Gebze Technical University

Department Of Computer Engineering

CSE 312 Spring 2023

Operating System

Homework 3

Due Date: 18.06.2023

Abdurrahman Bulut
1901042258

Introduction

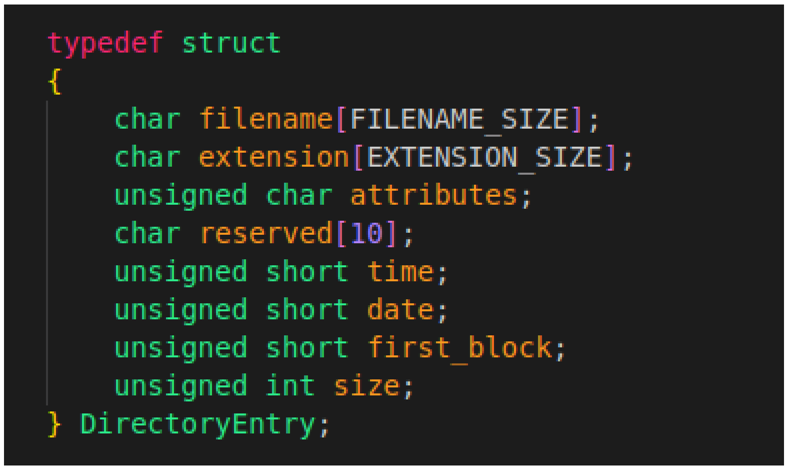
In this project, I built a file system using the FAT12 structure in C. The file system can store files with attributes like size, last modification date, and name. I followed the FAT12 structure in our textbook, and our file system can handle some possible FAT12 blocks.

I created a file system that can be accessed and manipulated through a set of operations. These operations include creating directories, listing directory contents, removing directories, and managing files within the file system. I also developed a command-line program that interacts with the file system, allowing users to perform various operations on the file system.

I made part-1, part-2 and some operations of part-3. In part-3, I couldn't implement read and del commands. The other commands are in the test part. I sent a design report that describes the system design, data structures, and key functionality implemented. The source code for the file system implementation. A Makefile to build the project and a readMe file that provides instructions for compiling the program.

Part 1 explanation

The file system had to store files with attributes like size, last modification date, and name. The design includes a directory table to organize and manage files and directories. The directory table is like a directory structure in our book example:



```c
typedef struct
{
    char filename[FILENAME_SIZE];
    char extension[EXTENSION_SIZE];
    unsigned char attributes;
    char reserved[10];
    unsigned short time;
    unsigned short date;
    unsigned short first_block;
    unsigned int size;
} DirectoryEntry;
```

Each directory entry stores information about a file or directory, like the name, extension, size, last modification date, and first block number. The file system also needs to keep track of free blocks to allocate them to new files and directories. The design includes a mechanism to manage and maintain the list of free blocks efficiently.

I used this entry structure as expected.

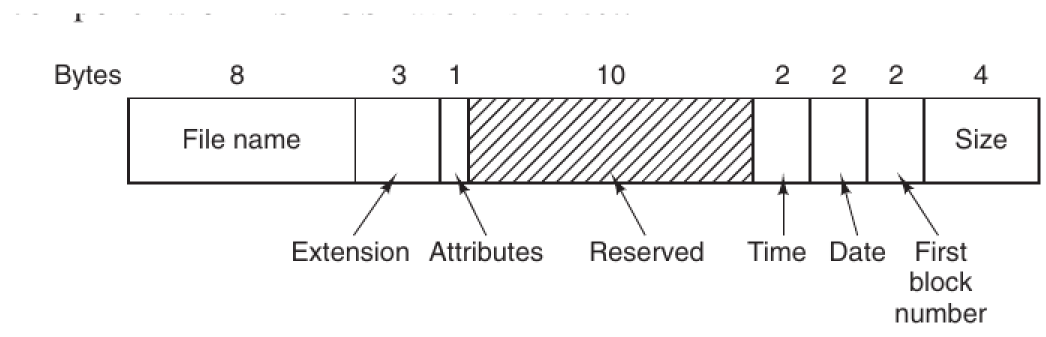
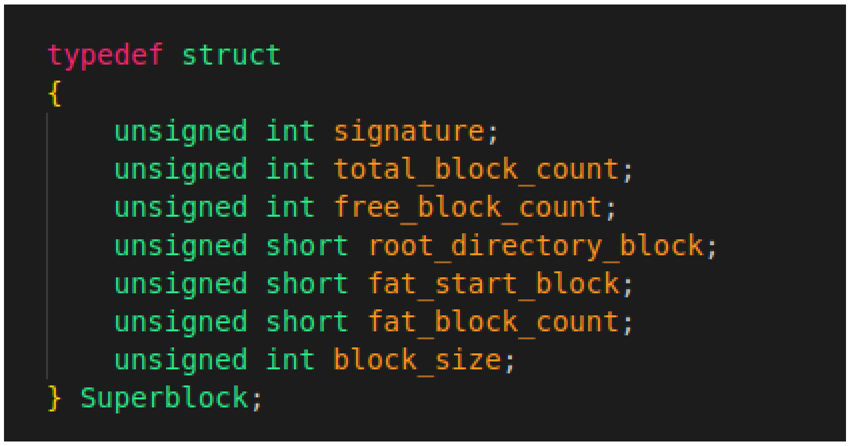


