Gebze Technical University

Department of Computer Engineering

CSE312 Spring 2024

Operating Systems

Homework #02

Mehmet Mahir Kayadelen, Student no. 210104004252

In this report, I have implemented a file system structure. This file system, crafted using C++.

The core structures designed and discussed in this report include **SuperBlock**, **DirectoryEntry**, and **FATEntry**, each serving critical functions within our file system. **DirectoryEntry** facilitates the organization of files and directories, including attributes like names, sizes, and permissions. **FATEntry** is crucial for managing the actual data blocks and maintaining the integrity of files stored across multiple blocks.

The implementation covers file system operations coded in separate modules, namely **makeFileSystem.cpp** and **fileSystemOper.cpp**. These operations include creating and removing directories (**mkdir** and **rmdir**), managing file contents (**write**, **read**, **and del**), modifying access controls (**chmod**, **addpw**) , **dumpe2fs** for giving information about system and **dir** for listing directory contents.

The report details the testing phase where all functionalities, such as directory listings (dir), system dumps (dumpe2fs), and error handling mechanisms have been tested. The **complete test results and analysis are provided at the end of this report**, demonstrating the performance of the implemented file system under various scenarios given in assignment PDF.

1) fileSystem.h

SuperBlock Structure

SuperBlock holds essential metadata about the file system, such as block size , the total number of blocks and free blocks

```
// Structure for Super Block
struct SuperBlock {
    unsigned int blockSize;    // Block size in bytes
    unsigned int totalBlocks;  // Total number of blocks
    unsigned int freeBlocks;   // Number of free blocks
};
```

blockSize: Specifies the size of each block in the file system in bytes.

totalBlocks: The total number of blocks available in the file system.

freeBlocks: The number of blocks that are currently unallocated.

DirectoryEntry Structure

```
// Structure for Directory Entry
struct DirectoryEntry {
    char name[MAX_FILENAME]; // Keep file/dir name
    unsigned int size;
    unsigned char ownerPermissions; // permissions default : 0b00000011 -> 3 -> +rw
    time_t lastModification; // modification time
    time_t creation;
    char password[32]; // default '\0' null
    unsigned short startBlock; // Keeps first block of the file/dir
    unsigned short parentBlock; // Keep the parent directory's start block
    bool isDirectory; // Indicate if it's a directory
};
```

name: Stores the name of the file or directory.

size: Indicates the size of the file or directory.

ownerPermissions: Binary permissions for the owner (3 is read/write).

lastModification: Records the last modification time.

creation: Records the creation time of the file or directory.

password: An optional password for the file or directory, typically initialized to null.

startBlock: Points to the first block in the FAT where the file or directory starts.

parentBlock: Points to the block in the FAT where the parent directory starts.

isDirectory: A boolean flag that indicates whether the entry is a directory.

FATEntry Structure

```
// Structure for File Allocation Table Entry
struct FATEntry {
    unsigned short nextBlock; // 16 bit
};
```

nextBlock: Stores the index of the next block in the file. This is essential for linking blocks of a file scattered throughout the file system.

FileSystem Structure

```
// Structure for File System
struct FileSystem {
    SuperBlock superBlock;
    std::vector<DirectoryEntry> rootDirectory;
    std::vector<FATEntry> fat;
    std::vector<std::vector<char>> data;
};
```

superBlock: An instance of SuperBlock that stores metadata about the file system.

rootDirectory: A dynamic array that holds the directory entries starting from the root.

fat: The File Allocation Table, implemented as a dynamic array of FATEntry structures.

data: A dynamic array of data blocks, where each block is represented as a vector of characters.

