CSE312 OPERATING SYSTEMS

HW3 REPORT

COŞKUN HASAN ŞALTU

1801042631

In this assignment, it is requested to make a file system based on the FAT12 file system. A representation of the file system made is written to a file and operations are expected to be performed on that file. For these stages, it is necessary to design the architecture of the file system first.

1) Designing of the File System Structure

FAT TABLE

Certain components are required to design this net system. One FAT12 table is required. The FAT12 table is a table that holds which block will come after a block.

```
typedef struct
{
    unsigned int fat_entry; // FAT entry value
} FATEntry;
```

```
FATEntry fat_table[MAX_BLOCKS];
```

The max block here was created by calculating the size. Since it is a FAT12 table, it has been determined as 4KB from 2 to the 12th.

DIRECTORY TABLE

Then, a directory table was needed to keep the contents of the directories.

```
typedef struct
{
    char filename[MAX_FILENAME_LENGTH];
    unsigned int size;        // Size of the file
    unsigned int start_block; // Starting block of the file
} DirectoryEntry;
```

```
DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
```

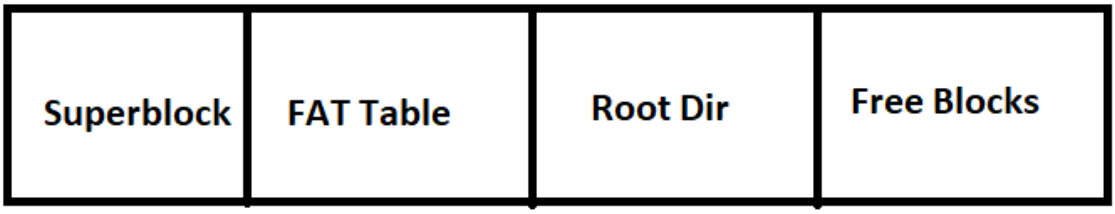
If we want to check the contents of a directory, we need to navigate through the directory table of that directory.

SUPERBLOCK

The superblock is a data structure that contains crucial information about a file system. It typically includes details such as the block size (the size of each data block), the position of the root directory (the top-level directory in the file system), the position of the FAT table (which tracks the allocation of data blocks), the position of the data blocks (where file data is stored), and the total number of blocks in the file system. The superblock provides important metadata that helps in managing and accessing the file system efficiently.

```
typedef struct
{
    unsigned short int block_size;   // Block size in KB
    unsigned int root_directory_pos; // Position of the root directory
    unsigned int fat_table_pos;      // Position of the FAT table
    unsigned int data_blocks_pos;    // Position of the data blocks
    unsigned int total_blocks;       // Total number of blocks
} Superblock;
```

The file is managed using superblock



FILE SYSTEM ARCHITECTURE

2) Implementing of the File System Structure

In the homework, a sample file system has been implemented and tested according to what was explained above.

First, the superblock must be filled in by making calculations. Because it is the building block of the superblock file system. All operations should be done by checking the superblock.

```
void createFileSystem(char *filename, int block_size)
{

    char **fileSystem;

    int fat_table_size;
    int fat_table_pos;
    int directory_table_size;
    int directory_table_pos;
    int data_blocks_pos;

    fileSystem = (char **)malloc(sizeof(char *) * MAX_BLOCKS);
    for (int i = 0; i < MAX_BLOCKS; i++)
    {
        fileSystem[i] = (char *)malloc(sizeof(char) * block_size);
    }

    directory_table_size = sizeof(DirectoryEntry) * MAX_DIRECTORY_ENTRIES;
    fat_table_size = sizeof(FATEntry) * MAX_BLOCKS;
    fat_table_pos = 1;
    directory_table_pos = fat_table_pos + fat_table_size / block_size;
    data_blocks_pos = directory_table_pos + directory_table_size / block_size;

    initializeFilesystem(block_size, fat_table_pos, directory_table_pos, data_blocks_pos);
```

```
void initializeFilesystem(int block_size, int fat_table_pos, int directory_table_pos, int data_blocks_pos)
{
    superblock.block_size = block_size;
    superblock.root_directory_pos = directory_table_pos;
    superblock.fat_table_pos = fat_table_pos;
    superblock.data_blocks_pos = data_blocks_pos;
    superblock.total_blocks = MAX_BLOCKS;

    printf("Block size: %d\n", superblock.block_size);
    printf("Root directory position: %d\n", superblock.root_directory_pos);
    printf("FAT table position: %d\n", superblock.fat_table_pos);
    printf("Data blocks position: %d\n", superblock.data_blocks_pos);
    printf("Total blocks: %d\n", superblock.total_blocks);
}
```

Next, the fat table and root directory table are be defined.

