REPORT

FILE SYSTEM STRUCTURE

File System consists of 1 metadata block to keep necessary information about the system in the first block of the system. Then, free table, fat table and directory table are written to the system file respectively.

```
// Write the metadata in the system file
int metadata_blocks= 1;
fwrite(&block_size, sizeof(int), 1, file);
fwrite(&free_table_blocks, sizeof(int), 1, file);
fwrite(&fat_blocks, sizeof(int), 1, file);
fwrite(&directory_blocks, sizeof(int), 1, file);
fwrite(&metadata_blocks, sizeof(int), 1, file);

// Fill the rest of the metadata block with empty data
int buffer1_size = (metadata_blocks * block_size) - (5 * sizeof(int));
char buffer1[buffer1_size];

memset(buffer1, 0, sizeof(buffer1));

fwrite(buffer1, sizeof(char), buffer1_size, file);
```

Image 1: Metadata block

```
int free_table[NUMBEROFBLOCKS];

// fat blocks free table blocks and directory blocks are filled
// do not mark them as free
// everything else is free initially
for(int i = 0; i < NUMBEROFBLOCKS; i++){
    if(i < 1 + free_table_blocks + fat_blocks + directory_blocks)
        free_table[i] = 0;
    else
        free_table[i] = 1;
    fwrite(&free_table[i], sizeof(int), 1, file);
}
```

Image 2: Free table blocks (Fat table blocks and directory block are similar in structure)

That free table is 0 for a specific block shows that the block is not free and 1 shows that it is free. The fat table entries are initialized to -1 since there is no block to follow for no block initially.

Directory Entries

A directory entry consists of necessary data about the file such as filename, last modification, first block etc. A function called init_directory_entry created for the initializing an entry.

```
typedef struct directory_entry {

    char filename[128];
    int parent_index;
    time_t last_modification;
    int size;
    int first_block;
    int isDirectory;
    int isExist;
    int permission;
    char password[32];

} directory_entry;
```

Image 3: Directory Entry Structure

The size of the filename is 128 characters which will be enough arbitrary file name lengths. Parent index holds the position of the filename's parent in the directory entry table. There is no password, and all permissions are allowed by default but can be changed using commands. In the next section, these commands will be inspected in detail.

Functions

Dir

Whether the path exists should be known before dir function is called. This is done by the find_file_indices function whose task is to find indexes of the files in the path. The last file index is passed to dir function.

```c
void dir(const int file_index, const directory_entry directory_entries[]){

    //Look for directories whose parent is this folder in the directory entries
    for(int i = 0; i < MAXNUMBEROFFILE; ++i){
        if(directory_entries[i].isExist && directory_entries[i].parent_index == file_index){
            printf("%s last modified: %s", directory_entries[i].filename,  ctime(&directory_entries[i].last_modification));
            if(strcmp(directory_entries[i].password, "NOPASSWORD") != 0) printf("is protected ");
            int permission = directory_entries[i].permission;
            if(permission == READWRITE) printf("rw\n");
            else if(permission == WRITE) printf("w\n");
            else if(permission == READ) printf("r\n");
            else printf("no permission");
        }
    }

    printf("\n");
}
```

Image 4: Dir function

Mkdir

The function begins by calling find_file to check if a directory with the same name (filename) already exists within the provided indices (file_indices) of the directory_entries. The function find_file returns an index. If it returns -1, it indicates that no directory with the given name exists. If the directory already exists (i.e., find_file does not return -1), the function prints a message stating that the file already exists and then exits using return. If the directory does not exist, the function iterates through the directory_entries array to find an available slot (a slot where isExist is not 1).

Upon finding an available entry, the function retrieves the current time using time(&current_time) and initializes the new directory entry using init_directory_entry.

