Operating Systems

HW3 REPORT

Elifnur KABALCI
1801042617

General:

In this project, I designed the FAT12 file system. I created a superblock structure by getting blocksize and data from the user. While doing this, I used the data equivalents of FAT12 in the table. The maximum fat size specified in the homework pdf was 16mb. It is expressed in the same way here. In Part2, I control this in the code.

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Figure 4-31. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

I used some struct structures in the 2nd part to hold the file system elements. In the second part, it reads the data saved in the data file and saves it to its own storage area and performs the operations expected from a file system by using them. Therefore, I will explain the design through struct structures.

All the sources I looked at were writing the data file as binary. However, since I could not provide reading, I used normal file typing instead of doing this. Thus, I was able to extract data easily in part3.

SuperBlock:

```
struct Superblock {
    unsigned int block_size;
    unsigned int root_directory_position;
    unsigned int fat_position; // fat start block
    unsigned int data_position;

    unsigned int total_block;
    unsigned int free_block;
    unsigned int fat_block; // fat block count
    unsigned int num_files;
};
```

Superblock is a structure that keeps the elements used in a file management, which is usually used 1 for each location. In this struct structure, I collected some elements in the content of the superblock structure.

Block size is the value that determines the size of each block, which can be a maximum of 4 (because we are designing fat12 in this assignment) received from the user.

Root directory position is the initial area for file management. Branch structure is formed from here.

Fat position is the position where Fat blocks start.

Data Position is the location where data will be obtained in write and read situations.

Since the total block should be a maximum of 16 mb, 4096 bytes can be allocated in a file system with block size I 4. The maximum state is therefore direct 4096.

Free block is the amount obtained by subtracting the root block from the total block for the initial state, and then allowing the use of the block as it is used.

Fat block shows the amount of fat in use.

Directory Entry:

```
struct DirectoryEntry {
    string file_name;
    unsigned int file_size;
    time_t last_modified;
    unsigned short first_block;

    string extension;
    unsigned char attributes;
    char reserved[10];
};
```

This part was designed by analogy with the MSDOS part given in the book. Extension, attribute and reserved parts represent bits contained here.

This struct simply holds data about the file to be created or deleted. For Part2, these are all blank. In Part 3, these parts are filled separately for each use.

There is only 1 method in Part2. This allows us to assign the data and write it to the file. For each line, it also writes what the opposite value is. Thus, when we read, for example, if we want to assign the value of block size, we can go to the line that starts with the block size at the beginning of the carriage and get the value there.

When we come to part 3, we get the .data file created in part2 here. So we can keep the data about the filesystem.

There is also another class structure here. This is used to hold the data received from the user. I called filesystem I in this class and positioned it here like an object. Because the changes to be made in the filesystem are made according to the data received from the user.

```
class Parameter{
private:
    string parameter1;
    string parameter2;
    string operation;
    FileSystem fat;

public:
    string getPar1(){ return parameter1;}
    string getPar2(){ return parameter2;}
    string getOp() const{ return operation;}

    FileSystem getFat(){ return fat; }

    void setpar1(string par){ this->parameter1 = par; }
    void setpar2(string par){ this->parameter2 = par; }
    void setop(const string& op){ this->operation = op; }
    void setFat(FileSystem fat){ this->fat = fat; }
    Parameter(string filename);
    void process();
    void setS(int argc, char* argv[]);

    void dirCommand();
    void mkdirCommand();
    void rmdirCommand();
    void writeCommand();
    void readCommand();
    void delCommand();
    void dumpCommand();
};
```

The part3.cpp file serves as the main file for this project. The accuracy of the data received from the user is checked. In this part, I got an error in reading the ./filesystemOper string received from the user, and I solved it with a library called filesystem.

```
std::filesystem::path programPath = argv[0]; // ı take error from recognize argv
string programname=programPath.filename().string();
if(programname != "fileSystemOper"){ // cmp example
    cerr<<"argv[0] is wrong. "<<endl;
    exit(EXIT_FAILURE);
}
```

After this check I created the parameter object and sent the filename directly so I created the filesystem element here as well. And I kept it. So I was able to access the filesystem elements directly from the parameter files.