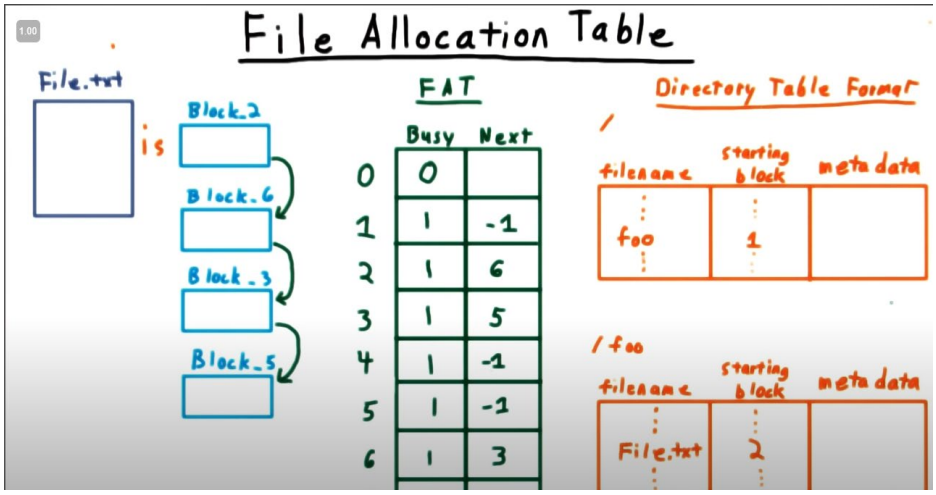
```
// Fill FAT table with empty entries
for (int i = 0; i < MAX_BLOCKS; i++)
{
    if (i < data_blocks_pos)
        fat_table[i].fat_entry = -1;
    else
        fat_table[i].fat_entry = -2;
}
// Fill directory table with empty entries
for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
{
    directory_table[i].size = -1;
    directory_table[i].start_block = -1;
    strcpy(directory_table[i].filename, "");
}
```

Then superblock , fat table and root directory table are written to the file system array respectively.

```
// Put FAT table in the following blocks
char *fat_table_ptr = fileSystem[fat_table_pos];
for (int i = 0; i < fat_table_size / block_size; i++)
{
    for (int j = 0; j < superblock.block_size / sizeof(FATEntry); j++)
    {
        memcpy(fat_table_ptr, &fat_table[i * (superblock.block_size / sizeof(FATEntry)) + j], sizeof(FATEntry));
        fat_table_ptr += sizeof(FATEntry);
    }
}

// Put directory table in the following blocks
char *directory_table_ptr = fileSystem[directory_table_pos];
for (int i = 0; i < directory_table_size / block_size; i++)
{
    for (int j = 0; j < superblock.block_size / sizeof(DirectoryEntry); j++)
    {
        memcpy(directory_table_ptr, &directory_table[i * (superblock.block_size / sizeof(DirectoryEntry)) + j], sizeof(DirectoryEnt
        directory_table_ptr += sizeof(DirectoryEntry);
    }
}
```

In this case, the first block belonged to the superblock. Afterwards, several blocks are allocated to the fat table according to the calculations. The remaining blocks are reserved for directory and files.



The whole system is designed approximately as in this image.

3) Implementing of the File System Operations

The file system has certain operations. The operations are as follows:

Operation	Parameters	Explanation	Example
dir	Path	Lists the contents of the directory shown by path on the screen.	`fileSystemOper fileSystem.data dir "\"` lists the contents of the root directory. The output will be similar to dir command of DOS
mkdir rmdir	Path and dir name	Makes or removes a directory	`fileSystemOper fileSystem.data mkdir "\ysa\fname"` makes a new directory under the directory "ysa" if possible. These two works exactly like mkdir and rmdir commands of DOS shell
dumpe2fs	None	Gives information about the file system.	`fileSystemOper fileSystem.data dumpe2fs` works like simplified and modified Linux dumpe2fs command. It will list block count, free blocks, number of files and directories, and block size. Different from regular dumpe2fs, this command lists all the occupied blocks and the file names for each of them.
write	Path and file name	Creates and writes data to the file	`fileSystemOper fileSystem.data write "\ysa\file" linuxFile` Creates a file named `file` under `"\usr\ysa"` in your file system, then copies the contents of the Linux file into the new file.
read	Path and file name	Reads data from the file	`fileSystemOper fileSystem.data read "\ysa\file" linuxFile` Reads the file named file under `"/usr/ysa"` in your file system, then writes this data to the Linux file. This again works very similar to Linux copy command.
del	Path and file name	Deletes file from the path	`fileSystemOper fileSystem.data del "\ysa\file"` Deletes the file named `file` under `"\ysa\file"` in your file system. This again works very similar to Linux del command.

