GIT Department of Computer Engineering
CSE 312 – Spring 2023

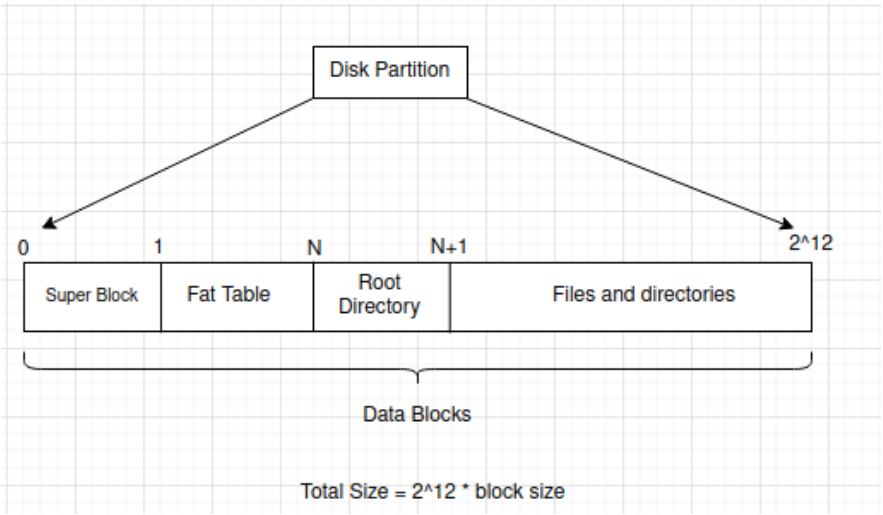
Homework 3 #Report

Okan Torun
1801042662

File System Design

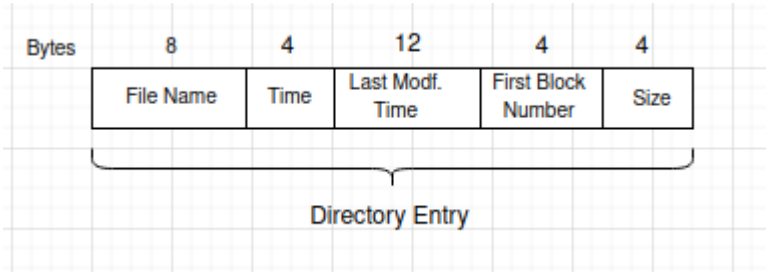
In the homework, 6 different structures are designed. Directory Entry ,Directory Table,Fat12 Entry,Fat12 Table,Super Block and Data Block , This file system is designed in accordance with FAT12 structure. Details are mentioned below.

File-System Layout



File-System Layout was designed as above. Data blocks include Super BLock, Fat Table, Root Directory and Files and directories, respectively. Super Block and Root Directory are designed to be represented in a data block. According to the block size value, the number of blocks occupied by Fat Table varies. Since the Fat12 structure is used, there are 2^12 data blocks.

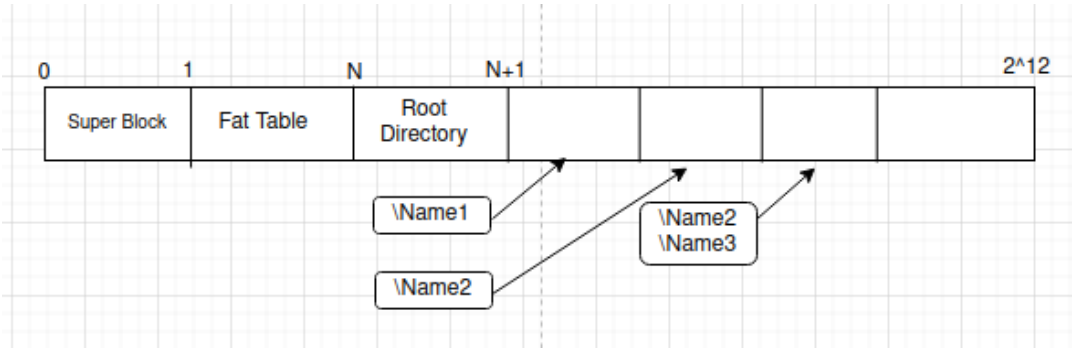
Directory Entry



The FAT structure uses the MS-DOS directory entry structure. Therefore, the directory entry structure is designed accordingly and is as above.Each added file has these properties.

```
class DirectoryEntry {
    public:
        char filename[8];
        char lastModfDate[12];
        int time;
        int firstblocknumber;
        short size;
};
```