After creating the directory, a global or externally scoped flag is_directories_changed is set to 1, indicating that there has been a change in the directory structure. This is useful for triggering updates.

```c
void mkdir(const char* filename, const int* file_indices, const int file_count, directory_entry directory_entries[]){

    int index = find_file(filename, file_indices, file_count, directory_entries);

    if(index != -1){
        printf("The file is already exists in the given path\n");
        return;
    }

    for(int i = 0; i < MAXNUMBEROFFILE; ++i){
        if(directory_entries[i].isExist != 1){
            time_t current_time;
            time(&current_time);
            init_directory_entry(&directory_entries[i], filename, file_indices[file_count - 1], current_time, 0, 0, 1);
            is_directories_changed = 1;
            return;
        }
    }
}
```

Image 5: mkdir function

Rmdirectory

Rmdirectory function simply checks the directory flag and whether the folder has children then proceeds with setting the existence flag to zero.

```
void rmdirectory(const int file_index, directory_entry directory_entries[]){

    if(!directory_entries[file_index].isDirectory){
        printf("%s is not a directory\n", directory_entries[file_index].filename);
        return;
    }

    for(int i = 0; i < MAXNUMBEROFFILE; ++i){
        if(directory_entries[i].isExist && directory_entries[i].parent_index == file_index){
            printf("%s is not an empty directory\n", directory_entries[file_index].filename);
            return;
        }
    }

    directory_entries[file_index].isExist = 0;
    is_directories_changed = 1;
}
```

Image 6: Rmdirectory

Dumpe2fs

The function dumpe2fs is designed to display information about a simulated filesystem. The function starts with printing the total number of blocks (NUMBEROFBLOCKS) and printing the block size. Then it continues to iterate through the free_table array to count how many blocks are marked as free and iterates through the directory_entries array, incrementing the respective counters based on whether the entry is marked as existing (isExist) and whether it is a directory (isDirectory).

Then it iterates again through directory_entries, this time focusing on entries that represent files (not directories).

For each file, prints the filename followed by the sequence of blocks occupied by the file. This involves navigating the fat_table, which links each block to the next block in the file until a sentinel value (-1) is encountered, indicating the end of the file's block chain.

```
void dumpe2fs(const int block_size, const int free_table[], const int fat_table[],
const directory_entry directory_entries[]){


    printf("Block Count: %d\n", NUMBEROFBLOCKS);
    printf("Block Size: %d\n", block_size);

    int number_of_freeblocks= 0;
    for(int i = 0; i < NUMBEROFBLOCKS; ++i){
        number_of_freeblocks += free_table[i];
    }
    printf("Number of free blocks: %d\n", number_of_freeblocks);

    int number_of_files = 0, number_of_directories = 0;
    for(int i = 0; i < MAXNUMBEROFFILE; ++i){
        if(directory_entries[i].isExist){
            if(directory_entries[i].isDirectory){ ++number_of_directories; }
            else {++number_of_files;}
        }
    }

    printf("Number of files: %d\n", number_of_files);
    printf("Number of directories: %d\n", number_of_directories);


    for(int i = 0; i < MAXNUMBEROFFILE; ++i){
        if(directory_entries[i].isExist && !directory_entries[i].isDirectory){
            printf("Occupied blocks for %s: ", directory_entries[i].filename);
            int current_block = directory_entries[i].first_block;
            while(current_block != -1){
                printf("%d ", current_block);
                current_block = fat_table[current_block];
            }
            printf("\n");
        }
    }

}
```

Image 7: dumpe2fs

Change_permissions

The function change_permission is designed to modify the permission settings of a file. First, the function compares the provided password with the password stored in directory entry. If they do not match, prints "False password" and exits the function, ensuring that permissions can only be changed by someone with the correct password.

Depending on the content of new_permission, the function modifies the permission variable. If the new_permission string is not recognized, prints "Unidentified permission request" and exits. READ, WRITE, and READWRITE are constants that represent numerical values for read and write permissions.

```c
void change_permission(const int file_index, directory_entry directory_entries[], const char* new_permission,
const char* password){


    if(strcmp(directory_entries[file_index].password, password) != 0){
        printf("False password\n");
        return;
    }

    int permission = directory_entries[file_index].permission;

    if(strcmp(new_permission, "+r") == 0){
        if(permission % 2 == 0)permission += READ;
    }else if(strcmp(new_permission, "+w") == 0){
        if(permission < 2)permission += WRITE;
    }else if(strcmp(new_permission, "+rw") == 0) permission = READWRITE;
    else if(strcmp(new_permission, "-r") == 0){
        if(permission % 2 == 1)permission -= READ;
    }else if(strcmp(new_permission, "-w") == 0){
        if(permission > 1)permission -= WRITE;
    }else if(strcmp(new_permission, "-rw") == 0) permission = 0;
    else {
        printf("Unidentified permission request\n");
        return;
    }

    directory_entries[file_index].permission = permission;

    is_directories_changed = 1;
}
```

Image 8: Change permissions

Addpw

The addpw function is designed to add or update a password for a specific file in a directory. Initially, the function checks if the current password for the specified file is anything other than "NOPASSWORD", a placeholder indicating that no password is currently set. If a password is set, the function prompts the user to enter the current password. The input is then compared to the existing password. If they do not match, it outputs "False password" and exits the function, denying the password change.