Mkdir

```
void mkdir(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        if (findDirectoryEntry(directory_table, dirs[count - 1]) != -1)
        {
            printf("Directory already exists.\n");
            return;
        }
        int blockCount = MAX_DIRECTORY_ENTRIES * sizeof(DirectoryEntry) / superblock.block_size;
        int new_block = findEmptyBlock(blockCount + 1);
        if (block == -1)
        {
            printf("No empty blocks.\n");
            return;
        }
        DirectoryEntry directory_entry;
        directory_entry.start_block = new_block;
        directory_entry.size = 0;
        strcpy(directory_entry.filename, dirs[count - 1]);
        putDirectoryEntry(directory_table, directory_entry);
        writeDirectoryTable(directory_table, block);
        createDirectoryTable(new_block);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count - 1]) == 0)
            {
                mkdir(directory_table[i].start_block, dirs, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `mkdir` function is used to create a new directory within the file system. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function starts by reading the directory table of the current directory using the `readDirectoryTable` function. It then checks if the count of directories is equal to the size. If they are equal, it means that the function has reached the desired directory level and needs to create the new directory.

It first checks if the new directory already exists by calling the `findDirectoryEntry` function on the directory table. If it finds a matching entry, it displays an error message and returns.

If the new directory does not exist, it proceeds to find an empty block in the file system using the `findEmptyBlock` function. If no empty blocks are available, it displays an error message and returns.

Next, it creates a new directory entry (`DirectoryEntry`) and sets its start block to the newly found empty block. The size is set to 0, and the directory name is copied from the `dirs` array. The new directory entry is then added to the directory table using the `putDirectoryEntry` function.

The modified directory table is written back to the current directory using the `writeDirectoryTable` function. Finally, a new directory table is created for the newly created directory using the `createDirectoryTable` function.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count - 1]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

Rmdir

```
void rmdir(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        int index = findDirectoryEntry(directory_table, dirs[count - 1]);
        if (index == -1)
        {
            printf("There is no such directory.\n");
            return;
        }
        int start_block = directory_table[index].start_block;
        // edit Fat table
        while (1)
        {
            if (fat_table[start_block].fat_entry == -1)
            {
                writeFATEntry(start_block, -2);
                break;
            }
            else if (fat_table[start_block].fat_entry == -2)
            {
                break;
            }
            int temp = fat_table[start_block].fat_entry;
            writeFATEntry(start_block, -2);
            start_block = temp;
        }
        removeDirectoryEntry(directory_table, index);
        writeDirectoryTable(directory_table, block);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count - 1]) == 0)
            {
                rmdir(directory_table[i].start_block, dirs, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `rmdir` function is used to remove a directory from the file system. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function begins by reading the directory table of the current directory using the `readDirectoryTable` function.

If the count of directories is equal to the size, it means that the desired directory level has been reached and the function needs to remove the directory.

It first checks if the directory exists by calling the `findDirectoryEntry` function on the directory table. If it does not find a matching entry, it displays an error message and returns.

If a matching entry is found, it retrieves the start block of the directory from the directory entry. Then, it traverses the linked list of blocks in the file allocation table (FAT) starting from the retrieved start block.

While traversing the linked list, it updates the FAT entries to mark the blocks as free. It checks the current block's FAT entry value and performs the following actions:

- If the FAT entry is -1 (end of file), it marks the block as free by writing -2 (empty) to the FAT entry and breaks the loop.

- If the FAT entry is -2 (empty), it breaks the loop.

If the FAT entry is neither -1 nor -2, it temporarily stores the current block's FAT entry value, writes -2 to the FAT entry (marking it as empty), and updates the start block to the next block indicated by the stored FAT entry value.

After updating the FAT entries, the directory entry for the removed directory is removed from the directory table using the `removeDirectoryEntry` function.