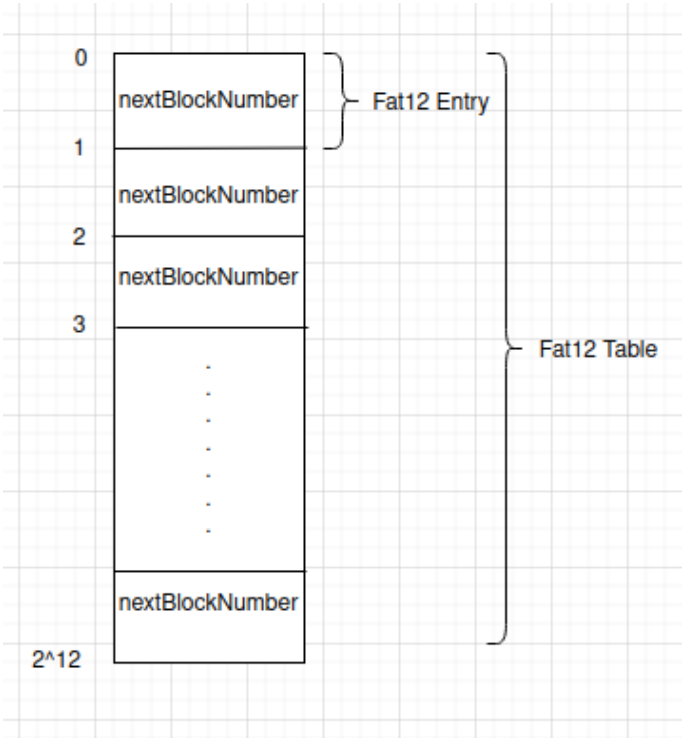
Directory Table



Directory table, on the other hand, is a place where the directory entries are kept collectively, the directory table is designed as an array, and each index corresponds to the relevant data block. As can be seen in the figure, each box under root is a directory entry, and the sum of these directory entries is the directory table. The blocks pointed by the directory entries represent the first block number of that entry.

```
class DirectoryTable {
    public:
        DirectoryEntry *directoryEntry;
```

Fat Table



Fat Table is a structure that shows the status of the blocks in the file system. When a new file or directory is to be added, first the first empty place in the fat table is found, then the necessary information is assigned to the data block with the corresponding index. If it does not fit, the continuation is written to another block, in this case, the block number of the file with the continuation is written to the nextBlockNumber value of the file in the fat table. This is how the hierarchy of the file is provided. If the added file can fit in a single block, the nextBlockNumber value of the corresponding index in the fat table is assigned -1.

```
class Fat12Entry {
    public:
        short nextBlockNumber;
};
```

Free Space Management

```
class Fat12Table {
    public:
        Fat12Entry *fat12Entry;

        Fat12Table(int fatEntryCount) {
            fat12Entry = new Fat12Entry[fatEntryCount]; // Allocate memory for fat12Entry array

            for (int i = 0; i < fatEntryCount; i++) {
                fat12Entry[i].nextBlockNumber = -2; // Initialize each element to -2,free blocks
            }
    }
};
```

I provided the free space management status in the file system with the fat table. Normally, the entries where the nextblock number is 0 in the file fat table means free. However, since my super block is in the 0. data block, I assigned the value -2 so that it would not cause any confusion. . In other words, each entry of the fat table corresponds to a data block, if I assign -2 to the next block number value in the fat table entry, that data block means empty. There are data blocks that are suitable by checking this situation in the fat table.

Super Block

```
class SuperBLock {
    public:
        int blockSize;
        int rootDirPosition;
        int blockCount;
        int freeBlockCount;
        int filesCount;
};
```

Super Block file is the most important part about the system, because it contains all the important information about the system, such as block size, root dir position, free block count etc. Actions are taken based on this information. In addition, in the designed system, the super block is located in the 0th data block.

