File Systems

Homework 2 Report

Gebze Technical University

Computer Engineering - CSE 312 Operating Systems

BUKET GENCER

210104004298

INTRODUCTION

This assignment aims to prepare the cd-rom file system. The cd-rom file system is in "write-once" structure. that is, it is written only once and then not written again. Only read operations can be performed. Doing the homework includes the following steps: First, we create an empty file system. Our "mySystem.dat" file represents our empty file system. Then, we copy a folder we prepared for testing to our file system to fill our empty file system. The test folder is the folder in the "/ysa/start" path mentioned in the assignment.

I copied the test folder to the "mySystem.dat" folder in the "file-system" structure I designed.

At this stage, I have finished the 2. part of the assignment. At this stage, how I design my file system is very important.

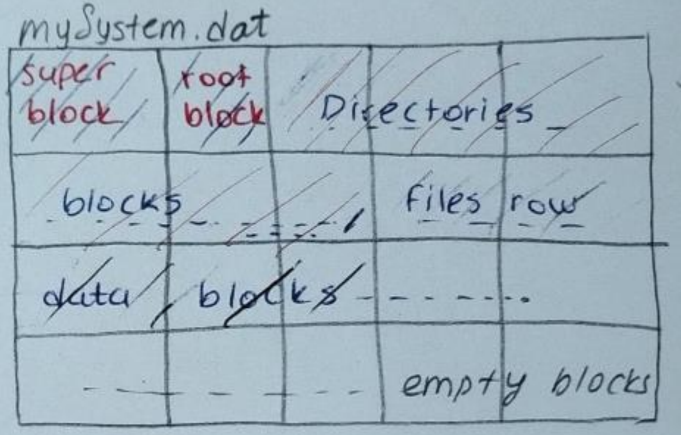
I explained my design in the drawing I made on paper.

If I have to explain again: the first block belongs to super_block. super block points to root_block. Thus, I can access the root block of my file system using the super block.

I have 3 different block types. The first one is super block. The second one is directory blocks. directory blocks consist only of directory_enrtys. These entries are the entries of the files and folders in that directory. The third is file blocks. file blocks hold raw data. To access this data, I can get information about where the block is located from the entry.

Finally, I created my general file system structure as follows: I recursively browse the test folder given to us once and create my directory blocks. I do not receive raw data at this stage. Then, I navigate the test folder recursively once again and write the raw data starting from the first empty block after the directory blocks. Thanks to this structure, my file system becomes more consistent.

my file system structure

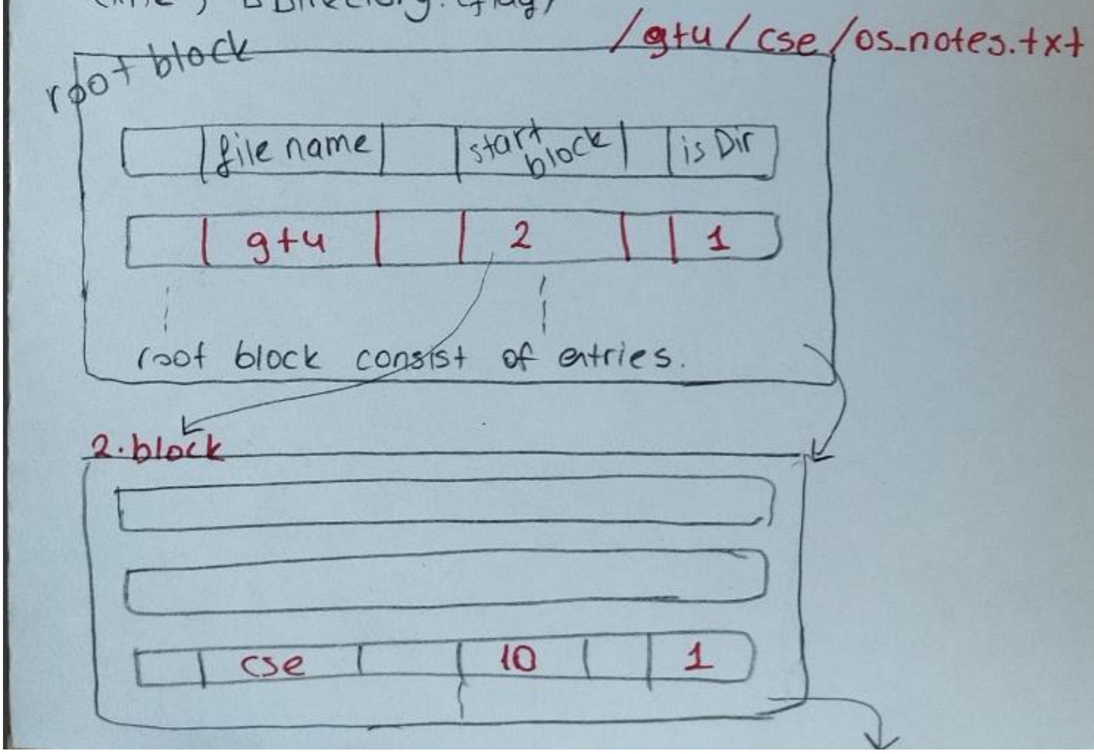


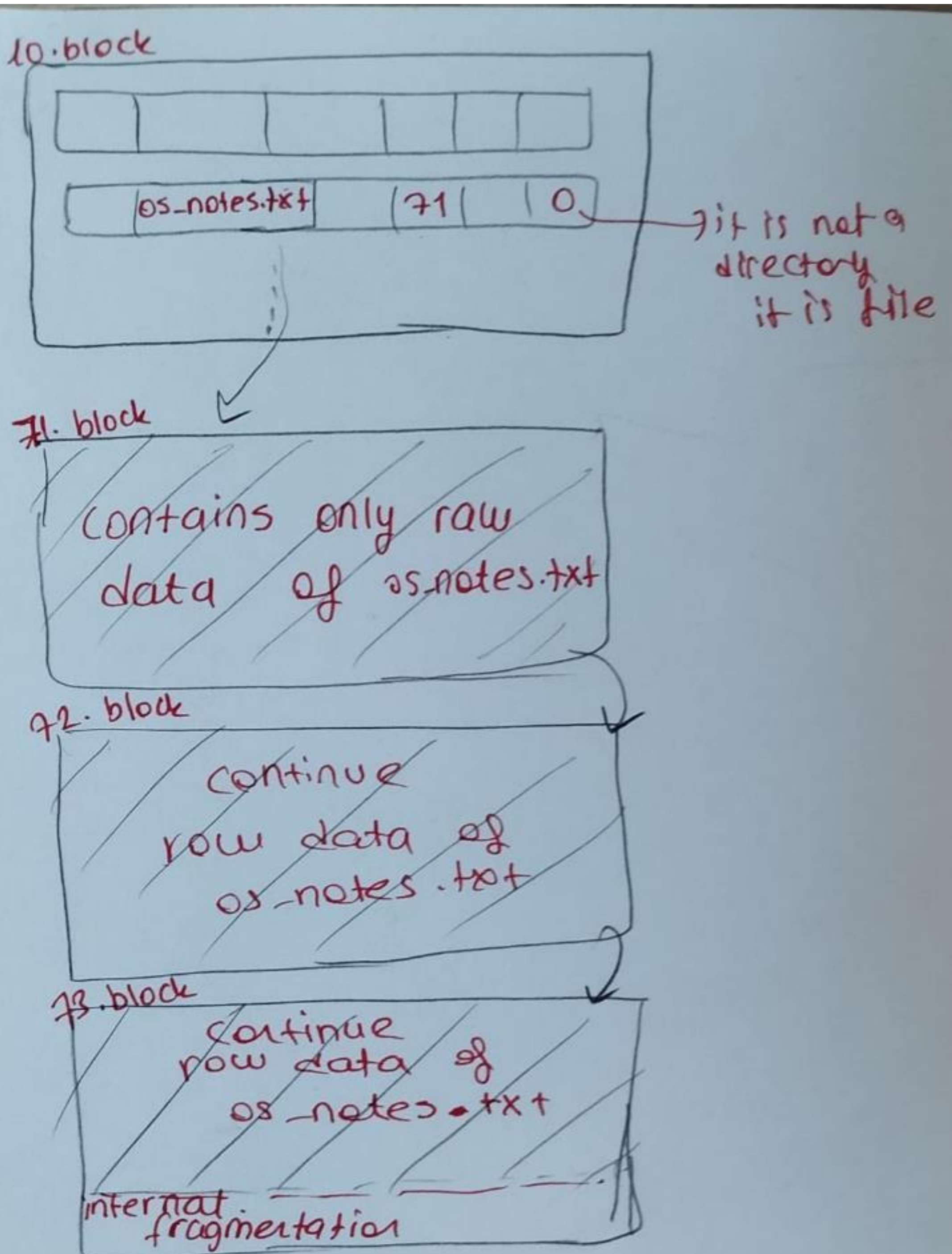
$\rightarrow \frac{\text{Block size}}{\text{size}} = \text{Block num}$

Supe block: special blocks. it holds a informations about file system and it point to root block.

Entry: it holds directory or file information

Entry ⇒ file name, blockLocation of Entry, fileSize, date, time, isDirectory. (flag)




10.block
os_notes.txt
71
0
if is not a directory it is file
71. block
contains only raw data of os_notes.txt
72. block
continue row data of os_notes.txt
73.block
continue row data of os_notes.txt
internal fragmentation

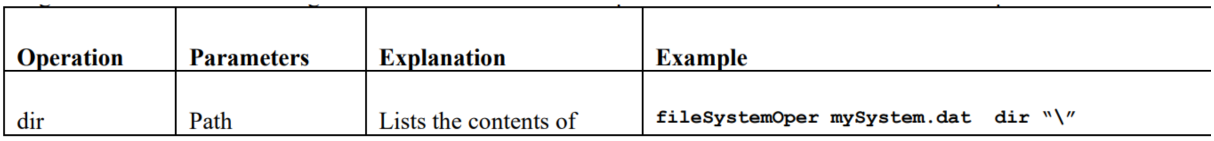
my directory blocks consist only of entiries.

First, I placed my directory blocks into my file sytem, Then I placed the raw data blocks sequentially.

The 3. stage of the assignment consists of operating on fileSystem.dat, that is, the file system we created. When we receive the correct data as a result of these operations, we can understand that we have constructed our file system structure correctly.

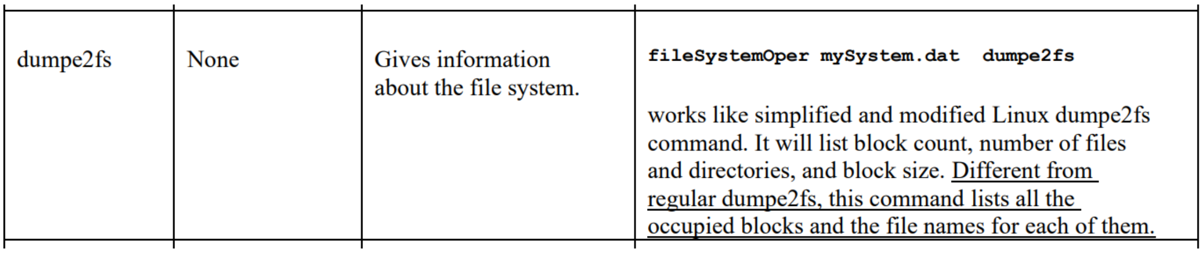
There are 3 operations in the 3rd stage of the assignment.

Dir operation: The purpose of the dir function is to provide information about the entries in the given path. We can say that dir is a bit like the ls -la command.



Operation	Parameters	Explanation	Example
dir	Path	Lists the contents of	`fileSystemOper mySystem.dat  dir "\"`

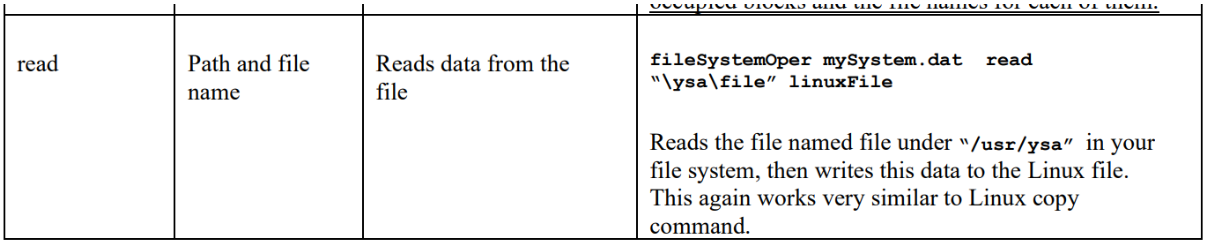
Dumpe2fs operation: dumpe2fs function gives general file system information. This information is read from the super block, dumpe2fs lists the occupied blocks and files.



dumpe2fs	None	Gives information about the file system.	`fileSystemOper mySystem.dat  dumpe2fs` works like simplified and modified Linux dumpe2fs command. It will list block count, 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.

