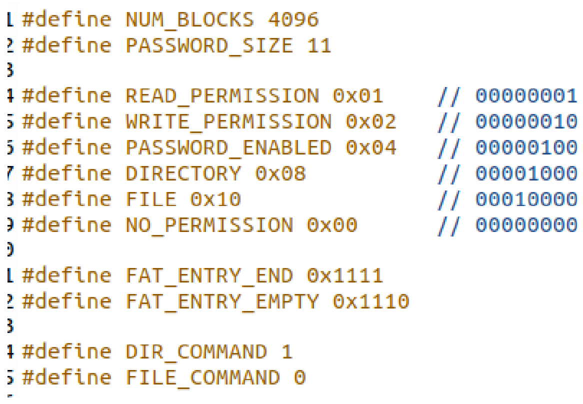
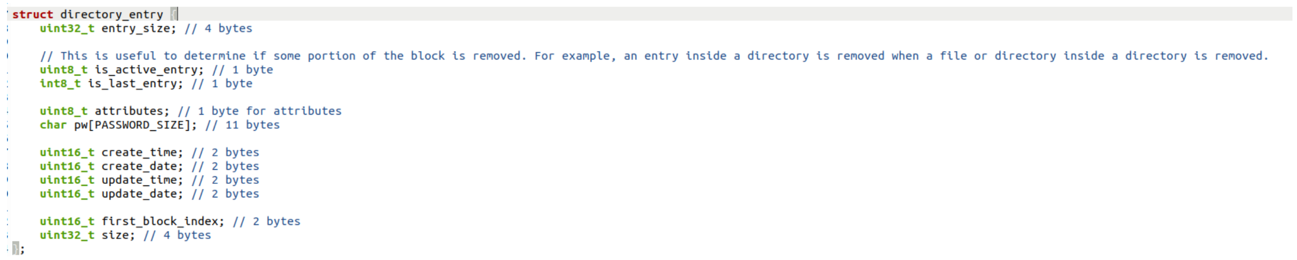
GEBZE TECHNICAL UNIVERSITY
CSE312 – HW2 FILE SYSTEM REPORT

Student Name: Emre Oytun
Student No: 200104004099

1) Directory Table and Directory Entries:



```c
#define NUM_BLOCKS 4096
#define PASSWORD_SIZE 11

#define READ_PERMISSION 0x01     // 00000001
#define WRITE_PERMISSION 0x02    // 00000010
#define PASSWORD_ENABLED 0x04    // 00000100
#define DIRECTORY 0x08           // 00001000
#define FILE 0x10                // 00010000
#define NO_PERMISSION 0x00       // 00000000

#define FAT_ENTRY_END 0x1111
#define FAT_ENTRY_EMPTY 0x1110

#define DIR_COMMAND 1
#define FILE_COMMAND 0
```



```c
struct directory_entry {
    uint32_t entry_size; // 4 bytes

    // This is useful to determine if some portion of the block is removed. For example, an entry inside a directory is removed when a file or directory inside a directory is removed.
    uint8_t is_active_entry; // 1 byte
    int8_t is_last_entry; // 1 byte

    uint8_t attributes; // 1 byte for attributes
    char pw[PASSWORD_SIZE]; // 11 bytes

    uint16_t create_time; // 2 bytes
    uint16_t create_date; // 2 bytes
    uint16_t update_time; // 2 bytes
    uint16_t update_date; // 2 bytes

    uint16_t first_block_index; // 2 bytes
    uint32_t size; // 4 bytes
};
```

“directory_entry” struct keeps necessary information for both a file and a directory.

- “entry_size” field keeps this entry’s total size with its name which comes after this entry. This is used for providing arbitrary file names.

- “is_active_entry” field is used in checking if this entry is active or not. It could be passive when an entry is removed and there can be fragments in the data table.

- “is_last_entry” field is used to check if this entry is the last entry in the directory table of a directory.

- “attributes” field keeps attributes like defined as above such as read, write permissions, password enabled and type of this entry as file or directory.

- “pw” field keeps password if any. It has limit of 10 characters at max.

- “first_block_index” keeps the index of first block as an entry point to the data of this entry. If it is file, the blocks contain file data; if it is directory, the blocks contain directory entries inside.

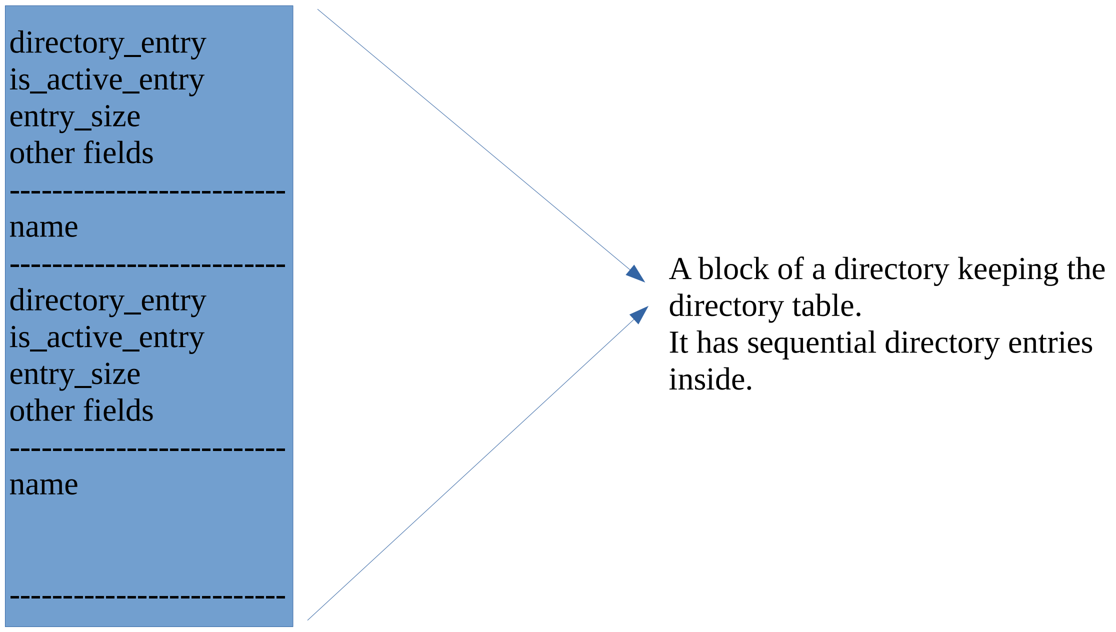
- “size” field keeps the size of the file if it is a file, or total size of the files if it is a directory.