2) makeFileSystem.cpp

Function: initializeFileSystem

```cpp
void initializeFileSystem(const char* fileName, unsigned int blockSize) {
    ofstream file(fileName, ios::binary);
    if (!file) {
        cerr << "Error creating file." << endl;
        return;
    }

    // Calculate number of blocks
    unsigned int totalBlocks;
    if(blockSize == 512)
        totalBlocks= (MAX_FS_SIZE / 2) / blockSize;
    else
        totalBlocks= MAX_FS_SIZE / blockSize;

    // Initialize Super Block
    SuperBlock superBlock;
    superBlock.blockSize = blockSize;
    superBlock.totalBlocks = totalBlocks;
    superBlock.freeBlocks = totalBlocks - 1; // one block used for super block
```

```cpp
// Write Super Block to file
file.write(reinterpret_cast<const char*>(&superBlock), sizeof(SuperBlock));

// Initialize and write FAT
FATEntry fatEntry;
fatEntry.nextBlock = FAT_ENTRY_FREE;  // Correctly initialize as free
for (unsigned int i = 0; i < totalBlocks; ++i) {
    file.write(reinterpret_cast<const char*>(&fatEntry), sizeof(FATEntry));
}

// Initialize and write Root Directory
DirectoryEntry emptyEntry = {0};
for (unsigned int i = 0; i < MAX_FILES; ++i) {
    file.write(reinterpret_cast<const char*>(&emptyEntry), sizeof(DirectoryEntry));
}

// Initialize and write Data Blocks
char emptyBlock[blockSize];
memset(emptyBlock, 0, blockSize);
for (unsigned int i = 0; i < totalBlocks; ++i) {
    file.write(emptyBlock, blockSize);
}

file.close();
cout << "File system initialized in " << fileName <<  "\nBlock size: "<<  superBlock.blockSize<<endl;
```

Parameters:

const char* **fileName**: The name of the file where the file system will be initialized.

unsigned int **blockSize**: The size of each block within the file system.

Functionality: This function initializes a new file system with the given block size and writes it to a file specified by fileName.

Steps within initializeFileSystem:

Calculate Number of Blocks:

Based on the provided blockSize, it calculates the total number of blocks (totalBlocks). This calculation is adjusted if the block size is 512 bytes to ensure half the maximum file system size is utilized.

Initialize Super Block:

A SuperBlock instance is created and initialized with the block size, total number of blocks, and the number of free blocks (which is one less than the total blocks to account for the super block itself).

Write Super Block:

The super block data is written to the file.

Initialize and Write FAT:

A FATEntry struct (fatEntry) is initialized with FAT_ENTRY_FREE to indicate that all blocks are initially free. This entry is repeatedly written to the file for each block.

Initialize and Write Root Directory:

An empty DirectoryEntry (emptyEntry) is created and written MAX_FILES times to set up an initially empty root directory.

Initialize and Write Data Blocks:

An array of chars (emptyBlock) of size blockSize is zero-initialized to represent an empty block. This empty block is written to the file for each block in the file system.

3) fileSystemOper.cpp

Function: mkdir

Steps within mkdir:

Path Validation:

Checks if the provided path is empty or does not start with a backslash (\). If the path is invalid, it prints an error message and returns immediately.

Path Segmentation:

Uses a stringstream to split the path into segments based on the backslash delimiter. The first backslash is skipped to start processing from the root.

Directory Traversal:

Initializes currentDir to nullptr to start at the root, and parentBlock is set to FAT_ENTRY_FREE (root directory). Iteratively processes each segment of the path to navigate through the directory structure.

Segment Processing:

For each segment in the path, the function searches through the entries in the rootDirectory of the FileSystem to find a match.It verifies whether the current directory entry is a directory and matches the segment name. Also, it checks if the parent directory block is correct (either the root or the last found directory).

Directory Creation Check:

If the directory segment is not found (found is false) and it is the last segment in the path (ss.eof() returns true), the function attempts to create a new directory.It looks for an empty directory entry (entry.name[0] == '\0') within the rootDirectory.

Setup New Directory:

If an empty entry is found, the segment name is copied to entry.name, and directory-related properties such as size, permissions, creation/modification times, and parent block are set.

The function then attempts to allocate a block for the directory by scanning the FAT (File Allocation Table) for a free entry (FAT_ENTRY_FREE).

When a free block is found, it is assigned to entry.startBlock, marked as FAT_ENTRY_EOF in the FAT to indicate the end of the file/directory chain, and the number of free blocks in the SuperBlock is decremented.

Error Handling:

If no free directory entry or free block in the FAT is found during the directory creation phase, appropriate error messages are displayed.

Function: rmdir

Steps within rmdir:

Path Validation:

Checks if the path is empty or does not start with a backslash (\). If the path is invalid, an error message is displayed, and the function returns immediately.

Path Segmentation:

Utilizes a stringstream to break the path into segments using the backslash as a delimiter. The leading backslash is skipped for processing from the root.

Directory Traversal:

Iterates through each segment in the path to navigate the directory structure.

Uses pointers currentDir and parentDir to keep track of the current directory being processed and its parent directory, respectively. parentBlock is initially set to FAT_ENTRY_FREE, indicating the root.