Read operation: The read function is actually the function that best shows that I have created mySystem.dat correctly. This function copies a file data from mySystem.dat and pastes the data into the file I created in the current path.

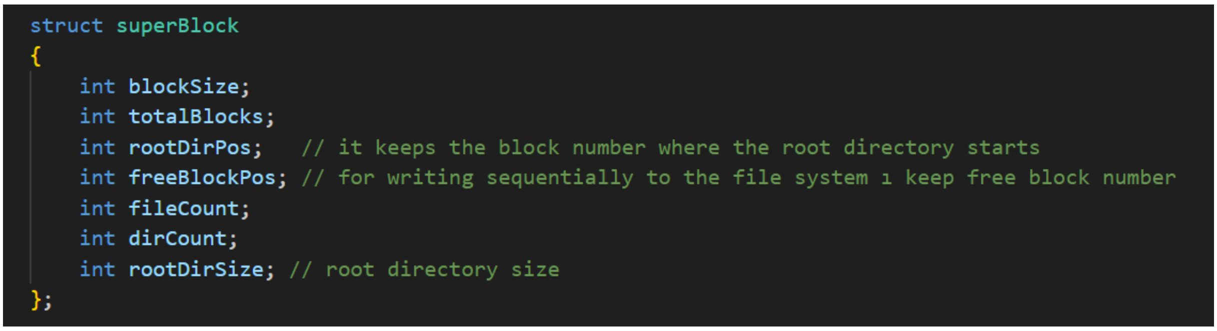
It is very important to get correct results at this stage because correct results prove that we have constructed our file system correctly.



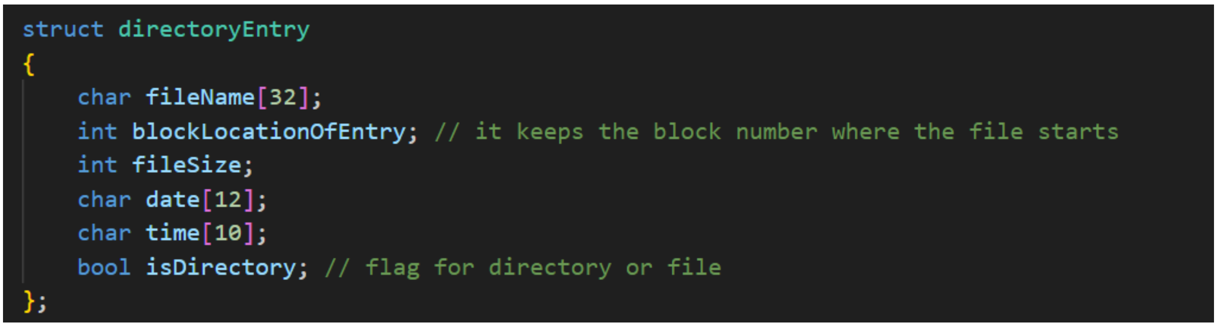
read	Path and file name	Reads data from the file	`fileSystemOper mySystem.dat  read "\ysa\file" linuxFile` Reads the file named file under `"/usr/ysa"` in your file system, then writes this data to the Linux file. This again works very similar to Linux copy command.

CODE EXPLANATIONS

File System Structure



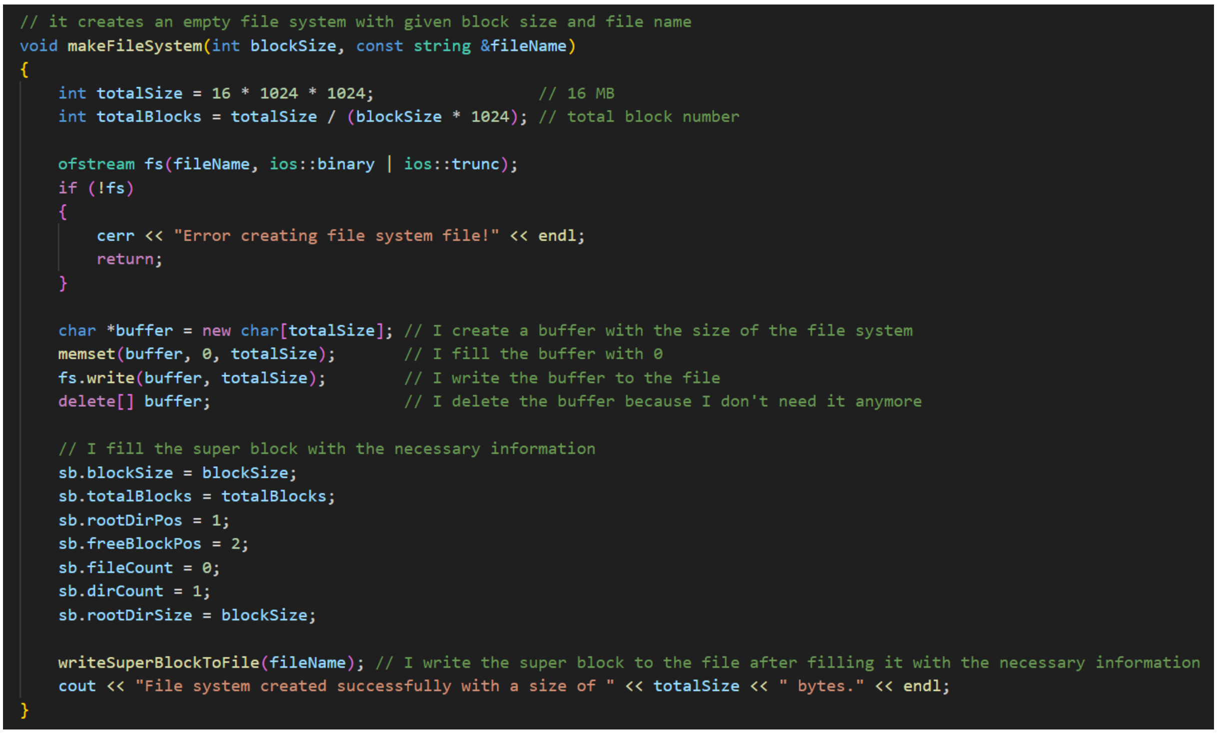
```
struct superBlock
{
    int blockSize;
    int totalBlocks;
    int rootDirPos;   // it keeps the block number where the root directory starts
    int freeBlockPos; // for writing sequentially to the file system I keep free block number
    int fileCount;
    int dirCount;
    int rootDirSize; // root directory size
};
```



```
struct directoryEntry
{
    char fileName[32];
    int blockLocationOfEntry; // it keeps the block number where the file starts
    int fileSize;
    char date[12];
    char time[10];
    bool isDirectory; // flag for directory or file
};
```

Creating a File System (PART 2)

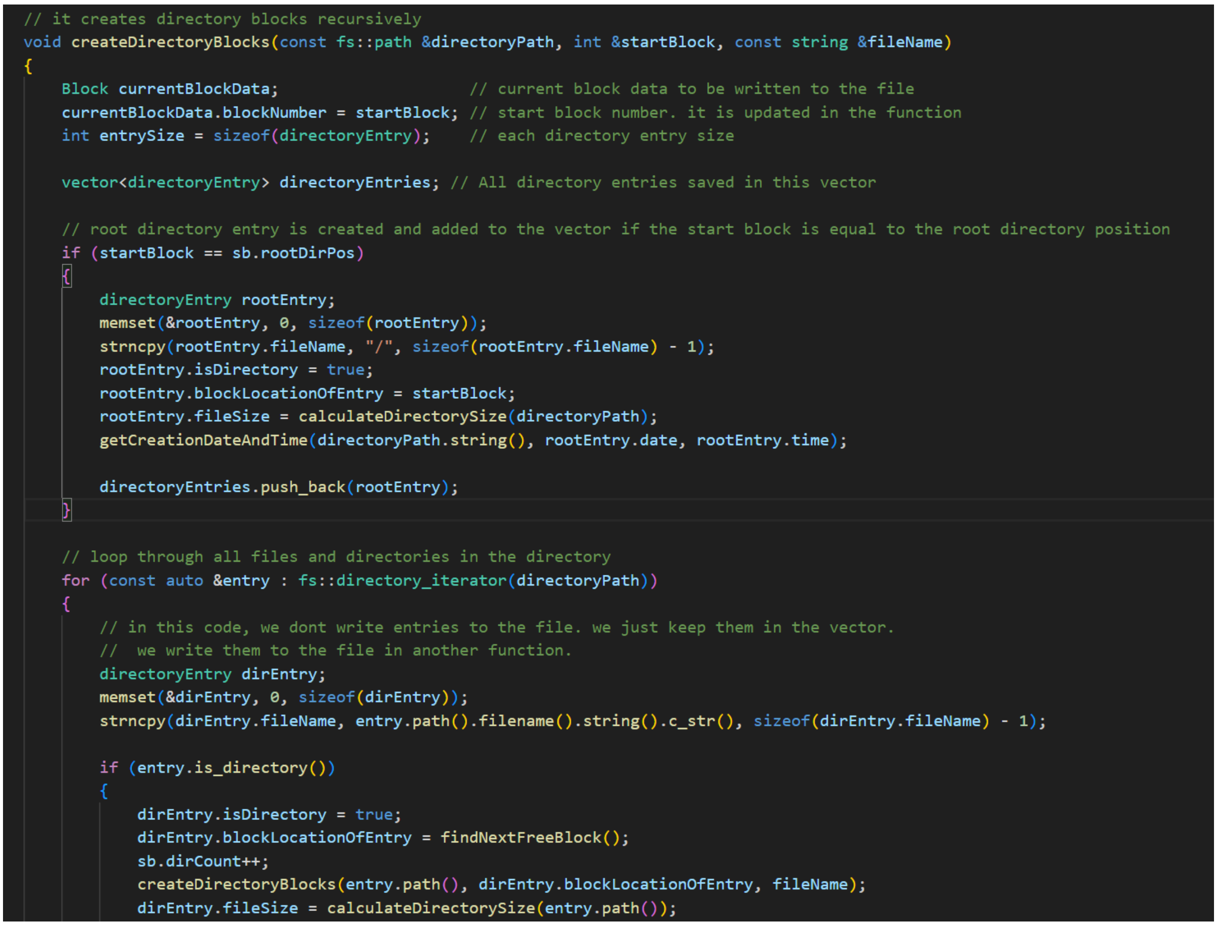
This function creates an empty file system with the specified block size. A file system with a size of 16 MB is created and the super block and other necessary information are stored in this file.



```
// it creates an empty file system with given block size and file name
void makeFileSystem(int blockSize, const string &fileName)
{
    int totalSize = 16 * 1024 * 1024;                 // 16 MB
    int totalBlocks = totalSize / (blockSize * 1024); // total block number

    ofstream fs(fileName, ios::binary | ios::trunc);
    if (!fs)
    {
        cerr << "Error creating file system file!" << endl;
        return;
    }

    char *buffer = new char[totalSize]; // I create a buffer with the size of the file system
    memset(buffer, 0, totalSize);       // I fill the buffer with 0
    fs.write(buffer, totalSize);        // I write the buffer to the file
    delete[] buffer;                    // I delete the buffer because I don't need it anymore

    // I fill the super block with the necessary information
    sb.blockSize = blockSize;
    sb.totalBlocks = totalBlocks;
    sb.rootDirPos = 1;
    sb.freeBlockPos = 2;
    sb.fileCount = 0;
    sb.dirCount = 1;
    sb.rootDirSize = blockSize;

    writeSuperBlockToFile(fileName); // I write the super block to the file after filling it with the necessary information
    cout << "File system created successfully with a size of " << totalSize << " bytes." << endl;
}
```