NOTE: There is no seperate directory table region in the file system. The directories are kept as the same way with files such that they keep directory entries inside the blocks instead of file data. This is exactly the same way of keeping the directories in the FAT system.

2) Arbitrary Length of File Names:

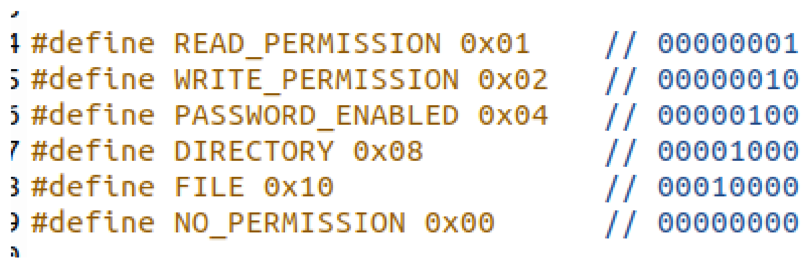
- Directory entries have “entry_size” field inside and when a directory entry is inserted its name is inserted after the entry. The name can be written by detecting the name size by subtracting size of struct directory entry from “entry_size”.

- Directory entries of a directory are kept sequentially inside blocks. When traversing the entries inside a directory, we need to traverse it sequentially. While traversing a directory, the program starts from the entry block and reads the entries one-by-one by first reading the struct and then reading the name. If there is not enough remaining space in the current block, it checks if this directory has another block from FAT and continue from there if available or finishes.



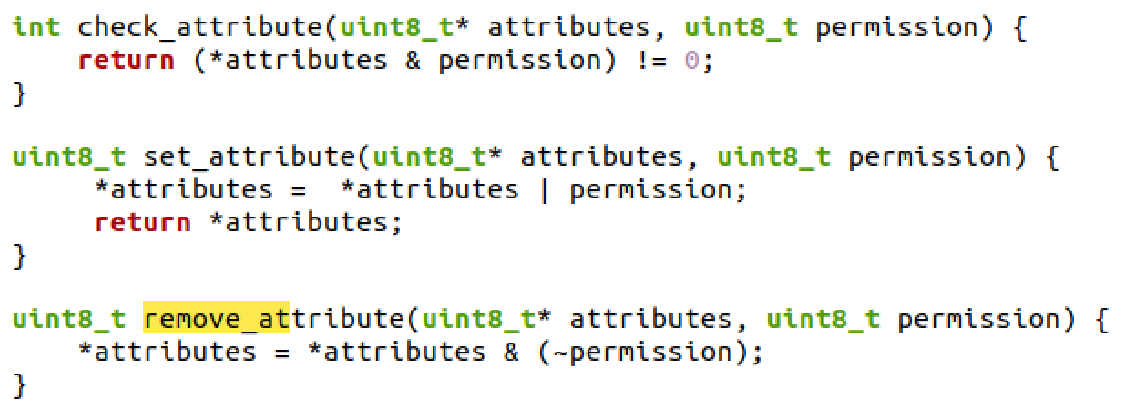
3) Handling Permissions:

- “attributes” field in directory entry keeps the permissions, password enabled information and type of the file such as file or directory.



```
#define READ_PERMISSION 0x01     // 00000001
#define WRITE_PERMISSION 0x02    // 00000010
#define PASSWORD_ENABLED 0x04    // 00000100
#define DIRECTORY 0x08           // 00001000
#define FILE 0x10                // 00010000
#define NO_PERMISSION 0x00       // 00000000
```

- “attributes” field keeps the permissions like that using bitwise operations. Set attribute, remove attribute, and check attribute functions are below.



```
int check_attribute(uint8_t* attributes, uint8_t permission) {
    return (*attributes & permission) != 0;
}

uint8_t set_attribute(uint8_t* attributes, uint8_t permission) {
     *attributes =  *attributes | permission;
     return *attributes;
}

uint8_t remove_attribute(uint8_t* attributes, uint8_t permission) {
    *attributes = *attributes & (~permission);
}
```

- “check_attribute” function: It checks if the given attribute like “READ_PERMISSION” is set in the attributes by AND’ing with the given attribute and checking if the result is not 0. If result is not 0, then it means bitwise 1-1 combination is found and attribute is set.

