CSE312 HW2

Eren Çakar
1901042656

Sunday 2nd June, 2024

Introduction

In this project, a file system similar to FAT12 file system is designed. Which has **16 bits** sized entries for the FAT and 512 bytes for the data blocks. The implemented file system supports recursive directories, file copying in both directions, setting permissions and passwords, and deleting directories and files.

Part 1: File System Design

Entry Table and Entries

The entry table is designed to store information about each file and directory data blocks. Each **entry** consists of 16 bits of which represent:

- **Directory flag**: The leftmost bit of the entry is to check if the entry is for a directory or not.
- **Read flag**: The second leftmost bit of the entry is to check if the entry has read permissions or not.
- **Write flag**: The third leftmost bit of the entry is to check if the entry has write permissions or not.
- **Address**: And the latter 13 bits are used for addressing the next entry of the file.

The FAT12 structure uses 12 bits to store entries which is a hassle to store on C arrays due to it being not divisible by 8. So instead in the design introduced in this report 16 bits are used instead.

Usage of 16 bits for entries enables having flags mentioned above to be stored in the entry itself. And with the remaining 13 bits the file system can address 8192 data blocks, which is the number of maximum data blocks that can be allocated on a 4MB disk using data blocks of size 512 bytes. With these reasons the FAT was structured to be an array of 16 bit entries

File Attributes and Data

To store the attributes regarding the file, the head data block is made use of. The head data block's first 44 bytes are reserved for these attributes, and the any following bytes are considered as the ASCII characters of the name of the file, which must finish with the null terminator character. The structure of the head data block is as follows:

- **Size**: Size of the file in bytes. This value is stored int the first 4 bytes of the block to represent an unsigned 32 bit integer.
- **File creation date and time**: Timestamp of file creation. Which is also an unsigned 32 bit integer and therefore stored in the next 4 bytes.
- **Last modification date and time**: Timestamp of the last modification. Which is also an unsigned 32 bit integer and therefore stored in the next 4 bytes.
- **Password**: Optional password for file protection. Since it is not stated that the passwords are free to be as long as the user likes, the maximum length of 32 characters is deemed to be sufficient for the passwords. Thus the next 32 bytes are reserved for the password of the file.
- **File name**: Up until here total of $4 + 4 + 4 + 32 = 44$ bytes are reserved. Starting from the 45th byte, all the bytes until a null terminator character (a byte of value zero) is reached are treated as the characters of the file name. This enables users to write file names as long as $512 - 44 - 1 = 467$ characters, which must suffice the user.

After the null terminator character is reached for the file name the rest of the head block is treated as data. For the any data block that comes after the head data block, they are all treated as whole blocks of data and do not have attributes mentioned in the bullet list above.

With the mentioned allocation of data blocks and FAT the total size (in bytes) of the file that represent the implemented file system is:

$$
\begin{aligned}
\texttt{diskSize} &= 4 \cdot 1024 \cdot 1024 = 4194304 \\
\texttt{blockSize} &= 512 \\
\texttt{\#ofBlokcs} &= \frac{\texttt{diskSize}}{\texttt{blockSize}} = 8192 \\
\texttt{FATSize} &= \texttt{\#ofBlocks} \cdot 2 = 8192 \cdot 2 = 16384 \\
\texttt{totalSize} &= \texttt{diskSize} + \texttt{FATSize} = 4194304 + 16384 = 4210688 \ \texttt{bytes}
\end{aligned}
$$

Which is proven by the screenshot of the details of the file on Linux:

Figure 1: File system file created by `makeFileSystem`

Free Blocks Management

Any data block whose FAT entry is set as `FAT_FREE` (which is zero) is treated as a free data block. And any data block whose FAT entry is not `FAT_FREE` is treated as the part of that file.

Handling Permissions

Permissions are stored as two separate bits for read (R) and write (W) permissions in the FAT entry. Operations check these bits to determine if the requested action is allowed.

Password Protection

Password protection is implemented by storing an optional password entry's head data block. Operations that require a password will prompt the user to enter it and verify against the stored hash.

Directories

A directory is basically a special type of file, whose directory flag is set to one. Any file whose directory flag is set to one is treated as a directory, whose structure is the same as a file. The attributes section for directories do not differ from files' at all.

The only difference of the directories is that they get to reserve only one data block, which means the maximum number of files that they can store is $\frac{512-44-1}{2} = 233$, which is deemed to be a considerable amount of files to store, thus one data block is deemed to be sufficient for directories, therefore their respective FAT entry of their head data block is always set to `FAT_END` value.

The other different part of directories is that their data is always a sequence of entries of head data blocks of files they store in them, this way a directory can contain another directory achieving recursive directory structure.

Root Directory

The implementation of the root directory is done by reserving the second data block as the root directory.

File System Metadata

The implementation of the super block is handled by reserving the first data block of the disk to store values for the super block, which are:

- **Total blocks**: Number of total blocks in the disk.
- **Free blocks**: Number of total free blocks in the disk.
- **Block size**: The size of a block in bytes.