If the current password is "NOPASSWORD" or if the user successfully enters the correct password, the function proceeds to update the password with the value of new_pw using strcpy.

```
void addpw(const int file_index, directory_entry directory_entries[], const char* new_pw){

    if(strcmp(directory_entries[file_index].password, "NOPASSWORD") != 0){

        printf("Enter current password: ");

        char input[128];

        if(fgets(input, sizeof(input), stdin) != NULL){

            int len = strlen(input);
            if (len > 0 && input[len-1] == '\n') {
                input[len-1] = '\0';
            }

            if (strcmp(input, directory_entries[file_index].password) != 0){
                printf("False password\n");
                return;
            }

        }else{
            printf("Error reading input.\n");
            return;
        }
    }

    strcpy(directory_entries[file_index].password, new_pw);

    is_directories_changed = 1;
}
```

Image 9: addpw function

Write_file

First, the fucntion checks if the file already exists in the system using find_file. If it exists, several checks follow:

Determine if the existing file is a directory (and return an error if so).

Validate password access.

Check if the file has write permissions; if not, it prints an error message. If permissions are granted, any existing data is deleted using del.

Then it opens the source file in binary read mode and calculates its size and reads the entire file into a dynamically allocated array (fileArray) of char*, divided into blocks of block_size.

Then it continues iterating over the free_table to find free blocks and writes data into these blocks in reverse order (from last to first), updating the fat_table to establish links between blocks.

After writing the data, it finds an available slot in directory_entries for a new file or an updated file if it doesn't exist and initializes the directory entry with the new file information including size, block index, and current time.

Read_file

Similar to write file function, read_file starts with validation checks. After that it opens the destination_file for writing and system_file for reading and initializes reading operations based on the starting block (first_block) from the file's directory entry.

Then continues using a loop to traverse the fat_table starting from first_block and writes the contents of the buffer to the destination_file. Then clears the buffer with memset to prepare for the next block.

Continues until a -1 is encountered in the fat_table, indicating the end of the file data.

```
void read_file(const char* destination_file, const char* system_file,
const int file_index, int block_size, directory_entry directory_entries[], int fat_table[], const char* password){

    directory_entry source_file = directory_entries[file_index];

    if(!check_permission(source_file, 0)){
        printf("The file has no read permission\n");
        return;
    }

    if(source_file.isDirectory){
        printf("The file to be read is a directory\n");
        return;
    }

    if(strcmp(source_file.password, password) != 0){
        printf("False password\n");
        return;
    }

    FILE* file = fopen(destination_file, "w");
    FILE* fsystem = fopen(system_file, "rb+");

    //advance through blocks and read them
    int read_block = source_file.first_block;

    while(read_block != -1){
        char buffer[block_size];
        fseek(fsystem, read_block * block_size, SEEK_SET);
        fread(buffer, sizeof(char), block_size, fsystem);
        fwrite(buffer, sizeof(char), strlen(buffer), file);
        memset(buffer, 0, sizeof(buffer));
        rewind(fsystem);
        read_block = fat_table[read_block];
    }

    fclose(file);
    fclose(fsystem);
}
```