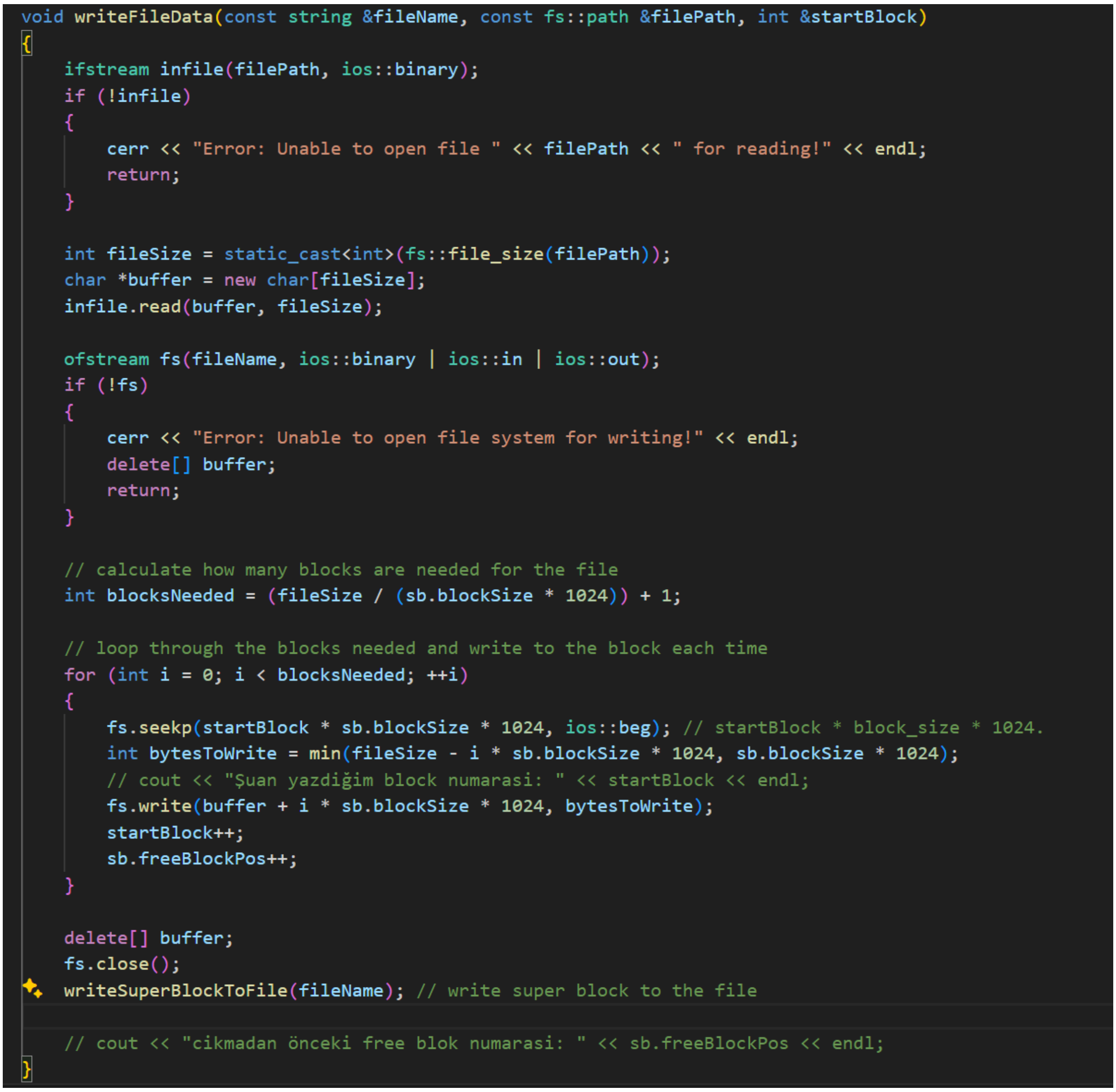
This function writes the data of a particular file to the file system. File data is written in blocks and the necessary information is updated for each block.

In this function, raw data is not written yet. I only create directory blocks that contain entries.



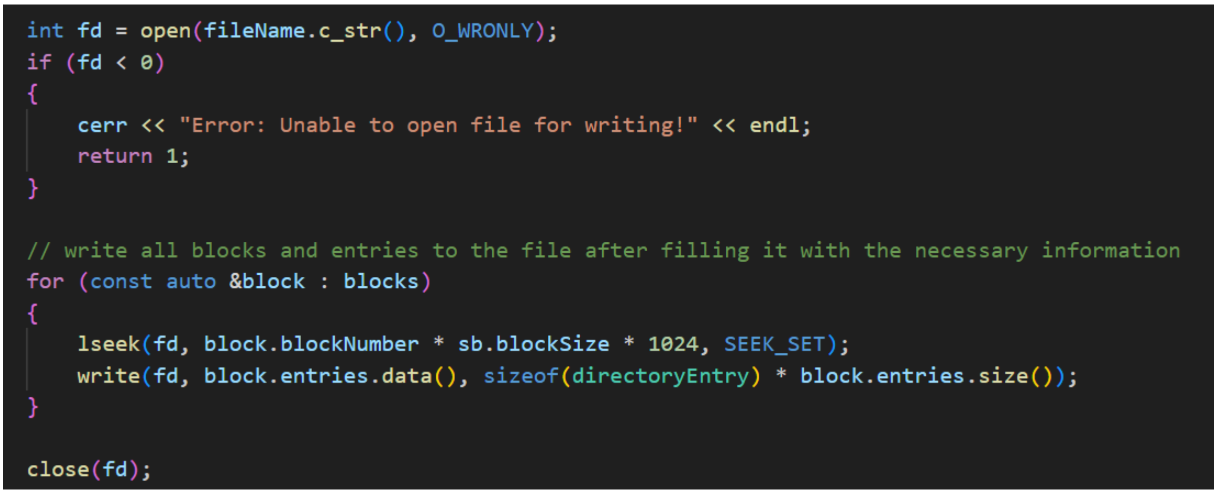
```
// it creates directory blocks recursively
void createDirectoryBlocks(const fs::path &directoryPath, int &startBlock, const string &fileName)
{
    Block currentBlockData;                     // current block data to be written to the file
    currentBlockData.blockNumber = startBlock; // start block number. it is updated in the function
    int entrySize = sizeof(directoryEntry);    // each directory entry size

    vector<directoryEntry> directoryEntries; // All directory entries saved in this vector

    // root directory entry is created and added to the vector if the start block is equal to the root directory position
    if (startBlock == sb.rootDirPos)
    {
        directoryEntry rootEntry;
        memset(&rootEntry, 0, sizeof(rootEntry));
        strncpy(rootEntry.fileName, "/", sizeof(rootEntry.fileName) - 1);
        rootEntry.isDirectory = true;
        rootEntry.blockLocationOfEntry = startBlock;
        rootEntry.fileSize = calculateDirectorySize(directoryPath);
        getCreationDateAndTime(directoryPath.string(), rootEntry.date, rootEntry.time);

        directoryEntries.push_back(rootEntry);
    }

    // loop through all files and directories in the directory
    for (const auto &entry : fs::directory_iterator(directoryPath))
    {
        // in this code, we dont write entries to the file. we just keep them in the vector.
        //  we write them to the file in another function.
        directoryEntry dirEntry;
        memset(&dirEntry, 0, sizeof(dirEntry));
        strncpy(dirEntry.fileName, entry.path().filename().string().c_str(), sizeof(dirEntry.fileName) - 1);

        if (entry.is_directory())
        {
            dirEntry.isDirectory = true;
            dirEntry.blockLocationOfEntry = findNextFreeBlock();
            sb.dirCount++;
            createDirectoryBlocks(entry.path(), dirEntry.blockLocationOfEntry, fileName);
            dirEntry.fileSize = calculateDirectorySize(entry.path());
```

This function transfers the raw data in the files to the file system in blocks.



```cpp
void writeFileData(const string &fileName, const fs::path &filePath, int &startBlock)
{
    ifstream infile(filePath, ios::binary);
    if (!infile)
    {
        cerr << "Error: Unable to open file " << filePath << " for reading!" << endl;
        return;
    }

    int fileSize = static_cast<int>(fs::file_size(filePath));
    char *buffer = new char[fileSize];
    infile.read(buffer, fileSize);

    ofstream fs(fileName, ios::binary | ios::in | ios::out);
    if (!fs)
    {
        cerr << "Error: Unable to open file system for writing!" << endl;
        delete[] buffer;
        return;
    }

    // calculate how many blocks are needed for the file
    int blocksNeeded = (fileSize / (sb.blockSize * 1024)) + 1;

    // loop through the blocks needed and write to the block each time
    for (int i = 0; i < blocksNeeded; ++i)
    {
        fs.seekp(startBlock * sb.blockSize * 1024, ios::beg); // startBlock * block_size * 1024.
        int bytesToWrite = min(fileSize - i * sb.blockSize * 1024, sb.blockSize * 1024);
        // cout << "Şuan yazdiğim block numarasi: " << startBlock << endl;
        fs.write(buffer + i * sb.blockSize * 1024, bytesToWrite);
        startBlock++;
        sb.freeBlockPos++;
    }

    delete[] buffer;
    fs.close();
    writeSuperBlockToFile(fileName); // write super block to the file

    // cout << "cikmadan önceki free blok numarasi: " << sb.freeBlockPos << endl;
}
```

I copy the test folder structure to mySystem.dat.



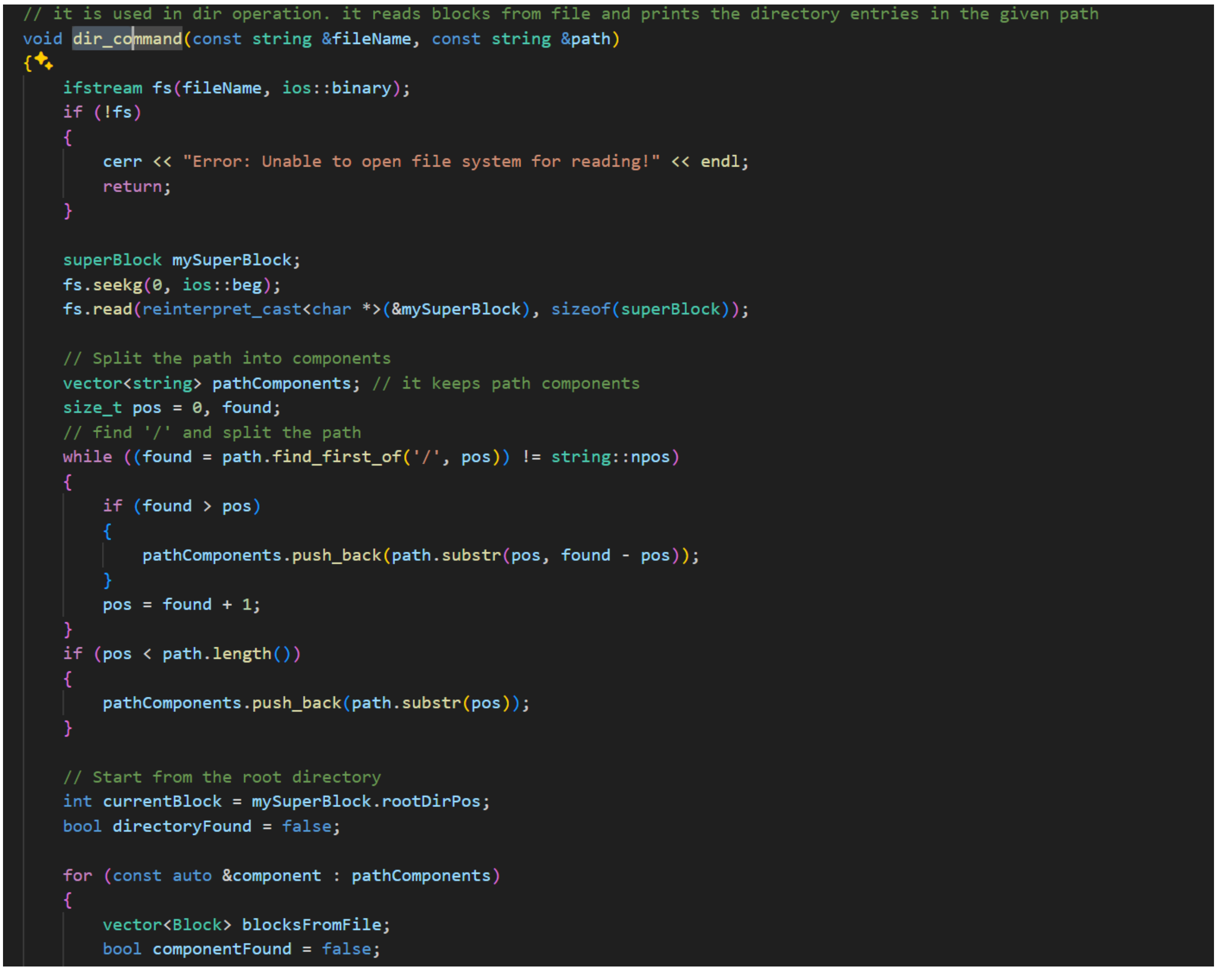
```cpp
int fd = open(fileName.c_str(), O_WRONLY);
if (fd < 0)
{
    cerr << "Error: Unable to open file for writing!" << endl;
    return 1;
}

// write all blocks and entries to the file after filling it with the necessary information
for (const auto &block : blocks)
{
    lseek(fd, block.blockNumber * sb.blockSize * 1024, SEEK_SET);
    write(fd, block.entries.data(), sizeof(directoryEntry) * block.entries.size());
}

close(fd);
```

File System Operations

1. dir Operation

The dir operation lists the contents of a directory at a specified path in the file system.

- The dir_command function is responsible for reading the directory structure from the file system and displaying its contents.
- It first opens the file system, locates the superblock, and splits the provided path into components.
- It navigates through the blocks corresponding to each directory in the path, eventually reaching the final directory block.
- Finally, it lists the entries (files or directories) within the final directory.