I normally sent string filename in the parameter constructer, but got an error in the definition. When I researched, I found out that it is the first definition from another class. That's why I changed the definition to this.

```
Parameter::Parameter(string filename) : fat(filename){
    this->parameter1 = "";
    this->parameter2 = "";
    this->operation = "";
    FileSystem fat(filename); // take only filename
    this->fat = fat;

}
```

In this section, the filesystem element is created after the first definitions. It goes to the Filesystem constructer. Here it calls the readFile method.

```
FileSystem::FileSystem(string filename){
    this->filename = filename;
    readfile();
}
```

In the Read file method, as I mentioned before, it opens the file and assigns the star to the data stored in the filesystem class according to the data at the beginning. Example of this usage:

```
while (getline(file, line)) {
    if (line.find("Block size") != string::npos) {
        setBlockSize(stoul(line.substr(line.find(":") + 1)));
    }
    else if (line.find("Root Directory Position") != string::npos) {
        setRootDirectoryPosition(stoul(line.substr(line.find(":") + 1)));
    }
    else if (line.find("FAT Position") != string::npos) {
        setFatPosition(stoul(line.substr(line.find(":") + 1)));
    }
    else if (line.find("Data position") != string::npos) {
        setDataPosition(stoul(line.substr(line.find(":") + 1)));
    }
    else if (line.find("Total Blocks") != string::npos) {
        setTotalBlock(stoul(line.substr(line.find(":") + 1)));
```

The program returns to main in part3.cpp. Here I called some methods respectively.

```
Parameter par(argv[1]);
par.setS(argc, argv);
par.process();
```

In the SetS method, its arguments are assigned to the objects of the parameter class given above. Some operators take 2 parameters, while others take 1 parameter.

```
void Parameter::setS(int argc, char* argv[]){
    setop(string(argv[2]));
    string temp = getOp();
    if (argc == 4 && (temp.compare("mkdir") == 0 || temp.compare("rmdir") == 0 || temp.compare("dir") == 0 || temp.compare("del") == 0)){
        setpar1(argv[3]);
    }
    else if (argc == 5 && (temp.compare("write") == 0 || temp.compare("read") == 0)){
        setpar1(argv[3]);
        setpar2(argv[4]);
    }
}
```

In the Process method, the operator is directed to the reserved method according to what it is. Thus, the processing begins.

I thought about using string to open and read the file, but when I tried to assign it with string I got an error. When I researched, I saw that unsigned short is used in such works. So I used my storage as an array of this type. However, when I try to pull data of this type, for example, the directory entry must be a string, when I want to pull it, it reads and keeps a single char. That's why I sent the start and end index, not a single index, to the method that I obtained data from. I thought I could access the whole string like this. However, when I tried to get the data, I saw that it returned blank. In methods like mkdir, it is necessary to make changes in the entry set and this is done in the methods. However, I noticed that the data is saved only in the method and does not affect the class structure.

```
std::string FileSystem::getEntriesContent(int startIndex, int endIndex) const {
    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> converter;
    std::string content;

    for (int i = startIndex; i <= endIndex; ++i) {
        std::u16string encodedContent(1, static_cast<char16_t>(entries[i]));
        std::string entry = converter.to_bytes(encodedContent);
        content += entry;
    }
    return content;
}
```

Dir Command

```
void Parameter::dirCommand() {
    const string& directoryPath = getPar1();
    FileSystem fileSystem = getFat();

    cout << "Contents of directory " << directoryPath << ":" << endl;
    cout << "Name\tSize\tLast Modified" << endl;

    // Scan the FAT entries for directory entries
    for (int i = 0; i < 4096; i++) {
        if (fileSystem.getEntriesContent(i) == directoryPath) {
            // Read the file/directory name
            string name = fileSystem.getEntriesContent(i);

            // Read the file size (0 for directories)
            unsigned int size = fileSystem.getFileSize(); // Simplified. Should actually be read from the entry

            // Read the last modified time
            time_t lastModified = fileSystem.getLastModified(); // Simplified. Should actually be read from the entry

            // Convert the last modified time to string
            char timeStr[20];
            strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", localtime(&lastModified));

            // Print the details
            cout << name << "\t" << size << "\t" << timeStr << endl;
        }
    }
}
```

In this method, it outputs the information of path, file or folder created or deleted in the file system. The parameter received from the user gives us which path to examine. It prints the name, size and modification date of all data saved with this path in the Entries array.