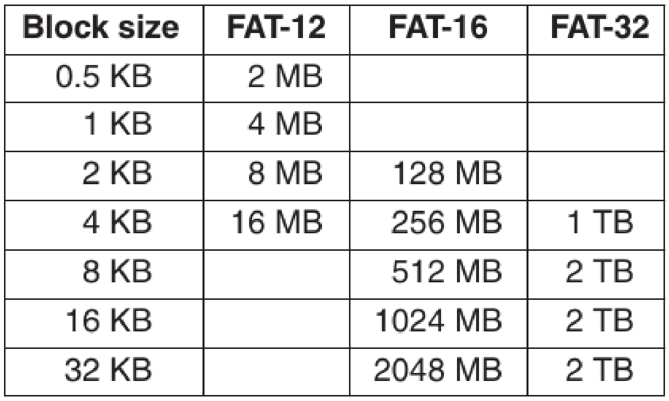
Figure 4-30. The MS-DOS directory entry.

The superblock is a critical component that contains important information about the file system. It serves as a reference point for the file system's configuration and organization. The design specifies the block size for the file system and stores the position of the root directory in the file system. It also contains information about the signature, the starting positions of the various blocks in the file system, like the FAT, root directory, and data blocks.



```
typedef struct
{
    unsigned int signature;
    unsigned int total_block_count;
    unsigned int free_block_count;
    unsigned short root_directory_block;
    unsigned short fat_start_block;
    unsigned short fat_block_count;
    unsigned int block_size;
} Superblock;
```

I adjust the maximum partition size by looking at the block size that entered via command line argument.



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.

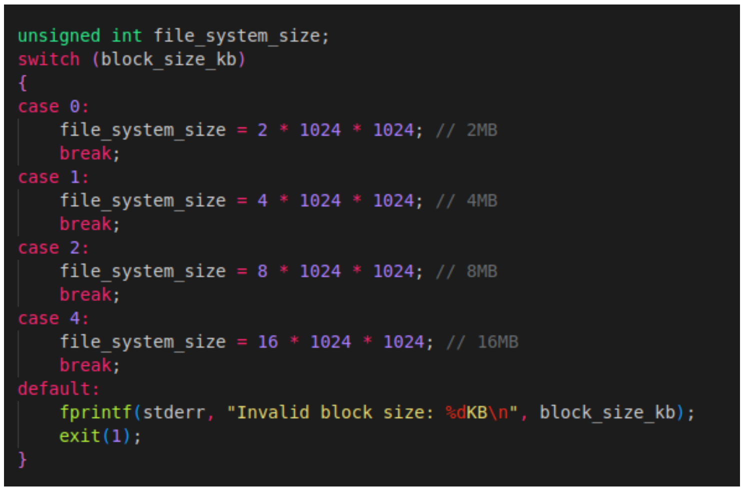
Part 2 explanation

The second part of the project was to create the file system. I implemented the makeFileSystem function, which creates the file system based on the specified block size and disk name. It creates an empty file system as a maximum 16MB file.