Segment Processing:

Searches for the directory segment within the root or the current directory. If found, updates the currentDir and parentDir accordingly. If a directory segment is not found, it prints an error message and exits.

Empty Directory Check:

Before attempting to remove the directory, verifies that no entries (files or directories) exist within the target directory. If any entries are found, an error message is displayed, and the function returns.

Directory Entry Clearance:

Clears the directory entry by setting all its attributes to zero or appropriate default values (like FAT_ENTRY_FREE for block pointers). This effectively "removes" the directory from the file system visually and structurally.

FAT Cleanup:

Iterates through the File Allocation Table (FAT) entries linked to the directory to free up the associated blocks. This involves traversing the chain of blocks starting from the directory's startBlock and marking each as free (FAT_ENTRY_FREE).

Increment Free Blocks:

Increments the count of free blocks in the superBlock to reflect the newly freed blocks.

Function: write

Steps within write:

Path Validation:

Checks if the provided path is empty or does not start with a backslash. If invalid, it prints an error message and returns.

Input File Opening:

Opens the external data file in binary mode for reading. If the file cannot be opened, it outputs an error message and returns.

Data Size Determination:

Seeks to the end of the file to determine its size and then resets to the beginning. Calculates the number of blocks needed based on the file size and block size in the file system.

Block Allocation:

Checks if there are enough free blocks to store the file. If not, it prints an error message and returns.Allocates blocks in the file system by finding free blocks in the FAT, marking them as used, and chaining them together to form the file.

Data Writing:

Reads data from the external file and writes it block by block into the allocated blocks in the file system.

Directory Entry Update:

Calls createOrUpdateDirectoryEntry to update the file system's directory structure, reflecting the new or updated file entry.

Function read:

Steps within read:

Path Validation:

Checks if the provided path is empty or does not start with a backslash (\). If invalid, it prints an error message and returns.

Output File Creation:

Opens an output file stream in binary mode. If the file cannot be created, it outputs an error message and returns.

Path Segmentation:

Utilizes a stringstream to break the path into segments using the backslash as a delimiter, starting from after the leading backslash.

Directory Traversal:

Iteratively processes each segment in the path to navigate the directory structure, tracking the current directory (currentDir) and its starting block (parentBlock).

Segment Processing:

Searches for the directory segment within the root or the current directory. If found, updates currentDir and parentBlock. If a directory segment is not found before reaching the end of the path, it prints an error message and exits.

File Lookup:

If the path traversal completes without finding the target directory (i.e., the last segment is supposed to be a file), the function searches for the file in the current or parent directory based on the last segment's name and the last known parent block.

Data Reading:

Once the file is found, it reads the data from the file system block by block. The function uses the file allocation table (FAT) to navigate through the blocks that make up the file.

Data from each block is written to the output file until all data specified by the file size has been written or the end of the file chain (FAT_ENTRY_EOF) is reached.

Function del:

Steps within del:

Path Validation:

Checks if the provided path is empty or does not start with a backslash (\). If invalid, it prints an error message and returns.

Path Segmentation:

Uses a stringstream to break the path into segments using the backslash as a delimiter, starting after the leading backslash.

Directory Traversal:

Iteratively processes each segment in the path to navigate through the directory structure. Uses currentDir to keep track of the current directory and parentBlock for the starting block of the current directory.

Segment Processing:

Searches for the directory segment within the root or the current directory. If found, updates currentDir and parentBlock. If a directory segment is not found before reaching the end of the path, it prints an error message and exits.

File Lookup:

If the path traversal completes without finding the target directory (i.e., the last segment is supposed to be a file), the function searches for the file in the current or parent directory based on the last segment's name and the last known parent block.

File Deletion:

Once the file is found, it frees all blocks allocated to the file by following the chain of blocks starting from the file's startBlock in the FAT, marking each as free (FAT_ENTRY_FREE).

Increments the count of free blocks in the superBlock for each block freed.

Clears the directory entry for the file by zeroing out its memory, effectively removing all trace of the file from the directory.

Error Handling:

If the file is not found at the expected location, it prints a "File not found" error message and returns.

Function chmod:

Steps within chmod:

Path Validation and Segmentation:

Initializes a stringstream to break the path into segments using the backslash as a delimiter, starting from after the leading backslash.

Directory Traversal:

Iteratively processes each segment in the path to navigate through the directory structure. Uses currentDir to keep track of the current directory or file and updates parentBlock as it progresses.