Part 2: Creating an Empty File System

`makeFileSystem` program was written to create an empty file system in a Linux file. The program basically writes the binary data of the initialized C struct:

```
typedef struct
{
    uint16_t FAT[FAT_SIZE];
    uint8_t dataBlocks[TOTAL_BLOCKS][BLOCK_SIZE_512];
} FileSystem;
```

Static allocation of the FAT and data block arrays ensures the size of the written file system file does not change and is always 4 megabytes.

The program for creating the file system file consists of its `main` and `createFileSystem` functions, which are explained below.

main

The `main` function is the entry point of the `makeFileSystem` program that creates a file system with a specified block size and writes it to a file.

The function begins by checking if the number of command-line arguments (argc) is less than 3. If it is, the function prints a usage message to the standard error stream and returns a failure status. This check ensures that the user has provided the necessary arguments to create the file system.

Next, the function retrieves the block size and filename from the command-line arguments (argv). The user is supposed to enter the block size in kilobytes, which is then converted to the block size from a string to a float using the atof function.

The function then determines the actual block size in bytes based on the block size in kilobytes. If the given block size is 0.5, it sets the block size to `BLOCK_SIZE_512`. If the given block size is 1, it sets the block size to `BLOCK_SIZE_1024`.

The function then checks if the block size is either `BLOCK_SIZE_512` or `BLOCK_SIZE_1024`. If it isn't, the function prints an error message to the standard error stream and returns a failure status. This check ensures that the block size is either 0.5 KB or 1 KB, which are the supported block sizes.

If the block size is valid, the function calls the `createFileSystem` function to create the file system with the specified block size and filename. It then prints a success message to the standard output stream.

Finally, the function returns a success status.

createFileSystem

The `createFileSystem` function is used to create a new file system with a specified block size and write it to a file. The function takes two parameters: an integer blockSize representing the size of the blocks in the file system, and a string filename representing the name of the file to which the file system will be written.

The function begins by calling the `initFileSystem` function to initialize the file system with the specified block size. This function sets up the `fileSystem` data structure for the file system, which consists of the File Allocation Table (FAT) and the data blocks.

Next, the function attempts to open the specified file in binary write mode. If the file fails to open, the function prints an error message and terminates the program with a failure status.

Once the file is successfully opened, the function writes the file system to the file. It does this by calling the `fwrite` function with the address of the file system, the size of the file system, the number of elements to be written, and the file. This effectively serializes the file system and writes it to the file.

Finally, the function closes the file.

Part 3: File System Operations

The following operations were implemented for the file system:

Directory Listing (dir)

The `dir` function is used to list the contents of a directory in our custom file system. The function takes one parameter: a string path representing the path of the directory in our file system.

The function begins by calling the `getDirectoryFAT` function to retrieve the File Allocation Table (FAT) entry of the directory. If the directory does not exist (i.e., `getDirectoryFAT` returns `0xFFFF`), the function prints an error message saying the directory does not exist and returns.

Next, the function initializes an offset variable to 0. This variable is used to keep track of the current position in the data block of the directory.

The function then retrieves the data block of the directory by indexing into the `dataBlocks` array of the file system with the address of the directory (obtained by bitwise ANDing the `directoryFAT` with `FAT_ADDRESS_MASK`).

The function then retrieves the FAT entry of the first file or subdirectory in the directory by calling the `getFirstEntryFAT` function with the directory block and the address of the offset.

The function then enters a loop where it continues to retrieve and print the information of the files or subdirectories in the directory until it reaches the end of the directory (i.e., `getFirstEntryFAT` or `getNextEntryFAT` returns `0xFFFF`). In each iteration of the loop, if the FAT entry is not free (i.e., the file or subdirectory exists), the function calls the `printEntryInfo` function to print the information of the file or subdirectory. It then retrieves the FAT entry of the next file or subdirectory by calling the `getNextEntryFAT` function with the directory block and the address of the offset.

Make Directory (mkdir)

The `mkdir` function is used to create a new directory in our custom file system. The function takes one parameter: a string `path` representing the path of the directory to be created.

The function begins by declaring two character arrays: `parentDirectory` and `name`. It then calls the `parsePath` function to split the `path` into a directory path and a directory name.

Next, the function attempts to create the directory by calling the `createDirectory` function with the parsed directory path, directory name, and read and write permissions. If the directory creation is successful, `createDirectory` returns a value other than `0xFFFF`, and the function prints a success message.

If the directory creation fails (i.e., `createDirectory` returns `0xFFFF`), the function prints an error message. It then checks the global `fsError` variable to determine the specific error that occurred and prints a corresponding error message. The possible errors are:

not enough blocks to create the directory, the parent directory is full, the File Allocation Table (FAT) is full, the parent directory does not exist, or the directory already exists.

createDirectory

To further explain the inner workings of directory creation the function `createDirectory` mentioned above is explained here.

The createDirectory function is used to create a new directory. The function takes three parameters: a string parentDirectory representing the path of the parent directory, a string name representing the name of the new directory, and a permissions bitmask representing the permissions of the new directory.