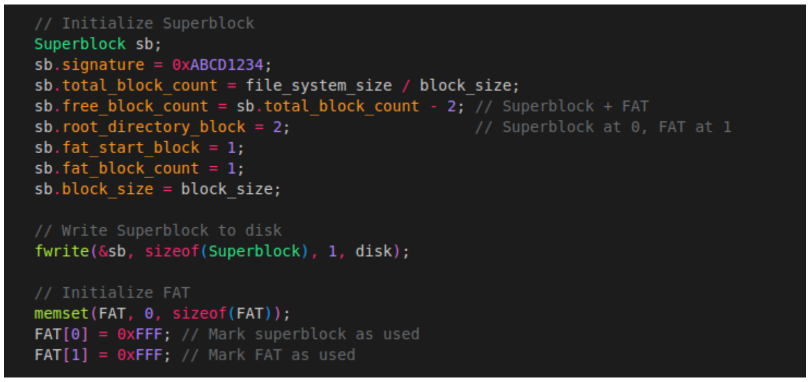
The sample run of the program is like:

```
makeFileSystem 4 mySystem.dat
```

If block size is entered bigger than 4KB, it is exiting from the program because it is forbidden in FAT12.


```c
unsigned int file_system_size;
switch (block_size_kb)
{
case 0:
    file_system_size = 2 * 1024 * 1024; // 2MB
    break;
case 1:
    file_system_size = 4 * 1024 * 1024; // 4MB
    break;
case 2:
    file_system_size = 8 * 1024 * 1024; // 8MB
    break;
case 4:
    file_system_size = 16 * 1024 * 1024; // 16MB
    break;
default:
    fprintf(stderr, "Invalid block size: %dKB\n", block_size_kb);
    exit(1);
}
```

Firstly, it calculates the actual block size in bytes and the total size of the file system in bytes. Then, it opens the disk file in "wb+" mode. Then, the function initializes the superblock with the relevant information about the file system. After that, it initializes the FAT array with all entries set to zero. The first two entries in the FAT are marked as used to indicate the superblock and the FAT itself.

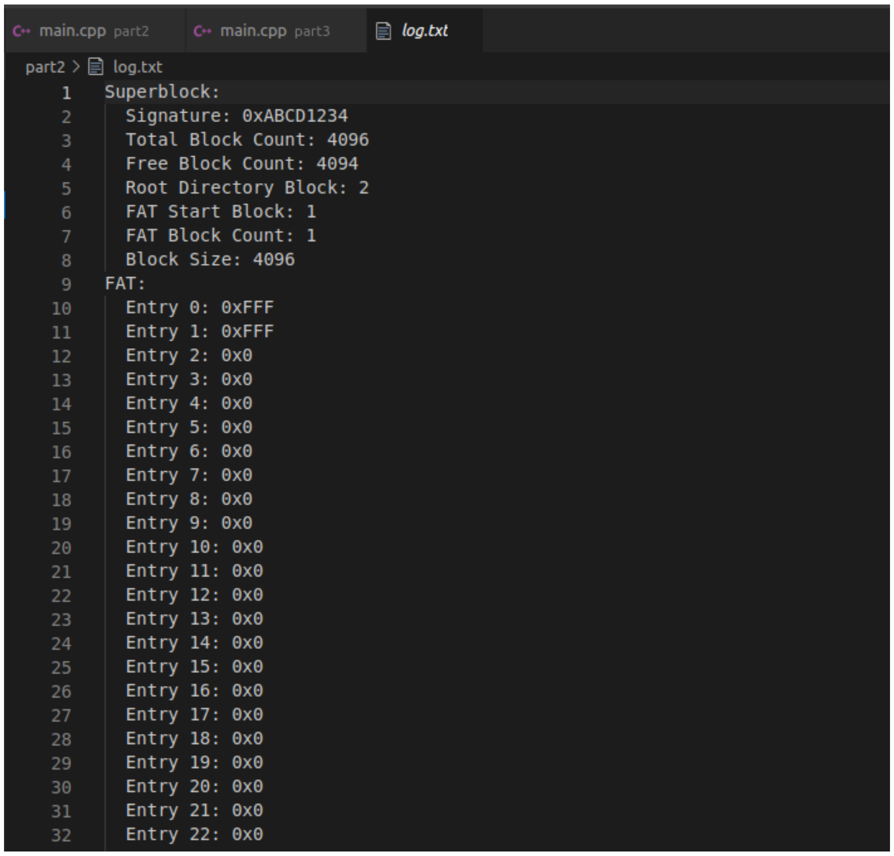

```c
// Initialize Superblock
Superblock sb;
sb.signature = 0xABCD1234;
sb.total_block_count = file_system_size / block_size;
sb.free_block_count = sb.total_block_count - 2; // Superblock + FAT
sb.root_directory_block = 2;                    // Superblock at 0, FAT at 1
sb.fat_start_block = 1;
sb.fat_block_count = 1;
sb.block_size = block_size;

// Write Superblock to disk
fwrite(&sb, sizeof(Superblock), 1, disk);

// Initialize FAT
memset(FAT, 0, sizeof(FAT));
FAT[0] = 0xFFF; // Mark superblock as used
FAT[1] = 0xFFF; // Mark FAT as used
```