- “set_attribute” function: It sets the given attribute like “WRITE_PERMISSION”. It OR’s the current attributes with the given attribute.

- “remove_attribute” function: It removes the given attribute like “PASSWORD_ENABLED”. It AND’s the attribute with the NOT of the given attribute.

Permission Semantically:

- READ permission is checked in “read” command. When a file is being read, the permission is checked.
- WRITE permission is checked in “write” command. When a file already exists in the system, but a “write” command is entered for the same file, it checks the WRITE permission first. If it has permission, it overwrites the content of the file.

4) Password Protection:

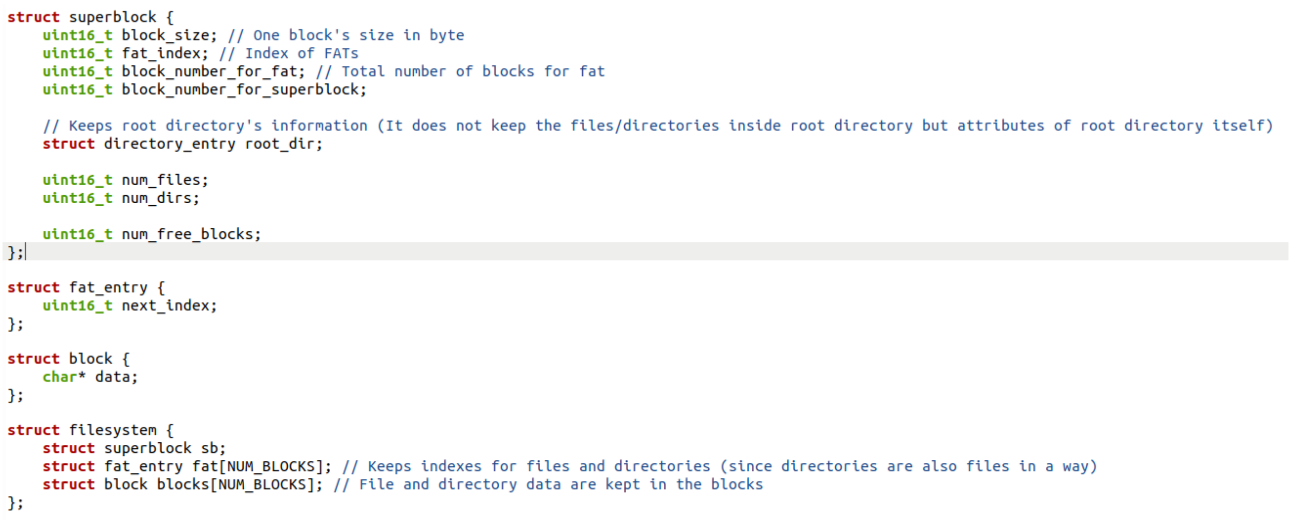
- “attributes” field in directory entry keeps PASSWORD_ENABLED attribute which indicates if this file is password protected or not. If this file is password protected, its password is checked before processing the command.

- “pw” field in directory entry keeps password of this file if any. It has 10 characters limitation.

- “addpw” command: It finds the entry of the file, updates its “attributes” and “pw” field and overwrites the found entry in the exact same position using “lseek” and “write” functions.

NOTE: In “addpw” command, if the file is already password protected, then the current password should be provided.

5) Superblock and Free Blocks:



```c
struct superblock {
    uint16_t block_size; // One block's size in byte
    uint16_t fat_index; // Index of FATs
    uint16_t block_number_for_fat; // Total number of blocks for fat
    uint16_t block_number_for_superblock;

    // Keeps root directory's information (It does not keep the files/directories inside root directory but attributes of root directory itself)
    struct directory_entry root_dir;

    uint16_t num_files;
    uint16_t num_dirs;

    uint16_t num_free_blocks;
};

struct fat_entry {
    uint16_t next_index;
};

struct block {
    char* data;
};

struct filesystem {
    struct superblock sb;
    struct fat_entry fat[NUM_BLOCKS]; // Keeps indexes for files and directories (since directories are also files in a way)
    struct block blocks[NUM_BLOCKS]; // File and directory data are kept in the blocks
};
```

In superblock:

- “block_size”: Keeps the block size of the system since this information is needed to figure out the filesystem.

- “fat_index”: Keeps the block number of FAT. Using “block_size” and this field, we can go to the offset of FAT.

- “block_number_for_fat”: Keeps total number of blocks allocated for FAT.

- “block_number_for_superblock”: Keeps total number of blocks allocated for superblock.

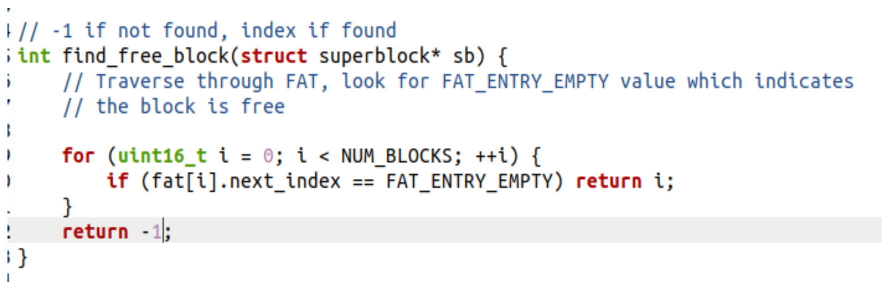
- “root_dir”: Root directory’s directory entry is kept inside the superblock so that we can find the block number of root directory and traverse through the root as entry point to our system.

