GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CSE 312 /CSE504

Operating Systems

Homework #2 Report

171044073

Berkan AKIN

1. Requirement

1.1 Introduction and General Information

- Assignment topic: Designing and implementing a simplified FAT-like file system.
- Programming language: C or C++.
- Maximum file size: 4 MB.
- File system type: FAT12-like structure.

1.2 Part 1: File System Design

- The file system will use the FAT12 structure.
- Supported block sizes: 0.5 KB and 1 KB.
- File attributes:
    - File name (any length is possible).
    - Size.
    - Owner permissions (Read (R) and Write (W) only, no Execute (X)).
    - Last modification date and time.
    - File creation date and time.
    - Password protection.

Design Report Content:

- Define your directory table and directory entries.
- Define how and where you keep the free blocks.
- Define how you solve arbitrary length of file names.
- Define how you handle the permissions.
- Define how you handle password protection.
- Include the function names of your source code that handle the file system operations listed in Part 3.

1.3 Part 2: Creating an Empty File System

- Write a C/C++ program to create an empty file system.
- Sample command: makeFileSystem 1 mySystem.dat
    - 1: Block size of the file system in KB.
    - mySystem.dat: Linux file containing all the file system information.
- File system information includes: super block, data blocks, free blocks, directories, etc.

1.4. Part 3: File System Operations

- Program to be written: fileSystemOper
- File system operations and parameters:
    - dir <path>: Lists the contents of the specified directory.
    - mkdir <path>: Creates a new directory.
    - rmdir <path>: Removes a directory.
    - dumpe2fs: Provides information about the file system.

- write <path> <file>: Creates and writes data to a file in the file system.
- read <path> <file>: Reads data from a file in the file system.
- del <path> <file>: Deletes a file from the file system.
- chmod <path> <file> <permissions>: Changes the owner permissions of a file.
- addpw <path> <file> <password>: Adds a protection password to a file.

1.5 Test Commands

- Use the provided sample command sequence to test your file system operations:

```
shell
Kodu kopyala
makeFileSystem 4 mySystem.data
fileSystemOper fileSystem.data mkdir “\usr”
fileSystemOper fileSystem.data mkdir “\usr\ysa”
fileSystemOper fileSystem.data mkdir “\bin\ysa” ; Should print error!
fileSystemOper fileSystem.data write “\usr\ysa\file1” linuxFile.data
fileSystemOper fileSystem.data write “\usr\file2” linuxFile.data
fileSystemOper fileSystem.data write “\file3” linuxFile.data
fileSystemOper fileSystem.data dir “\” ; Should list 1 dir, 1 file
fileSystemOper fileSystem.data del “\usr\ysa\file1”
fileSystemOper fileSystem.data dumpe2fs
fileSystemOper fileSystem.data read “\usr\file2” linuxFile2.data
cmp linuxFile2.data linuxFile.data ; Should not print any difference
fileSystemOper fileSystem.data chmod “\usr\file2” -rw
fileSystemOper fileSystem.data read “\usr\file2” linuxFile2.data ;Should produce an error.
fileSystemOper fileSystem.data chmod “\usr\file2” +rw
fileSystemOper fileSystem.data addpw “\ysa\file2” test1234
fileSystemOper fileSystem.data read “\usr\file2” linuxFile2.data ;Should produce an error.
fileSystemOper fileSystem.data read “\usr\file2” linuxFile2.data test1234 ;Should be OK
```

In your report, you should explain these sections in detail and use the provided examples to test your file system. Prepare a report that details your file system design and implementation according to these requirements.