The function then logs the information from the superblock and FAT to the log file. Because, .data file is a binary file and we cannot see the content of it. That’s why I printed it into a log file to see what it has. It also ensures that the file size of the disk is adjusted to match the maximum partition size based on the block size. Finally, the function closes the disk and log files.

The main function serves as the entry point for the program. It checks the command-line arguments, calls the makeFileSystem function with the provided block size and disk name, and displays a success message. By executing the program with the appropriate command-line arguments, the file system is created successfully, and detailed information about the superblock and FAT is logged in the "log.txt" file.

In the subsequent parts of the project, additional functionality and operations will be implemented to interact with the created file system.

This is my log file:



```
Superblock:
  Signature: 0xABCD1234
  Total Block Count: 4096
  Free Block Count: 4094
  Root Directory Block: 2
  FAT Start Block: 1
  FAT Block Count: 1
  Block Size: 4096
FAT:
  Entry 0: 0xFFF
  Entry 1: 0xFFF
  Entry 2: 0x0
  Entry 3: 0x0
  Entry 4: 0x0
  Entry 5: 0x0
  Entry 6: 0x0
  Entry 7: 0x0
  Entry 8: 0x0
  Entry 9: 0x0
  Entry 10: 0x0
  Entry 11: 0x0
  Entry 12: 0x0
  Entry 13: 0x0
  Entry 14: 0x0
  Entry 15: 0x0
  Entry 16: 0x0
  Entry 17: 0x0
  Entry 18: 0x0
  Entry 19: 0x0
  Entry 20: 0x0
  Entry 21: 0x0
  Entry 22: 0x0
```

Part 3 explanation

This part is about file operations. It is asked to use fileSystem.data file system data and modify it using some operations. I created a function for each operation.

Operation “dir”

I created a “list_directory” method for this operation.

```
void list_directory(const char *disk_name, char *path)
```

It first opens the File System. The list_directory function takes the disk_name parameter, which specifies the file system data file (e.g., "fileSystem.data"). The function opens the file system using the fopen function with the "rb" (read binary) mode because I wrote it in binary format in part 2. Then it reads the superblock. The function reads the superblock from the file system using the fread function. The Superblock struct, sb, is used to store the superblock information. The fread function reads the size of Superblock (sizeof(Superblock)) from the disk file and stores it in sb. Then the function compares the provided path parameter with the root directory path (""). If the path is not equal to "", it means the function only supports listing the root directory.

The function then seeks to the position of the root directory within the file system using fseek and the root directory block number from the superblock. It starts a loop to read and display each directory entry within the root directory. It reads each entry using fread and stores the information in the DirectoryEntry struct, entry. The loop continues until it encounters an entry with a filename of 0, indicating the end of the directory entries. For each entry, the function displays the filename, extension, size, date, time, and attributes. After listing the directory contents, the file system data file is closed as expected.