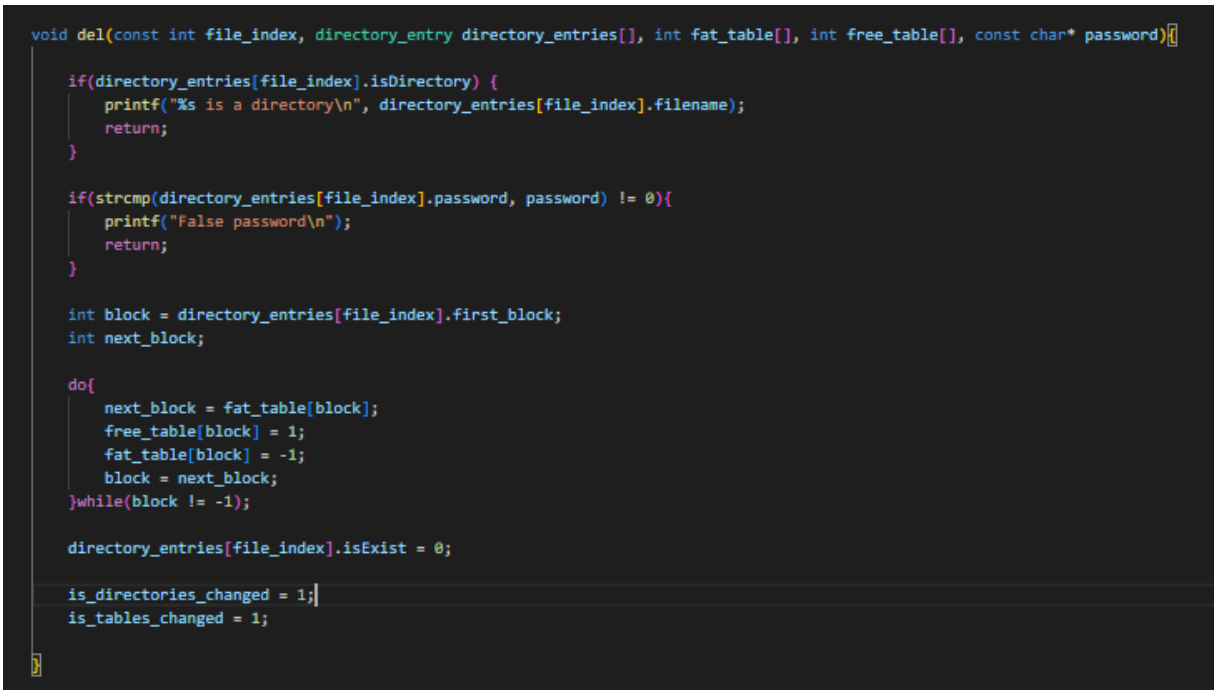
Image 10: Read file

Del

The function ensures that the specified index does not represent a directory, since directories are deleted using rmdir.

Then it Compares the stored password for the file with the provided password. If they don't match, it prints "False password" and terminates the operation.

After that, it starts from the first_block of the file and iterates through the FAT to trace and free all blocks associated with the file and marks the current block as free in the free_table by setting its value to 1. Then marks the directory entry of the file as non-existent by setting isExist to 0. This effectively removes the file from the directory listing and makes the entry available for new files.



```
void del(const int file_index, directory_entry directory_entries[], int fat_table[], int free_table[], const char* password){

    if(directory_entries[file_index].isDirectory) {
        printf("%s is a directory\n", directory_entries[file_index].filename);
        return;
    }

    if(strcmp(directory_entries[file_index].password, password) != 0){
        printf("False password\n");
        return;
    }

    int block = directory_entries[file_index].first_block;
    int next_block;

    do{
        next_block = fat_table[block];
        free_table[block] = 1;
        fat_table[block] = -1;
        block = next_block;
    }while(block != -1);

    directory_entries[file_index].isExist = 0;

    is_directories_changed = 1;
    is_tables_changed = 1;

}
```

Image 11: Del Function

Test Results

```
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./makeFileSystem 1 mySystem.data
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data mkdir /ysa
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data mkdir /bin/usr
False filepath. Exiting...
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data write /ysa/file2 data.txt
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data write /file1 data.txt
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data dir /
ysa last modified: Sat Jun  8 16:45:15 2024
rw
file1 last modified: Sat Jun  8 16:45:59 2024
rw

esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data dumpe2fs
Block Count: 4096
Block Size: 1024
Number of free blocks: 4036
Number of files: 2
Number of directories: 2
Occupied blocks for file2: 58
Occupied blocks for file1: 59
```

```
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data chmod /file1 -rw
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data read /file1 data.txt
The file has no read permission
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data chmod /file1 +rw
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data addpw /file1 1234
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data read /file1 data.txt
False password
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$ ./fileSystemOper mySystem.data read /file1 data.txt 1234
esat@DESKTOP-MI9NITJ:/mnt/c/MYFAT12$
```