The function begins by checking if there are any free blocks left in the file system by calling the getSuperBlockFreeBlocks function. If there are no free blocks, the function sets the global fsError variable to FS_NOT_ENOUGH_BLOCKS and returns 0xFFFF to indicate an error.

Next, the function retrieves the File Allocation Table (FAT) entry of the parent directory by calling the getDirectoryFAT function. If the parent directory does not exist (i.e., getDirectoryFAT returns 0xFFFF), the function sets fsError to FS_DIRECTORY_DOES_NOT_EXIST and returns 0xFFFF.

The function then retrieves the data block of the parent directory by indexing into the dataBlocks array of the file system with the address of the parent directory (obtained by bitwise ANDing the directoryFAT with FAT_ADDRESS_MASK).

The function then checks if the new directory already exists in the parent directory by calling the getEntryOffset function with the parent directory block and the new directory name. If the new directory already exists (i.e., getEntryOffset returns a value other than -1), the function sets fsError to FS_DIRECTORY_EXISTS and returns 0xFFFF.

The function then finds an available spot in the parent directory block to store the new directory by calling the getDirectoryAvailableSpot function. If the parent directory is full (i.e., getDirectoryAvailableSpot returns -1), the function sets fsError to FS_DIRECTORY_FULL and returns 0xFFFF.

The function then enters a loop where it searches for a free FAT entry to store the new directory. In each iteration of the loop, it checks if the current FAT entry is free. If it is, the function sets the FAT entry to indicate that it is the end of a directory with the specified permissions, adds the FAT entry to the parent directory block, sets the directory entry with the new directory name, current time as creation and modification time, and an empty description, decreases the number of free blocks in the super block by 1, and returns the FAT entry with the directory and permissions flags set.

If the function cannot find a free FAT entry (i.e., it has iterated through all the FAT entries), it sets fsError to FS_FAT_FULL and returns 0xFFFF.

Remove Directory (rmdir)

The function `rmdir` removes a directory from our file system. The function takes a single argument, path, which is a string representing the path of the directory to be removed.

The function begins by checking if the provided path is the root directory (denoted by "/"). If it is, the function prints an error message and returns, as the root directory cannot be deleted.

Next, the function declares two character arrays, `parentDirectory` and `name`, which will hold the parent directory's path and the name of the directory to be deleted, respectively. The `parsePath` function is then called to split the provided path into these two components.

The function then retrieves the File Allocation Table (FAT) entry for the parent directory using the `getDirectoryFAT` function. If the parent directory does not exist (indicated by a return value of `0xFFFF`), an error message is printed and the function returns.

The same process is repeated for the directory to be deleted. If it does not exist, an error message is printed and the function returns.

Next, the function retrieves the data block of the directory to be deleted from the file system's data blocks array. The `isDirectoryEmpty` function is then called to check if the directory is empty. If it is not, an error message is printed and the function returns, as non-empty directories cannot be deleted.

If all these checks pass, the function proceeds to delete the directory. It calls the `deleteFromDirectory` function to remove the directory's entry from its parent directory. It then marks the directory's FAT entry as free using the `setFATEntry` function and clears the directory's data block using the `memset` function.

Finally, the function prints a message indicating that the directory was deleted successfully.

deleteFromDirectory

To further explain the inner workings of the function `rmdir`, the function `deleteFromDirectory` is explained here.

The function `deleteFromDirectory` is used to delete an entry from a directory. The function takes two arguments: a pointer to a `uint8_t` (an 8-bit unsigned integer) named `directoryDataBlock`, which represents the data block of the directory from which an entry is to be deleted, and a `const char *` named `name`, which represents the name of the entry to be deleted.

The function begins by calling the `getEntryOffset` function, passing in `directoryDataBlock` and `name` as arguments. This function is expected to return the offset of the entry named `name` within the directory's data block. The offset is stored in the `int` variable `offset`.

Next, the function checks if `offset` is `-1`. If it is, this means that the `getEntryOffset` function did not find an entry named `name` in the directory's data block, so the `deleteFromDirectory` function immediately returns without doing anything.

If `offset` is not `-1`, this means that an entry named `name` was found in the directory's data block. The function then calls the `freeFAT` function, passing in `directoryDataBlock` and `offset` as arguments. This function is expected to free the File Allocation Table (FAT) entry corresponding to the entry at the given offset in the directory's data block, effectively deleting the entry from the directory.

Dump File System Information (dumpe2fs)

The function `dumpe2fs` doesn't take any arguments. This function is used to print out information about the file system, including the total number of blocks, the number of free blocks, the block size, and the File Allocation Table (FAT).

The function begins by declaring three `uint32_t` variables: `totalBlocks`, `freeBlocks`, and `blockSize`. It then calls the `getSuperBlock` function, passing the addresses of these three variables as arguments. The `getSuperBlock` function is expected to fill these variables with the total number of blocks in the file system, the number of free blocks, and the size of a block, respectively.

Next, the function prints out the total number of blocks, the number of free blocks, and the block size.