Segment Processing:

Searches for the directory or file segment within the current directory or root. If found, updates currentDir and parentBlock. If not found before reaching the end of the path, it prints an error message and exits.

Permission Parsing and Application:

Checks if the permissions string starts with a valid modifier ('+' or '-'). If not, prints an error message and returns.

Iterates through the permission characters (after the first character) and modifies the permissions of the found directory entry accordingly:

For 'r' (read permission), sets or clears the respective bit in ownerPermissions based on whether '+' or '-' is used.

For 'w' (write permission), similarly adjusts the write permission bit.

Error Handling:

If the currentDir is not found (i.e., no file or directory entry is available at the end of the path), it prints a "File not found" error.

If an invalid permission character is encountered (not 'r' or 'w'), it prints an "Invalid permission flag" error.

Function addpw:

Steps within addpw:

Path Validation:

Checks if the provided path is empty or does not start with a backslash (\). If invalid, it prints an error message and returns.

Path Segmentation:

Uses a stringstream to break the path into segments using the backslash as a delimiter, starting from after the leading backslash.

Directory Traversal:

Iteratively processes each segment in the path to navigate through the directory structure. Uses currentDir to keep track of the current directory or file and updates parentBlock as it progresses.

Segment Processing:

Searches for the directory or file segment within the current directory or root. If found, updates currentDir and parentBlock. If not found before reaching the end of the path, it prints an error message and exits.

Password Addition or Update:

If the target directory or file is found (currentDir is not nullptr), the password is added or updated:

Copies the provided password into the password field of the DirectoryEntry using strncpy, ensuring that it does not exceed the buffer size (31 characters). It explicitly sets the last character of the password buffer to '\0' to ensure null termination.

If no directory entry is found by the end of the path (i.e., currentDir is nullptr), it prints a "File or directory not found for password addition" error.

Function dir:

Steps within dir:

Path Handling:

If the path is empty or just the root directory ("\\"), it sets parentBlock to FAT_ENTRY_FREE, indicating the root directory is the target.

Directory Traversal:

If not targeting the root, uses a stringstream to split the path into directory names, ignoring the leading backslash.

Iteratively processes each segment to navigate through the directory structure:

Searches for each directory segment in the rootDirectory. If found, updates currentDir and parentBlock. If not found, prints an error message and exits.

Listing Directory Contents:

Sets up column headers using cout and setw for formatting, including columns for Name, Size, Permissions, Last Modified, Creation Date, and Password Protection.

Iterates over all entries in fs.rootDirectory:

For each entry that is a child of the target directory (parentBlock matches), prints the entry details:

Name: The name of the file or directory.

Size: The size of the file or directory.

Permissions: Displays the binary representation of the permissions using bitset.

Last Modified and Creation Date: Uses a hypothetical formatTime function to format the timestamps. This function would convert time_t values to a readable string (function definition not provided in your code).

Password Protected: Indicates if a password is set (yes/no).

Error Handling:

If the specified path does not lead to a valid directory, an error message is displayed, indicating that the directory was not found.

Function dumpe2fs:

Steps within dumpe2fs:

SuperBlock Information Display:

Begins by printing a header for SuperBlock information.

Displays the block size, total number of blocks, and the number of free blocks from the SuperBlock structure, providing an overview of the file system's capacity and usage.

Counting Files and Directories:

Iterates over all entries in the rootDirectory to count the number of files and directories:

Checks if the directory entry is not empty (entry.name[0] != '\0') to ensure the entry is active.

Increments fileCount or directoryCount based on whether the entry is a file or a directory.

Display File and Directory Counts:

Prints the total number of files and directories, providing a quick summary of the file system's contents.

File Information Display:

Iterates again over the rootDirectory, this time focusing on files (ignores directories).

For each file:

Prints a separator line for readability.

Displays the file's name, size, and permissions. Permissions are shown as a binary representation using bitset<2> for a two-bit permissions model (read and write).

Lists the blocks occupied by the file:

Starts from the startBlock and follows the chain of blocks in the FAT (File Allocation Table), printing each block number.

Continues until it reaches FAT_ENTRY_EOF (end of file marker) or FAT_ENTRY_FREE (although typically it should only end at FAT_ENTRY_EOF).

Blocks Occupied Display:

For each file, prints a list of all blocks occupied by the file. This detail is crucial for understanding how the file's data is distributed across the file system.