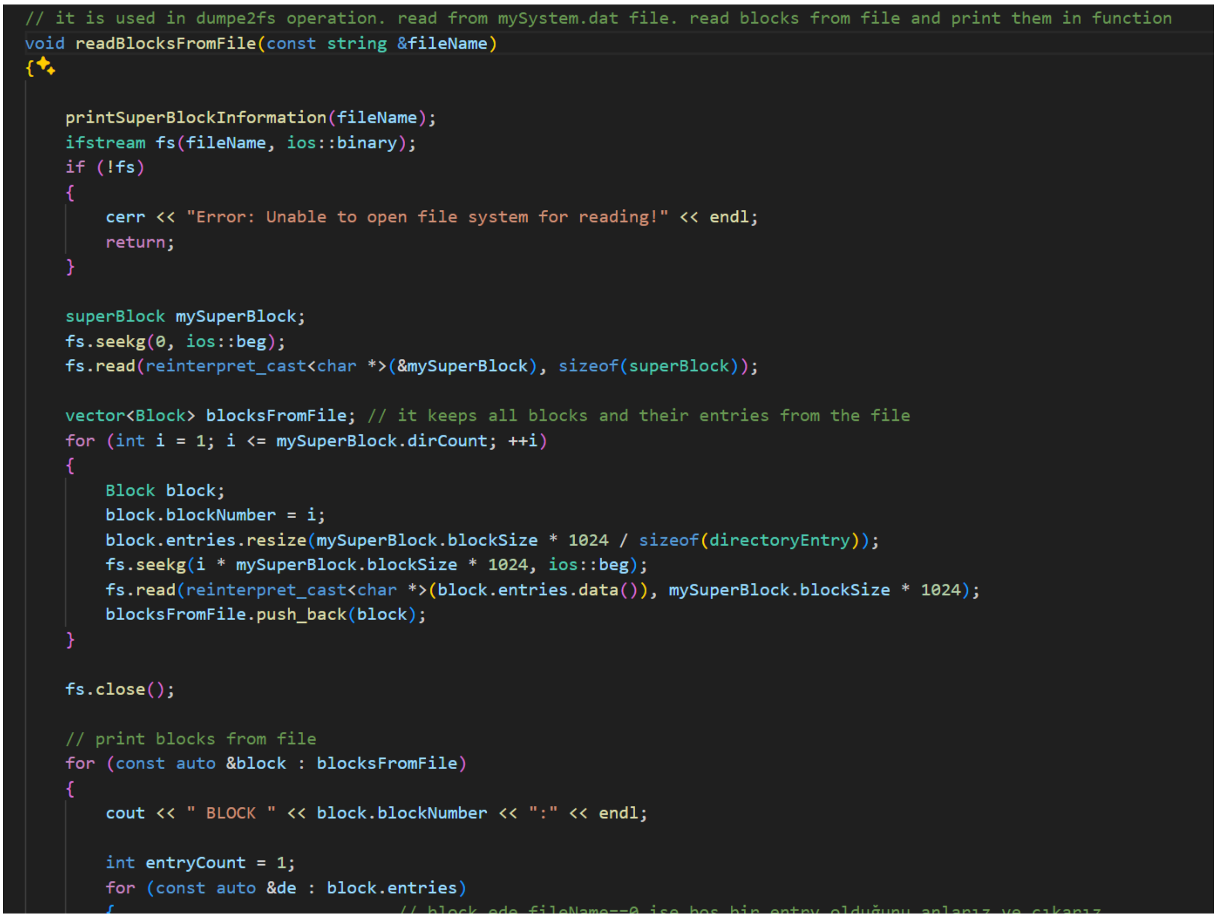


```
// it is used in dir operation. it reads blocks from file and prints the directory entries in the given path
void dir_command(const string &fileName, const string &path)
{
    ifstream fs(fileName, ios::binary);
    if (!fs)
    {
        cerr << "Error: Unable to open file system for reading!" << endl;
        return;
    }

    superBlock mySuperBlock;
    fs.seekg(0, ios::beg);
    fs.read(reinterpret_cast<char *>(&mySuperBlock), sizeof(superBlock));

    // Split the path into components
    vector<string> pathComponents; // it keeps path components
    size_t pos = 0, found;
    // find '/' and split the path
    while ((found = path.find_first_of('/', pos)) != string::npos)
    {
        if (found > pos)
        {
            pathComponents.push_back(path.substr(pos, found - pos));
        }
        pos = found + 1;
    }
    if (pos < path.length())
    {
        pathComponents.push_back(path.substr(pos));
    }

    // Start from the root directory
    int currentBlock = mySuperBlock.rootDirPos;
    bool directoryFound = false;

    for (const auto &component : pathComponents)
    {
        vector<Block> blocksFromFile;
        bool componentFound = false;
```

2. dumpe2fs Operation

The dumpe2fs operation provides a summary of the file system, including block usage and file names.

- The readBlocksFromFile function reads and displays the file system's blocks, including all directory entries.
- It prints the superblock information first, and then iterates over the blocks to display each entry's details, including file name, type (file or directory), size, and creation date/time.
- This operation is akin to the Linux dumpe2fs command, but tailored to your file system's structure.



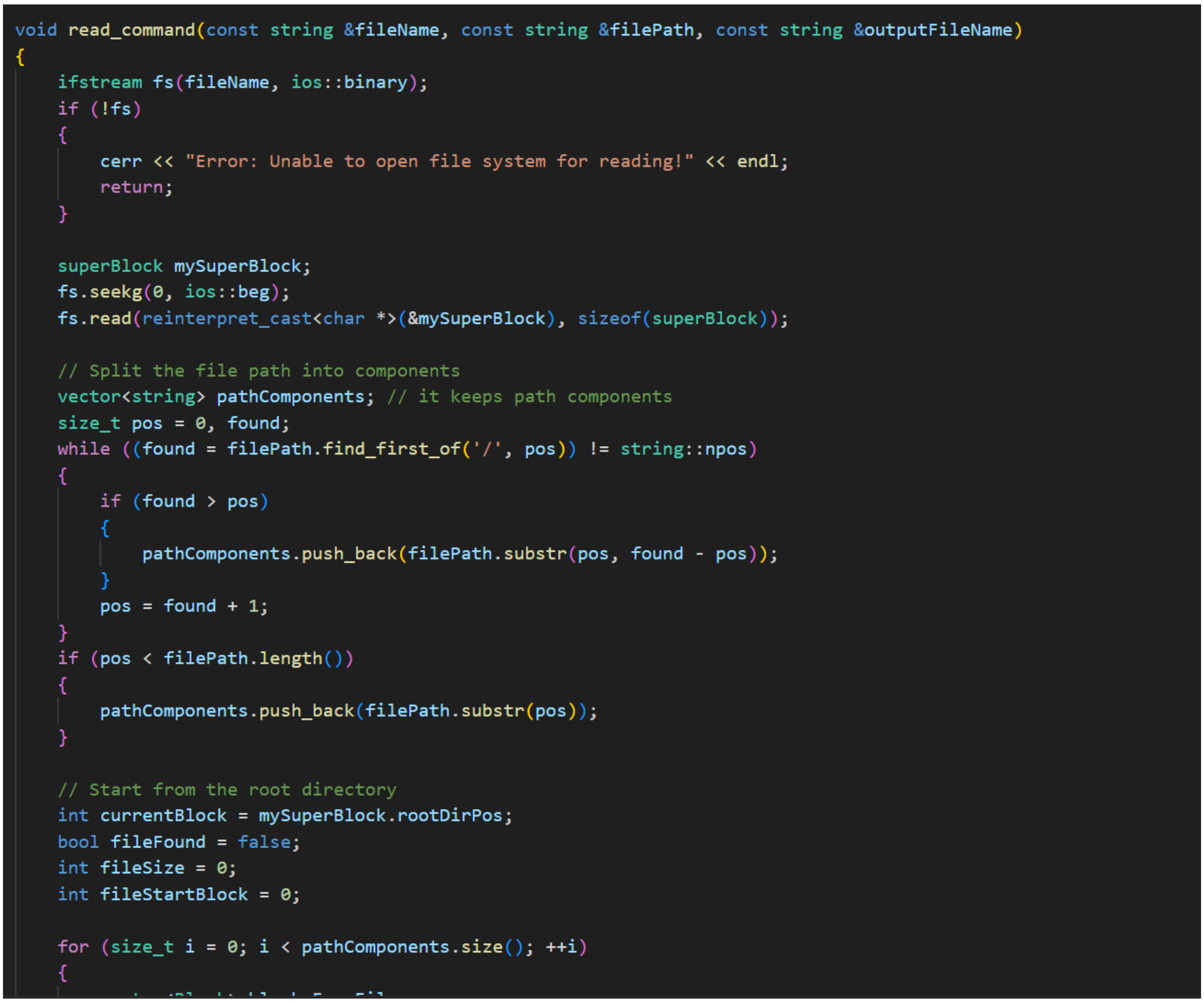
```cpp
// it is used in dumpe2fs operation. read from mySystem.dat file. read blocks from file and print them in function
void readBlocksFromFile(const string &fileName)
{

    printSuperBlockInformation(fileName);
    ifstream fs(fileName, ios::binary);
    if (!fs)
    {
        cerr << "Error: Unable to open file system for reading!" << endl;
        return;
    }

    superBlock mySuperBlock;
    fs.seekg(0, ios::beg);
    fs.read(reinterpret_cast<char *>(&mySuperBlock), sizeof(superBlock));

    vector<Block> blocksFromFile; // it keeps all blocks and their entries from the file
    for (int i = 1; i <= mySuperBlock.dirCount; ++i)
    {
        Block block;
        block.blockNumber = i;
        block.entries.resize(mySuperBlock.blockSize * 1024 / sizeof(directoryEntry));
        fs.seekg(i * mySuperBlock.blockSize * 1024, ios::beg);
        fs.read(reinterpret_cast<char *>(block.entries.data()), mySuperBlock.blockSize * 1024);
        blocksFromFile.push_back(block);
    }

    fs.close();

    // print blocks from file
    for (const auto &block : blocksFromFile)
    {
        cout << " BLOCK " << block.blockNumber << ":" << endl;

        int entryCount = 1;
        for (const auto &de : block.entries)
        {                          // block.ede.fileName==0 ise bos bir entry olduğunu anlarız ve çıkarız
```

3. read Operation

The read operation extracts a file from the file system and saves it to the host's file system.

- The read_command function reads a file from the file system and writes it to a specified output file on the host system.

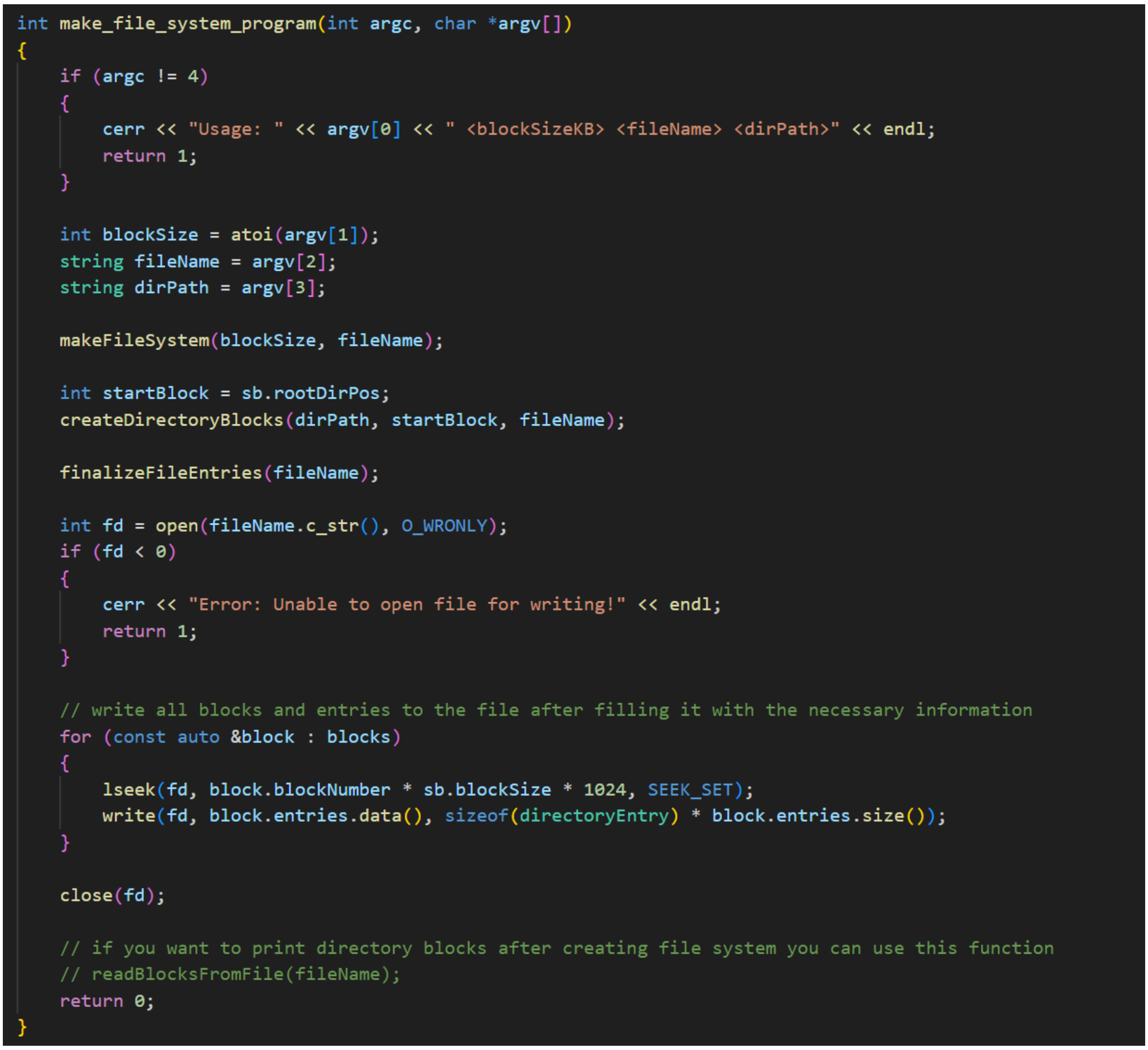
- It navigates through the directory structure to locate the file, determines its size, and reads the corresponding blocks from the file system.
- The data is then written to the specified output file, making it accessible on the host system.



```
void read_command(const string &fileName, const string &filePath, const string &outputFileName)
{
    ifstream fs(fileName, ios::binary);
    if (!fs)
    {
        cerr << "Error: Unable to open file system for reading!" << endl;
        return;
    }

    superBlock mySuperBlock;
    fs.seekg(0, ios::beg);
    fs.read(reinterpret_cast<char *>(&mySuperBlock), sizeof(superBlock));

    // Split the file path into components
    vector<string> pathComponents; // it keeps path components
    size_t pos = 0, found;
    while ((found = filePath.find_first_of('/', pos)) != string::npos)
    {
        if (found > pos)
        {
            pathComponents.push_back(filePath.substr(pos, found - pos));
        }
        pos = found + 1;
    }
    if (pos < filePath.length())
    {
        pathComponents.push_back(filePath.substr(pos));
    }

    // Start from the root directory
    int currentBlock = mySuperBlock.rootDirPos;
    bool fileFound = false;
    int fileSize = 0;
    int fileStartBlock = 0;

    for (size_t i = 0; i < pathComponents.size(); ++i)
    {
```