The function then declares an array of `uint8_t` named `continuationOf` with a size of `FAT_SIZE`, and initializes all its elements to `0`. This array is used to keep track of the FAT entries that are not addressing a head data block but are addressing a data block that is a continuation of a file.

The function then enters a loop that iterates over each entry in the FAT. For each entry, it checks if it's a continuation of a previous entry. If it is, and the entry is not the end of a file or directory, it updates the `continuationOf` array to indicate that the entry value in the position current entry addressing is a continuation of the current entry. It then prints the information of the file which this entry is addressing continuation of, by calling `printEntryInfo` with the value obtained by checking from which entry is the current entry continues.

If the current entry is not a continuation of a previous entry and it's not free, the function prints the entry and its information. If the entry is not the end of a file or directory, it updates the `continuationOf` array to indicate that the entry pointed is a continuation of the current entry.

In this way, the `dumpe2fs` function provides a detailed dump of the file system's state, including the status of all blocks and the structure of the FAT.

And any non-empty FAT entries that are not addressing the head data block but a part of it is printed as the same as the head data block addressing entry is printed, this is done to let user see what file allocates how many blocks instead of not printing non-head data block addressing entries at all.

Write File (write)

The function writeFile is used to write the contents of a file from a Linux system to a file in our custom file system. The function takes two arguments: a string named path, which represents the path of the file in the custom file system, and a string named linuxFile, which represents the path of the file on the Linux system.

The function begins by opening the Linux file in binary read mode. If the file fails to open, the function prints an error message and returns immediately.

Next, the function declares two character arrays: parentDirectory and name. It then calls the parsePath function, passing in path, parentDirectory, and name as arguments. This function is expected to parse the path into a parent directory path and a file name.

The function then calls the createFile function, passing in parentDirectory, name, and a permission mask that grants read and write permissions. This function is expected to create a new file in the custom file system and return the File Allocation Table (FAT) entry of the file's first data block. If the file fails to be created, the function prints an error message based on the value of fsError, closes the Linux file, and returns immediately.

The function then declares an int variable fileSize and initializes it to 0. This variable is used to keep track of the size of the file being written.

The function then enters a loop that continues until the end of the Linux file is reached. In each iteration of the loop, the function reads a block of data from the Linux file and writes it to a data block in the custom file system. If a free data block cannot be found, the function prints an error message, closes the Linux file, frees the data blocks allocated to the file in the custom file system, deletes the file's entry from its parent directory, and returns immediately.

After all the data has been written, the function closes the Linux file, sets the permissions of the file's first data block in the FAT to read and write, sets the size of the file's entry in its parent directory to fileSize, and prints a success message.

createFile

To further explain the inner workings of the writeFile function, the createFile function is explained here.

The createFile function is used to create a new file in our custom file system. It takes three arguments: a string named parentDirectory, which represents the path of the parent directory in the file system; a string named name, which represents the name of the file to be created; and a uint16_t named permissions, which represents the permission bitmask of the file.

The function begins by checking if there are any free blocks in the file system by calling the getSuperBlockFreeBlocks function. If there are no free blocks, it sets the global variable fsError to FS_NOT_ENOUGH_BLOCKS and returns 0xFFFF, indicating an error.

Next, the function gets the File Allocation Table (FAT) entry of the parent directory by calling the getDirectoryFAT function with parentDirectory as the argument. If the parent directory does not exist (indicated by a return value of 0xFFFF), it sets fsError to FS_DIRECTORY_DOES_NOT_EXIST and returns 0xFFFF.

The function then gets a pointer to the data block of the parent directory in the file system. It does this by indexing into the dataBlocks array of the file system with the address part of directoryFAT.

The function then checks if a file with the same name already exists in the parent directory by calling the getFileFAT function with parentDirectoryBlock and name as arguments. If such a file exists (indicated by a return value not equal to 0xFFFF), it sets fsError to FS_FILE_EXISTS and returns 0xFFFF.

The function then tries to find an available spot in the parent directory for the new file by calling the getDirectoryAvailableSpot function with parentDirectoryBlock as the argument. If the parent directory is full (indicated by a return value of -1), it sets fsError to FS_DIRECTORY_FULL and returns 0xFFFF.

The function then iterates over each entry in the FAT, starting from RESERVED_FATS and ending at FAT_SIZE - 1. For each entry, it checks if it's free. If it finds a free entry, it sets the entry to value obtained by ANDing FAT_END and permission bitmask by calling the setFATEntry function, adds the entry to the parent directory by calling the putFAT function, sets the directory entry for the new file by calling the setDirectoryEntry function, and returns the entry with the permissions applied.

If the function cannot find a free entry in the FAT, it sets fsError to FS_FAT_FULL and returns 0xFFFF.

Read File (read)

The readFile function is designed to read a file from our custom file system and write it to a file in a Linux system. The function takes two arguments: path, which is the path of the file in the custom file system, and linuxFile, which is the path of the file where the data will be written in the Linux system.

The function starts by declaring two character arrays, parentDirectory and name, which will hold the directory and the name of the file respectively. The parsePath function is then called to split the path into the directory and the file name.