Data Blocks And Data

```
char **data; //data blockları ve içindeki veriler
```

An entire row of the char array represents a data block, and the information in each row represents data.

Creating File System

```
int fatEntryCount = 1024 * 4;//fat tableda kaç tane entry var
int blockSize;
int maxBlockSize = (1024 * 1024 * 16) / fatEntryCount;  // 16 MB/ 4k toplam block sayısı
int dirEntryCount;
```

Fat Entry Count :Fat table represents the number of entries and the number of blocks.
Block Size:It represents the size of each block. Total data size is found with -> block size * fat entry count.
Max Block Size :With this information, the maximum block size is calculated.

When the user enters the block size value, the data blocks are initialized first. Then the oil table is started. These are calculated according to their sizes in which data block they will be stored. Finally, the blocks where they are stored are marked as full in the fat table. And the final version of all of them is written to the disk. The next time the program works, it will continue knowing the initial information about the file system.

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

As stated in Figure 4.31, if the disk size is maximum 16 MB in fat-12, the block size should be maximum 4kb.

Fat table is initialized and each block is set as empty.

```
Fat12Table fat12Table(fatEntryCount);
int fat12EntrySize = fatEntryCount * sizeof(Fat12Entry);
int dataBlockCountForFat = fat12EntrySize / blockSize;
```

The attributes in the super block are set.

```
SuperBLock superBlock;
superBlock.blockSize = blockSize;
superBlock.rootDirPosition = dataBlockCountForFat + 1;
superBlock.blockCount = fatEntryCount;
superBlock.freeBlockCount = freeBlockCount(fat12Table);
superBlock.filesCount = 0;
```

In the Fat table, the corresponding blocks are shown as full. The areas that super block , root, and fat will occupy.And the total number of entries to be created in the root directory is calculated. And the directory table is also created. Initially, they are shown as empty.

```
DirectoryTable *directoryTable = new DirectoryTable[dirTableSize];

fat12Table.fat12Entry[0].nextBlockNumber = -1; // super block
for (int i = 1; i <= dataBlockCountForFat; i++)
    fat12Table.fat12Entry[i].nextBlockNumber = -1;                    // fat table
fat12Table.fat12Entry[dataBlockCountForFat + 1].nextBlockNumber = -1; // root directory
```

This initialized information is saved to the disk by casting.This information is saved in the file as binary.

```
file.write(reinterpret_cast<char *>(&superBlock), sizeof(SuperBLock));
file.write(reinterpret_cast<char *>(fat12Table.fat12Entry), sizeof(Fat12Entry) * fatEntryCount);
for (int i = 0; i < dirTableSize; i++)
{
    file.write(reinterpret_cast<char *>(directoryTable[i].directoryEntry), sizeof(DirectoryEntry) * rootDirEntryCount);
}
```

Operations on the File System

If you want to test the operating parts of the homework, first of all, the file system byte created in the previous part is read.

```
char tempData[50];
file2.read(tempData, sizeof(SuperBLock));
SuperBLock *ptrSuperBlock = reinterpret_cast<SuperBLock *>(tempData);
superBlock = *ptrSuperBlock;
initializeData(superBlock.blockSize);
memcpy(data[0], tempData, sizeof(SuperBLock));
```

In order to understand the necessary information about the file system in advance, the super block is read first, and the state recorded on the disk with the cast operation is restored, now we know the necessary information about the system and we will do our operations accordingly.The super block read is saved in the 0th data block.

```
int fat12EntrySize = fatEntryCount * sizeof(Fat12Entry);
int dataBlockCountForFat = fat12EntrySize / superBlock.blockSize;

// Fat12Entry'leri dosyadan data blocka okuma
for (int i = 1; i <= dataBlockCountForFat; i++)
{
    file2.read(data[i], superBlock.blockSize);
}
```

After learning the relevant information, we found how many blocks the fat table consists of, and we read the fat table from the disk.

```
// Fat12Entry'leri okuma, data block dan
Fat12Entry *entries = new Fat12Entry[fatEntryCount];
int count = 0;
for (int i = 1; i <= dataBlockCountForFat; i++)
{
    for (int j = 0; j < superBlock.blockSize / sizeof(Fat12Entry); j++)
    {
        memcpy(&entries[count], &(data[i][j * sizeof(Fat12Entry)]), sizeof(Fat12Entry));
        count++;
    }
}
fat12Table.fat12Entry = entries;
```