- “num_files”: Keeps the total number of files in the system. 0 at the beginning.

- “num_dirs”: Keeps the total number of directories in the system. 1 at the beginning for root directory.

- “num_free_blocks”: Keeps the total number of remaining free blocks in the file system.

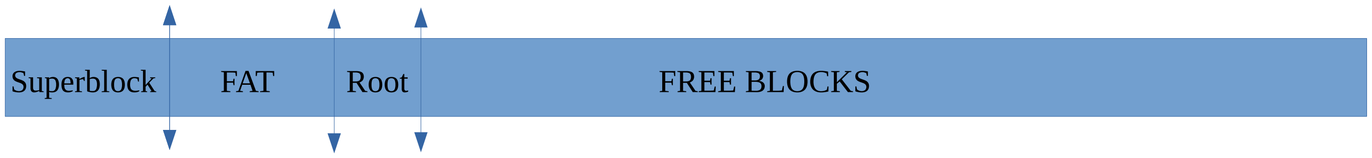
Free Blocks Handling:



```c
// -1 if not found, index if found
int find_free_block(struct superblock* sb) {
    // Traverse through FAT, look for FAT_ENTRY_EMPTY value which indicates
    // the block is free

    for (uint16_t i = 0; i < NUM_BLOCKS; ++i) {
        if (fat[i].next_index == FAT_ENTRY_EMPTY) return i;
    }
    return -1;
}
```

When a free block is needed, this function traverse through the FAT and tries to find an entry whose value is “FAT_ENTRY_EMPTY” which is “0x1110”. This value is selected because there are not many blocks in the FAT so we can use this value as an indicator of empty block.

6) General Structure of File System:



Superblock:

- It is at the beginning of the file system data. It contains necessary information to figure out the filesystem as I explained its fields above.
- It takes 5 blocks if block size is 1024, 9 blocks if block size is 512. These values are precisely calculated according to the size of superblock and blocks.

FAT:

- It comes after the superblock.
- Its starting block number is kept in superblock.
- It is used to figure out the blocks for a file or a directory starting from an entry index exactly same as FAT system.
- It takes 8 blocks if block size is 1024, 16 blocks if block size is 512. These values are precisely calculated according to the size of FAT and blocks since FAT has size of 2^12 entry * 2 bytes (16-bit) = 2^13 bytes = 8KB.

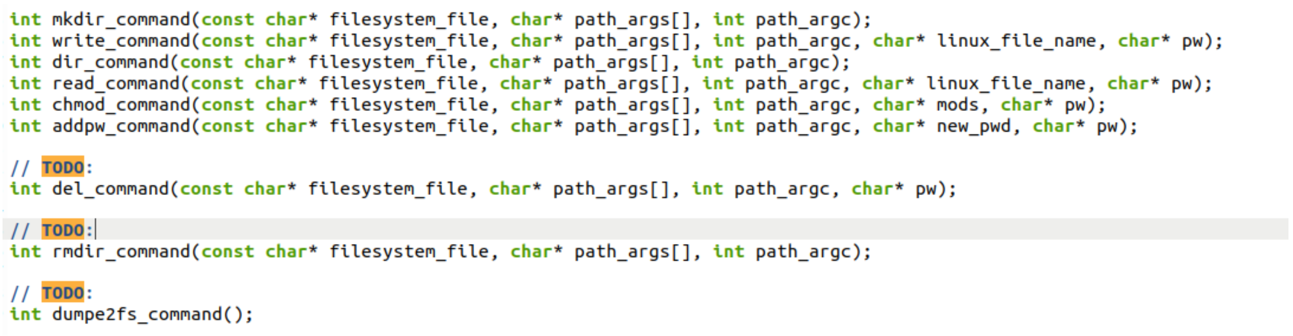
Root:

- It comes after the FAT.
- Its entry and starting block number is kept in superblock.
- It is used as an entry point while traversing our file system.

Free Blocks:

- Remaining blocks are free at the beginning. They have value of “FAT_ENTRY_EMPTY” (0x1110).
- When a file or data is created in the file system, one or more free blocks are allocated and FAT is updated accordingly.
- If we need to find free blocks, we need to traverse through the FAT as I explained with function screenshot above.
- A block keeps file data if it is file, or it keeps directory table as sequential directory entries if it is a directory.

7) Function Signatures for Commands in Part 3:



```
int mkdir_command(const char* filesystem_file, char* path_args[], int path_argc);
int write_command(const char* filesystem_file, char* path_args[], int path_argc, char* linux_file_name, char* pw);
int dir_command(const char* filesystem_file, char* path_args[], int path_argc);
int read_command(const char* filesystem_file, char* path_args[], int path_argc, char* linux_file_name, char* pw);
int chmod_command(const char* filesystem_file, char* path_args[], int path_argc, char* mods, char* pw);
int addpw_command(const char* filesystem_file, char* path_args[], int path_argc, char* new_pwd, char* pw);

// TODO:
int del_command(const char* filesystem_file, char* path_args[], int path_argc, char* pw);

// TODO:
int rmdir_command(const char* filesystem_file, char* path_args[], int path_argc);

// TODO:
int dumpe2fs_command();
```

- These functions are defined in “fileSystemOper.h” and implemented in “fileSystemOper.c”.