Next, the function retrieves the File Allocation Table (FAT) entry for the parent directory using the getDirectoryFAT function. If the returned FAT entry is 0xFFFF, it means the parent directory does not exist, so an error message is printed and the function returns.

The function then retrieves a pointer to the block of data in the custom file system that represents the parent directory. It uses the getFileFAT function to get the FAT entry for the file. If the file does not exist (indicated by a FAT entry of 0xFFFF), an error message is printed and the function returns.

The function then checks if the file has read permissions using the checkForReadPermission function. If the file does not have read permissions, an error message is printed and the function returns. The function also checks if the file is password protected using the checkForPassword function. If the password is incorrect, an error message is printed and the function returns.

The function then opens the Linux file for writing. If the file cannot be opened, an error message is printed and the function returns.

The function then retrieves the size of the file in the custom file system using the getDirectoryEntrySize function. It then reads the data from the file in the custom file system and writes it to the Linux file. This is done in a loop that continues until all the data has been read from the file in the custom file system and written to the Linux file.

Finally, the Linux file is closed and a success message is printed.

Delete File (del)

The `deleteFile` function is designed to delete a file from our custom file system. The function takes one argument, `path`, which is the path of the file in the custom file system that should be deleted.

The function starts by declaring two character arrays, `parentDirectory` and `name`, which will hold the directory and the name of the file respectively. The `parsePath` function is then called to split the `path` into the directory and the file name.

Next, the function retrieves the File Allocation Table (FAT) entry for the parent directory using the `getDirectoryFAT` function. If the returned FAT entry is `0xFFFF`, it means the parent directory does not exist, so an error message is printed and the function returns.

The function then retrieves a pointer to the block of data in the custom file system that represents the parent directory. It uses the `getFirstEntryFAT` function to get the FAT entry for the first file in the directory. If the file does not exist (indicated by a FAT entry of `0xFFFF`), an error message is printed and the function returns.

The function then enters a loop where it compares the name of the file to be deleted with the names of the files in the directory. This is done using the `strncmp` function. If the names do not match, the function gets the FAT entry for the next file in the directory using the `getNextEntryFAT` function. If the file to be deleted does not exist in the directory, an error message is printed and the function returns.

Before the file is deleted, the function checks if the file is password protected using the `checkForPassword` function. If the password is incorrect, an error message is printed and the function returns.

Finally, the function deletes the file by freeing the FAT entry for the file using the `freeFAT` function and freeing the data blocks of the file using the `freeDataBlocks` function. A success message is then printed.

Change Permissions (chmod)

The chmodFile function is designed to change the permissions of a file in our custom file system for this command. The function takes two arguments: path, which is the path of the file in the custom file system, and permissions, which is a string that specifies the changes to the file's permissions.

The function starts by declaring two character arrays, parentDirectory and name, which will hold the directory and the name of the file respectively. The parsePath function is then called to split the path into the directory and the file name.

Next, the function retrieves the File Allocation Table (FAT) entry for the parent directory using the getDirectoryFAT function. If the returned FAT entry is 0xFFFF, it means the parent directory does not exist, so an error message is printed and the function returns.

The function then retrieves a pointer to the block of data in the custom file system that represents the parent directory. It uses the getFileFAT function to get the FAT entry for the file. If the file does not exist (indicated by a FAT entry of 0xFFFF), an error message is printed and the function returns.

The function then checks if the file is password protected using the checkForPassword function. If the password is incorrect, an error message is printed and the function returns.

The function then checks the first character of the permissions string. If it is a '+', the function enters a loop where it adds the specified permissions to the file. This is done by setting the corresponding bits in the file's FAT entry. If the permissions string contains an 'r', the read permission bit is set. If it contains a 'w', the write permission bit is set. If the permissions string contains any other character, an error message is printed and the function returns.

If the first character of the permissions string is a '-', the function enters a loop where it removes the specified permissions from the file. This is done by clearing the corresponding bits in the file's FAT entry. If the permissions string contains an 'r', the read permission bit is cleared. If it contains a 'w', the write permission bit is cleared. If the permissions string contains any other character, an error message is printed and the function returns.

Finally, the function updates the modification time of the file using the setDirectoryEntryModificationTime function, updates the FAT entry for the file using the updateFAT function, and prints a success message.

Add Password (addpw)

The addPassword function is designed to add a password to a file in our custom file system for this command. The function takes two arguments: path, which is the path of the file in the custom file system, and password, which is the password to be added to the file.

The function starts by declaring two character arrays, parentDirectory and name, which will hold the directory and the name of the file respectively. The parsePath function is then called to split the path into the directory and the file name.

Next, the function retrieves the File Allocation Table (FAT) entry for the parent directory using the getDirectoryFAT function. If the returned FAT entry is 0xFFFF, it means the parent directory does not exist, so an error message is printed and the function returns.

The function then retrieves the FAT entry for the file using the getFileFAT function. It does this by passing the block of data in the custom file system that represents the parent directory and the name of the file. If the file does not exist (indicated by a FAT entry of 0xFFFF), an error message is printed and the function returns.