After reading from the disk to the data block, I re-save the fat table data in the data block into the fat table object.

```
// Root directory entry'leri dosyadan data blocka okuma
int dirTableSize = fatEntryCount - dataBlockCountForFat - 1;
for (int i = dataBlockCountForFat + 1; i < dirTableSize + 1; i++)
{
    file2.read(data[i], superBlock.blockSize);
}
```

From the point where the fat table ends, the root directory is also read.

```
// Root directory entry'leri okuma, data block dan
DirectoryEntry *entries2 = new DirectoryEntry[rootDirEntryCount];
for (int i = 0; i < dirTableSize; i++)
{
    count = 0;
    for (int j = 0; j < superBlock.blockSize / sizeof(DirectoryEntry); j++)
    {
        memcpy(&entries2[count], &(data[dataBlockCountForFat + 1 + i][j * sizeof(DirectoryEntry)]), sizeof(DirectoryEntry)
        count++;
    }
    directoryTable[i].directoryEntry = entries2;
    entries2 = new DirectoryEntry[rootDirEntryCount];
}
```

Then, the number of directory counts is calculated and in the light of this information, the rest of the disk is read into the directories.

mkdirCommand Operation

```
int isInvalidPath = 0;
int uniqueNameControl = 0;
std::vector<std::string> tokens;
tokens = splitPath(path, '\\');
```

First of all, splitting the given path is done with split.

```
for (int i = 0; i < fatEntryCount; i++)
{
    if (fatTable.fat12Entry[i].nextBlockNumber == -2)
    {
        fatTable.fat12Entry[i].nextBlockNumber = -1;
        firstBlockNumber = i;
        std::cout << "firstBlockNumber: " << firstBlockNumber << std::endl;
        break;
    }
}
```

If the token size is 1, it means that it wants to be added to the roo directory, first of all, the fat table free block is found.

```
for (int i = 0; i < directoryTable[0].getEntryCount(); i++)
{
    if (directoryTable[0].directoryEntry[i].filename[0] == '\0')
    {
        int control = isNameUniqueInDir(tokens[0], directoryTable, 0);
        if(control == 0)
        {
            std::cout<<"Name is not unique !"<<std::endl;
            fatTable.fat12Entry[firstBlockNumber].nextBlockNumber = -2;
            uniqueNameControl = 1;
            break;
        }
        else
        {
            strcpy(directoryTable[0].directoryEntry[i].filename, tokens[0].c_str());
            directoryTable[0].directoryEntry[i].firstblocknumber = firstBlockNumber;
            directoryTable[0].directoryEntry[i].size = 0;
            time_t now = time(0);
            std::string dateTimeString = std::ctime(&now);
            std::string timeString = dateTimeString.substr(11, 8); // Saat, dakika ve saniye bilgilerini alır
            std::strcpy(directoryTable[0].directoryEntry[i].lastModfDate, timeString.c_str());
            superBlock.freeBlockCount = superBlock.freeBlockCount - 1;
            superBlock.filesCount = superBlock.filesCount + 1;
            break;
        }

    }
```

Since it wants to be added to the root directory, index 0 of the directory table is navigated, and the first empty directory is entered. Afterwards, it is checked whether there is another folder with the same name in the root directory. After this process, the information of the relevant directory is entered and the process is finished.

```
// root directoryde değilse
int firstBlockNumber = 0;
for (int i = 0; i < tokens.size(); i++)
{
    if (i == 0)
    {
        for (int j = 0; j < directoryTable[0].getEntryCount(); j++)
        {
            if (strcmp(directoryTable[0].directoryEntry[j].filename, tokens[i].c_str()) == 0)
            {
                firstBlockNumber = directoryTable[0].directoryEntry[j].firstblocknumber;
                isInvalidPath = 1;
                break;
            }
        }
    }
```