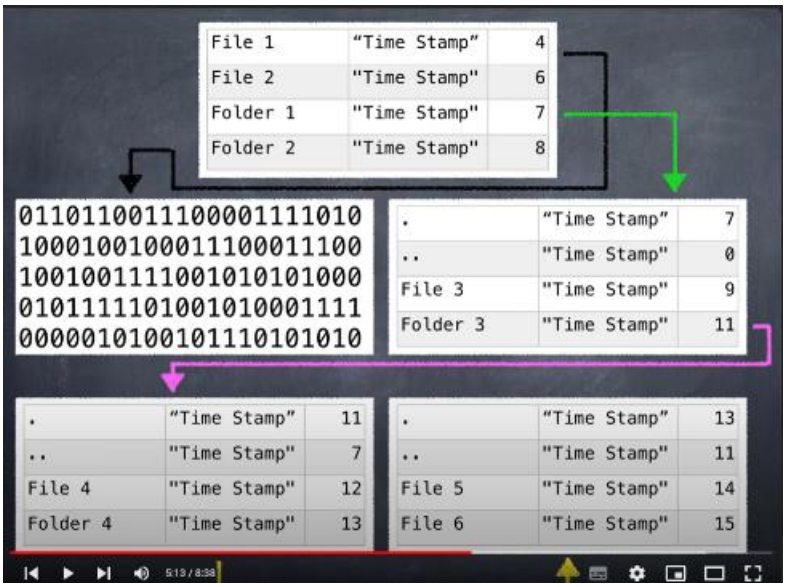
2.0 File System Design

The file system design is based on the simplified FAT12 structure, as described in the given code. The design includes the following features:

1. **File Attributes**:
    - **File Name**: The file name can be of any length, but for simplicity, it is limited to 20 characters (FILENAME_LEN).
    - **Size**: The size of the file is stored as an integer.
    - **Start Block**: The starting block of the file in the file system.
    - **Number of Blocks**: The number of blocks occupied by the file.
    - **Permissions**: Read (R) and Write (W) permissions are managed as a string of length 3.
    - **Creation and Modification Time**: Both are stored as strings of length 20.
    - **Password**: The password protection for the file is stored as a string of length 20.

```
typedef struct {
    char name[FILENAME_LEN];
    int size;
    int start_block;
    int num_blocks;
    char permissions[3]; // rw
    char creation_time[20];
    char modification_time[20];
    char password[PASSWORD_LEN];
} File;
```

2. **Directory Structure**:
    - The directory structure supports nested directories and files within directories.
    - Each directory can have a maximum of 100 subdirectories and files (MAX_FILES).
    - Directory information includes:
        - Name
        - Subdirectories (array of pointers to Directory)
        - Files (array of File)
        - Count of subdirectories and files.




```c
typedef struct Directory {
    char name[FILENAME_LEN];
    struct Directory *subdirectories[MAX_FILES];
    File files[MAX_FILES];
    int subdirectory_count;
    int file_count;
} Directory;
```

3. **File Allocation Table (FAT12)**:
    - The FAT12 table is implemented as an array of unsigned chars.
    - Each entry in the FAT12 table is 12 bits, and the table size is calculated as FAT_SIZE = (MAX_BLOCKS * 3 / 2).

4. **File System Metadata**:
    - The file system metadata includes:
        - Name
        - Block size
        - Number of blocks
        - Free blocks
        - Number of files and directories
        - Root directory
        - Data block for storing file data
        - FAT12 table.

```c
typedef struct {
    char name[FILENAME_LEN];
    int block_size;
    int num_blocks;
    int free_blocks;
    int num_files;
    int num_dir;
    Directory root;
    char *data;
    unsigned char fat[FAT_SIZE];  // FAT12 table
} FileSystem;
```

Design Report Content

1. Directory Table and Directory Entries: The directory table is defined within the Directory struct, which includes arrays for subdirectories and files. Each directory has a name, count of subdirectories, and count of files.

```c
typedef struct Directory {
    char name[FILENAME_LEN];
    struct Directory *subdirectories[MAX_FILES];
    File files[MAX_FILES];
    int subdirectory_count;
    int file_count;
} Directory;
```

