**

---

# **OPERATING SYSTEMS - HW2**

**EMRE YILMAZ - 1901042606**

**SPRING 2023-24**

---

**Lecturer: Yusuf Sinan AKGÜL**

**Student No: 1901042606**

**June 8, 2024**

# Contents

<b>1</b>	<b>General Notes</b>	<b>5</b>
<b>2</b>	<b>How to Run</b>	<b>5</b>
<b>3</b>	<b>General Structure</b>	<b>6</b>
<b>4</b>	<b>Superblock</b>	<b>6</b>
4.1	Superblock Struct . . . . .	6
<b>5</b>	<b>Directory Tables and Directory Entries</b>	<b>7</b>
5.1	Directory Tables . . . . .	7
5.1.1	Structure of a Directory Table . . . . .	7
5.2	Directory Entries . . . . .	7
5.2.1	Structure of a Directory Entry . . . . .	7
5.2.2	Fields in a Directory Entry . . . . .	8
5.3	Storing Nested Directory Tables . . . . .	8
5.4	Directory Navigation . . . . .	8
5.5	Example . . . . .	8
5.6	Summary . . . . .	9
<b>6</b>	<b>How to Solve Arbitrary Length of File Names</b>	<b>9</b>
6.1	File Name Area . . . . .	9
6.2	Storing File Names . . . . .	9
6.3	Algorithm . . . . .	9
6.4	Implementation Summary . . . . .	9
<b>7</b>	<b>How to Keep Free Blocks</b>	<b>10</b>
7.1	File Allocation Table (FAT) . . . . .	10
7.2	Purpose and Usage of the Bitmap . . . . .	10
7.2.1	Free Block Bitmap . . . . .	10
7.2.2	Marking Blocks as Free or Used . . . . .	11
7.2.3	Finding a Free Block . . . . .	11
7.2.4	Implementation Summary . . . . .	12
<b>8</b>	<b>How to Handle File Permissions</b>	<b>12</b>
8.1	Permission Flags . . . . .	12
8.2	Checking Permissions . . . . .	12
8.3	Changing Permissions . . . . .	12
8.4	Implementation Summary . . . . .	12
<b>9</b>	<b>How to Handle Passwords</b>	<b>13</b>
9.1	Password Storage . . . . .	13
9.2	Setting Passwords . . . . .	13
9.3	Implementation Summary . . . . .	13

<b>10 Base Pointer to File System</b>	<b>13</b>
10.1 Base Pointer Definition . . . . .	13
10.2 Setting the Base Pointer . . . . .	13
10.3 Accessing File System Structures . . . . .	14
10.4 Implementation Summary . . . . .	14
<b>11 How to Save and Load the File System</b>	<b>14</b>
11.1 Saving the File System . . . . .	14
11.1.1 Purpose . . . . .	14
11.1.2 Use Cases . . . . .	14
11.1.3 Algorithm . . . . .	14
11.1.4 Implementation Summary . . . . .	15
11.2 Loading the File System . . . . .	15
11.2.1 Purpose . . . . .	15
11.2.2 Use Cases . . . . .	15
11.2.3 Algorithm . . . . .	15
11.2.4 Implementation Summary . . . . .	15
<b>12 Functions That Are Used For File System Implementation</b>	<b>15</b>
12.1 initializeFileSystem . . . . .	15
12.1.1 Purpose . . . . .	15
12.1.2 Use Cases . . . . .	16
12.1.3 Algorithm . . . . .	16
12.1.4 Implementation Summary . . . . .	16
12.2 setBlockFree . . . . .	16
12.2.1 Purpose . . . . .	16
12.2.2 Use Cases . . . . .	16
12.2.3 Algorithm . . . . .	16
12.2.4 Implementation Summary . . . . .	16
12.3 setBlockUsed . . . . .	17
12.3.1 Purpose . . . . .	17
12.3.2 Use Cases . . . . .	17
12.3.3 Algorithm . . . . .	17
12.3.4 Implementation Summary . . . . .	17
12.4 findFreeBlock . . . . .	17
12.4.1 Purpose . . . . .	17
12.4.2 Use Cases . . . . .	17
12.4.3 Algorithm . . . . .	17
12.4.4 Implementation Summary . . . . .	17
12.5 createFile . . . . .	17
12.5.1 Purpose . . . . .	17
12.5.2 Use Cases . . . . .	18
12.5.3 Algorithm . . . . .	18
12.5.4 Implementation Summary . . . . .	18
12.6 deleteFile . . . . .	18
12.6.1 Purpose . . . . .	18
12.6.2 Use Cases . . . . .	18
12.6.3 Algorithm . . . . .	18

12.6.4	Implementation Summary . . . . .	19
12.7	printFileDetails . . . . .	19
12.7.1	Purpose . . . . .	19
12.7.2	Use Cases . . . . .	19
12.7.3	Algorithm . . . . .	19
12.7.4	Implementation Summary . . . . .	19
12.8	saveFileSystem . . . . .	19
12.8.1	Purpose . . . . .	19
12.8.2	Use Cases . . . . .	19
12.8.3	Algorithm . . . . .	19
12.8.4	Implementation Summary . . . . .	20
12.9	loadFileSystem . . . . .	20
12.9.1	Purpose . . . . .	20
12.9.2	Use Cases . . . . .	20
12.9.3	Algorithm . . . . .	20
12.9.4	Implementation Summary . . . . .	20
12.10	createDirectory . . . . .	20
12.10.1	Purpose . . . . .	20
12.10.2	Use Cases . . . . .	20
12.10.3	Algorithm . . . . .	21
12.10.4	Implementation Summary . . . . .	21
12.11	deleteDirectory . . . . .	21
12.11.1	Purpose . . . . .	21
12.11.2	Use Cases . . . . .	21
12.11.3	Algorithm . . . . .	21
12.11.4	Implementation Summary . . . . .	21
12.12	printDirectoryDetails . . . . .	22
12.12.1	Purpose . . . . .	22
12.12.2	Use Cases . . . . .	22
12.12.3	Algorithm . . . . .	22
12.12.4	Implementation Summary . . . . .	22
12.13	findDirectory . . . . .	22
12.13.1	Purpose . . . . .	22
12.13.2	Use Cases . . . . .	22
12.13.3	Algorithm . . . . .	22
12.13.4	Implementation Summary . . . . .	22
12.14	write . . . . .	23
12.14.1	Purpose . . . . .	23
12.14.2	Use Cases . . . . .	23
12.14.3	Algorithm . . . . .	23
12.14.4	Implementation Summary . . . . .	23
12.15	read . . . . .	23
12.15.1	Purpose . . . . .	23
12.15.2	Use Cases . . . . .	23
12.15.3	Algorithm . . . . .	23
12.15.4	Implementation Summary . . . . .	24
12.16	findFileInDirectory . . . . .	24
12.16.1	Purpose . . . . .	24