“mkdir_command”:
- Used in “mkdir” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is not present, then it creates the directory. Otherwise, it gives error.

“write_command”:
- Used in “write” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is not present, it creates and writes to file as specified in the PDF. Otherwise, it checks if the existing file has write permission. If it has write permission, it overwrites the data inside.

“dir_command”:
- Used in “dir” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is present and it is a directory, then it goes to directory table of this directory and traverses through the directory entries while printing the entry information. Otherwise, it gives error.

“read_command”:
- Used in “read” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is present and it is a file which has read permission, it reads the file and writes its content to the given file as its specified in the PDF while preserving the owner permissions. Otherwise, it gives error.

“chmod_command”:
- Used in “chmod” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is present, it changes its permission as given in the command. Otherwise, it gives error.

“addpw_command”:
- Used in “addpw” command.
- It traverses through the directories given in the path starting from the root directory.
- It checks whether there are such directories as in the path.
- If the last name in the path is present, it adds the password as given in the command. Otherwise, it gives error.

8) Running the Program:

- Use “make” to compile the program.
- Then enter your commands like:

```
./makeFileSystem 1 mysystem.data (1 – 1KB block size, 0.5 – 0.5KB block size)
./fileSystemOper mysystem.data dir "/"
```

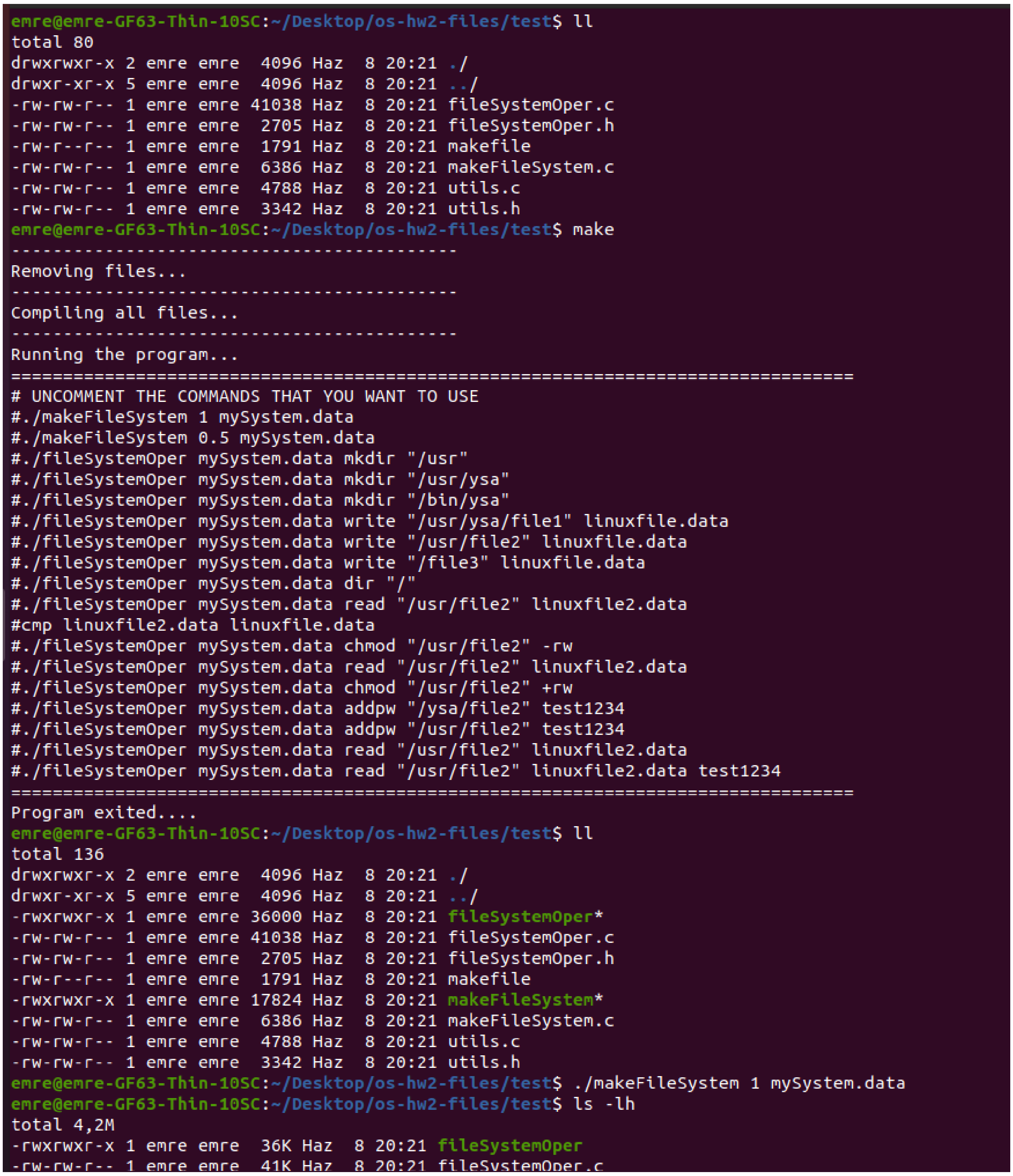
NOTE: You should write the path inside double quotes and the slash should be normal slash ‘/’ instead of backslash ‘\’ since ‘\’ is ignored in C while passing as argument.