2. Free Blocks Management: Free blocks are managed using the FAT12 table. Each entry in the FAT table indicates whether a block is free or occupied by a file. The allocateCluster function finds the first free block and marks it as occupied.

```c
int allocateCluster() {
    for (int i = 0; i < fs.num_blocks; i++) {
        int index = i * 3 / 2;
        int value;
        if (i % 2 == 0) {
            value = fs.fat[index] | (fs.fat[index + 1] << 8);
            value &= 0xFFF;
        } else {
            value = (fs.fat[index] >> 4) | (fs.fat[index + 1] << 4);
            value &= 0xFFF;
        }
        if (value == 0) {
            fs.free_blocks--;
            return i;
        }
    }
    return -1;
}
```

2. **Handling Arbitrary Length File Names:** The file name length is restricted to FILENAME_LEN (20 characters) for simplicity. This limitation ensures that file names are easy to manage within the directory structure.

```c
#define FILENAME_LEN 30
```

```c
char name[FILENAME_LEN];
```

4. Permissions Management: Permissions are managed as a string of length 3, containing characters for read (r) and write (w) permissions. The chmodFile function updates the permissions for a specified file.

```c
typedef struct {
    char name[FILENAME_LEN];
    int size;
    int start_block;
    int num_blocks;
    char permissions[3]; // rw
    char creation_time[20];
    char modification_time[20];
    char password[PASSWORD_LEN];
} File;
```

```c
void chmodFile(const char *path, const char *permissions) {
    char parent_path[MAX_PATH_LEN];
    char file_name[FILENAME_LEN];
    splitPath(path, parent_path, file_name);

    Directory *parentDir = navigateToDirectory(parent_path);
    if (parentDir == NULL) {
        printf("Directory not found: %s\n", parent_path);
        return;
    }

    for (int i = 0; i < parentDir->file_count; i++) {
        if (strcmp(parentDir->files[i].name, file_name) == 0) {
            strcpy(parentDir->files[i].permissions, permissions);
            printf("File permissions changed: %s ,%s\n", path,permissions);
            return;
        }
    }
    printf("File not found: %s\n", file_name);
}
```

Write permission

```c
    }
    if (strchr(parentDir->files[i].permissions, '+') == NULL) {
        printf("Read permission denied for file: %s\n", file_name);
        return;
    }
    if (strchr(parentDir->files[i].permissions, 'w') == NULL) {
        printf("Read permission denied for file: %s\n", file_name);
        return;
    }
        existingFile = &parentDir->files[i];
        break;
    }
}
```

Read Permission

```c
if (strchr(parentDir->files[i].permissions, '+') == NULL) {
    printf("Read permission denied for file: %s\n", file_name);
    return;
}
if (strchr(parentDir->files[i].permissions, 'r') == NULL) {
    printf("Read permission denied for file: %s\n", file_name);
    return;
}
```

5. **Password Protection Management:** Password protection for files is handled by storing the password as a string of length PASSWORD_LEN (20 characters). The addPassword function adds or updates the password for a specified file.

```c
if (strlen(parentDir->files[i].password) > 0 && strcmp(parentDir->files[i].password, password) != 0) {
    printf("Password incorrect for file: %s\n", file_name);
    return;
}
```

```c
void addPassword(const char *path, const char *password) {
    char parent_path[MAX_PATH_LEN];
    char file_name[FILENAME_LEN];
    splitPath(path, parent_path, file_name);

    Directory *parentDir = navigateToDirectory(parent_path);
    if (parentDir == NULL) {
        printf("Directory not found: %s\n", parent_path);
        return;
    }

    for (int i = 0; i < parentDir->file_count; i++) {
        if (strcmp(parentDir->files[i].name, file_name) == 0) {
            strcpy(parentDir->files[i].password, password);
            printf("Password added to file: %s\n", path);
            return;
        }
    }
    printf("File not found: %s\n", file_name);
}
```

6. File System Operations Functions: The report should include the function names handling various file system operations:

- **Creating File System**: createFileSystem
- **Saving File System**: saveFileSystem
- **Loading File System**: loadFileSystem
- **Writing to File**: writeToFile
- **Reading from File**: readFromFile
- **Deleting File**: deleteFile
- **Listing Directory Contents**: listDirectory
- **Changing File Permissions**: chmodFile
- **Adding Password to File**: addPassword
- **Creating Directory**: makeDirectory
- **Removing Directory**: removeDirectory
- **Dumping File System Information**: dumpFileSystem

```c
    while (1) {
        printf("> ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = 0; // Remove trailing newline

        if (sscanf(command, "makeFileSystem %d %s", &block_size, file_system_name) == 2) {
            createFileSystem(file_system_name, block_size);
            saveFileSystem(file_system_name);
        } else if (sscanf(command, "fileSystemOper %s %s %s %s %s", arg1, arg2, arg3, arg4, arg5) >= 2) {
            loadFileSystem(arg1); // Load the specified file system

            removeQuotes(arg3);

            if (strcmp(arg2, "dir") == 0) {
                listDirectory(arg3);
            } else if (strcmp(arg2, "dumpe2fs") == 0) {
                dumpFileSystem();
            } else if (strcmp(arg2, "write") == 0) {
                writeToFile(arg3, arg4,arg5);
            } else if (strcmp(arg2, "read") == 0) {
                readFromFile(arg3, arg4, arg5);
            } else if (strcmp(arg2, "del") == 0) {
                deleteFile(arg3);
            } else if (strcmp(arg2, "chmod") == 0) {
                chmodFile(arg3, arg4);
            } else if (strcmp(arg2, "addpw") == 0) {
                addPassword(arg3, arg4);
            } else if (strcmp(arg2, "mkdir") == 0) {
                makeDirectory(arg3); // Renamed to avoid conflict with POSIX mkdir
            } else if (strcmp(arg2, "rmdir") == 0) {
                removeDirectory(arg3);
            } else {
                printf("Unknown operation: %s\n", arg2);
            }
            saveFileSystem(arg1); // Save the file system after each operation
        } else {
            printf("Unknown command: %s\n", command);
        }
    }
```

The design of this file system aligns with the requirements of a simplified FAT-like structure. By adhering to the constraints and functionalities specified, the file system ensures efficient management of files and directories, supports password protection, and handles read/write permissions effectively.

Part 2: Creating an Empty File System

This section outlines the steps and code necessary to create an empty file system. The empty file system is created as a file with the specified size and includes the file system's metadata, FAT12 table, and data blocks.

1. Creating the File System

To create an empty file system, the createFileSystem function is used. This function initializes the necessary settings by taking the name and block size of the file system.

- fs.name: Stores the name of the file system.

- fs.block_size: Stores the block size (in KB).
- fs.num_blocks: Sets the maximum number of blocks (4096 blocks).
- fs.free_blocks: Initially indicates that all blocks are free.
- fs.num_files: Initially sets the number of files to zero.
- fs.num_dir: Indicates that there is one directory initially (the root directory).
- fs.root: Stores the root directory's information.
- fs.data: Allocates memory for storing file data.
- fs.fat: Initializes the FAT12 table.

```
if (sscanf(command, "makeFileSystem %d %s", &block_size, file_system_name) == 2) {
    createFileSystem(file_system_name, block_size);
    saveFileSystem(file_system_name);
```