12.16.2 Use Cases . . . . .	24
12.16.3 Algorithm . . . . .	24
12.16.4 Implementation Summary . . . . .	24
12.17 duple2fs . . . . .	24
12.17.1 Purpose . . . . .	24
12.17.2 Use Cases . . . . .	24
12.17.3 Algorithm . . . . .	25
12.17.4 Implementation Summary . . . . .	25
12.18 printDirectoryContents . . . . .	25
12.18.1 Purpose . . . . .	25
12.18.2 Use Cases . . . . .	25
12.18.3 Algorithm . . . . .	25
12.18.4 Implementation Summary . . . . .	25
12.19 countFilesAndDirectories . . . . .	25
12.19.1 Purpose . . . . .	25
12.19.2 Use Cases . . . . .	25
12.19.3 Algorithm . . . . .	26
12.19.4 Implementation Summary . . . . .	26
12.20 chmodFile . . . . .	26
12.20.1 Purpose . . . . .	26
12.20.2 Use Cases . . . . .	26
12.20.3 Algorithm . . . . .	26
12.20.4 Implementation Summary . . . . .	26
12.21 addPassword . . . . .	26
12.21.1 Purpose . . . . .	26
12.21.2 Use Cases . . . . .	26
12.21.3 Algorithm . . . . .	27
12.21.4 Implementation Summary . . . . .	27
<b>13 Test Cases and Results</b>	<b>28</b>
13.1 Test Case in PDF . . . . .	28
13.2 Read - Write Additional Proof . . . . .	30
13.3 rmdir Proof . . . . .	32
13.4 addpw and chmod Error Handling . . . . .	33
13.5 Testing 4KB Data Write and Read . . . . .	34

## 1 General Notes

1. File paths and directories should not include " " characters. You can see how you should give the inputs in the section13.
2. When you compile the code, two executables are created. One is `./makeFileSystem`, and the other is `./fileSystemOper`. You can run the program as described in the assignment PDF.
3. The block size is 1KB or 0.5KB as stated in the PDF. When running the program, you should provide these sizes in BYTES. I have added the necessary warning message to the program. An example usage is: `./makeFileSystem 512 test.data`. This command creates a file system with a block size of 512 bytes.
4. I have configured the program to work exactly as shown in the test cases in the PDF. While `./makeFileSystem` creates the file system, `./fileSystemOper` allows you to run the desired operations. The program operates with command line arguments, as stated in the PDF, and not in a loop.
5. There are no commands that do not work. Every command in the example test case in the PDF has been tested and demonstrated. Additionally, the `rmdir` command has been tested separately since it is not included in the example test case in the PDF.
6. The file system sizes, including everything, are 2.1MB with a 0.5KB block size and 4.2MB with a 1KB block size. You can see the details in the report.
7. In the report, I tried to explain each function in its simplest form. I did not want to complicate the report with deep implementation details. Still, first, you can easily understand everything by examining the `header` files under the `include` directory and then reviewing the relevant implementation files in `src` folder.
8. I am ready to provide a demo for any questions you might have. I believe I have prepared the report clearly and comprehensibly, but you can reach me at any time if needed: `eyilmaz2019@gtu.edu.tr`, +905319346629.
9. **All operations are tested with VALGRIND. There is no memory leak. You can verify it by testing.**

## 2 How to Run

Steps to follow:

1. Compile the program in the directory where the `makefile` is located by using the `make` command.
2. Create the file system using the command `./makeFileSystem <block-size> <file-system-name>`. (As stated in the PDF.)
3. Run the desired operation using the command `./fileSystemOper <command> <arguments...>`. (As stated in the PDF.)

You can see how to run the file system in test case results section: Please click section: 13

### 3 General Structure

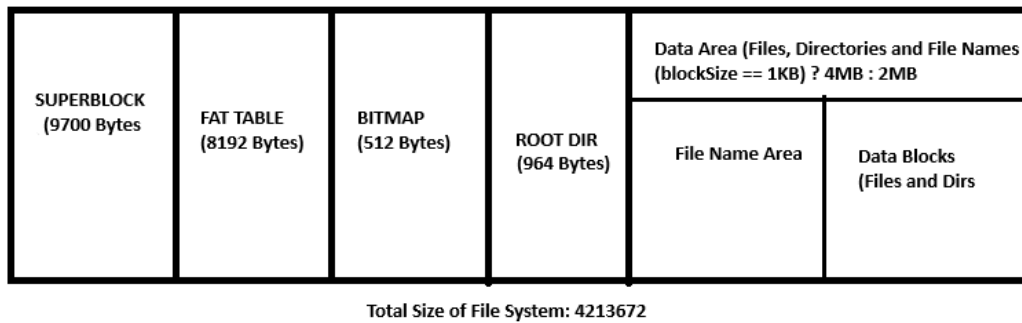


Fig: General Structure

As you can see, the file system has been developed in accordance with the general structure in the book. Since the Superblock, FAT, Bitmap, and Root Directory are not stored in the data area, the total file system size is equal to 4MB plus the space occupied by these areas. Arbitrary file names are also handled by allocating a portion of the data area. You will be able to see the details in the following sections.

## 4 Superblock

The superblock is a crucial component of our file system, containing metadata that defines the overall structure and state of the file system. It includes information about block sizes, free blocks, FAT entries, and more. This structure is initialized when the file system is created and loaded into memory when the file system is mounted.