The modified directory table is then written back to the current directory using the `writeDirectoryTable` function.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count - 1]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

Dir

```
void dir(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        printDirectoryTable(directory_table);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count]) == 0)
            {
                dir(directory_table[i].start_block, dirs, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `dir` function is used to list the contents of a directory in the file system. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function begins by reading the directory table of the current directory using the `readDirectoryTable` function.

If the count of directories is equal to the size, it means that the desired directory level has been reached and the function needs to print the directory table.

It calls the `printDirectoryTable` function, passing the directory table as an argument, to display the contents of the current directory.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

Write

```
void write(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], char *linuxFile, int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        if (findDirectoryEntry(directory_table, dirs[count - 1]) != -1)
        {
            printf("File already exists.\n");
            return;
        }
        char *buffer = readLinuxFile(linuxFile, buffer);
        int blockCount = sizeof(buffer) / superblock.block_size;
        int new_block = findEmptyBlock(blockCount + 1);
        if (block == -1)
        {
            printf("No empty blocks.\n");
            return;
        }
        DirectoryEntry directory_entry;
        directory_entry.start_block = new_block;
        directory_entry.size = strlen(buffer);
        strcpy(directory_entry.filename, dirs[count - 1]);
        putDirectoryEntry(directory_table, directory_entry);
        writeDirectoryTable(directory_table, block);
        createFile(buffer, new_block);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count - 1]) == 0)
            {
                write(directory_table[i].start_block, dirs, linuxFile, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `write` function is used to create and write data to a file in the file system. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the name of the Linux file to be written (`linuxFile`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function begins by reading the directory table of the current directory using the `readDirectoryTable` function.

If the count of directories is equal to the size, it means that the desired directory level has been reached and the function can proceed to write the file.

It checks if the file already exists in the current directory by using the `findDirectoryEntry` function. If a matching directory entry is found, it displays an error message indicating that the file already exists and returns.

The function then reads the contents of the Linux file into a buffer using the `readLinuxFile` function.

It calculates the number of blocks needed to store the file data based on the buffer size and the block size defined in the superblock.

The `findEmptyBlock` function is used to find a sequence of empty blocks in the file system that can accommodate the file data. If no empty blocks are available, it displays an error message and returns.

A new directory entry is created for the file, specifying the start block, size (determined from the buffer length), and filename. The directory entry is added to the directory table using the `putDirectoryEntry` function.

The updated directory table is written back to the file system using the `writeDirectoryTable` function.

Finally, the `createFile` function is called to write the file data from the buffer to the allocated blocks in the file system.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count - 1]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

Read

```
void read(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], char *linuxFile, int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        int index = findDirectoryEntry(directory_table, dirs[count - 1]);
        if (index == -1)
        {
            printf("There is no such file.\n");
            return;
        }
        int start_block = directory_table[index].start_block;
        char *buffer = malloc(sizeof(char) * directory_table[index].size);
        buffer = readFile(buffer, start_block, directory_table[index].size);
        writeLinuxFile(linuxFile, buffer, directory_table[index].size);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count - 1]) == 0)
            {
                read(directory_table[i].start_block, dirs, linuxFile, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `read` function is used to read data from a file in the file system and write it to a Linux file. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the name of the Linux file to write the data to (`linuxFile`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function begins by reading the directory table of the current directory using the `readDirectoryTable` function.

If the count of directories is equal to the size, it means that the desired directory level has been reached, and the function can proceed to read the file.

It checks if the file exists in the current directory by using the `findDirectoryEntry` function. If a matching directory entry is not found, it displays an error message indicating that the file does not exist and returns.

The start block of the file is retrieved from the directory entry, and a buffer of appropriate size is dynamically allocated to hold the file data.

The `readFile` function is called to read the data of the file from the file system, starting from the start block and with the size specified in the directory entry. The data is stored in the buffer.

Finally, the `writeLinuxFile` function is used to write the data from the buffer to the Linux file specified by `linuxFile`.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count - 1]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

Del