2. Saving the File System

The created file system is saved to a specified file. The saveFileSystem function writes the file system's metadata, FAT12 table, directory structures, and data blocks to the file.

```
139 void createFileSystem(const char *name, int block_size) {
140     strcpy(fs.name, name);
141     fs.block_size = block_size * 1024;
142     fs.num_blocks = MAX_BLOCKS;  // 4 MB file system
143     fs.free_blocks = fs.num_blocks;
144     fs.num_files = 0;
145     fs.num_dir=1;
146     strcpy(fs.root.name, "\\");
147     fs.root.subdirectory_count = 0;
148     fs.root.file_count = 0;
149     fs.data = (char *)malloc(fs.num_blocks * fs.block_size);
150     memset(fs.data, 0, fs.num_blocks * fs.block_size);
151     memset(fs.fat, 0, FAT_SIZE);  // Initialize FAT12 table
152     printf("File system created with block size: %d KB\n", block_size);
153 }
154
```

```
92  void saveFileSystem(const char *filename) {
93      FILE *file = fopen(filename, "wb");
94      if (!file) {
95          perror("Failed to open file");
96          return;
97      }
98
99      fwrite(fs.name, sizeof(char), FILENAME_LEN, file);
100     fwrite(&fs.block_size, sizeof(int), 1, file);
101     fwrite(&fs.num_blocks, sizeof(int), 1, file);
102     fwrite(&fs.free_blocks, sizeof(int), 1, file);
103     fwrite(&fs.num_files, sizeof(int), 1, file);
104     fwrite(&fs.num_dir, sizeof(int), 1, file);
105
106     fwrite(fs.fat, sizeof(unsigned char), FAT_SIZE, file);  // Save FAT12 table
107     write_directory_to_file(&fs.root, file);
108
109     fwrite(fs.data, sizeof(char), fs.num_blocks * fs.block_size, file);  // Save data
110
111     fclose(file);
112     printf("File system saved to: %s\n", filename);
113 }
```

fwrite: Writes the file system's name, block size, number of blocks, number of free blocks, and number of files and directories to the file.

- write_directory_to_file: Writes the root directory and its subdirectories' information to the file.
- fwrite: Writes the data blocks to the file.

3. Loading the File System

The saved file system is loaded from a specified file. The loadFileSystem function reads the file system's metadata, FAT12 table, directory structures, and data blocks from the file.

```c
void loadFileSystem(const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (!file) {
        perror("Failed to open file");
        return;
    }

    fread(fs.name, sizeof(char), FILENAME_LEN, file);
    fread(&fs.block_size, sizeof(int), 1, file);
    fread(&fs.num_blocks, sizeof(int), 1, file);
    fread(&fs.free_blocks, sizeof(int), 1, file);
    fread(&fs.num_files, sizeof(int), 1, file);
    fread(&fs.num_dir, sizeof(int), 1, file);

    fread(fs.fat, sizeof(unsigned char), FAT_SIZE, file);  // Load FAT12 table
    read_directory_from_file(&fs.root, file);

    fs.data = (char *)malloc(fs.num_blocks * fs.block_size);
    fread(fs.data, sizeof(char), fs.num_blocks * fs.block_size, file);  // Load data

    fclose(file);
    printf("File system loaded from: %s\n", filename);
}
```

4. Example Usage

To create and save an empty file system, the following command is used:

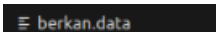
shell

Kodu kopyala

makeFileSystem 4 mySystem.dat

- 4: Specifies the block size (in KB).
- mySystem.dat: The name of the file to store the file system.

```
berkan@beko:~/Desktop/os_hw_2$ ./test
> makeFileSystem 1 mySystem.dat
File system created with block size: 1 KB
File system saved to: mySystem.dat
> makeFileSystem 4 berkan.data
File system created with block size: 4 KB
File system saved to: berkan.data
>
```



This command creates a file system with a 4 KB block size and saves it to a file named mySystem.dat.

The file system is created and saved using the createFileSystem and saveFileSystem functions. This process writes the file system's metadata, FAT12 table, and data blocks to the specified file, making the file system fully functional.

Part 3: File System Operations

In this section, the functions and operations necessary for managing the file system are outlined. The file system operations include creating directories, listing directory contents, writing to and reading from files, deleting files, changing file permissions, and adding passwords to files.

1. Listing Directory Contents

The listDirectory function lists the contents of a specified directory, including subdirectories and files with their attributes.

```c
void listDirectory(const char *path) {
    Directory *currentDir = navigateToDirectory(path);
    if (currentDir == NULL) {
        printf("Directory not found: %s\n", path);
        return;
    }

    for (int i = 0; i < currentDir->subdirectory_count; i++) {
        printf("dir  %s\n", currentDir->subdirectories[i]->name);
    }

    for (int i = 0; i < currentDir->file_count; i++) {
        printf("%s %d %s %s %s %s\n", currentDir->files[i].permissions, currentDir->files[i].size, currentDir->files[i].creation_time, currentDir->files[i].modification_time, currentDir
    }
}
void listBlocks(File *file) {
```