### 4.1 Superblock Struct

1. blockSize: Specifies the size of each block in the file system, typically 1024 bytes (1 KB).
2. totalBlocks: Indicates the total number of blocks in the file system (e.g., 4096 blocks).
3. freeBlocks: Tracks the number of blocks that are currently free.
4. fatBlocks: Specifies the number of blocks used for the File Allocation Table (FAT).
5. rootDirectory: Contains the directory table for the root directory.
6. freeBlocksBitmap: A bitmap that tracks the allocation status of each block.
7. fat: The File Allocation Table, managing block linkage for files and directories.
8. fileNameArea: Manages the storage space for file and directory names.
9. dataArea: Contains user data and files, specifying the offset and size of the data segment.



## 5 Directory Tables and Directory Entries

In our file system, directories and files are managed using directory tables and directory entries. This section provides a detailed overview of how these structures are stored and managed.

### 5.1 Directory Tables

A directory table is a structure that contains metadata for the files and subdirectories within a directory. Each directory table is stored in a specific block in the file system. The root directory is represented by a directory table stored in the superblock, while nested directories have their own directory tables stored in different blocks.

#### 5.1.1 Structure of a Directory Table

A directory table consists of an array of directory entries and a counter that tracks the number of entries. The structure is defined as follows:

```
typedef struct
{
    DirectoryEntry entries[MAX_FILES];
    uint16_t fileCount;
} DirectoryTable;
```

Each directory table can hold up to `MAX_FILES` entries, which can be either files or subdirectories.

### 5.2 Directory Entries

A directory entry stores metadata about a file or subdirectory. The structure of a directory entry includes information such as the type of entry (file or directory), the name of the entry, permissions, size, and the first block where the entry's data or subdirectory table is stored.

#### 5.2.1 Structure of a Directory Entry

A directory entry is defined as follows:

```
typedef struct
{
    EntryType entryType;           // Type of entry (file or directory)
    uint32_t fileNameOffset;       // Offset of the file name within the file system
    uint16_t fileNameLength;       // Length of the file name
    uint32_t size;                 // File size (0 for directories)
    uint16_t permissions;         // Owner permissions (R and W)
    uint32_t creationDate;         // File creation date
    uint32_t modificationDate;    // Last modification date
    char password[32];             // Password protection
    uint16_t firstBlock;           // First block of the file or directory
} DirectoryEntry;
```

### 5.2.2 Fields in a Directory Entry

- **entryType**: Indicates whether the entry is a file or a directory.
- **fileNameOffset**: The offset within the file system where the file name is stored.
- **fileNameLength**: The length of the file name, including the null terminator.
- **size**: The size of the file in bytes. For directories, this is set to 0.
- **permissions**: The permissions for the file or directory (e.g., read, write).
- **creationDate**: The date when the file or directory was created.
- **modificationDate**: The date when the file or directory was last modified.
- **password**: A password for accessing the file or directory, if applicable.
- **firstBlock**: The first block where the file's data or the subdirectory's table is stored.

### 5.3 Storing Nested Directory Tables

Nested directories are handled by storing their directory tables in blocks allocated specifically for them. When a new directory is created within another directory, the system performs the following steps:

1. Allocate a free block for the new directory's table.
2. Create a new directory entry in the parent directory's table.
3. Set the **firstBlock** field of the new directory entry to the block number allocated for the new directory's table.
4. Initialize the new directory in the allocated block.

The function `createDirectory` shows how a nested directory is created and stored. You can examine it.

### 5.4 Directory Navigation

To navigate through directories, the system starts from the root directory and follows the **firstBlock** pointers of directory entries. Each subdirectory is accessed by loading its directory table from the block specified in its parent directory entry. This allows for efficient traversal of nested directories.

### 5.5 Example

Consider the following directory structure:

```
/
|-- documents
    |-- images
        |-- vacation
```

In this example, the `documents` directory has a directory table stored in a specific block. Within `documents`, the `images` directory has its directory table stored in another block, and within `images`, the `vacation` directory has its directory table stored in yet another block.

Each directory entry for `documents`, `images`, and `vacation` contains a `firstBlock` field pointing to the block where their respective directory tables are stored.

## 5.6 Summary

Directory tables and directory entries provide a flexible and efficient way to manage files and directories in the file system. By using the `firstBlock` field to store the location of nested directory tables, the system can easily navigate and manage hierarchical directory structures.

# 6 How to Solve Arbitrary Length of File Names

In our file system, file names can have arbitrary lengths, but they are managed within a fixed-size file name area to ensure efficient storage and easy managing.

## 6.1 File Name Area

The file name area is a contiguous memory region allocated for storing file names. It is defined within the superblock as follows:

```
typedef struct
{
    uint32_t offset; // Offset where file names start
    uint32_t size;   // Size of the file name storage area
    uint32_t used;   // Amount of used space in the file name storage area
} FileNameArea;
```

## 6.2 Storing File Names

When a file or directory is created, its name is stored in the file name area, and the corresponding directory entry stores the offset and length of the name. This allows for arbitrary length file names, as long as the total size does not exceed the allocated file name area.

## 6.3 Algorithm

1. Check if there is enough space in the file name area for the new file name.
2. Copy the file name to the file name area at the current offset.
3. Update the directory entry with the offset and length of the file name.
4. Increment the used space in the file name area by the length of the file name.

## 6.4 Implementation Summary

By using a dedicated file name area and storing offsets and lengths in directory entries, the system can efficiently manage file names of arbitrary lengths without wasting space.

## 7 How to Keep Free Blocks

The file system uses two mechanisms to track the status of blocks: a File Allocation Table (FAT) and a bitmap. The FAT serves as the primary method for managing free and used blocks, while the bitmap is used experimentally for tracking free blocks.

### 7.1 File Allocation Table (FAT)

The FAT is a table that maps each block to the next block in a file, or marks the end of a file or a free block. It allows efficient tracking of file fragments and free blocks. Each entry in the FAT corresponds to a block in the file system. If the FAT entry for a block contains a special value (e.g., 0xFFFF), it indicates that the block is free. Otherwise, the value points to the next block in the file.