Test Case Commands:

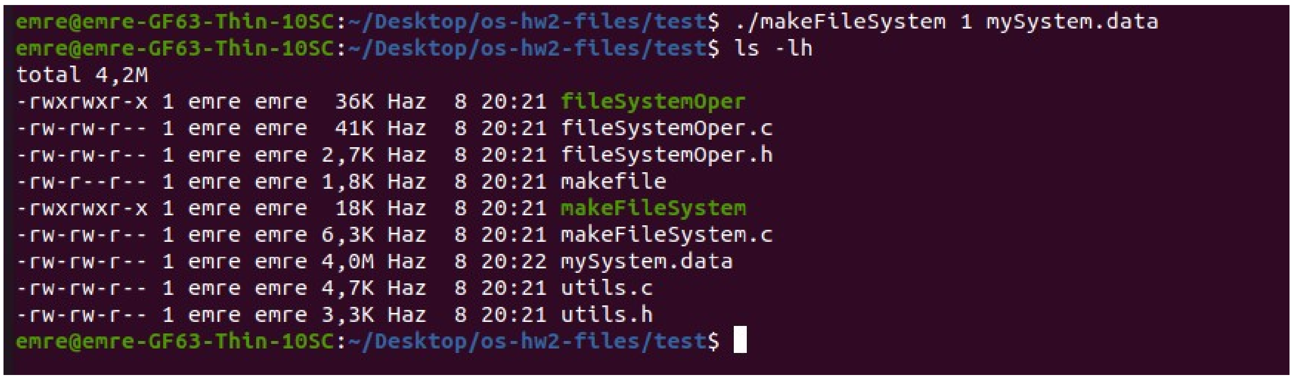
```
./makeFileSystem 1 mySystem.data
./makeFileSystem 0.5 mySystem.data

./fileSystemOper mySystem.data mkdir "/usr"
./fileSystemOper mySystem.data mkdir "/usr/ysa"
./fileSystemOper mySystem.data mkdir "/bin/ysa"
./fileSystemOper mySystem.data write "/usr/ysa/file1" linuxfile.data
./fileSystemOper mySystem.data write "/usr/file2" linuxfile.data
./fileSystemOper mySystem.data write "/file3" linuxfile.data
./fileSystemOper mySystem.data dir "/"
./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
cmp linuxfile2.data linuxfile.data
./fileSystemOper mySystem.data chmod "/usr/file2" -rw
./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
./fileSystemOper mySystem.data chmod "/usr/file2" +rw
./fileSystemOper mySystem.data addpw "/ysa/file2" test1234
./fileSystemOper mySystem.data addpw "/usr/file2" test1234
./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data test1234
```

8) Screenshots:

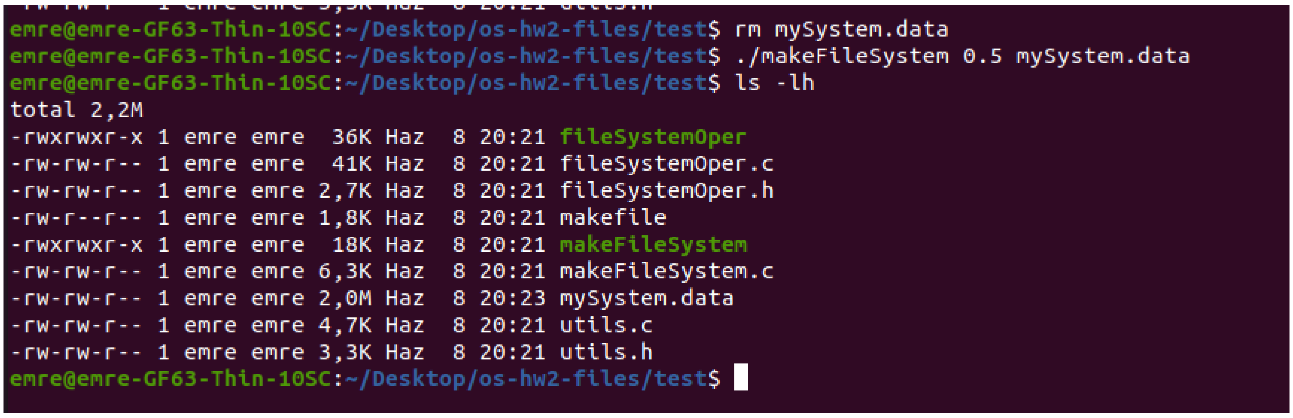
makeFileSystem:



```
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ll
total 80
drwxrwxr-x 2 emre emre  4096 Haz  8 20:21 ./
drwxr-xr-x 5 emre emre  4096 Haz  8 20:21 ../
-rw-rw-r-- 1 emre emre 41038 Haz  8 20:21 fileSystemOper.c
-rw-rw-r-- 1 emre emre  2705 Haz  8 20:21 fileSystemOper.h
-rw-r--r-- 1 emre emre  1791 Haz  8 20:21 makefile
-rw-rw-r-- 1 emre emre  6386 Haz  8 20:21 makeFileSystem.c
-rw-rw-r-- 1 emre emre  4788 Haz  8 20:21 utils.c
-rw-rw-r-- 1 emre emre  3342 Haz  8 20:21 utils.h
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ make
-------------------------------------------
Removing files...
-------------------------------------------
Compiling all files...
-------------------------------------------
Running the program...
================================================================================
# UNCOMMENT THE COMMANDS THAT YOU WANT TO USE
#./makeFileSystem 1 mySystem.data
#./makeFileSystem 0.5 mySystem.data
#./fileSystemOper mySystem.data mkdir "/usr"
#./fileSystemOper mySystem.data mkdir "/usr/ysa"
#./fileSystemOper mySystem.data mkdir "/bin/ysa"
#./fileSystemOper mySystem.data write "/usr/ysa/file1" linuxfile.data
#./fileSystemOper mySystem.data write "/usr/file2" linuxfile.data
#./fileSystemOper mySystem.data write "/file3" linuxfile.data
#./fileSystemOper mySystem.data dir "/"
#./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
#cmp linuxfile2.data linuxfile.data
#./fileSystemOper mySystem.data chmod "/usr/file2" -rw
#./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
#./fileSystemOper mySystem.data chmod "/usr/file2" +rw
#./fileSystemOper mySystem.data addpw "/ysa/file2" test1234
#./fileSystemOper mySystem.data addpw "/usr/file2" test1234
#./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
#./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data test1234
================================================================================
Program exited....
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ll
total 136
drwxrwxr-x 2 emre emre  4096 Haz  8 20:21 ./
drwxr-xr-x 5 emre emre  4096 Haz  8 20:21 ../
-rwxrwxr-x 1 emre emre 36000 Haz  8 20:21 fileSystemOper*
-rw-rw-r-- 1 emre emre 41038 Haz  8 20:21 fileSystemOper.c
-rw-rw-r-- 1 emre emre  2705 Haz  8 20:21 fileSystemOper.h
-rw-r--r-- 1 emre emre  1791 Haz  8 20:21 makefile
-rwxrwxr-x 1 emre emre 17824 Haz  8 20:21 makeFileSystem*
-rw-rw-r-- 1 emre emre  6386 Haz  8 20:21 makeFileSystem.c
-rw-rw-r-- 1 emre emre  4788 Haz  8 20:21 utils.c
-rw-rw-r-- 1 emre emre  3342 Haz  8 20:21 utils.h
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./makeFileSystem 1 mySystem.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ls -lh
total 4,2M
-rwxrwxr-x 1 emre emre  36K Haz  8 20:21 fileSystemOper
-rw-rw-r-- 1 emre emre  41K Haz  8 20:21 fileSystemOper.c
```



```
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./makeFileSystem 1 mySystem.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ls -lh
total 4,2M
-rwxrwxr-x 1 emre emre  36K Haz  8 20:21 fileSystemOper
-rw-rw-r-- 1 emre emre  41K Haz  8 20:21 fileSystemOper.c
-rw-rw-r-- 1 emre emre 2,7K Haz  8 20:21 fileSystemOper.h
-rw-r--r-- 1 emre emre 1,8K Haz  8 20:21 makefile
-rwxrwxr-x 1 emre emre  18K Haz  8 20:21 makeFileSystem
-rw-rw-r-- 1 emre emre 6,3K Haz  8 20:21 makeFileSystem.c
-rw-rw-r-- 1 emre emre 4,0M Haz  8 20:22 mySystem.data
-rw-rw-r-- 1 emre emre 4,7K Haz  8 20:21 utils.c
-rw-rw-r-- 1 emre emre 3,3K Haz  8 20:21 utils.h
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ 
```