Test

```
> fileSystemOper fileSystem.data dir "\"
File system loaded from: fileSystem.data
dir  usr
+rw2024-06-08 09:45:36 107490 2024-06-08 09:45:36 2024-06-08 09:45:36 file3
File system saved to: fileSystem.data
>
```

```
> fileSystemOper fileSystem.data dir "\usr"
File system loaded from: fileSystem.data
dir  ysa
+rw2024-06-08 09:44:27 107490 2024-06-08 09:44:27 2024-06-08 09:44:27 file2
File system saved to: fileSystem.data
>
```

2. Creating and Removing Directories

The makeDirectory function creates a new directory, while the removeDirectory function removes an existing directory.

```c
} else if (strcmp(arg2, "mkdir") == 0) {
    makeDirectory(arg3);
} else if (strcmp(arg2, "rmdir") == 0) {
    removeDirectory(arg3);
} else {
    printf("Unknown operation: %s\n", arg2);
```

Mkdir Test

```
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data mkdir "\usr"
File system loaded from: fileSystem.data
Directory created: \usr
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data mkdir "\usr\ysa"
File system loaded from: fileSystem.data
Directory created: \usr\ysa
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data mkdir "\bin\ysa"
File system loaded from: fileSystem.data
Directory not found: \bin
File system saved to: fileSystem.data
```

Rmdir Test

```
> fileSystemOper fileSystem.data mkdir "\berkan"
File system loaded from: fileSystem.data
Directory created: \berkan
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data dir "\"
File system loaded from: fileSystem.data
dir  usr
dir  berkan
+rw2024-06-08 12:01:34 751 2024-06-08 12:01:34 2024-06-08 12:04:13 file4
+rw2024-06-08 12:05:22 751 2024-06-08 12:05:22 2024-06-08 12:06:38 file5
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data rmdir "\berkan"
File system loaded from: fileSystem.data
Directory deleted: \berkan
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data dir "\"
File system loaded from: fileSystem.data
dir  usr
+rw2024-06-08 12:01:34 751 2024-06-08 12:01:34 2024-06-08 12:04:13 file4
+rw2024-06-08 12:05:22 751 2024-06-08 12:05:22 2024-06-08 12:06:38 file5
File system saved to: fileSystem.data
>
```

3. Writing to and Reading from Files

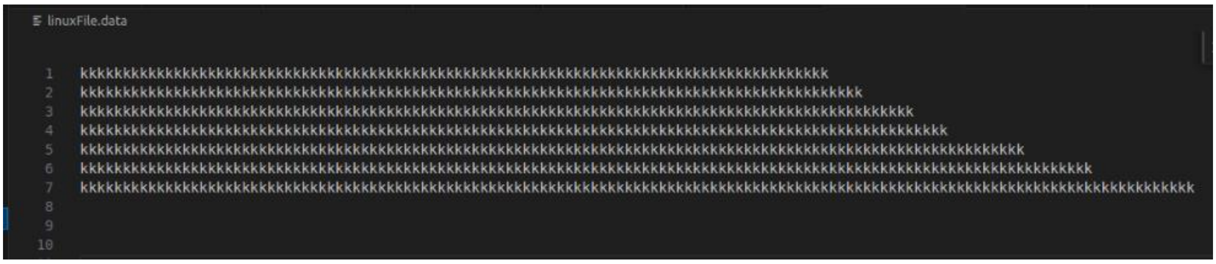
The writeToFile function writes data to a file in the file system, while the readFromFile function reads data from a file in the file system.

```
> fileSystemOper fileSystem.data write "\usr\ysa\file1" linuxFile.data
File system loaded from: fileSystem.data
File written: linuxFile.data
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data write "\usr\file2" linuxFile.data
File system loaded from: fileSystem.data
File written: linuxFile.data
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data write "\file3" linuxFile.dat
File system loaded from: fileSystem.data
Failed to open file: No such file or directory
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data dir "\"
```