The function then sets the password for the file using the setDirectoryEntryPassword function. It does this by passing the FAT entry for the file (with the address mask applied) and the password.

The function then updates the modification time of the file using the setDirectoryEntry-ModificationTime function. It does this by passing the FAT entry for the file (with the address mask applied) and the current time.

Finally, the function prints a success message indicating that the password has been added successfully.

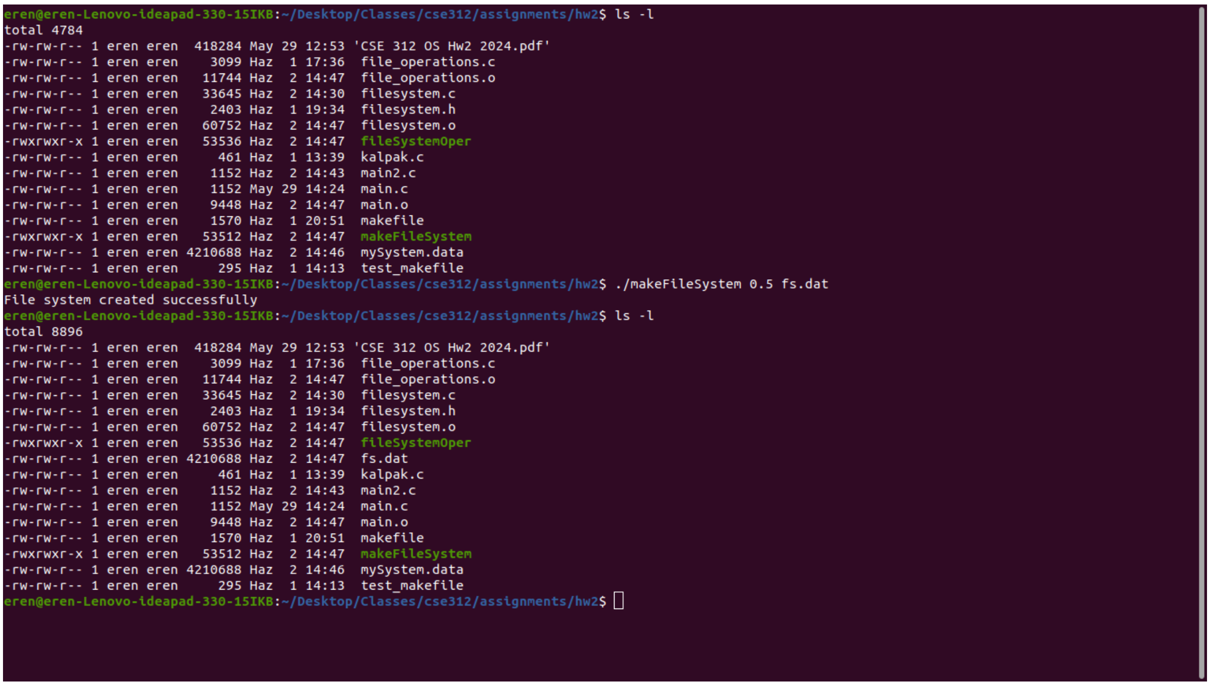
Tests

The design choice differences from the assignment document that affect the execution and that must be known of the programs are:

- **Path slash**: The slash used to indicate hierarchy among directories is / and not , since that is how Linux does.
- **Password asking**: The password is asked in the execution time when any operation is done that needs checking password on the file with a password, instead of accepting as an argument of the executable.