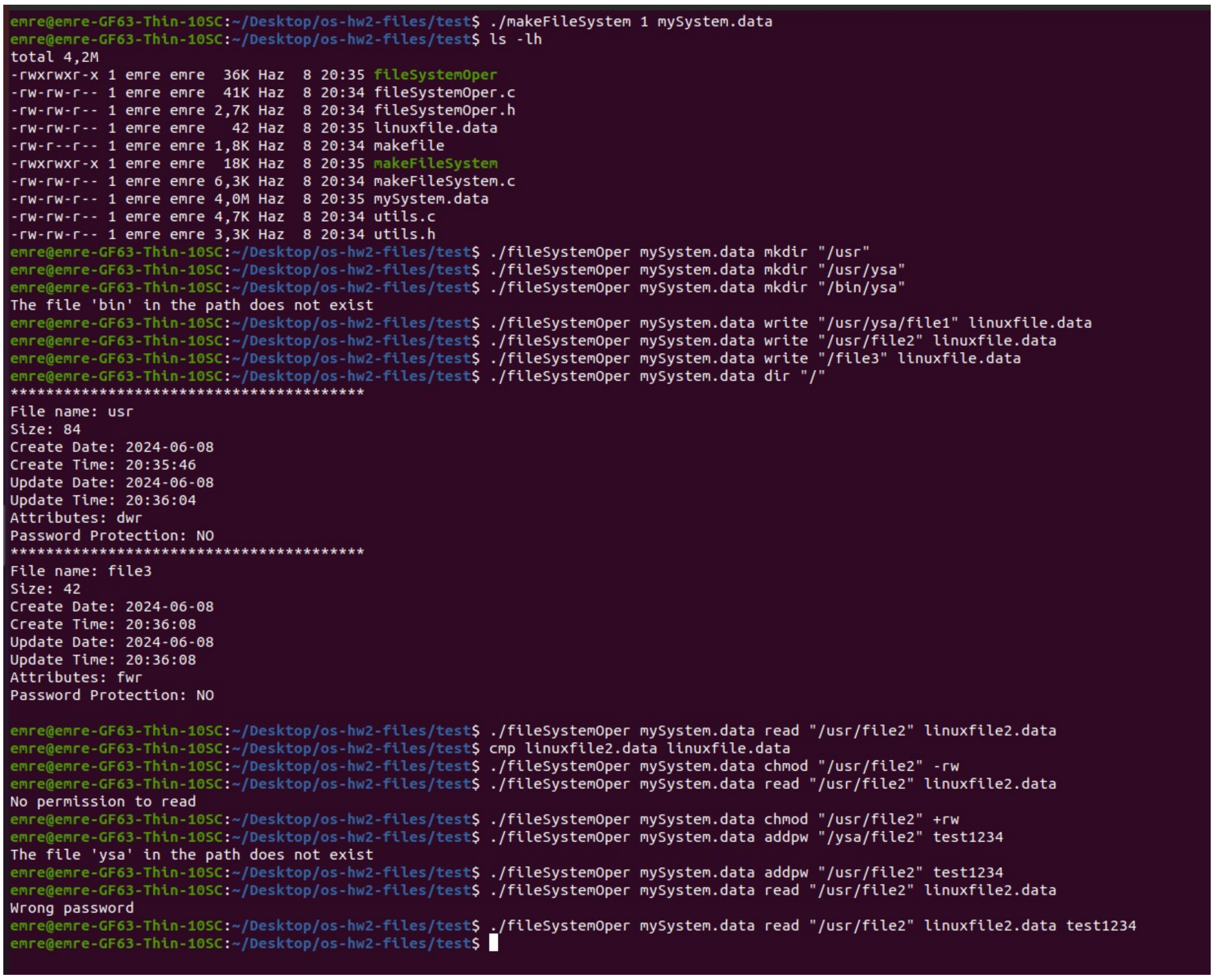


```
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ rm mySystem.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./makeFileSystem 0.5 mySystem.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ls -lh
total 2,2M
-rwxrwxr-x 1 emre emre  36K Haz  8 20:21 fileSystemOper
-rw-rw-r-- 1 emre emre  41K Haz  8 20:21 fileSystemOper.c
-rw-rw-r-- 1 emre emre 2,7K Haz  8 20:21 fileSystemOper.h
-rw-r--r-- 1 emre emre 1,8K Haz  8 20:21 makefile
-rwxrwxr-x 1 emre emre  18K Haz  8 20:21 makeFileSystem
-rw-rw-r-- 1 emre emre 6,3K Haz  8 20:21 makeFileSystem.c
-rw-rw-r-- 1 emre emre 2,0M Haz  8 20:23 mySystem.data
-rw-rw-r-- 1 emre emre 4,7K Haz  8 20:21 utils.c
-rw-rw-r-- 1 emre emre 3,3K Haz  8 20:21 utils.h
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ 
```

- As it is seen from the screenshots, a 2MB .data file is created for 0.5KB block size and a 4MB .data file is created for 1KB block size.

CONTINUE IN NEXT PAGE REMAINING SCREENSHOTS ARE THERE ...

fileSystemOper:


```
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./makeFileSystem 1 mySystem.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ls -lh
total 4,2M
-rwxrwxr-x 1 emre emre  36K Haz  8 20:35 fileSystemOper
-rw-rw-r-- 1 emre emre  41K Haz  8 20:34 fileSystemOper.c
-rw-rw-r-- 1 emre emre 2,7K Haz  8 20:34 fileSystemOper.h
-rw-rw-r-- 1 emre emre   42 Haz  8 20:35 linuxfile.data
-rw-r--r-- 1 emre emre 1,8K Haz  8 20:34 makefile
-rwxrwxr-x 1 emre emre  18K Haz  8 20:35 makeFileSystem
-rw-rw-r-- 1 emre emre 6,3K Haz  8 20:34 makeFileSystem.c
-rw-rw-r-- 1 emre emre 4,0M Haz  8 20:35 mySystem.data
-rw-rw-r-- 1 emre emre 4,7K Haz  8 20:34 utils.c
-rw-rw-r-- 1 emre emre 3,3K Haz  8 20:34 utils.h
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data mkdir "/usr"
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data mkdir "/usr/ysa"
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data mkdir "/bin/ysa"
The file 'bin' in the path does not exist
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data write "/usr/ysa/file1" linuxfile.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data write "/usr/file2" linuxfile.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data write "/file3" linuxfile.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data dir "/"
****************************************
File name: usr
Size: 84
Create Date: 2024-06-08
Create Time: 20:35:46
Update Date: 2024-06-08
Update Time: 20:36:04
Attributes: dwr
Password Protection: NO
****************************************
File name: file3
Size: 42
Create Date: 2024-06-08
Create Time: 20:36:08
Update Date: 2024-06-08
Update Time: 20:36:08
Attributes: fwr
Password Protection: NO

emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ cmp linuxfile2.data linuxfile.data
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data chmod "/usr/file2" -rw
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
No permission to read
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data chmod "/usr/file2" +rw
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data addpw "/ysa/file2" test1234
The file 'ysa' in the path does not exist
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data addpw "/usr/file2" test1234
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data
Wrong password
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ ./fileSystemOper mySystem.data read "/usr/file2" linuxfile2.data test1234
emre@emre-GF63-Thin-10SC:~/Desktop/os-hw2-files/test$ 
```

- As it is seen from the screenshots, given test cases in the homework PDF are working properly.

NOTE: Since I did not implement “dumpe2fs”, “rmdir”, and “del” commands I skipped these parts.