Writing Test and Reading Test

```
> fileSystemOper fileSystem.data write "\file5" linuxFile.data
File system loaded from: fileSystem.data
File written: linuxFile.data
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data read "\usr\file5" abc.data
File system loaded from: fileSystem.data
File not found: file5
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data read "\file5" abc.data
File system loaded from: fileSystem.data
File read: abc.data
File system saved to: fileSystem.data
>
```

linucFile.data



abc.data


```
berkan@beko:~/Desktop/os_hw_2$ cmp abc.data linuxFile.data
berkan@beko:~/Desktop/os_hw_2$ 
```

```
} else if (strcmp(arg2, "write") == 0) {
    writeToFile(arg3, arg4,arg5);
} else if (strcmp(arg2, "read") == 0) {
    readFromFile(arg3, arg4, arg5);
```

4. Deleting Files

The deleteFile function deletes a file from the file system by freeing its blocks and removing it from the directory structure.

```
void deleteFile(const char *path) {
    char parent_path[MAX_PATH_LEN];
    char file_name[FILENAME_LEN];
    splitPath(path, parent_path, file_name);

    Directory *parentDir = navigateToDirectory(parent_path);
    if (parentDir == NULL) {
        printf("Directory not found: %s\n", parent_path);
        return;
    }

    for (int i = 0; i < parentDir->file_count; i++) {
        if (strcmp(parentDir->files[i].name, file_name) == 0) {
            int current_block = parentDir->files[i].start_block;
            while (current_block != 0xFFF) {
                int next_block = getFATValue(current_block);
                updateFAT(current_block, 0);  // Mark as free
                current_block = next_block;
            }
            fs.free_blocks += parentDir->files[i].num_blocks;

            for (int j = i; j < parentDir->file_count - 1; j++) {
                parentDir->files[j] = parentDir->files[j + 1];
            }
            parentDir->file_count--;
            fs.num_files--;
            printf("File deleted: %s\n", file_name);
            fs.num_files--;
            return;
        }
    }
    printf("File not found: %s\n", file_name);
}
```

file3 deleted

```
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data dir "\"
File system loaded from: fileSystem.data
dir  usr
+rw2024-06-08 09:45:36 31648 2024-06-08 09:45:36 2024-06-08 11:59:24 file3
+rw2024-06-08 12:01:34 751 2024-06-08 12:01:34 2024-06-08 12:04:13 file4
+rw2024-06-08 12:05:22 751 2024-06-08 12:05:22 2024-06-08 12:06:38 file5
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data del "\file3"
File system loaded from: fileSystem.data
File deleted: file3
File system saved to: fileSystem.data
> > fileSystemOper fileSystem.data dir "\"
Unknown command: > fileSystemOper fileSystem.data dir "\"
> fileSystemOper fileSystem.data dir "\"
File system loaded from: fileSystem.data
dir  usr
+rw2024-06-08 12:01:34 751 2024-06-08 12:01:34 2024-06-08 12:04:13 file4
+rw2024-06-08 12:05:22 751 2024-06-08 12:05:22 2024-06-08 12:06:38 file5
File system saved to: fileSystem.data
>
```

5. Changing File Permissions

The chmodFile function changes the permissions of a specified file.

```
476  void chmodFile(const char *path, const char *permissions) {
477      char parent_path[MAX_PATH_LEN];
478      char file_name[FILENAME_LEN];
479      splitPath(path, parent_path, file_name);
480
481      Directory *parentDir = navigateToDirectory(parent_path);
482      if (parentDir == NULL) {
483          printf("Directory not found: %s\n", parent_path);
484          return;
485      }
486
487      for (int i = 0; i < parentDir->file_count; i++) {
488          if (strcmp(parentDir->files[i].name, file_name) == 0) {
489              strcpy(parentDir->files[i].permissions, permissions);
490              printf("File permissions changed: %s ,%s\n", path,permissions);
491              return;
492          }
493      }
494      printf("File not found: %s\n", file_name);
495  }
```

file2 was first opened with read and write privileges, but could not be opened after the authorization was obtained.

```
> fileSystemOper fileSystem.data read "\usr\file2" linuxFile2.data
File system loaded from: fileSystem.data
File read: linuxFile2.data
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data chmod "\usr\file2" -rw
File system loaded from: fileSystem.data
File permissions changed: \usr\file2 ,-rw
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data read "\usr\file2" linuxFile2.data
File system loaded from: fileSystem.data
Read permission denied for file: file2
File system saved to: fileSystem.data
>
```