Mkdir Command

```
void Parameter::mkdirCommand() {
    const string& directoryName = getPar1();
    FileSystem fileSystem = getFat();

    // Check if a directory with the same name already exists
    for (int i = 0; i < 4096; i++) {
        if (fileSystem.getEntriesContent(i) == directoryName) {
            cerr << "Error: Directory with the same name already exists." << endl;
            return;
        }
    }
    // Find a free FAT entry
    int freeFATEntry = -1;
    for (int i = 0; i < 4096; i++) {
        if (fileSystem.getEntry(i) == 0) {
            freeFATEntry = i;
            break;
        }
    }
    // If no free FAT entry is found, the disk is full
    if (freeFATEntry == -1) {
        cerr << "Error: No free space available on disk." << endl;
        return;
    }
    time_t currentTime;
    time(&currentTime); // Get current time

    fileSystem.setEntriesContent(freeFATEntry, directoryName); // Set directory name
    fileSystem.setFirstBlock(freeFATEntry); // This is a new directory, so it's the first block
    fileSystem.setLastModified(currentTime); // Set the last modified time to the current time
    fileSystem.setFileSize(0); // Directory, so file size is 0

    // Update the FAT entry to indicate that it's used
    fileSystem.setFatEntry(freeFATEntry, 0xFFFF); // 0xFFFF usually indicates end-of-file in FAT-based file systems
    cout << "Directory " << directoryName << " created successfully." << endl;
}
```

The mkdir command works with a parameter. This parameter contains the path I to create the directory. In this method, we first check whether the directory we want to create already exists. We search for free space by scanning the list of entries. When we find the empty space, we fill the space by processing the data in this location. Then we make sure that the changed filesystem object is protected by equating it to the class object.

Rmdir Command

```
void Parameter::rmdirCommand() {
    const string& directoryPath = getPar1();
    FileSystem fileSystem = getFat();

    int fatIndex = -1;
    bool isEmpty = true;

    // Scan the FAT entries for directory entries
    for (int i = 0; i < 4096; i++) {
        if (fileSystem.getEntriesContent(i) == directoryPath) {
            fatIndex = i;

            if (fileSystem.getFileSize() != 0) {
                isEmpty = false;
            }
            break;
        }
    }
    // Check if the directory was found
    if (fatIndex == -1) {
        cerr << "Error: Directory not found." << endl;
        return;
    }
    // Check if directory is empty
    if (!isEmpty) {
        cerr << "Error: Directory is not empty." << endl;
        return;
    }
    fileSystem.removeEntry(fatIndex);

    cout << "Directory " << directoryPath << " has been deleted successfully." << endl;
}
```

In this command, we scan the set of emtries with the for loop. First of all, it is checked whether there is a directory to be deleted or not. If it is found, it is checked whether it is empty or full. If it is empty, the contents of the directory should also be deleted. This is achieved with the removeEntry method.

Del Command

```
void Parameter::delCommand() {
    const string& filePath = getPar1();
    FileSystem fileSystem = getFat();

    int fatIndex = -1;
    // Scan the FAT entries for file entries
    for (int i = 0; i < 4096; i++) {
        // Check if this entry matches the file path to be deleted
        if (fileSystem.getEntriesContent(i) == filePath) {
            fatIndex = i;
            break;
        }
    }
    // Check if the file was found
    if (fatIndex == -1) {
        cerr << "Error: File not found." << endl;
        return;
    }
    fileSystem.removeEntry(fatIndex);
    // Get the first block of the file from the FAT
    unsigned short firstBlock = fileSystem.getEntry(fatIndex);

    // Iterate through the FAT to free the blocks used by this file
    unsigned short currentBlock = firstBlock;
    while (currentBlock != 0) { // Assuming 0 indicates the end of the file
        unsigned short nextBlock = fileSystem.getEntry(currentBlock);
        fileSystem.setFatEntry(currentBlock, 0); // Mark block as free
        currentBlock = nextBlock;
    }
    cout << "File " << filePath << " has been deleted successfully." << endl;
}
```

The del command takes one parameter, which is the path of the file. The purpose here is to delete all files and paths in the entered path. For this, the fields related to this path in the entries set are filled with zero. Thus, it becomes available for use.

Dumpe2fs Command

```
void Parameter::dumpCommand() {
    FileSystem fileSystem = getFat();
    // Printing the Superblock information
    cout << "Super Block Information:" << endl;
    cout << "Block size               : " << fileSystem.getBlockSize() << endl;
    cout << "Data position            : " << fileSystem.getDataPosition() << endl;
    cout << "FAT Position             : " << fileSystem.getFatPosition() << endl;
    cout << "Root Directory Position  : " << fileSystem.getRootDirectoryPosition() << endl;

    // Printing the Directory Entry information
    //cout << "\nRoot Directory Entry Information:" << endl;
    //cout << "File Name                : " << fileSystem.getFilename() << endl;
    //cout << "File Size                : " << fileSystem.getFileSize() << endl;
    //cout << "First Block              : " << fileSystem.getFirstBlock() << endl;

    time_t lastModified = fileSystem.getLastModified();
    char buffer[80];
    //strftime(buffer, 80, "%c", localtime(&lastModified));
    //cout << "Last Modified            : " << buffer << endl;

    // Printing the FAT Entries information
    //cout << "\nFAT Entries:" << endl;
    /*for (int i = 0; i < 4096; i++) {
        cout << "FAT Entry " << i << " : " << fileSystem.getEntry(i) << endl;
    }*/
}
```