If we are not going to add anything to the root, all paths are checked one by one in the loop as much as the size of the path.

```
// Eklenmek istenen directory için
else if (i == tokens.size() - 1)
{
    int targetDir = firstBlockNumber - blockWithoutDir;
    int firstBlockNumber = 0;
    for (int k = 0; k < fatEntryCount; k++)
    {
        if (fatTable.fat12Entry[k].nextBlockNumber == -2)
        {
            fatTable.fat12Entry[k].nextBlockNumber = -1;
            firstBlockNumber = k;
            std::cout << "firstBlockNumber: " << firstBlockNumber << std::endl;
            break;
        }
    }
```

```
for (int k = 0; k < directoryTable[targetDir].getEntryCount(); k++)
{
    if (directoryTable[targetDir].directoryEntry[k].filename[0] == '\0')
    {
        int control = isNameUniqueInDir(tokens[0], directoryTable, targetDir);
        if(control == 0)
        {
            std::cout<<"Name is not unique !"<<std::endl;
            fatTable.fat12Entry[firstBlockNumber].nextBlockNumber = -2;
            uniqueNameControl = 1;
            break;
        }
        else
        {
            strcpy(directoryTable[targetDir].directoryEntry[k].filename, tokens[i].c_str());
            directoryTable[targetDir].directoryEntry[k].firstblocknumber = firstBlockNumber;
            directoryTable[targetDir].directoryEntry[k].size = 0;
            time_t now = time(0);
            std::string dateTimeString = std::ctime(&now);
            std::string timeString = dateTimeString.substr(11, 8); // Saat, dakika ve saniye bilgilerini alır
            std::strcpy(directoryTable[targetDir].directoryEntry[k].lastModfDate, timeString.c_str());
            superBlock.freeBlockCount = superBlock.freeBlockCount - 1;
            superBlock.filesCount = superBlock.filesCount + 1;
            updateSizeOfDir(path, fatTable, directoryTable, superBlock, blockWithoutDir, timeString);
            break;
        }

    }
}
isInvalidPath = 1;
```

If "mkdir /okan/burak" is entered, firstly the root directory is found with arrow, then the block pointed by the arrow is gone, it is checked if there is a file named burak in the directories there, then the deletion is performed.In short, until I reach the file I want to add, control operations are performed by navigating the paths before it with the help of the firstblock number, and finally, the relevant file is added.
Finally, the final version of the data block is re-saved to disk.At the same time, the size and modified time times of the directories in the subdirectory of the added directory are updated.

dirCommand Operation

```
int isInvalidPath = 0;
std::vector<std::string> tokens;
tokens = splitPath(path, '\\');
if (tokens.size() == 0)
{
    // root directoryde oluştur
    printDirectoryTable(directoryTable, 0);
}
```

The file path given in the dir command is split with split. Afterwards, if the directory we are looking for is in the root directory, the files in it are suppressed by navigating in the root directory.

If it's not in the root directory

```
// root directoryde değilse
int firstBlockNumber = 0;
for (int i = 0; i < tokens.size(); i++)
{
    if (i == 0)
    {
        for (int j = 0; j < directoryTable[0].getEntryCount(); j++)
        {
            if (strcmp(directoryTable[0].directoryEntry[j].filename, tokens[i].c_str()) == 0)
            {
                firstBlockNumber = directoryTable[0].directoryEntry[j].firstblocknumber;
                if (i == tokens.size() - 1)
                {
                    int targetDir = firstBlockNumber - blockWithoutDir;
                    printDirectoryTable(directoryTable, targetDir);
                }
                isInvalidPath = 1;
                break;
            }
        }
    }
else if (i == tokens.size() - 1)
{
    int targetDir = firstBlockNumber - blockWithoutDir;
    for (int j = 0; j < directoryTable[targetDir].getEntryCount(); j++)
    {
        if (strcmp(directoryTable[targetDir].directoryEntry[j].filename, tokens[i].c_str()) == 0)
        {
            firstBlockNumber = directoryTable[targetDir].directoryEntry[j].firstblocknumber;
            if (i == tokens.size() - 1)
            {
                int targetDir = firstBlockNumber - blockWithoutDir;
                printDirectoryTable(directoryTable, targetDir);
            }
            isInvalidPath = 1;
            break;
        }
    }
}
```

```
else
{
    int targetDir = firstBlockNumber - blockWithoutDir;
    for (int j = 0; j < directoryTable[targetDir].getEntryCount(); j++)
    {
        if (strcmp(directoryTable[targetDir].directoryEntry[j].filename, tokens[i].c_str()) == 0)
        {
            firstBlockNumber = directoryTable[targetDir].directoryEntry[j].firstblocknumber;
            isInvalidPath = 1;
            break;
        }
    }
}
```