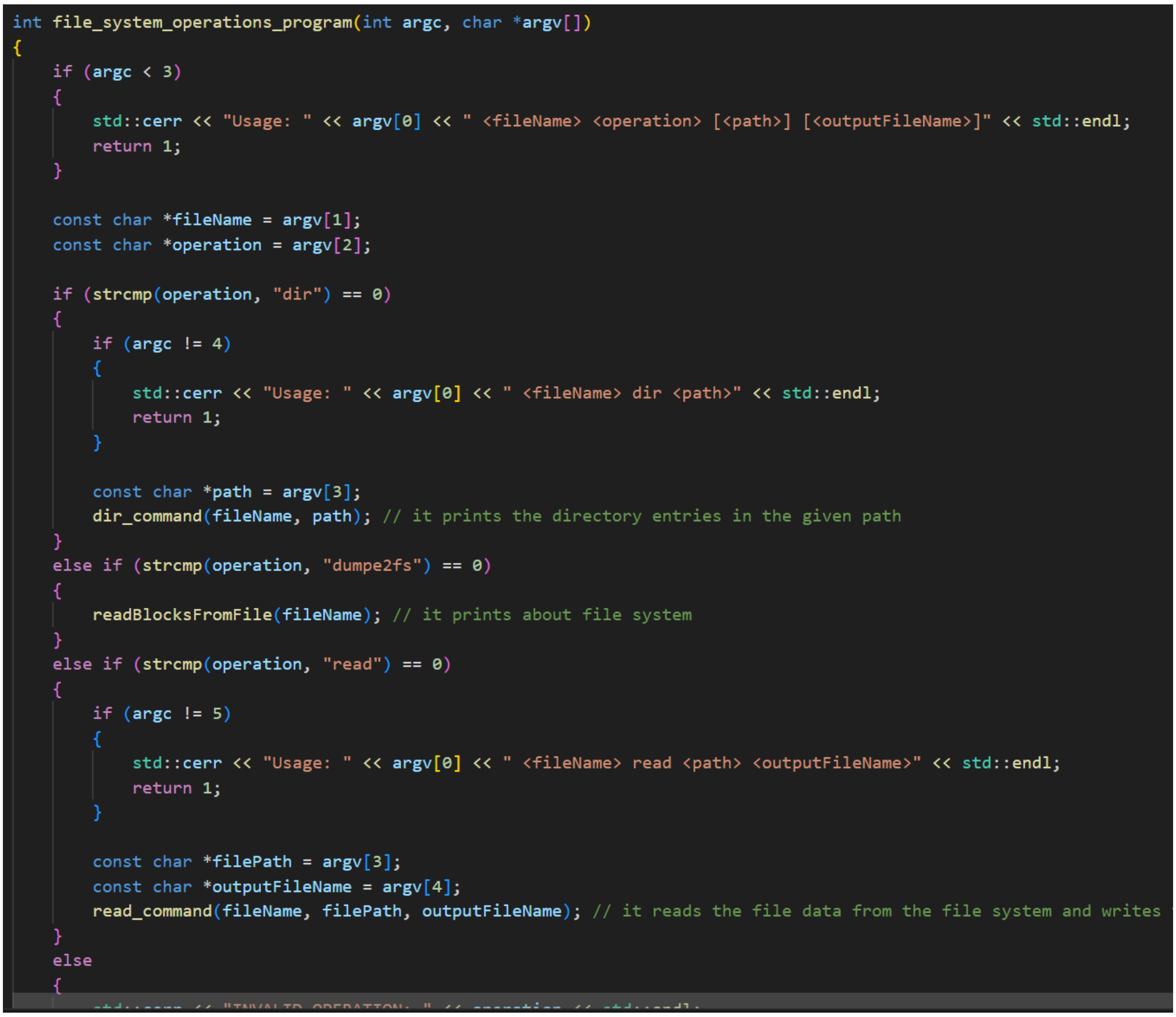
Main Functions

Main funtciton for PART 2 .This function processes the commands and arguments given for part 2.



```
int make_file_system_program(int argc, char *argv[])
{
    if (argc != 4)
    {
        cerr << "Usage: " << argv[0] << " <blockSizeKB> <fileName> <dirPath>" << endl;
        return 1;
    }

    int blockSize = atoi(argv[1]);
    string fileName = argv[2];
    string dirPath = argv[3];

    makeFileSystem(blockSize, fileName);

    int startBlock = sb.rootDirPos;
    createDirectoryBlocks(dirPath, startBlock, fileName);

    finalizeFileEntries(fileName);

    int fd = open(fileName.c_str(), O_WRONLY);
    if (fd < 0)
    {
        cerr << "Error: Unable to open file for writing!" << endl;
        return 1;
    }

    // write all blocks and entries to the file after filling it with the necessary information
    for (const auto &block : blocks)
    {
        lseek(fd, block.blockNumber * sb.blockSize * 1024, SEEK_SET);
        write(fd, block.entries.data(), sizeof(directoryEntry) * block.entries.size());
    }

    close(fd);

    // if you want to print directory blocks after creating file system you can use this function
    // readBlocksFromFile(fileName);
    return 0;
}
```

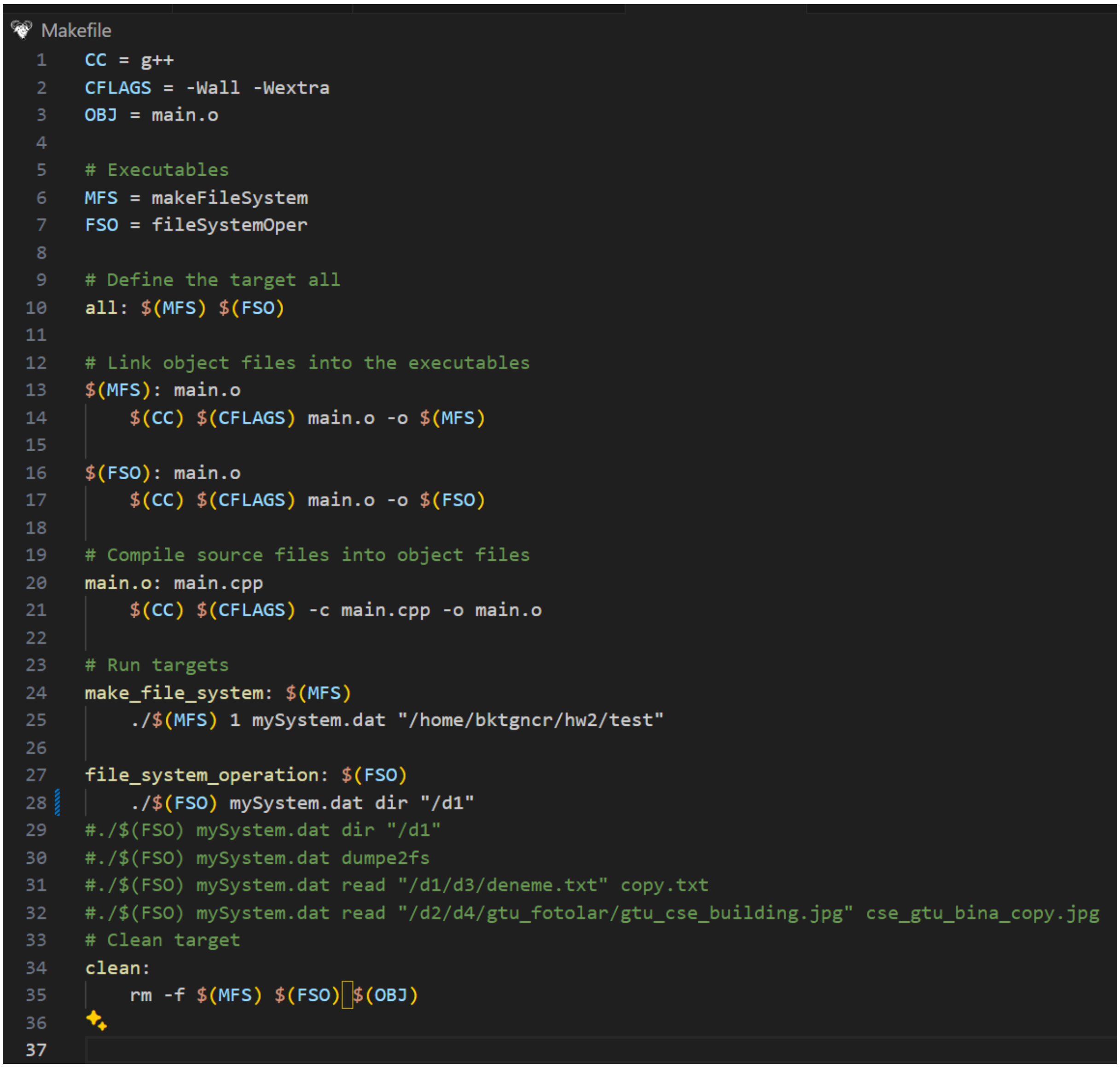
Main function for Part 3 .This function processes the commands and arguments given for part 3.



```
int file_system_operations_program(int argc, char *argv[])
{
    if (argc < 3)
    {
        std::cerr << "Usage: " << argv[0] << " <fileName> <operation> [<path>] [<outputFileName>]" << std::endl;
        return 1;
    }

    const char *fileName = argv[1];
    const char *operation = argv[2];

    if (strcmp(operation, "dir") == 0)
    {
        if (argc != 4)
        {
            std::cerr << "Usage: " << argv[0] << " <fileName> dir <path>" << std::endl;
            return 1;
        }

        const char *path = argv[3];
        dir_command(fileName, path); // it prints the directory entries in the given path
    }
    else if (strcmp(operation, "dumpe2fs") == 0)
    {
        readBlocksFromFile(fileName); // it prints about file system
    }
    else if (strcmp(operation, "read") == 0)
    {
        if (argc != 5)
        {
            std::cerr << "Usage: " << argv[0] << " <fileName> read <path> <outputFileName>" << std::endl;
            return 1;
        }

        const char *filePath = argv[3];
        const char *outputFileName = argv[4];
        read_command(fileName, filePath, outputFileName); // it reads the file data from the file system and writes
    }
    else
    {
        std::cerr << "INVALID OPERATION: " << operation << std::endl;
```

Makefile

The Makefile includes run targets, make_file_system and file_system_operation, which execute the corresponding executables with specific arguments. The make_file_system target runs the makeFileSystem executable with parameters to create a file system, while the file_system_operation target runs the fileSystemOper executable to perform operations such as listing directory contents.