This command returns information about the data about the filesystem. Here I have the entries data in the comment line. Because it had 4096 lines, it was causing trouble in examining other data. I also commented the data of the Directory entry section. Because these are the data held for the processed file. It is determined and used at the time of operation. That's why I chose not to press the screen.

Read Command

```
void Parameter::readCommand() {
    FileSystem fileSystem = getFat();
    const string& customFilePath = getPar1();
    const string& linuxFilePath = getPar2();

    FileSystem entry = fileSystem.findDirectoryEntryByPath(customFilePath);
    if (entry.getFilename().empty()) {
        cerr << "Error: File not found in the custom file system" << endl;
        return;
    }

    string content = fileSystem.readFileData(entry.getFirstBlock(), entry.getFileSize());

    ofstream linuxFile(linuxFilePath, ios::binary);
    if (!linuxFile.is_open()) {
        cerr << "Error: Unable to open the Linux file for writing" << endl;
        return;
    }
    linuxFile.write(content.c_str(), content.size());
    linuxFile.close();
    cout << "File data has been successfully read from the custom file system and written to the Linux file." << endl;
}
```

This command takes 2 parameters, one of which is the path to be read. The other is the file where copies of the data are saved after reading.

Write Command

Since the code of this command is longer than the others, I wanted to explain it without adding a screenshot. This command takes two parameters. One of these parameters is the path where the operation will be applied. The other is where the data will be written after it is read. Here, we read the given file as we did at the beginning, and if there is no such file as defined, we create a new file to write and write it into it. Here, creating a file is done by saving the data in the entries set, as in the mkdir command.

Outputs:

I use the makefile that make compile and run.

I try to run all commands but I take these. I try it. But it doesn’t work.

I take these outputs.

```
latulipenoirez@Elifnur-PC:/mnt/c/Users/e.kabalci2018/Desktop/os3/part3$ make
g++ -c fileSystem.cpp
g++ part3.o fileSystem.o -o fileSystemOper
./fileSystemOper fileSystem.data mkdir "/usr"
Directory /usr created successfully.


./fileSystemOper fileSystem.data mkdir "/usr/ysa"
Directory /usr/ysa created successfully.


./fileSystemOper fileSystem.data mkdir "/bin/ysa"
Directory /bin/ysa created successfully.


./fileSystemOper fileSystem.data write "/usr/ysa/file1" fileSystem.cpp
Error: Unable to open the file system file /usr/ysa/file1
Error: Unable to open the file system file /usr/ysa/file1
File /usr/ysa/file1 has been created and written to successfully.
```

```
./fileSystemOper fileSystem.data write "/usr/file2" Makefile
Error: Unable to open the file system file /usr/file2
Error: Unable to open the file system file /usr/file2
File /usr/file2 has been created and written to successfully.


./fileSystemOper fileSystem.data write "/file3" Makefile
Error: Unable to open the file system file /file3
Error: Unable to open the file system file /file3
File /file3 has been created and written to successfully.


./fileSystemOper fileSystem.data dir "/"
Contents of directory /:
Name    Size    Last Modified
```

```
./fileSystemOper fileSystem.data del "/usr/ysa/file1"
Error: File not found.


./fileSystemOper fileSystem.data dumpe2fs
Super Block Information:
Block size              : 4096
Data position           : 21536
FAT Position            : 13344
Root Directory Position  : 32
```

```
./fileSystemOper fileSystem.data read "/usr/file2" linuxFile2.data
Error: Unable to open the file system file  /
Error: File not found in the custom file system


./fileSystemOper fileSystem.data dir "/usr"
Contents of directory /usr:
Name    Size    Last Modified


./fileSystemOper fileSystem.data write "/usr/linkedfile2" Makefile
Error: Unable to open the file system file /usr/linkedfile2
Error: Unable to open the file system file /usr/linkedfile2
File /usr/linkedfile2 has been created and written to successfully.


./fileSystemOper fileSystem.data dumpe2fs
Super Block Information:
FAT Position            : 13344
Root Directory Position  : 32


./fileSystemOper fileSystem.data dir "/"
Contents of directory /:
Name    Size    Last Modified
```