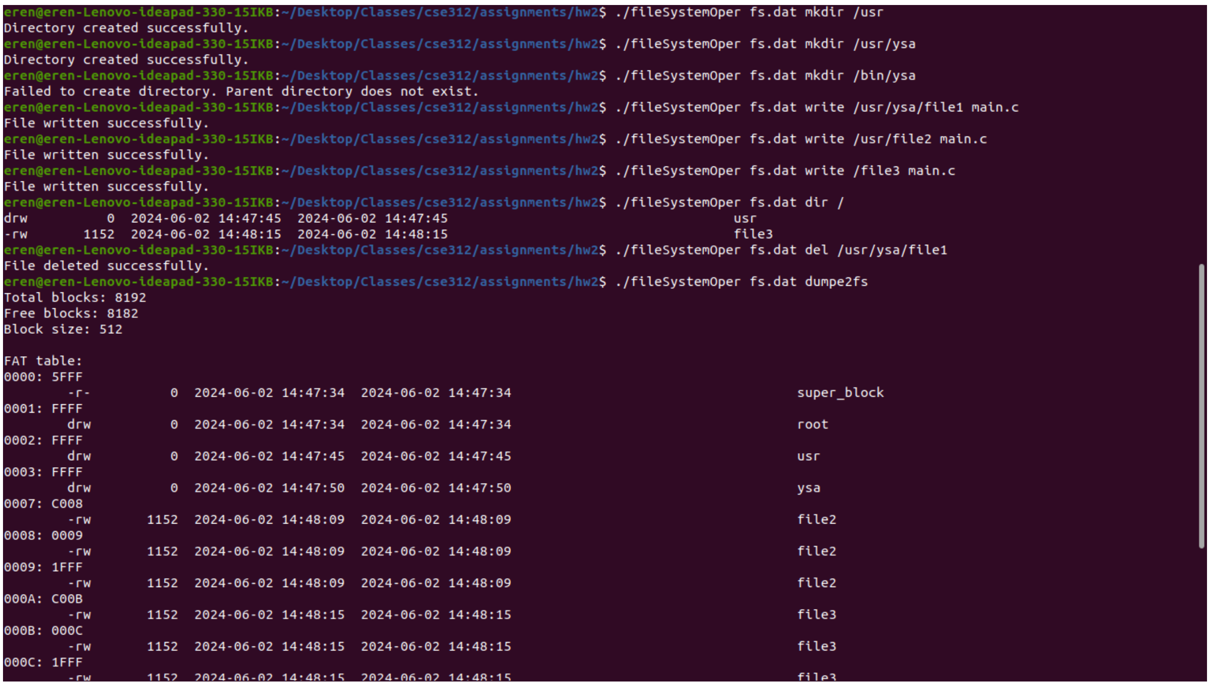
Given Test

The results of test lines given in the assignment document are shown below:



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ls -l
total 4784
-rw-rw-r-- 1 eren eren  418284 May 29 12:53 'CSE 312 OS Hw2 2024.pdf'
-rw-rw-r-- 1 eren eren    3099 Haz  1 17:36  file_operations.c
-rw-rw-r-- 1 eren eren   11744 Haz  2 14:47  file_operations.o
-rw-rw-r-- 1 eren eren   33645 Haz  2 14:30  filesystem.c
-rw-rw-r-- 1 eren eren    2403 Haz  1 19:34  filesystem.h
-rw-rw-r-- 1 eren eren   60752 Haz  2 14:47  filesystem.o
-rwxrwxr-x 1 eren eren   53536 Haz  2 14:47  fileSystemOper
-rw-rw-r-- 1 eren eren     461 Haz  1 13:39  kalpak.c
-rw-rw-r-- 1 eren eren    1152 Haz  2 14:43  main2.c
-rw-rw-r-- 1 eren eren    1152 May 29 14:24  main.c
-rw-rw-r-- 1 eren eren    9448 Haz  2 14:47  main.o
-rw-rw-r-- 1 eren eren    1570 Haz  1 20:51  makefile
-rwxrwxr-x 1 eren eren   53512 Haz  2 14:47  makeFileSystem
-rw-rw-r-- 1 eren eren 4210688 Haz  2 14:46  mySystem.data
-rw-rw-r-- 1 eren eren     295 Haz  1 14:13  test_makefile
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./makeFileSystem 0.5 fs.dat
File system created successfully
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ls -l
total 8896
-rw-rw-r-- 1 eren eren  418284 May 29 12:53 'CSE 312 OS Hw2 2024.pdf'
-rw-rw-r-- 1 eren eren    3099 Haz  1 17:36  file_operations.c
-rw-rw-r-- 1 eren eren   11744 Haz  2 14:47  file_operations.o
-rw-rw-r-- 1 eren eren   33645 Haz  2 14:30  filesystem.c
-rw-rw-r-- 1 eren eren    2403 Haz  1 19:34  filesystem.h
-rw-rw-r-- 1 eren eren   60752 Haz  2 14:47  filesystem.o
-rwxrwxr-x 1 eren eren   53536 Haz  2 14:47  fileSystemOper
-rw-rw-r-- 1 eren eren 4210688 Haz  2 14:47  fs.dat
-rw-rw-r-- 1 eren eren     461 Haz  1 13:39  kalpak.c
-rw-rw-r-- 1 eren eren    1152 Haz  2 14:43  main2.c
-rw-rw-r-- 1 eren eren    1152 May 29 14:24  main.c
-rw-rw-r-- 1 eren eren    9448 Haz  2 14:47  main.o
-rw-rw-r-- 1 eren eren    1570 Haz  1 20:51  makefile
-rwxrwxr-x 1 eren eren   53512 Haz  2 14:47  makeFileSystem
-rw-rw-r-- 1 eren eren 4210688 Haz  2 14:46  mySystem.data
-rw-rw-r-- 1 eren eren     295 Haz  1 14:13  test_makefile
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$
```

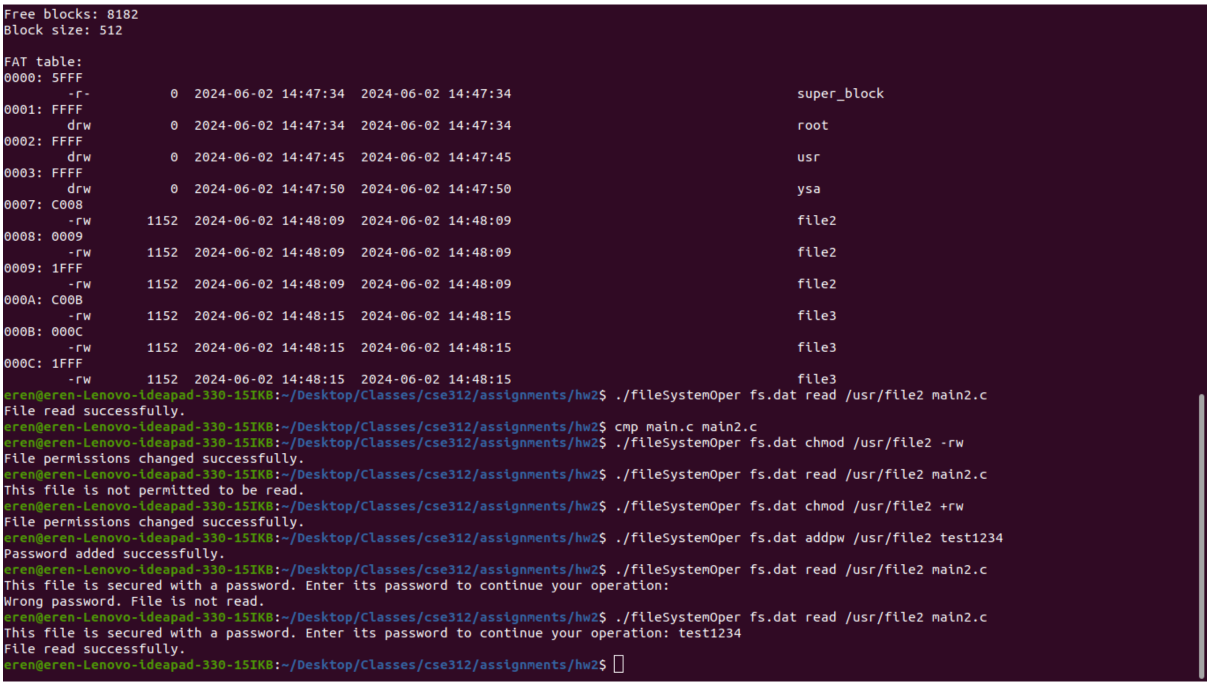
Figure 2: Creation of file system file