I am using “\\” instead of “\” because it is an escape character.

```
void list_directory(const char *disk_name, char *path)
{
// Open the file representing the file system
disk = fopen(disk_name, "rb");
if (!disk)
{
perror("Error opening filesystem");
exit(1);
}

printf("Opened filesystem %s successfully.\n", disk_name);
```

```c
// Read Superblock
Superblock sb;
fread(&sb, sizeof(Superblock), 1, disk);

printf("Read Superblock.\n");

// It only support listing the root directory
if (strcmp(path, "\\") != 0)
{
fprintf(stderr, "Only the root directory is supported.\n");
fclose(disk);
exit(1);
}

// Locate the root directory
fseek(disk, sb.root_directory_block * sb.block_size, SEEK_SET);

// List contents of the root directory
DirectoryEntry entry;
printf("Filename Extension Size Date Time Attributes\n");
printf("-----------------------------------------------------------------------------
----------\n");

while (fread(&entry, sizeof(DirectoryEntry), 1, disk))
{
if (entry.filename[0] == 0)
{
break; // End of directory entries
}
printf("%-8.8s %-7.3s %-2d %02d-%02d-%02d %02d:%02d:%02d %02X\n",
entry.filename, entry.extension, entry.size,
((entry.date >> 5) & 0x0F), (entry.date & 0x1F), ((entry.date >> 9) +
1980),
(entry.time >> 11), ((entry.time >> 5) & 0x3F), ((entry.time & 0x1F) * 2),
entry.attributes);
}

fclose(disk); }
```

Operation “mkdir”

The mkdir command creates directories within the file system recursively. It involves two functions: “make_directory_recursive” and “make_directory”.

```
void make_directory_recursive(FILE *disk, Superblock sb, unsigned short
current_block, char *remaining_path)
```

```
void make_directory(const char *disk_name, char *path)
```

make_directory_recursive

This function recursively creates a directory within the file system. It takes the following parameters:

- disk: A file pointer representing the file system.
- sb: The superblock containing information about the file system.
- current_block: The block number of the current directory.
- remaining_path: The remaining part of the path for creating the directory.

The function reads the current directory block from the disk using fseek and fread. It then iterates through the directory entries to check for empty entries or duplicate directory names.

If an empty entry is found, a new directory entry is created and written to the disk. The entity's attributes, time, and date are set accordingly. If a duplicate directory name with the same attributes is found, a message is displayed, indicating that the directory already exists.

If no empty entry is found, a new block is allocated for the directory, and a new directory entry is created with the appropriate attributes and data. The function then recursively calls itself to create nested subdirectories, iterating through the remaining parts of the path. This recursive process continues until all subdirectories are created.

```
// Function to create a directory
void make_directory_recursive(FILE *disk, Superblock sb, unsigned short
current_block, char *remaining_path)
{
// Read the current directory block
DirectoryEntry entry;
fseek(disk, current_block * sb.block_size, SEEK_SET);

while (fread(&entry, sizeof(DirectoryEntry), 1, disk))
{
if (entry.filename[0] == 0)
```

```
{
// Empty entry found, create new directory entry
memset(&entry, 0, sizeof(DirectoryEntry));
strncpy(entry.filename, remaining_path, FILENAME_SIZE);
entry.attributes = 0x10; // Directory attribute

// Set time and date
time_t current_time = time(NULL);
struct tm *tm_info = localtime(&current_time);
entry.time = (tm_info->tm_hour << 11) | (tm_info->tm_min << 5) |
(tm_info->tm_sec / 2);
entry.date = ((tm_info->tm_year - 80) << 9) | ((tm_info->tm_mon + 1) << 5)
| tm_info->tm_mday;

// Write the new directory entry
fseek(disk, -sizeof(DirectoryEntry), SEEK_CUR);
fwrite(&entry, sizeof(DirectoryEntry), 1, disk);

return; // Directory created
}

if (strcmp(entry.filename, remaining_path) == 0 && entry.attributes ==
0x10)
{
printf("Directory already exists, no need to create it..\n");
return;
}
}

// No empty entry found, need to allocate a new block for the directory
unsigned short next_block = 0;
for (unsigned short i = 2; i < FAT_ENTRIES; i++)
{
if (FAT[i] == 0)
{
next_block = i;
FAT[current_block] = next_block;
FAT[next_block] = 0xFFF;
break;
}
```

```c
}

if (next_block == 0)
{
fprintf(stderr, "No space left in directory.\n");
exit(1);
}

// Create new directory entry
memset(&entry, 0, sizeof(DirectoryEntry));
strncpy(entry.filename, remaining_path, FILENAME_SIZE);
entry.attributes = 0x10; // Directory attribute
entry.first_block = next_block;

// Set time and date
time_t current_time = time(NULL);
struct tm *tm_info = localtime(&current_time);
entry.time = (tm_info->tm_hour << 11) | (tm_info->tm_min << 5) |
(tm_info->tm_sec / 2);
entry.date = ((tm_info->tm_year - 80) << 9) | ((tm_info->tm_mon + 1) << 5)
| tm_info->tm_mday;

// Write the new directory entry
fseek(disk, current_block * sb.block_size, SEEK_SET);
fwrite(&entry, sizeof(DirectoryEntry), 1, disk);

// Create nested directory
make_directory_recursive(disk, sb, next_block, "");

// Create the remaining subdirectories recursively
char subdirectory[FILENAME_SIZE + EXTENSION_SIZE + 2];
char *next_slash = strchr(remaining_path, '\\');

while (next_slash)
{
strncpy(subdirectory, remaining_path, next_slash - remaining_path);
subdirectory[next_slash - remaining_path] = '\0';

make_directory_recursive(disk, sb, next_block, subdirectory);
```

```
remaining_path = next_slash + 1;
next_slash = strchr(remaining_path, '\\');
}

// Create the last subdirectory
make_directory_recursive(disk, sb, next_block, remaining_path);
}
```