Function verifyPassword:

Steps within verifyPassword:

Path Validation:

Checks if the path is empty or does not start with a backslash (\). If invalid, prints an error message and returns false.

Path Segmentation:

Uses a stringstream to split the path into directory names, ignoring the leading backslash, facilitating navigation through the file system structure.

Directory Traversal:

Iteratively processes each segment in the path to navigate through the directory structure, using currentDir to keep track of the current directory or file.

For each segment, searches the rootDirectory for a matching directory entry. Updates currentDir and parentBlock if found. If not found and not at the end of the path, prints an error and returns false.

End of Path Processing:

At the end of the path, if the target is not found in the current directory context, it checks for the file by its name and parent block. This occurs when no further segments are left to process (ss.eof() returns true).

If the file or directory is found:

Checks if a password is set (entry.password[0] != '\0). If so, compares the provided password with the stored password and returns true if they match, otherwise false.

If no password is set, returns true immediately, indicating no password protection.

Error Handling:

If the file or directory is not found after processing the entire path, prints "File not found" and returns false.

Function verifyPermissions

Steps within verifyPermissions:

Path Validation:

Checks if the path is empty or does not start with a backslash (\). If invalid, prints an error message and returns false.

Path Segmentation:

Uses a stringstream to split the path into directory names, removing the leading backslash to facilitate navigation through the file system.

Directory Traversal:

Iteratively processes each segment in the path to navigate through the directory structure, using currentDir to keep track of the current directory or file and parentBlock for the block ID of the current directory.

Segment Processing:

Searches the rootDirectory for each segment. If the segment matches a directory and the parent block matches parentBlock, updates currentDir and parentBlock. If not found and not at the end of the path, prints an error and returns false.

End of Path Processing:

At the end of the path, when no more segments are left to process (ss.eof() returns true), it searches for the file or directory by its name and parent block.

If the file or directory is found:

Checks if the actual permissions (entry.ownerPermissions) include all the required permissions (requiredPermissions). This is done using a bitwise AND operation and comparing the result with requiredPermissions.

Returns true if the permissions match the requirements; otherwise, returns false.

Error Handling:

If the file or directory is not found after processing the entire path, prints "File not found" and returns false.

Tests:

```
mahir@DESKTOP-9RRECEV:~/OS/hw2$ g++ makeFileSystem.cpp -o makeFileSystem
mahir@DESKTOP-9RRECEV:~/OS/hw2$ g++ fileSystemOper.cpp -o fileSystemOper
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./makeFileSystem 1 mySystem.data
File system initialized in mySystem.data
Block size: 1024
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data mkdir "\usr"
Directory created
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data mkdir "\usr\ysa"
Directory created
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data mkdir "\bin\ysa"
Directory not found: bin
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data write "\usr\ysa\file1" test.data
File written successfully
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data write "\usr\file2" test.data
File written successfully
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data write "\file3" test.data
File written successfully
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data dir "\\"
Name                Size      PermissionsLast Modified       Creation Date       Password Protected
usr                 0         11        2024-06-07 22:43:30 2024-06-07 22:43:30 No
file3               1350      11        2024-06-07 22:44:04 2024-06-07 22:44:04 No
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data del "\usr\ysa\file1"
File deleted: \usr\ysa\file1
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data dumpe2fs
SuperBlock Information:
Block Size: 1024 bytes
Total Blocks: 4096
Free Blocks: 4089
Total Files: 2
Total Directories: 2
------------------------------
File: file2, Size: 1350, Permissions: 11
Blocks occupied: 5 6
------------------------------
File: file3, Size: 1350, Permissions: 11
Blocks occupied: 7 8
```

```
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data read "\usr\file2" testread.data
File read: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ cmp test.data testread.data
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data chmod "\usr\file2" -rw
Permissions updated for: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data read "\usr\file2" testread2.data
Permission denied: Insufficient permissions for operation on file: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data chmod "\usr\file2" +rw
Permissions updated for: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data addpw "\usr\file2" test1234
Password added for: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data read "\usr\file2" testread2.data
Access denied: Incorrect or no password provided for file: \usr\file2
mahir@DESKTOP-9RRECEV:~/OS/hw2$ ./fileSystemOper mySystem.data read "\usr\file2" testread2.data test1234
File read: \usr\file2
```

All operations (mkdir,rmdir,dumpe2fs,write,read,del,chmod,addpw) working properly.