6. Adding Passwords to Files

The addPassword function adds or updates a password for a specified file.

```
497  void addPassword(const char *path, const char *password) {
498      char parent_path[MAX_PATH_LEN];
499      char file_name[FILENAME_LEN];
500      splitPath(path, parent_path, file_name);
501
502      Directory *parentDir = navigateToDirectory(parent_path);
503      if (parentDir == NULL) {
504          printf("Directory not found: %s\n", parent_path);
505          return;
506      }
507
508      for (int i = 0; i < parentDir->file_count; i++) {
509          if (strcmp(parentDir->files[i].name, file_name) == 0) {
510              strcpy(parentDir->files[i].password, password);
511              printf("Password added to file: %s\n", path);
512              return;
513          }
514      }
515      printf("File not found: %s\n", file_name);
516  }
```

```
> fileSystemOper fileSystem.data addpw "\usr\file2" test1234
File system loaded from: fileSystem.data
Password added to file: \usr\file2
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data read "\usr\file2" linuxFile2.data
File system loaded from: fileSystem.data
Password incorrect for file: file2
File system saved to: fileSystem.data
> fileSystemOper fileSystem.data read "\usr\file2" linuxFile2.data test1234
File system loaded from: fileSystem.data
File read: linuxFile2.data
File system saved to: fileSystem.data
>
```

7. Dumping File System Information

The dumpFileSystem function provides detailed information about the file system, including block usage and file attributes.

```
443  void printDirectoryDetails(Directory *dir, int level) {
444      for (int i = 0; i < dir->file_count; i++) {
445          listBlocks(&dir->files[i]);
446          //for (int j = 0; j < level; j++)
447          printf("| %-20s | %10d | %11d | %7d | %-3s | %-20s | %-20s | %s\n",
448                 dir->files[i].name,
449                 dir->files[i].size,
450                 dir->files[i].start_block,
451                 dir->files[i].num_blocks,
452                 dir->files[i].permissions,
453                 dir->files[i].creation_time,
454                 dir->files[i].modification_time,
455                 strlen(dir->files[i].password) > 0 ? "Yes" : "No");
456      }
457
458      for (int i = 0; i < dir->subdirectory_count; i++) {
459          printDirectoryDetails(dir->subdirectories[i], level + 1);
460      }
461  }
462
463  void dumpFileSystem() {
464      printf("File System Name: %s\n", fs.name);
465      printf("Block size: %d bytes\n", fs.block_size);
466      printf("Total blocks: %d\n", fs.num_blocks);
467      printf("Free blocks: %d\n", fs.free_blocks);
468      printf("Total directory: %d\n", fs.num_dir);
469      printf("Total files: %d\n", fs.num_files);
470      printf("\n");
471      printf("| %-20s | %10s | %11s | %7s | %-3s | %-20s | %-20s | %-3s |\n",
472             "Name", "Size", "Start Block", "Blocks", "Permissions", "Creation Time", "Modification Time", "Password Protected");
473      printf("|----------------------|------------|-------------|---------|-----|----------------------|----------------------|-----|\n");
474      printDirectoryDetails(&fs.root, 1);
475  }
```

The file system operations outlined in this section provide comprehensive functionality for managing files and directories in the FAT12-like file system. These operations include listing directory contents, creating and removing directories, writing to and reading from files, deleting files, changing file permissions, and adding passwords to files. Each function is designed to interact with the file system's metadata, FAT12 table, and data blocks to ensure efficient and secure file management.