If the root is not in the directoy, the same operations are repeated in mkdir, it is navigated between the blocks with the first block numbers until the last path is reached. And when the last path is reached, the files in it are printed from the directory table.

rmdirCommand Operation

```
if (i == 0)
{
    for (int j = 0; j < directoryTable[0].getEntryCount(); j++)
    {
        if (strcmp(directoryTable[0].directoryEntry[j].filename, tokens[i].c_str()) == 0)
        {
            firstBlockNumber = directoryTable[0].directoryEntry[j].firstblocknumber;
            if (i == tokens.size() - 1)
            {
                std::cout<<"filename: "<<directoryTable[0].directoryEntry[j].filename<<" is deleted"<<std::endl;
                directoryTable[0].directoryEntry[j].filename[0] = '\0';
                directoryTable[0].directoryEntry[j].size = 0;
                directoryTable[0].directoryEntry[j].lastModfDate[0] = '\0';
                targetBlockNumber = directoryTable[0].directoryEntry[j].firstblocknumber;
                fatTable.fat12Entry[firstBlockNumber].nextBlockNumber = -2;
                superBlock.freeBlockCount = superBlock.freeBlockCount + 1;
                superBlock.filesCount = superBlock.filesCount - 1;
            }
            break;
        }
    }
}
```

```
else if (i == tokens.size() - 1)
{
    int targetDir = firstBlockNumber - blockWithoutDir;
    int firstBlockNumber = 0;
    for (int k = 0; k < directoryTable[targetDir].getEntryCount(); k++)
    {
        if (std::strcmp(directoryTable[targetDir].directoryEntry[k].filename, tokens[i].c_str()) == 0)
        {
            std::cout<<"filename: "<<directoryTable[targetDir].directoryEntry[k].filename<<" is deleted"<<std::endl;
            directoryTable[targetDir].directoryEntry[k].filename[0] = '\0';
            directoryTable[targetDir].directoryEntry[k].size = 0;
            directoryTable[targetDir].directoryEntry[k].lastModfDate[0] = '\0';
            targetBlockNumber = directoryTable[targetDir].directoryEntry[k].firstblocknumber;
            firstBlockNumber = directoryTable[targetDir].directoryEntry[k].firstblocknumber;
            fatTable.fat12Entry[firstBlockNumber].nextBlockNumber = -2;
            superBlock.freeBlockCount = superBlock.freeBlockCount + 1;
            superBlock.filesCount = superBlock.filesCount - 1;
            break;
        }
    }
}
```

```
else
{
    int targetDir = firstBlockNumber - blockWithoutDir;
    for (int j = 0; j < directoryTable[targetDir].getEntryCount(); j++)
    {
        if (strcmp(directoryTable[targetDir].directoryEntry[j].filename, tokens[i].c_str()) == 0)
        {
            firstBlockNumber = directoryTable[targetDir].directoryEntry[j].firstblocknumber;
            break;
        }
    }
```

In the rmdir process, the following method was used, first the file to be deleted is accessed, the same process as in mkdir, this file needs to be deleted after this file is reached, but there may be other files in its subdirectory. All of them were deleted with a recursive structure.

```
void rmdirHelper(int targetBlockNumber,Fat12Table &fatTable, DirectoryTable *directoryTable, SuperBLock &superBlock , int b
{
    int endOfDir = 0;
    while (endOfDir != 1)
    {
        int targetDir = targetBlockNumber - blockWithoutDir;
        for (int k = 0; k < directoryTable[targetDir].getEntryCount(); k++)
        {
            if (directoryTable[targetDir].directoryEntry[k].filename[0] != '\0')
            {
                std::cout<<"filename: "<<directoryTable[targetDir].directoryEntry[k].filename<<" is deleted"<<std::endl;
                directoryTable[targetDir].directoryEntry[k].filename[0] = '\0';
                directoryTable[targetDir].directoryEntry[k].size = 0;
                directoryTable[targetDir].directoryEntry[k].lastModfDate[0] = '\0';
                targetBlockNumber = directoryTable[targetDir].directoryEntry[k].firstblocknumber;
                int firstBlockNumber = directoryTable[targetDir].directoryEntry[k].firstblocknumber;
                fatTable.fat12Entry[firstBlockNumber].nextBlockNumber = -2;
                superBlock.freeBlockCount = superBlock.freeBlockCount + 1;
                superBlock.filesCount = superBlock.filesCount - 1;
                rmdirHelper(targetBlockNumber, fatTable, directoryTable, superBlock, blockWithoutDir);
                //break;
            }
        }
        endOfDir = 1;
    }
}
```