When a block needs to be marked as free or used:

- To mark a block as free, the corresponding FAT entry is set to 0xFFFF.
- To mark a block as used, the corresponding FAT entry is set to the next block in the file or 0xFFFF if it is the last block of a file.

The FAT allows the system to manage file storage and free blocks without needing to scan through all blocks, providing quick lookups and updates.

### 7.2 Purpose and Usage of the Bitmap

Actually, bitmap was not necessary for this homework, but still, I wanted to implement it. The bitmap is used experimentally alongside the FAT to track the status of blocks. The primary purpose of the bitmap is to provide a visual and bit-level tracking of block usage. It offers a more granular view of block allocation status and can be used for debugging, testing, and educational purposes.

Each bit in the bitmap represents a block in the file system:

- A bit set to 1 indicates a free block.
- A bit set to 0 indicates a used block.

Although the FAT is the main mechanism for managing block allocation, the bitmap allows for quick scanning of free and used blocks, which can be useful for certain operations or analysis.

#### 7.2.1 Free Block Bitmap

The free block bitmap is an array that consists of 8 byte elements, and each bit in these elements represents a block in the file system. A bit set to 1 indicates a free block, while a bit set to 0 indicates a used block. In this way, we do not waste memory and use it very efficiently. We use each bit of the bitmap array efficiently; each bit has a meaning.

Each element of the bitmap array is 1 byte in size, and each bit in this element represents whether a block is free or used. Free blocks are marked with 1, and used blocks are marked with 0. When a block number needs to be marked as used, first, the corresponding bitmap element is found using  $(\text{blockNumber} / 8)$ , then the specific bit within this element is determined using  $(\text{blockNumber} \% 8)$ . The binary number 00000001 is then shifted to match the corresponding bit, thereby locating the correct block in the bitmap. For example, if we want to mark block 0 as used, we go to the 0th element in the bitmap and change its 0th bit. After finding the bit, its inverse is taken binary, and

an AND operation is performed with the bitmap element. This way, the desired block is correctly changed to 0 (used).

Making blocks free follows a similar logic. You can check the implementation. If you do not understand the logic, call me whenever you want: eyilmaz2019@gtu.edu.tr, +905319346629.

The structure is defined as follows:

```
typedef struct
{
    uint8_t bitmap[MAX_BLOCKS / 8]; // Bitmap to track free blocks
} FreeBlockBitmap;
```

### 7.2.2 Marking Blocks as Free or Used

```
void setBlockFree(uint16_t blockNumber)
{
    if (blockNumber < MAX_BLOCKS)
    {
        superBlock.freeBlocksBitmap.bitmap[blockNumber / 8] |= (1 << (blockNumber % 8));
        superBlock.freeBlocks++;
    }
}

void setBlockUsed(uint16_t blockNumber)
{
    if (blockNumber < MAX_BLOCKS)
    {
        superBlock.freeBlocksBitmap.bitmap[blockNumber / 8] &= ~(1 << (blockNumber % 8));
        superBlock.freeBlocks--;
    }
}
```

### 7.2.3 Finding a Free Block

To find a free block, the system scans the bitmap to locate the first bit set to 1:

```
uint16_t findFreeBlock(char *fsBase)
{
    for (int i = 0; i < MAX_BLOCKS; ++i)
    {
        if (superBlock.freeBlocksBitmap.bitmap[i / 8] & (1 << (i % 8)))
        {
            return i;
        }
    }
    return (uint16_t)-1; // No free block found
}
```

### 7.2.4 Implementation Summary

Using a bitmap to track free blocks allows the system to efficiently manage block allocation and deallocation, ensuring that blocks are reused and free space is maximized.

## 8 How to Handle File Permissions

File permissions control the access rights for files and directories, ensuring that only authorized users can read or write data.

### 8.1 Permission Flags

Permissions are stored as bit flags in the directory entry:

```
#define PERMISSION_READ 0x01
#define PERMISSION_WRITE 0x02

typedef struct
{
    uint16_t permissions; // Owner permissions (R and W)
} DirectoryEntry;
```

### 8.2 Checking Permissions

When accessing a file or directory, the system checks the permission flags to determine if the requested operation is allowed:

```
int checkPermissions(uint16_t permissions, uint16_t required)
{
    return (permissions & required) == required;
}
```

### 8.3 Changing Permissions

File permissions can be changed using the `chmodFile` function:

```
int chmodFile(char *fsBase, const char *filePath, uint16_t newPermissions)
{
    // Find the file and update its permissions
    // ...
    entry->permissions = newPermissions;
    entry->modificationDate = time(NULL); // Update the modification date
    return 0; // Success
}
```

### 8.4 Implementation Summary

By using bit flags for permissions, the system can efficiently manage and enforce access controls for files and directories, ensuring data security.

## 9 How to Handle Passwords

Passwords add an additional layer of security, restricting access to files and directories.

### 9.1 Password Storage

Passwords are stored as plain text in the directory entry. The `password` field in the `DirectoryEntry` structure holds the password:

```
typedef struct
{
    char password[32]; // Password protection
} DirectoryEntry;
```

### 9.2 Setting Passwords

```
void addPassword(char *fsBase, const char *filePath, const char *password)
{
    // Find the file and update its password
    // ...
    strcpy(entry->password, password, sizeof(entry->password));
    entry->modificationDate = time(NULL); // Update the modification date
}
```

### 9.3 Implementation Summary

Passwords are managed by storing them in the directory entries and providing functions to set and check passwords, enhancing security by restricting access to authorized users.

## 10 Base Pointer to File System

The base pointer to the file system memory is crucial for accessing and manipulating the file system's data structures.