Makefile

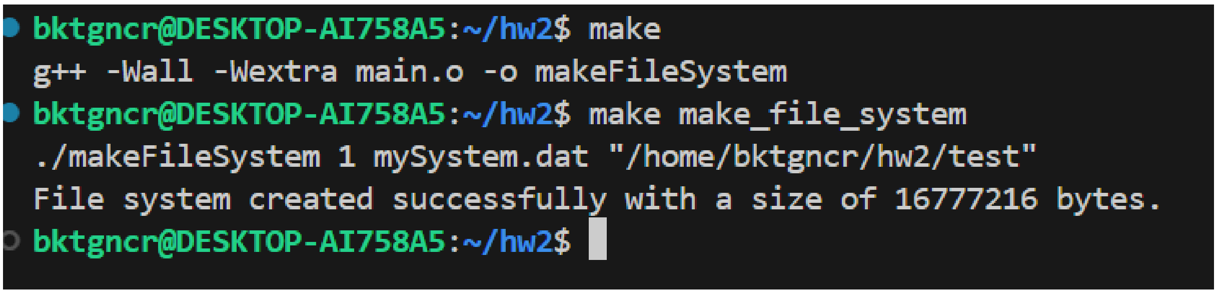
```
CC = g++
CFLAGS = -Wall -Wextra
OBJ = main.o

# Executables
MFS = makeFileSystem
FSO = fileSystemOper

# Define the target all
all: $(MFS) $(FSO)

# Link object files into the executables
$(MFS): main.o
	$(CC) $(CFLAGS) main.o -o $(MFS)

$(FSO): main.o
	$(CC) $(CFLAGS) main.o -o $(FSO)

# Compile source files into object files
main.o: main.cpp
	$(CC) $(CFLAGS) -c main.cpp -o main.o

# Run targets
make_file_system: $(MFS)
	./$(MFS) 1 mySystem.dat "/home/bktgncr/hw2/test"

file_system_operation: $(FSO)
	./$(FSO) mySystem.dat dir "/d1"
#./$(FSO) mySystem.dat dir "/d1"
#./$(FSO) mySystem.dat dumpe2fs
#./$(FSO) mySystem.dat read "/d1/d3/deneme.txt" copy.txt
#./$(FSO) mySystem.dat read "/d2/d4/gtu_fotolar/gtu_cse_building.jpg" cse_gtu_bina_copy.jpg
# Clean target
clean:
	rm -f $(MFS) $(FSO) $(OBJ)

```

Output Result

I'm adding my commands and output results here.

This command is for me to create the file system in part 2. I created mySystem.dat with a size of 16 mb.

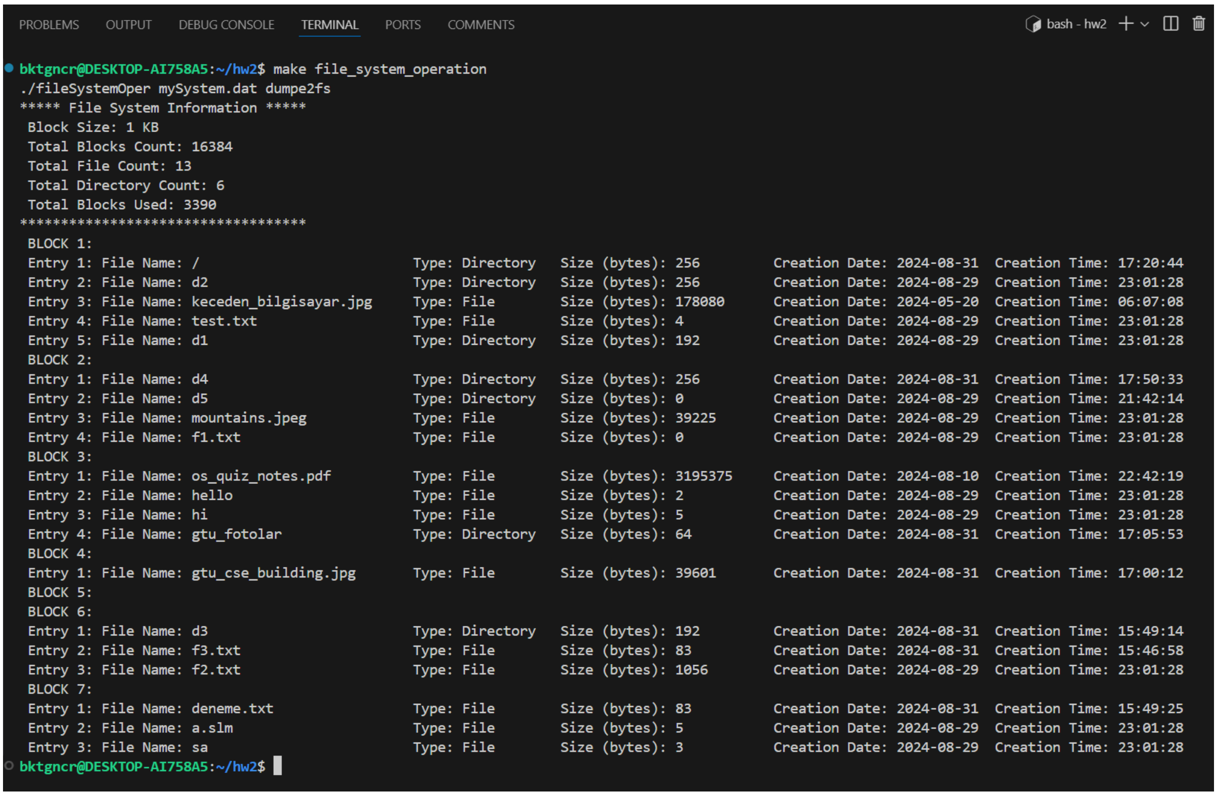
4 mySystem.dat "/home/bktgncr/hw2/test"



```
bktgncr@DESKTOP-AI758A5:~/hw2$ make
g++ -Wall -Wextra main.o -o makeFileSystem
bktgncr@DESKTOP-AI758A5:~/hw2$ make make_file_system
./makeFileSystem 1 mySystem.dat "/home/bktgncr/hw2/test"
File system created successfully with a size of 16777216 bytes.
bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

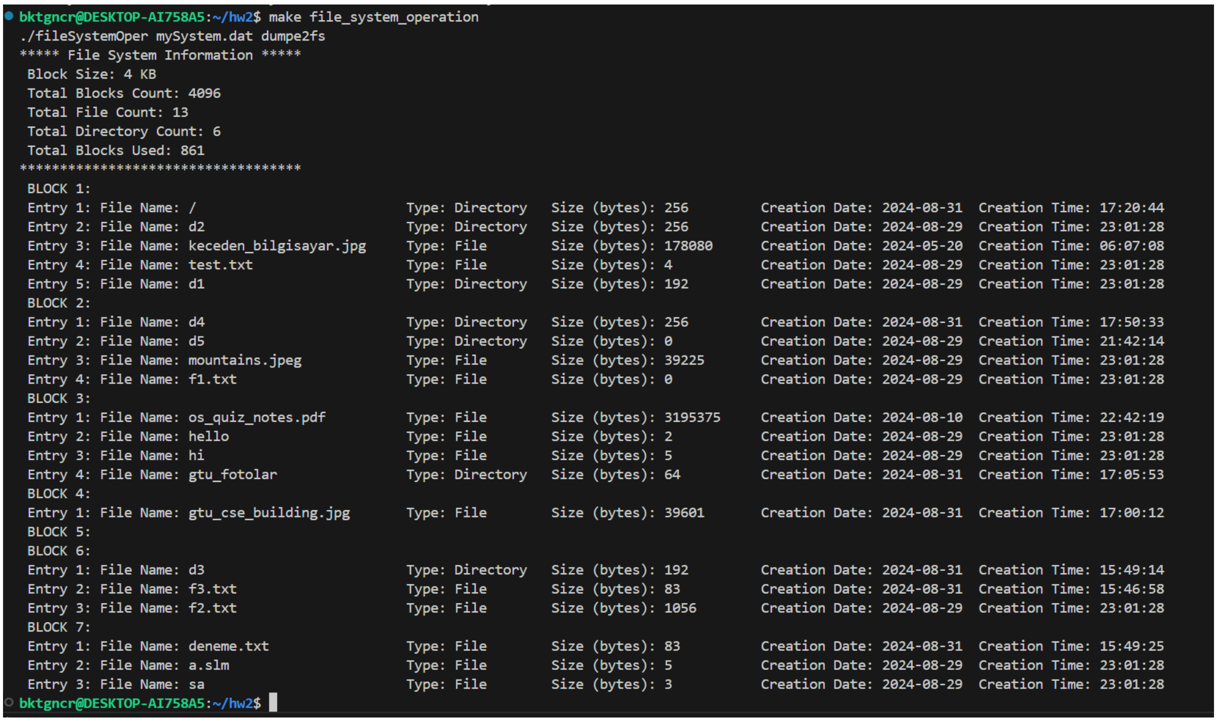
mySystem.dat dumpe2fs

1kb block size . dumpe2fs result:



```
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
./fileSystemOper mySystem.dat dumpe2fs
***** File System Information *****
 Block Size: 1 KB
 Total Blocks Count: 16384
 Total File Count: 13
 Total Directory Count: 6
 Total Blocks Used: 3390
***********************************
 BLOCK 1:
 Entry 1: File Name: /                          Type: Directory   Size (bytes): 256         Creation Date: 2024-08-31  Creation Time: 17:20:44
 Entry 2: File Name: d2                         Type: Directory   Size (bytes): 256         Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: keceden_bilgisayar.jpg     Type: File        Size (bytes): 178080      Creation Date: 2024-05-20  Creation Time: 06:07:08
 Entry 4: File Name: test.txt                   Type: File        Size (bytes): 4           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 5: File Name: d1                         Type: Directory   Size (bytes): 192         Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 2:
 Entry 1: File Name: d4                         Type: Directory   Size (bytes): 256         Creation Date: 2024-08-31  Creation Time: 17:50:33
 Entry 2: File Name: d5                         Type: Directory   Size (bytes): 0           Creation Date: 2024-08-29  Creation Time: 21:42:14
 Entry 3: File Name: mountains.jpeg             Type: File        Size (bytes): 39225       Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 4: File Name: f1.txt                     Type: File        Size (bytes): 0           Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 3:
 Entry 1: File Name: os_quiz_notes.pdf          Type: File        Size (bytes): 3195375     Creation Date: 2024-08-10  Creation Time: 22:42:19
 Entry 2: File Name: hello                      Type: File        Size (bytes): 2           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: hi                         Type: File        Size (bytes): 5           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 4: File Name: gtu_fotolar                Type: Directory   Size (bytes): 64          Creation Date: 2024-08-31  Creation Time: 17:05:53
 BLOCK 4:
 Entry 1: File Name: gtu_cse_building.jpg       Type: File        Size (bytes): 39601       Creation Date: 2024-08-31  Creation Time: 17:00:12
 BLOCK 5:
 BLOCK 6:
 Entry 1: File Name: d3                         Type: Directory   Size (bytes): 192         Creation Date: 2024-08-31  Creation Time: 15:49:14
 Entry 2: File Name: f3.txt                     Type: File        Size (bytes): 83          Creation Date: 2024-08-31  Creation Time: 15:46:58
 Entry 3: File Name: f2.txt                     Type: File        Size (bytes): 1056        Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 7:
 Entry 1: File Name: deneme.txt                 Type: File        Size (bytes): 83          Creation Date: 2024-08-31  Creation Time: 15:49:25
 Entry 2: File Name: a.slm                      Type: File        Size (bytes): 5           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: sa                         Type: File        Size (bytes): 3           Creation Date: 2024-08-29  Creation Time: 23:01:28
bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