```
void del(unsigned int block, char dirs[MAX_DIRECTORY_ENTRIES][MAX_FILENAME_LENGTH], int count, int size)
{
    DirectoryEntry directory_table[MAX_DIRECTORY_ENTRIES];
    // read directory table
    readDirectoryTable(directory_table, block);
    if (count == size)
    {
        int index = findDirectoryEntry(directory_table, dirs[count - 1]);
        if (index == -1)
        {
            printf("There is no such file.\n");
            return;
        }
        int start_block = directory_table[index].start_block;
        // edit Fat table
        while (1)
        {
            if (fat_table[start_block].fat_entry == -1)
            {
                writeFATEntry(start_block, -2);
                break;
            }
            else if (fat_table[start_block].fat_entry == -2)
            {
                break;
            }
            int temp = fat_table[start_block].fat_entry;
            writeFATEntry(start_block, -2);
            start_block = temp;
        }
        removeDirectoryEntry(directory_table, index);
        writeDirectoryTable(directory_table, block);
        return;
    }
    else
    {
        for (int i = 0; i < MAX_DIRECTORY_ENTRIES; i++)
        {
            if (strcmp(directory_table[i].filename, dirs[count - 1]) == 0)
            {
                del(directory_table[i].start_block, dirs, count + 1, size);
                return;
            }
        }
        printf("No such directory.\n");
        return;
    }
}
```

The `del` function is used to delete a file from the file system. It takes several parameters, including the block number of the current directory, an array of directory names (`dirs`), the count of directories in the array (`count`), and the total size of the array (`size`).

The function begins by reading the directory table of the current directory using the `readDirectoryTable` function.

If the count of directories is equal to the size, it means that the desired directory level has been reached, and the function can proceed to delete the file.

It checks if the file exists in the current directory by using the `findDirectoryEntry` function. If a matching directory entry is not found, it displays an error message indicating that the file does not exist and returns.

The start block of the file is retrieved from the directory entry. The function then proceeds to edit the FAT table to remove the file from the file allocation table. It iterates through the FAT table, starting from the start block, and follows the linked blocks until it reaches the end of the file, indicated by a FAT entry of -1. For each block, it writes a FAT entry of -2, indicating that the block is now available for reuse.

Next, the directory entry for the file is removed from the directory table using the `removeDirectoryEntry` function. The updated directory table is then written back to the file system using the `writeDirectoryTable` function.

If the count of directories is not equal to the size, it means that the desired directory level has not been reached yet. In this case, the function iterates through the directory table to find a directory entry that matches the current directory name (`dirs[count - 1]`). If a match is found, the function recursively calls itself with the start block of the matched directory, incrementing the count by 1. This recursive call continues until the desired directory level is reached.

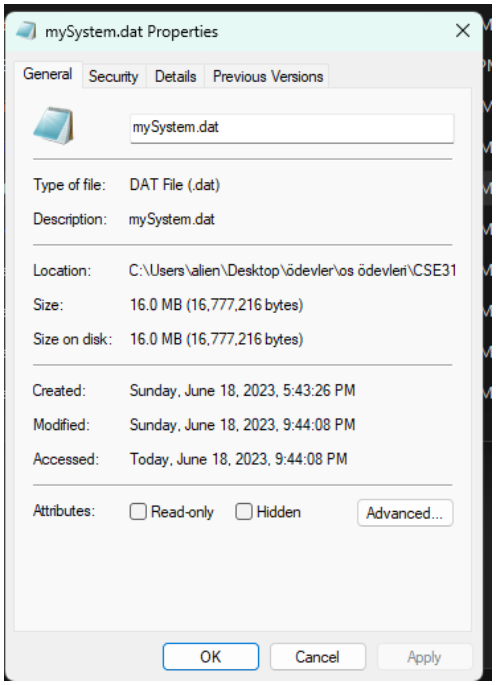
If no matching directory entry is found in the current directory, it displays an error message indicating that the directory does not exist.

4) TEST CASES

The program is tested all commands:

```
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./makeFileSystem 4 mySystem.dat
Block size: 4096
Root directory position: 5
FAT table position: 1
Data blocks position: 6
Total blocks: 4096
Root directory size: -1
Fat table first entry: -1
Supreblock block size: 4096
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$
```

This is create file system command…



16MB file system…

```
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat mkdir /usr
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /
usr
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat mkdir /usr
Directory already exists.
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat rmdir /usr
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat rmdir /usr
There is no such directory.
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat mkdir /usr
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat mkdir /usr/ysa
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat mkdir /bin
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /
usr
bin
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /usr
ysa
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /usr/ysa
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /usr/asdasd
No such directory.
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat write /file1 sample.pdf
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat read /file1 sample5.pdf
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat read /file2 sample5.pdf
There is no such file.
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat write /file1 sample.pdf
File already exists.
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat del /file1
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat write /file1 sample.pdf
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$ ./fileSystemOper mySystem.dat dir /
usr
bin
file1
saltu@DESKTOP-US0K338:/mnt/c/Users/alien/Desktop/ödevler/os ödevleri/CSE312-Operating-System-Homeworks/HW3$
```