### 10.1 Base Pointer Definition

The base pointer is defined globally and set during the initialization of the file system:

```
extern char *fsMemoryBase; // Base pointer for file system memory
```

### 10.2 Setting the Base Pointer

The base pointer is set in the `initializeFileSystem` function:

```
void initializeFileSystem(uint16_t blockSize, char *fsBase, int totalFsSize)
{
    fsMemoryBase = fsBase; // Set the global file system base pointer
    // ...
}
```

## 10.3 Accessing File System Structures

The base pointer allows access to various file system structures by calculating offsets:

```
char *fileNamePtr = fsMemoryBase + superBlock.fileNameArea.offset;
DirectoryTable *dirTable = (DirectoryTable *) (fsMemoryBase + blockNumber * superBlock.blockSize);
```

## 10.4 Implementation Summary

The base pointer is essential for navigating and manipulating the file system. It provides a reference point for accessing all other data structures, ensuring consistent and efficient memory management.

This pointer is central to the file system's operation, providing a reference point for all file system activities.

# 11 How to Save and Load the File System

In our file system, saving and loading the file system state is crucial for data persistence. This section details the process of writing the file system's current state to a file and reading it back into memory.

## 11.1 Saving the File System

The process of saving the file system involves writing all critical data structures to a file. This ensures that the file system's state can be restored later.

### 11.1.1 Purpose

To persist the current state of the file system to a file, allowing it to be restored at a later time.

### 11.1.2 Use Cases

- Regular backups of the file system.
- Saving the file system before shutting down the system.

### 11.1.3 Algorithm

1. Open the file in write-binary mode.
2. Write the superblock to the file.
3. Write the FAT (File Allocation Table) to the file.
4. Write the root directory table to the file.
5. Write the free block bitmap to the file.
6. Write the file name area to the file.
7. Write the data area to the file.
8. Close the file.



#### 11.1.4 Implementation Summary

The `saveFileSystem` function is responsible for saving the file system. This function opens a file, writes all necessary data structures, and closes the file, ensuring that the file system's state is accurately saved.

### 11.2 Loading the File System

Loading the file system involves reading the previously saved state from a file and reconstructing the in-memory data structures.

#### 11.2.1 Purpose

To restore the file system's state from a file, enabling continuation from where it was left off.

#### 11.2.2 Use Cases

- Restoring the file system after a shutdown or crash.
- Loading a backup or snapshot of the file system.

#### 11.2.3 Algorithm

1. Open the file in read-binary mode.
2. Get the size of the file.
3. Allocate memory for the file system.
4. Read the entire file into memory.
5. Copy the superblock from the file to memory.
6. Set pointers to the FAT, root directory table, free block bitmap, file name area, and data area based on offsets.
7. Close the file.

#### 11.2.4 Implementation Summary

The `loadFileSystem` function is responsible for loading the file system. This function opens a file, reads the saved state into memory, and reconstructs the file system's data structures, allowing the file system to be restored to its previous state.

## 12 Functions That Are Used For File System Implementation

### 12.1 initializeFileSystem

#### 12.1.1 Purpose

To initialize the file system by setting up the superblock, free block bitmap, FAT, and root directory.

### 12.1.2 Use Cases

- Creating a new file system.
- Setting up the initial state of the file system for the first use.

### 12.1.3 Algorithm

1. Set the global base pointer for the file system memory.
2. Clear all memory to zero.
3. Initialize the superblock fields including block size, total blocks, free blocks, FAT blocks, and data area.
4. Initialize the free block bitmap, marking all blocks as free.
5. Initialize the FAT with end-of-chain markers (0xFFFF).
6. Set up the root directory with no entries.
7. Configure the file name area, specifying its offset, size, and used space.
8. Configure the data area, specifying its offset, size, and block count.
9. Mark the blocks reserved for the file system metadata as used.

### 12.1.4 Implementation Summary

This function initializes all necessary data structures and fields in the superblock to set up a fresh file system. It ensures that the file system is ready for operations like creating files and directories.

## 12.2 setBlockFree

### 12.2.1 Purpose

To mark a specific block as free in the free block bitmap and update the count of free blocks.

### 12.2.2 Use Cases

- Reclaiming a block when a file is deleted.
- Making a block available for future allocations.

### 12.2.3 Algorithm

1. Check if the block number is within the valid range.
2. Set the corresponding bit in the free block bitmap to 1 to mark the block as free.
3. Increment the count of free blocks in the superblock.

### 12.2.4 Implementation Summary

This function updates the free block bitmap to mark a block as free and increments the count of free blocks, making it available for future use.

## **12.3 setBlockUsed**

### **12.3.1 Purpose**

To mark a specific block as used in the free block bitmap and update the count of free blocks.

### **12.3.2 Use Cases**

- Allocating a block when a file is created or extended.

### **12.3.3 Algorithm**

1. Check if the block number is within the valid range.
2. Clear the corresponding bit in the free block bitmap to 0 to mark the block as used.
3. Decrement the count of free blocks in the superblock.

### **12.3.4 Implementation Summary**

This function updates the free block bitmap to mark a block as used and decrements the count of free blocks, ensuring the block is not allocated again.

## **12.4 findFreeBlock**

### **12.4.1 Purpose**

To find the first free block in the file system and return its index.

### **12.4.2 Use Cases**

- Allocating a block when creating a new file or directory.
- Extending a file that requires additional blocks.

### **12.4.3 Algorithm**

1. Iterate through the free block bitmap.
2. Check each bit to find the first free block.
3. Return the index of the first free block found.
4. If no free block is found, return an error indicator (e.g., (uint16\_t)-1).

### **12.4.4 Implementation Summary**

This function scans the free block bitmap to locate the first available block and returns its index for allocation purposes.

## **12.5 createFile**

### **12.5.1 Purpose**

To create a new file in a specified directory with given permissions and an optional password.

### **12.5.2 Use Cases**

- Adding a new file to the file system.
- Storing data with specific access permissions.

### **12.5.3 Algorithm**

1. Find the specified directory.
2. Check if the directory has space for a new file.
3. Check if there is enough space for the file name in the file name area.
4. Find a free block for the file.
5. Create a new directory entry for the file with the specified permissions and password.
6. Copy the file name to the file name area.
7. Mark the allocated block as used and update the FAT.

### **12.5.4 Implementation Summary**

This function handles all steps required to create a new file, including directory lookup, space checks, block allocation, and directory entry creation.

## **12.6 deleteFile**

### **12.6.1 Purpose**

To delete a specified file from the file system, reclaiming its blocks.