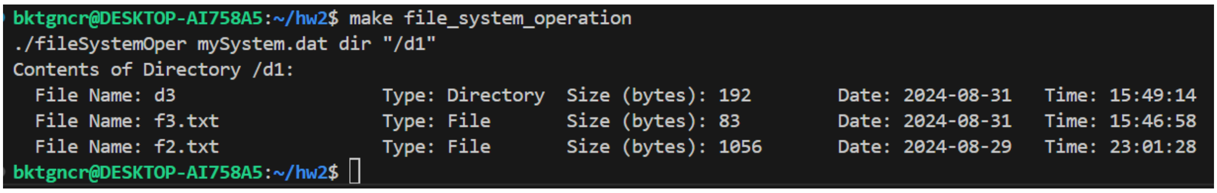
4 kb block size . dumpe2fs result:



```
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
./fileSystemOper mySystem.dat dumpe2fs
***** File System Information *****
 Block Size: 4 KB
 Total Blocks Count: 4096
 Total File Count: 13
 Total Directory Count: 6
 Total Blocks Used: 861
***********************************
 BLOCK 1:
 Entry 1: File Name: /                          Type: Directory   Size (bytes): 256         Creation Date: 2024-08-31  Creation Time: 17:20:44
 Entry 2: File Name: d2                         Type: Directory   Size (bytes): 256         Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: keceden_bilgisayar.jpg     Type: File        Size (bytes): 178080      Creation Date: 2024-05-20  Creation Time: 06:07:08
 Entry 4: File Name: test.txt                   Type: File        Size (bytes): 4           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 5: File Name: d1                         Type: Directory   Size (bytes): 192         Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 2:
 Entry 1: File Name: d4                         Type: Directory   Size (bytes): 256         Creation Date: 2024-08-31  Creation Time: 17:50:33
 Entry 2: File Name: d5                         Type: Directory   Size (bytes): 0           Creation Date: 2024-08-29  Creation Time: 21:42:14
 Entry 3: File Name: mountains.jpeg             Type: File        Size (bytes): 39225       Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 4: File Name: f1.txt                     Type: File        Size (bytes): 0           Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 3:
 Entry 1: File Name: os_quiz_notes.pdf          Type: File        Size (bytes): 3195375     Creation Date: 2024-08-10  Creation Time: 22:42:19
 Entry 2: File Name: hello                      Type: File        Size (bytes): 2           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: hi                         Type: File        Size (bytes): 5           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 4: File Name: gtu_fotolar                Type: Directory   Size (bytes): 64          Creation Date: 2024-08-31  Creation Time: 17:05:53
 BLOCK 4:
 Entry 1: File Name: gtu_cse_building.jpg       Type: File        Size (bytes): 39601       Creation Date: 2024-08-31  Creation Time: 17:00:12
 BLOCK 5:
 BLOCK 6:
 Entry 1: File Name: d3                         Type: Directory   Size (bytes): 192         Creation Date: 2024-08-31  Creation Time: 15:49:14
 Entry 2: File Name: f3.txt                     Type: File        Size (bytes): 83          Creation Date: 2024-08-31  Creation Time: 15:46:58
 Entry 3: File Name: f2.txt                     Type: File        Size (bytes): 1056        Creation Date: 2024-08-29  Creation Time: 23:01:28
 BLOCK 7:
 Entry 1: File Name: deneme.txt                 Type: File        Size (bytes): 83          Creation Date: 2024-08-31  Creation Time: 15:49:25
 Entry 2: File Name: a.slm                      Type: File        Size (bytes): 5           Creation Date: 2024-08-29  Creation Time: 23:01:28
 Entry 3: File Name: sa                         Type: File        Size (bytes): 3           Creation Date: 2024-08-29  Creation Time: 23:01:28
bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

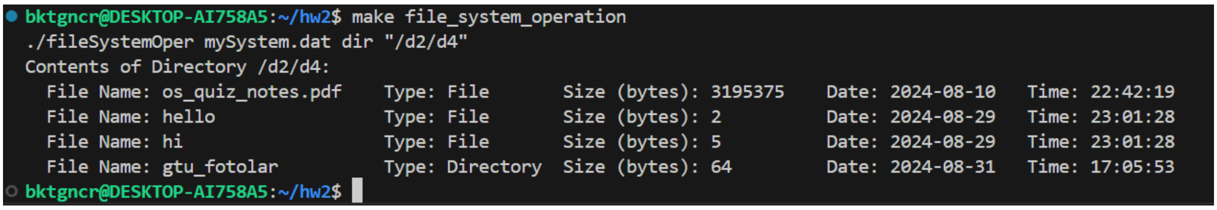
My only problem at this stage is this: the root directory appears as an entry in the 1st block. I didn't need this when I created the file system, but I added it in the "dir" function to follow the path correctly. **This entry still never disrupts my file system structure and consistency.**

mySystem.dat dir "/d1"



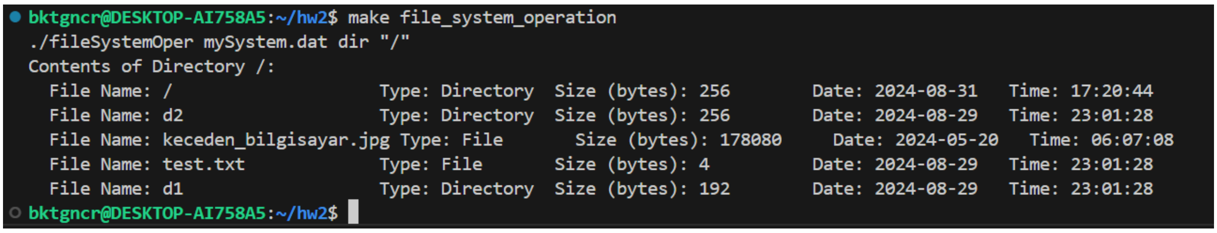
```
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
./fileSystemOper mySystem.dat dir "/d1"
Contents of Directory /d1:
  File Name: d3                Type: Directory  Size (bytes): 192       Date: 2024-08-31   Time: 15:49:14
  File Name: f3.txt            Type: File       Size (bytes): 83        Date: 2024-08-31   Time: 15:46:58
  File Name: f2.txt            Type: File       Size (bytes): 1056      Date: 2024-08-29   Time: 23:01:28
bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

mySystem.dat dir "/d2/d4"



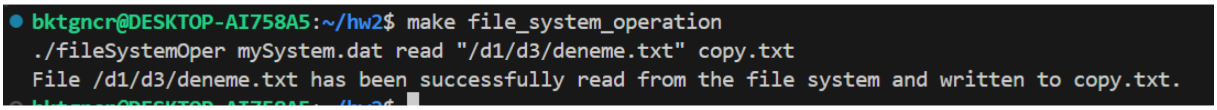
```
● bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
  ./fileSystemOper mySystem.dat dir "/d2/d4"
  Contents of Directory /d2/d4:
    File Name: os_quiz_notes.pdf   Type: File       Size (bytes): 3195375   Date: 2024-08-10   Time: 22:42:19
    File Name: hello               Type: File       Size (bytes): 2         Date: 2024-08-29   Time: 23:01:28
    File Name: hi                  Type: File       Size (bytes): 5         Date: 2024-08-29   Time: 23:01:28
    File Name: gtu_fotolar         Type: Directory  Size (bytes): 64        Date: 2024-08-31   Time: 17:05:53
○ bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

mySystem.dat dir "/"



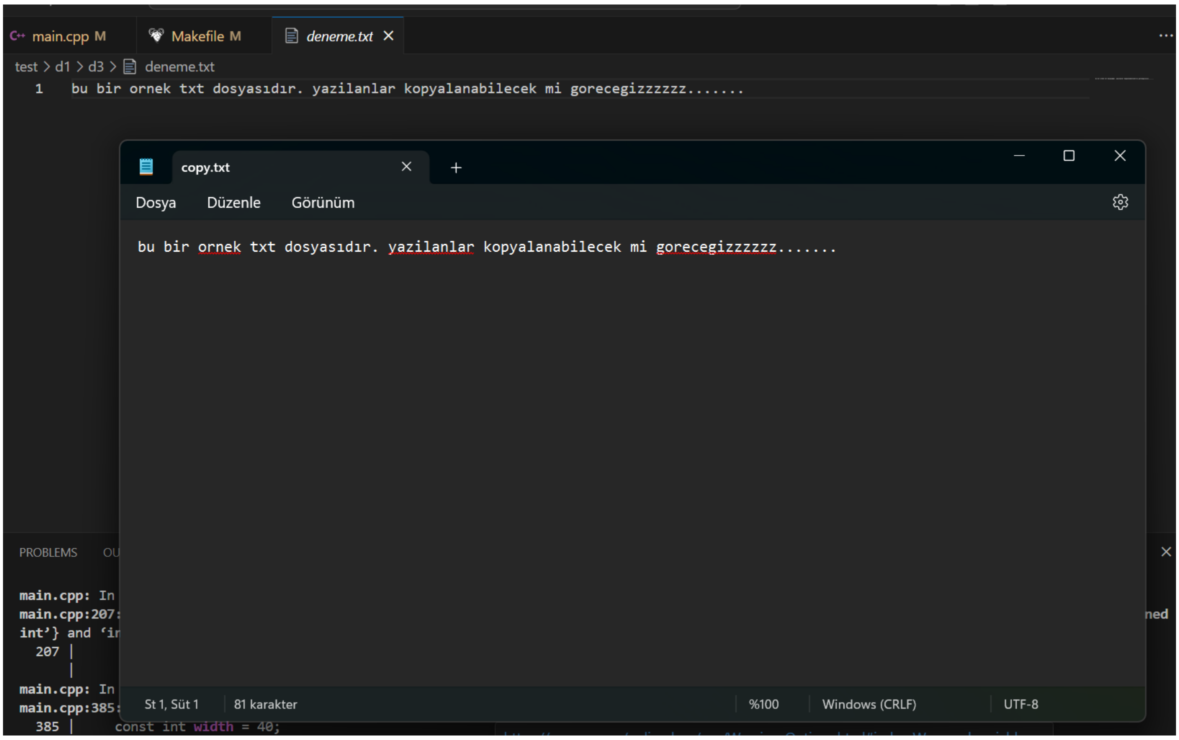
```
● bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
  ./fileSystemOper mySystem.dat dir "/"
  Contents of Directory /:
    File Name: /                   Type: Directory  Size (bytes): 256       Date: 2024-08-31   Time: 17:20:44
    File Name: d2                  Type: Directory  Size (bytes): 256       Date: 2024-08-29   Time: 23:01:28
    File Name: keceden_bilgisayar.jpg Type: File       Size (bytes): 178080    Date: 2024-05-20   Time: 06:07:08
    File Name: test.txt            Type: File       Size (bytes): 4         Date: 2024-08-29   Time: 23:01:28
    File Name: d1                  Type: Directory  Size (bytes): 192       Date: 2024-08-29   Time: 23:01:28
○ bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

mySystem.dat read "/d1/d3/deneme.txt" copy.txt



```
● bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
  ./fileSystemOper mySystem.dat read "/d1/d3/deneme.txt" copy.txt
  File /d1/d3/deneme.txt has been successfully read from the file system and written to copy.txt.
○ bktgncr@DESKTOP-AI758A5:~/hw2$ 
```

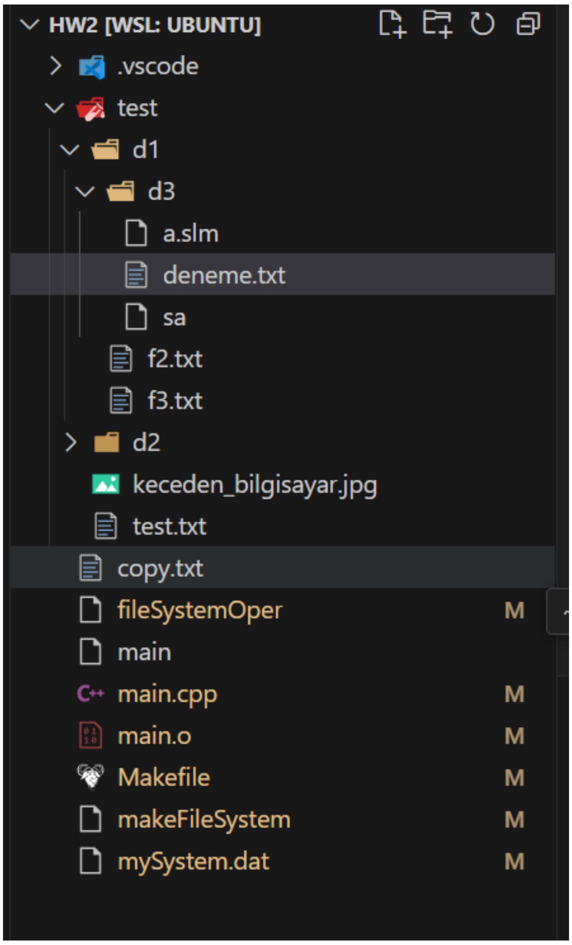
The images below show the original txt file and the contents of the file we copied.



```
bu bir ornek txt dosyasıdır. yazilanlar kopyalanabilecek mi gorecegizzzzzz.......
```

```
bu bir ornek txt dosyasıdır. yazilanlar kopyalanabilecek mi gorecegizzzzzz.......
```

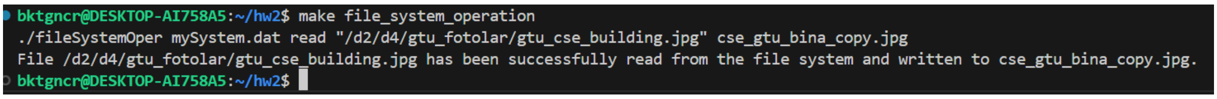
```
main.cpp: In
main.cpp:207:
int'} and 'in
  207 |
      |
main.cpp: In
main.cpp:385:
  385 |     const int width = 40;
```