The directory information of each deleted file is assigned null and the fat table is set as that block free.

Dumpe2fs Operation

```
int blockWithoutDir = dataBlockCountForFat + 1;
std::cout<<"Block Size: "<<superBlock.blockSize<<std::endl;
std::cout<<"Block Count: "<<superBlock.blockCount<<std::endl;
std::cout<<"Free Block Count: "<<superBlock.freeBlockCount<<std::endl;
std::cout<<"Files Count: "<<superBlock.filesCount<<std::endl;
occupiedBlocksWithName(fat12Table, directoryTable,blockWithoutDir);
```

Since the information about the file system had to be suppressed here, no separate function was required. All of this information is stored in the super block, and the necessary information is updated in each mkdir and rmdir operations.

On the other hand, occupied blocks are shown in the fattable by navigating one by one and printing the necessary information from the directries in the relevant block.

TEST CASES

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 makeFileSystem 4 mySystem.dat
```

The file system has been created so that the block sizes are 4kb

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\okan"
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\burak"
```

With the mkdir command, a file named burak was added to the root directory first.

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dir "\\"
FileName Size FirstBlockNumber Last Modified Time
okan     0          4          16:06:12
FileName Size FirstBlockNumber Last Modified Time
burak     0          5          16:06:28
```

With the dir command, the files in the root are printed.

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\okan\deniz"
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\okan\deniz\atakan"
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dir "\okan"
FileName Size FirstBlockNumber Last Modified Time
deniz     32          6          16:12:20
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dir "\okan\deniz"
FileName Size FirstBlockNumber Last Modified Time
atakan     0          7          16:12:20
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dir "\\"
FileName Size FirstBlockNumber Last Modified Time
okan     64          4          16:12:20
FileName Size FirstBlockNumber Last Modified Time
burak     0          5          16:12:09
```

The last file added is Atakan, and the modified times in its sub-directories have also been updated with the date of Atakan's addition. At the same time, the size of the files in the subdirectory is updated when a new file is added.

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dumpe2fs
Block Size: 4096
Block Count: 4096
Free Block Count: 4092
Files Count: 4
filename: SuperBlock , block number: 0
filename: FatTable , block number: 1
filename: FatTable , block number: 2
filename: Root Block , block number: 3
filename: okan block number: 4
filename: burak block number: 5
filename: deniz block number: 6
filename: atakan block number: 7
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat rmdir "\okan\deniz"
filename: deniz is deleted
filename: atakan is deleted
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dir "\okan\deniz"
Invalid Path !
```

When dir \okan\deniz is made, the deniz and the directories under the deniz are deleted.

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dumpe2fs
Block Size: 4096
Block Count: 4096
Free Block Count: 4094
Files Count: 2
filename: SuperBlock , block number: 0
filename: FatTable , block number: 1
filename: FatTable , block number: 2
filename: Root Block , block number: 3
filename: okan block number: 4
filename: burak block number: 5
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\okan\emir"
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat dumpe2fs
Block Size: 4096
Block Count: 4096
Free Block Count: 4093
Files Count: 3
filename: SuperBlock , block number: 0
filename: FatTable , block number: 1
filename: FatTable , block number: 2
filename: Root Block , block number: 3
filename: okan block number: 4
filename: burak block number: 5
filename: emir block number: 6
```

```
okan@okan-ABRA-A5-V16-4:~/Desktop/os-hw3$ ./hw3 fileSystemOpen mySystem.dat mkdir "\okan"
Name is not unique !
```