```
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat mkdir /usr
Directory created successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat mkdir /usr/ysa
Directory created successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat mkdir /bin/ysa
Failed to create directory. Parent directory does not exist.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat write /usr/ysa/file1 main.c
File written successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat write /usr/file2 main.c
File written successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat write /file3 main.c
File written successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat dir /
drw          0  2024-06-02 14:47:45  2024-06-02 14:47:45                              usr
-rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                              file3
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat del /usr/ysa/file1
File deleted successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat dumpe2fs
Total blocks: 8192
Free blocks: 8182
Block size: 512

FAT table:
0000: 5FFF
        -r-          0  2024-06-02 14:47:34  2024-06-02 14:47:34                      super_block
0001: FFFF
        drw          0  2024-06-02 14:47:34  2024-06-02 14:47:34                      root
0002: FFFF
        drw          0  2024-06-02 14:47:45  2024-06-02 14:47:45                      usr
0003: FFFF
        drw          0  2024-06-02 14:47:50  2024-06-02 14:47:50                      ysa
0007: C008
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                      file2
0008: 0009
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                      file2
0009: 1FFF
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                      file2
000A: C00B
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                      file3
000B: 000C
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                      file3
000C: 1FFF
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                      file3
```

Figure 3: First part of execution of fileSystemOper executable commands



```
Free blocks: 8182
Block size: 512

FAT table:
0000: 5FFF
        -r-          0  2024-06-02 14:47:34  2024-06-02 14:47:34                                    super_block
0001: FFFF
        drw          0  2024-06-02 14:47:34  2024-06-02 14:47:34                                    root
0002: FFFF
        drw          0  2024-06-02 14:47:45  2024-06-02 14:47:45                                    usr
0003: FFFF
        drw          0  2024-06-02 14:47:50  2024-06-02 14:47:50                                    ysa
0007: C008
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                                    file2
0008: 0009
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                                    file2
0009: 1FFF
        -rw       1152  2024-06-02 14:48:09  2024-06-02 14:48:09                                    file2
000A: C00B
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                                    file3
000B: 000C
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                                    file3
000C: 1FFF
        -rw       1152  2024-06-02 14:48:15  2024-06-02 14:48:15                                    file3
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat read /usr/file2 main2.c
File read successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ cmp main.c main2.c
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat chmod /usr/file2 -rw
File permissions changed successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat read /usr/file2 main2.c
This file is not permitted to be read.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat chmod /usr/file2 +rw
File permissions changed successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat addpw /usr/file2 test1234
Password added successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat read /usr/file2 main2.c
This file is secured with a password. Enter its password to continue your operation:
Wrong password. File is not read.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$ ./fileSystemOper fs.dat read /usr/file2 main2.c
This file is secured with a password. Enter its password to continue your operation: test1234
File read successfully.
eren@eren-Lenovo-ideapad-330-15IKB:~/Desktop/Classes/cse312/assignments/hw2$
```

Figure 4: Second part of execution of fileSystemOper executable commands

As it can be seen the outputs of the commands given in the assignment document, the fixed versions (some had typos and contradictions), are as expected in the assignment document.