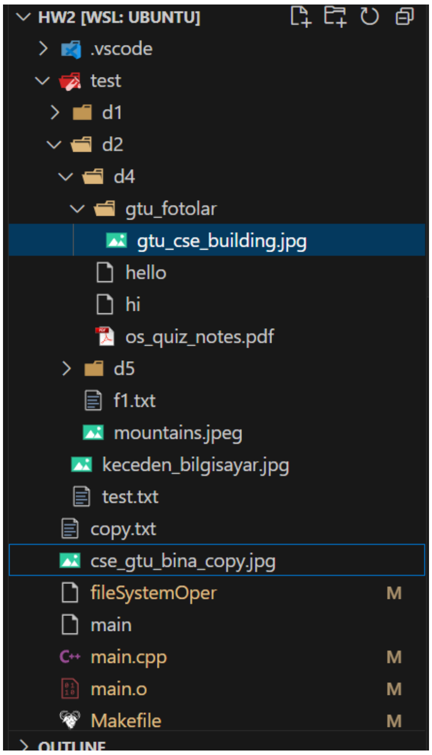
```
HW2 [WSL: UBUNTU]
  > .vscode
  v test
    v d1
      v d3
          a.slm
          deneme.txt
          sa
        f2.txt
        f3.txt
    > d2
      keceden_bilgisayar.jpg
      test.txt
    copy.txt
    fileSystemOper          M
    main
    main.cpp                M
    main.o                  M
    Makefile                M
    makeFileSystem          M
    mySystem.dat            M
```

mySystem.dat read "/d2/d4/gtu_fotolar/gtu_cse_building.jpg" cse_gtu_bina_copy.jpg

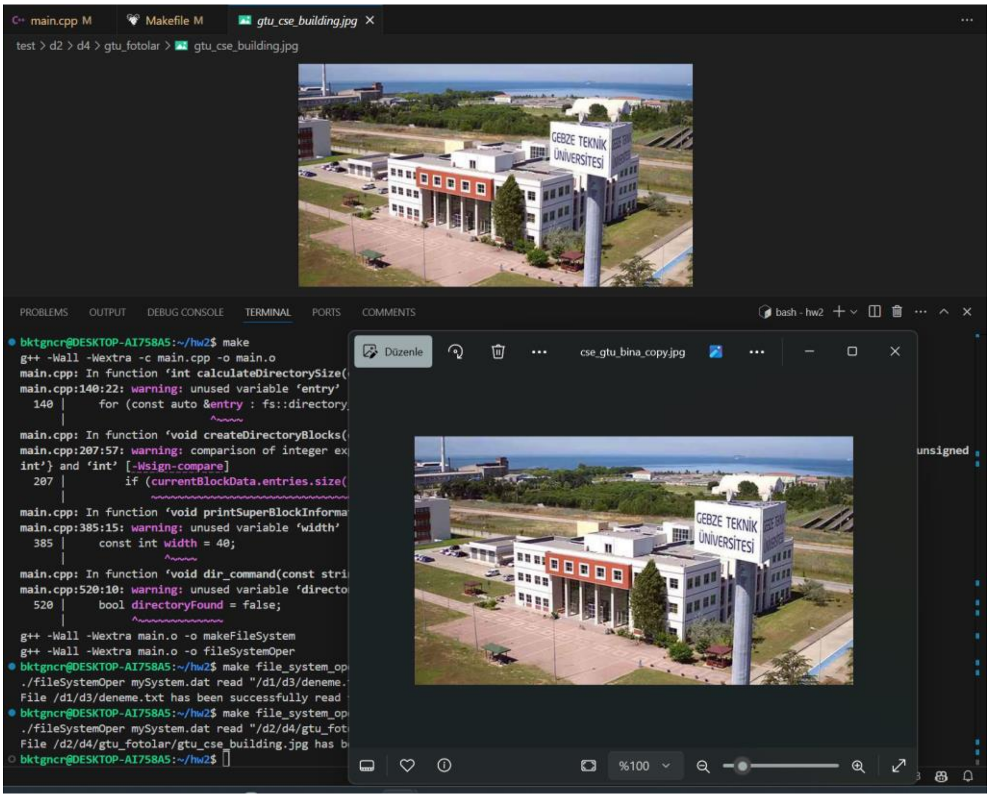
I copied the jpeg file here. I share the original and copy file by taking screenshots.



```
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
./fileSystemOper mySystem.dat read "/d2/d4/gtu_fotolar/gtu_cse_building.jpg" cse_gtu_bina_copy.jpg
File /d2/d4/gtu_fotolar/gtu_cse_building.jpg has been successfully read from the file system and written to cse_gtu_bina_copy.jpg.
bktgncr@DESKTOP-AI758A5:~/hw2$
```



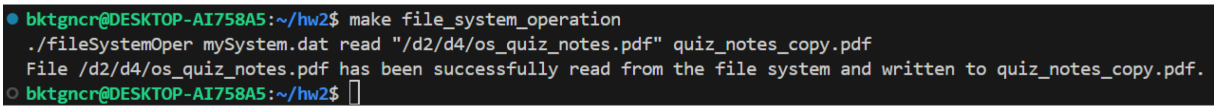
```
HW2 [WSL: UBUNTU]
  > .vscode
  v test
    > d1
    v d2
      v d4
        v gtu_fotolar
            gtu_cse_building.jpg
          hello
          hi
          os_quiz_notes.pdf
      > d5
      f1.txt
      mountains.jpeg
    keceden_bilgisayar.jpg
    test.txt
  copy.txt
  cse_gtu_bina_copy.jpg
  fileSystemOper          M
  main
  main.cpp                M
  main.o                  M
  Makefile                M
> OUTLINE
```



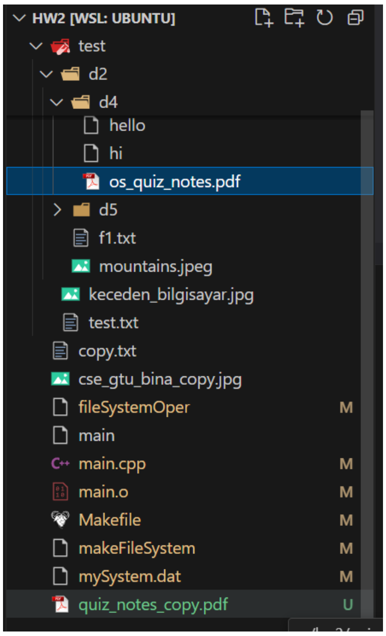
```
bktgncr@DESKTOP-AI758A5:~/hw2$ make
g++ -Wall -Wextra -c main.cpp -o main.o
main.cpp: In function ‘int calculateDirectorySize(
main.cpp:140:22: warning: unused variable ‘entry’
  140 |     for (const auto &entry : fs::directory
      |                      ^~~~~
main.cpp: In function ‘void createDirectoryBlocks(
main.cpp:207:57: warning: comparison of integer ex                unsigned
int’} and ‘int’ [-Wsign-compare]
  207 |         if (currentBlockData.entries.size(
      |             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.cpp: In function ‘void printSuperBlockInforma
main.cpp:385:15: warning: unused variable ‘width’
  385 |     const int width = 40;
      |               ^~~~~
main.cpp: In function ‘void dir_command(const stri
main.cpp:520:10: warning: unused variable ‘directo
  520 |     bool directoryFound = false;
      |          ^~~~~~~~~~~~~~
g++ -Wall -Wextra main.o -o makeFileSystem
g++ -Wall -Wextra main.o -o fileSystemOper
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_op
./fileSystemOper mySystem.dat read "/d1/d3/deneme.
File /d1/d3/deneme.txt has been successfully read
bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_op
./fileSystemOper mySystem.dat read "/d2/d4/gtu_fot
File /d2/d4/gtu_fotolar/gtu_cse_building.jpg has b
bktgncr@DESKTOP-AI758A5:~/hw2$
```

mySystem.dat read "/d2/d4/os_quiz_notes.pdf" quiz_notes_copy.pdf

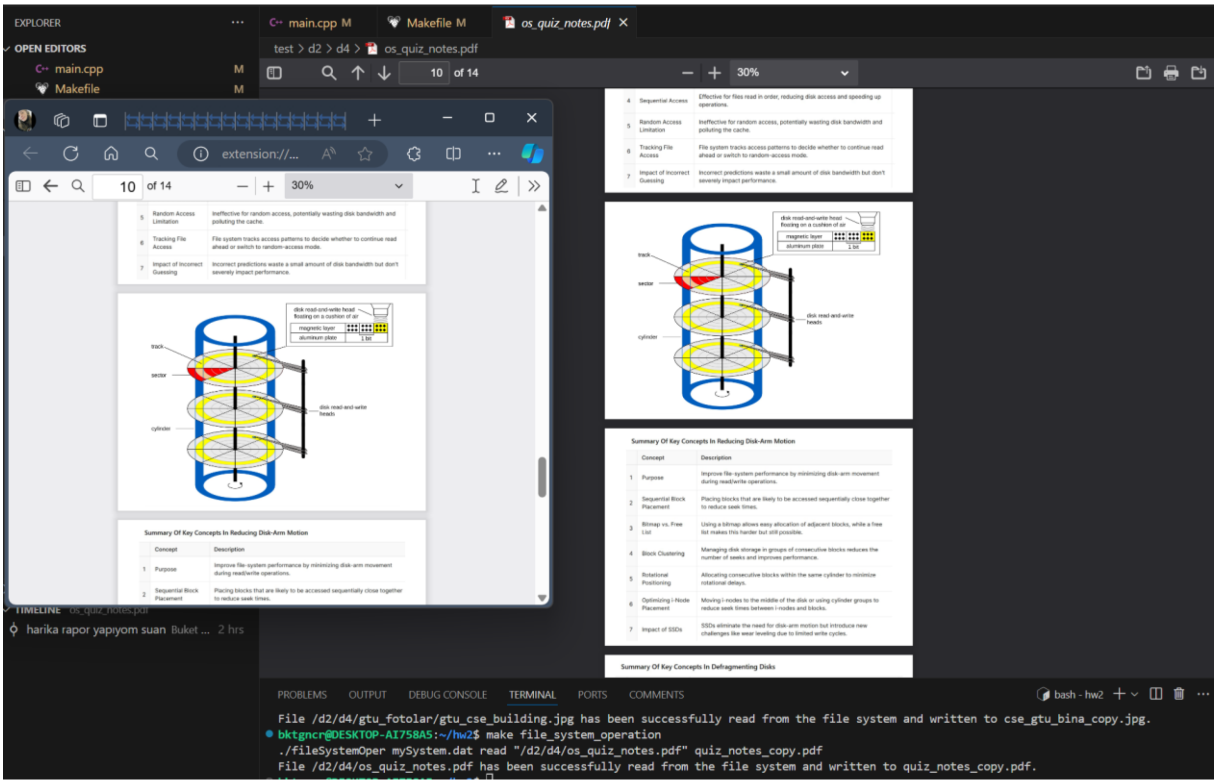
I am copying the pdf file here.



```
● bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
  ./fileSystemOper mySystem.dat read "/d2/d4/os_quiz_notes.pdf" quiz_notes_copy.pdf
  File /d2/d4/os_quiz_notes.pdf has been successfully read from the file system and written to quiz_notes_copy.pdf.
○ bktgncr@DESKTOP-AI758A5:~/hw2$ 
```



```
HW2 [WSL: UBUNTU]
  test
    d2
      d4
        hello
        hi
        os_quiz_notes.pdf
      d5
      f1.txt
      mountains.jpeg
    keceden_bilgisayar.jpg
    test.txt
  copy.txt
  cse_gtu_bina_copy.jpg
  fileSystemOper          M
  main
  main.cpp                M
  main.o                  M
  Makefile                M
  makeFileSystem          M
  mySystem.dat            M
  quiz_notes_copy.pdf     U
```



4	Sequential Access	Effective for files read in order, reducing disk access and speeding up operations.
5	Random Access Limitation	Ineffective for random access, potentially wasting disk bandwidth and polluting the cache.
6	Tracking File Access	File system tracks access patterns to decide whether to continue read ahead or switch to random-access mode.
7	Impact of Incorrect Guessing	Incorrect predictions waste a small amount of disk bandwidth but don't severely impact performance.

Summary Of Key Concepts In Reducing Disk-Arm Motion

	Concept	Description
1	Purpose	Improve file-system performance by minimizing disk-arm movement during read/write operations.
2	Sequential Block Placement	Placing blocks that are likely to be accessed sequentially close together to reduce seek times.
3	Bitmap vs. Free List	Using a bitmap allows easy allocation of adjacent blocks, while a free list makes this harder but still possible.
4	Block Clustering	Managing disk storage in groups of consecutive blocks reduces the number of seeks and improves performance.
5	Rotational Positioning	Allocating consecutive blocks within the same cylinder to minimize rotational delays.
6	Optimizing i-Node Placement	Moving i-nodes to the middle of the disk or using cylinder groups to reduce seek times between i-nodes and blocks.
7	Impact of SSDs	SSDs eliminate the need for disk-arm motion but introduce new challenges like wear leveling due to limited write cycles.

Summary Of Key Concepts In Defragmenting Disks

```
File /d2/d4/gtu_fotolar/gtu_cse_building.jpg has been successfully read from the file system and written to cse_gtu_bina_copy.jpg.
● bktgncr@DESKTOP-AI758A5:~/hw2$ make file_system_operation
  ./fileSystemOper mySystem.dat read "/d2/d4/os_quiz_notes.pdf" quiz_notes_copy.pdf
  File /d2/d4/os_quiz_notes.pdf has been successfully read from the file system and written to quiz_notes_copy.pdf.
```