### **12.6.2 Use Cases**

- Removing a file that is no longer needed.
- Freeing up space for other files.

### **12.6.3 Algorithm**

1. Duplicate the file path to avoid modifying the original.
2. Split the path into directory and file name.
3. Find the specified directory.
4. Locate the file in the directory.
5. Free all blocks used by the file.
6. Remove the directory entry and reclaim the file name space.
7. Update the directory's file count.

#### **12.6.4 Implementation Summary**

This function performs all necessary operations to safely delete a file, including block deallocation and directory entry removal.

### **12.7 printFileDetails**

#### **12.7.1 Purpose**

To print detailed information about a specified file.

#### **12.7.2 Use Cases**

- Viewing file metadata such as size, permissions, and creation date.
- Debugging and verifying file system integrity.

#### **12.7.3 Algorithm**

1. Find the specified directory.
2. Locate the file in the directory.
3. Print the file name, size, permissions, creation date, modification date, and first block.

#### **12.7.4 Implementation Summary**

This function retrieves and displays metadata for a specified file, aiding in file management and debugging.

### **12.8 saveFileSystem**

#### **12.8.1 Purpose**

To save the current state of the file system to a file.

#### **12.8.2 Use Cases**

- Persisting the file system state to disk.
- Creating backups of the file system.

#### **12.8.3 Algorithm**

1. Open the specified file for writing.
2. Write the superblock to the file.
3. Write the FAT to the file.
4. Write the root directory table to the file.
5. Write the free block bitmap to the file.

6. Write the file name area to the file.
7. Write the data area to the file.
8. Close the file.

#### **12.8.4 Implementation Summary**

This function serializes the entire file system state and writes it to a file, ensuring all data structures are saved correctly.

### **12.9 loadFileSystem**

#### **12.9.1 Purpose**

To load the file system state from a file into memory.

#### **12.9.2 Use Cases**

- Restoring the file system state from disk.
- Loading a previously saved file system.

#### **12.9.3 Algorithm**

1. Open the specified file for reading.
2. Read the file system size.
3. Allocate memory for the file system.
4. Read the superblock from the file.
5. Set pointers to the FAT, root directory, free block bitmap, file name area, and data area.
6. Close the file.

#### **12.9.4 Implementation Summary**

This function reads the serialized file system state from a file and restores it into memory, ready for use.

### **12.10 createDirectory**

#### **12.10.1 Purpose**

To create a new directory in a specified path with given permissions and an optional password.

#### **12.10.2 Use Cases**

- Organizing files into hierarchical structures.
- Creating subdirectories for better file management.

### 12.10.3 Algorithm

1. Check if the root directory has space for a new entry.
2. Check if there is enough space for the directory name in the file name area.
3. Find a free block for the directory.
4. If the directory is at the root level:
  - (a) Create a new directory entry in the root directory.
  - (b) Initialize the new directory table.
5. If the directory is nested:
  - (a) Find the parent directory.
  - (b) Create a new directory entry in the parent directory.
  - (c) Initialize the new directory table.
6. Copy the directory name to the file name area.
7. Mark the allocated block as used.

### 12.10.4 Implementation Summary

This function handles creating both root-level and nested directories, including directory lookup, space checks, block allocation, and directory entry creation.

## 12.11 deleteDirectory

### 12.11.1 Purpose

To delete a specified directory and its contents from the file system.

### 12.11.2 Use Cases

- Removing directories that are no longer needed.
- Freeing up space by deleting directories and their contents.

### 12.11.3 Algorithm

1. Use a helper function to recursively delete the contents of the directory.
2. Free all blocks used by the directory and its contents.
3. Remove the directory entry from the parent directory.
4. Update the parent directory's file count.

### 12.11.4 Implementation Summary

This function ensures safe and complete deletion of directories, including recursive deletion of nested directories and files.

## **12.12 printDirectoryDetails**

### **12.12.1 Purpose**

To print detailed information about a specified directory and its contents recursively.

### **12.12.2 Use Cases**

- Viewing directory structure and contents.
- Debugging and verifying file system integrity.

### **12.12.3 Algorithm**

1. Find the specified directory.
2. Print the details of each file and subdirectory in the directory.
3. Recursively print the contents of subdirectories.

### **12.12.4 Implementation Summary**

This function retrieves and displays metadata for a specified directory and its contents, aiding in directory management and debugging.

## **12.13 findDirectory**

### **12.13.1 Purpose**

To find a specified directory within the file system.

### **12.13.2 Use Cases**

- Locating directories for various file system operations.
- Navigating the directory structure.

### **12.13.3 Algorithm**

1. If the directory name is the root directory, return the root directory.
2. Iterate through the directory entries to find the specified directory.
3. If a match is found, return the directory table.
4. If not found, return NULL.

### **12.13.4 Implementation Summary**

This function locates a specified directory within the file system, enabling directory-related operations.



## **12.14 write**

### **12.14.1 Purpose**

To write the contents of a Linux file into a specified directory in the file system.

### **12.14.2 Use Cases**

- Importing files from the Linux file system into the custom file system.
- Storing data in the file system.

### **12.14.3 Algorithm**

1. Get the permissions of the Linux file.
2. Open the Linux file for reading.
3. Read the content of the Linux file.
4. Create a new file in the specified directory.
5. Check write permissions and password.
6. Write the content to the new file in the file system.
7. Free allocated memory.

### **12.14.4 Implementation Summary**

This function imports a file from the Linux file system, creating a new file in the custom file system and copying the content.

## **12.15 read**

### **12.15.1 Purpose**

To read the contents of a file from the file system and write it to a Linux file.

### **12.15.2 Use Cases**

- Exporting files from the custom file system to the Linux file system.
- Accessing data stored in the file system.

### **12.15.3 Algorithm**

1. Duplicate the file path to avoid modifying the original.
2. Split the path into directory and file name.
3. Find the specified directory.
4. Locate the file in the directory.

5. Check read permissions and password.
6. Open the Linux file for writing.
7. Read the content from the file system and write it to the Linux file.
8. Set Linux file permissions.

#### **12.15.4 Implementation Summary**

This function exports a file from the custom file system to the Linux file system, copying the content and setting appropriate permissions.

### **12.16 findFileInDirectory**

#### **12.16.1 Purpose**

To find a specified file within a given directory table.

#### **12.16.2 Use Cases**

- Locating files for various file system operations.
- Verifying file existence in a directory.

#### **12.16.3 Algorithm**

1. Iterate through the directory entries to find the specified file.
2. If a match is found, return the directory entry.
3. If not found, return NULL.

#### **12.16.4 Implementation Summary**

This function locates a specified file within a directory table, enabling file-related operations.

### **12.17 dumpe2fs**

#### **12.17.1 Purpose**

To print the current state of the file system, including block counts, free blocks, and directory contents.

#### **12.17.2 Use Cases**

- Debugging and verifying file system integrity.
- Monitoring file system status.

### **12.17.3 Algorithm**

1. Print the superblock information.
2. Count and print the number of files and directories.
3. Print the details of occupied blocks.
4. Recursively print the contents of the root directory.

### **12.17.4 Implementation Summary**

This function provides a comprehensive overview of the file system's current state, aiding in debugging and monitoring.

## **12.18 printDirectoryContents**

### **12.18.1 Purpose**

To print the contents of a specified directory recursively.

### **12.18.2 Use Cases**

- Viewing directory structure and contents.
- Debugging and verifying file system integrity.

### **12.18.3 Algorithm**

1. Iterate through the directory entries.
2. Print the details of each file and subdirectory.
3. Recursively print the contents of subdirectories.

### **12.18.4 Implementation Summary**

This function retrieves and displays the contents of a specified directory and its subdirectories, aiding in directory management and debugging.

## **12.19 countFilesAndDirectories**

### **12.19.1 Purpose**

To count the number of files and directories within a specified directory table recursively.

### **12.19.2 Use Cases**

- Gathering statistics about the file system.
- Verifying directory and file counts.

### **12.19.3 Algorithm**

1. Iterate through the directory entries.
2. Increment the file or directory count based on the entry type.
3. Recursively count the contents of subdirectories.

### **12.19.4 Implementation Summary**

This function counts the number of files and directories within a specified directory table and its subdirectories, providing useful statistics.

## **12.20 chmodFile**

### **12.20.1 Purpose**

To change the permissions of a specified file.

### **12.20.2 Use Cases**

- Modifying access permissions for files.
- Enforcing security policies.

### **12.20.3 Algorithm**

1. Duplicate the file path to avoid modifying the original.
2. Split the path into directory and file name.
3. Find the specified directory.
4. Locate the file in the directory.
5. Update the file permissions.
6. Update the modification date.

### **12.20.4 Implementation Summary**

This function updates the permissions of a specified file, allowing changes to access controls.

## **12.21 addPassword**

### **12.21.1 Purpose**

To add or update the password for a specified file.

### **12.21.2 Use Cases**

- Securing files with password protection.
- Enforcing access controls.

### **12.21.3 Algorithm**

1. Duplicate the file path to avoid modifying the original.
2. Split the path into directory and file name.
3. Find the specified directory.
4. Locate the file in the directory.
5. Update the file password.
6. Update the modification date.

### **12.21.4 Implementation Summary**

This function adds or updates the password for a specified file, enhancing security by requiring a password for access.

## 13 Test Cases and Results

### 13.1 Test Case in PDF

```
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./makeFileSystem 1024 enre.data
Total size of superblock: 9700 bytes
Total size of Root Directory Table: 964 bytes
Total size of Free Block Bitmap: 512 bytes
Total size of FAT: 8192 bytes
Total size of Data Area (File Name Area + Data Blocks): 4194304 bytes (4 MB)
Total size of File System: 4213672 bytes
Available commands are: dir, mkdir, rmdir, duple2fs, write, read, del, chmod, addpw
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data mkdir /usr
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data mkdir /usr/ysa
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data mkdir /bin/ysa
Parent directory not found: /bin
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data write /usr/ysa/file1.txt test2.txt
File created successfully
Allocating block 28 -> 29
Marking block 29 as end of chain
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data write /usr/file2.txt test2.txt
File created successfully
Allocating block 30 -> 31
Marking block 31 as end of chain
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data write /file3.txt test2.txt
File created successfully
Allocating block 32 -> 33
Marking block 33 as end of chain
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data dir /
Content of /:
Directory Name: /usr
File Name: file3.txt
File Size: 1465 bytes
Permissions: RW
Creation Date: 2024-06-02 13:56:31
Modification Date: 2024-06-02 13:56:31
Password Protected: No
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data del /usr/ysa/file1.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data duple2fs
Filesystem status:
Block count: 4096
Free blocks: 4064
Block size: 1024 bytes
First 26 blocks are reserved for file system
Number of files: 2
Number of directories: 2
Occupied blocks:
Directory: /usr (Block: 26)
Directory: /usr/ysa (Block: 27)
File: /usr/file2.txt (Block: 30)
Block 30: /usr/file2.txt (Next: 31)
Block 31: /usr/file2.txt (Next: 65535)
File: //file3.txt (Block: 32)
Block 32: //file3.txt (Next: 33)
Block 33: //file3.txt (Next: 65535)
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt linuxFile2.data
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ cmp linuxFile2.data test2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data chmod /usr/file2.txt -rw
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt linuxFile2.data
Read permission denied for file: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data chmod /usr/file2.txt +rw
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data addpw /usr/file2.txt test1234
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt linuxFile2.data
You did not provide correct password for file: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt linuxFile2.data test1234
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data duple2fs
Filesystem status:
Block count: 4096
Free blocks: 4064
Block size: 1024 bytes
First 26 blocks are reserved for file system
Number of files: 2
Number of directories: 2
Occupied blocks:
Directory: /usr (Block: 26)
Directory: /usr/ysa (Block: 27)
File: /usr/file2.txt (Block: 30)
Block 30: /usr/file2.txt (Next: 31)
Block 31: /usr/file2.txt (Next: 65535)
File: //file3.txt (Block: 32)
Block 32: //file3.txt (Next: 33)
Block 33: //file3.txt (Next: 65535)
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-2: General Test in PDF

All the tests in the assignment PDF have been completed and have successfully passed. The only operation not included in the assignment PDF, the rmdir operation, is shown in the figure in

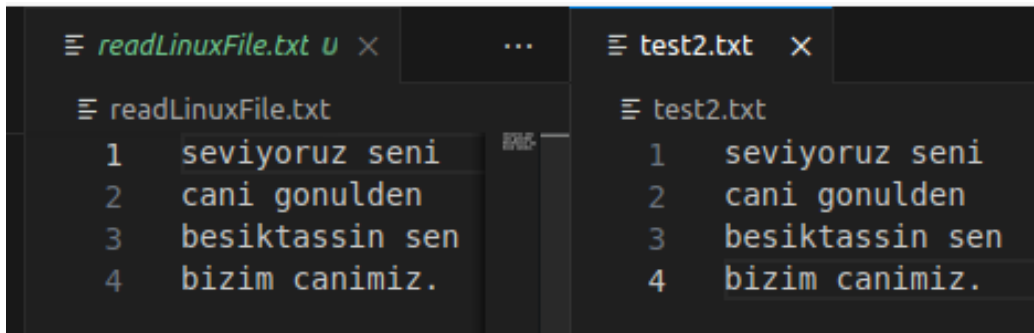
next page. The system is functioning successfully.

You can see that read operations cannot be performed on files that require a password, and we encounter an error message when we do not have the necessary permissions. Similarly, you can observe that an error is encountered when trying to create a file in a non-existent directory.

## 13.2 Read - Write Additional Proof

```
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt readLinuxFile.txt
You did not provide correct password for file: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt readLinuxFile.txt test1234
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-3: Read-Write Commands Proof



File	Line 1	Line 2	Line 3	Line 4
readLinuxFile.txt	1 seviyoruz seni	2 cani gonulden	3 besiktassin sen	4 bizim canimiz.
test2.txt	1 seviyoruz seni	2 cani gonulden	3 besiktassin sen	4 bizim canimiz.

Fig-4: Read-Write Commands Proof, Written and Read Files

On the previous page, you saw that the `test2.txt` file was successfully written and stored in our file system under the `/usr/ysa` directory as `file2.txt`. By using `cmp` Linux command to check, we confirmed that there are no differences, proving that the operations were successful. However, I also want to show you the contents of the created files to demonstrate that this process was completed successfully. As seen in Fig-4 and Fig-5, the contents of the file written to our file system and the contents read from the file system and written to Linux are identical.



```
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./makeFileSystem 1024 test.data
File system created and saved to test.data
Available commands are: dir, mkdir, rmdir, dunpe2fs, write, read, del, chmod, addpw

emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data mkdir /usr
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data write /ilhan/test.txt test2.txt
Invalid directory path: /ilhan
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data read /ilhan/test.txt atest2.txt
File not found: test.txt
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-4.1: Write-Read Command Proof, No Target Dir

As you see, if the path is not valid while performing a write operation, it gives error message successfully. Also, you can see if there is no such corresponding file in filesystem while performing read operation, it gives error message.

### 13.3 rmdir Proof

```
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./makeFileSystem 512 test.data
File system created and saved to test.data
Available commands are: dir, mkdir, rmdir, duple2fs, write, read, del, chmod, addpw

emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data mkdir /usr
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data mkdir /usr/ysa
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data duple2fs
Filesystem status:
Block count: 4096
Free blocks: 4020
Block size: 512 bytes
First 74 blocks are reserved for file system
Number of files: 0
Number of directories: 2
Occupied blocks:
Directory: /usr (Block: 74)
Directory: /usr/ysa (Block: 75)
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data rmdir /usr/ysa
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper test.data duple2fs
Filesystem status:
Block count: 4096
Free blocks: 4021
Block size: 512 bytes
First 74 blocks are reserved for file system
Number of files: 0
Number of directories: 1
Occupied blocks:
Directory: /usr (Block: 74)
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-5: rmdir Success Operation Test

```
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./makeFileSystem 1024 emre.data
File system created and saved to emre.data
Available commands are: dir, mkdir, rmdir, duple2fs, write, read, del, chmod, addpw

emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper emre.data mkdir /ysa
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper emre.data rmdir /ysa/asd
Directory not found
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper emre.data mkdir /ysa/usr
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper emre.data rmdir /ysa
Error while deleting directory: Directory is not empty
Directory not found
emre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-6: rmdir Error Operation Test

As you can see, the rmdir command also works successfully. Also, if target directory does not exist OR if the target directory is not empty, it gives error as Linux does.

### 13.4 addpw and chmod Error Handling

```
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data addpw /ilhan/file2.txt test1234
File not found: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data chmod /ilhan/file2.txt test1234
File not found: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-7: addpw and chmod Error Operation Test

```
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data addpw /usr/file2.txt newpsw
You did not provide correct password for file: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data addpw /usr/file2.txt newpsw test1234
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data read /usr/file2.txt linuxFile2.data newpsw
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data chmod /usr/file2.txt +rw
You did not provide correct password for file: file2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystemOper enre.data chmod /usr/file2.txt +rw newpsw
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-8: addpw and chmod Required Password

As you see, if the file does not exist, it gives error. Also, if file has a password, while performing a chmod or addPassword operation; you should provide correct password as you see in Fig-8.

### 13.5 Testing 4KB Data Write and Read

```
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystem0per enre2.data mkdir /usr
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystem0per enre2.data write /usr/f1.txt test2.txt
File created successfully
Allocating block 27 -> 28
Allocating block 28 -> 29
Allocating block 29 -> 30
Allocating block 30 -> 31
Marking block 31 as end of chain
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ ./fileSystem0per enre2.data read /usr/f1.txt linuxFile.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$ cmp linuxFile.txt test2.txt
enre@ubuntu:~/Downloads/cse312-operating-systems/HW2$
```

Fig-9: Write and Read 4KB data

As you see, if data is 4KB, it allocates 4 blocks from file system. And as you can see from `cmp` linux command we read it successfully.