make_directory

This function is the entry point for the mkdir command. It takes the following parameters:

- disk_name: The name of the file system data file.
- path: The path of the directory to be created.

The function opens the file system data file in "r+b" (read and write binary) mode using fopen. It reads the superblock from the file system using fread. The function then reads the FAT from the file system by seeking to the appropriate position and using fread. Finally, the make_directory_recursive function is called with the root directory block, the file system's superblock, and the provided path.

```
// Function to create a directory
void make_directory(const char *disk_name, char *path)
{
disk = fopen(disk_name, "r+b");
if (!disk)
{
perror("Error opening filesystem");
exit(1);
}

// Read Superblock
Superblock sb;
fread(&sb, sizeof(Superblock), 1, disk);

// Read FAT
fseek(disk, sb.fat_start_block * sb.block_size, SEEK_SET);
fread(&FAT, sizeof(FAT), 1, disk);
// Create the directory recursively
make_directory_recursive(disk, sb, sb.root_directory_block, path);

fclose(disk);
}
```

Operation “rmdir”

The rmdir allows the user to remove directories from the file system. It uses two functions: delete_directory_recursive and delete_directory.

delete_directory_recursive

This function recursively deletes a directory and its subdirectories from the file system. It takes the following parameters:

- disk: A file pointer representing the file system.
- sb: The superblock containing information about the file system.
- current_block: The block number of the current directory being examined.
- remaining_path: The remaining part of the path to the directory to be deleted.

The function reads the current directory block from the disk using fseek and fread. It then iterates through the directory entries to check for a match with the remaining path and a directory attribute. If a directory entry matches the remaining path and is a directory, the entry is zeroed out by using memset.

The modified directory entry is written back to the disk using fwrite. The function then marks the blocks used by the directory as free in the FAT. It starts with the first block of the directory and follows the linked list of blocks until it reaches the end (indicated by 0xFFF).

```
void delete_directory_recursive(FILE *disk, Superblock sb, unsigned short
current_block, char *remaining_path)
{
// Read the current directory block
DirectoryEntry entry;
fseek(disk, current_block * sb.block_size, SEEK_SET);

while (fread(&entry, sizeof(DirectoryEntry), 1, disk))
{
// If directory entry matches the remaining path
if (strcmp(entry.filename, remaining_path) == 0 && entry.attributes ==
0x10)
{
// Zero out the directory entry
memset(&entry, 0, sizeof(DirectoryEntry));

// Write back the modified directory entry
fseek(disk, -sizeof(DirectoryEntry), SEEK_CUR);
fwrite(&entry, sizeof(DirectoryEntry), 1, disk);

// Mark the blocks used by the directory as free in the FAT
unsigned short block = entry.first_block;
while (block != 0xFFF)
{
unsigned short next_block = FAT[block];
FAT[block] = 0;
block = next_block;
}

return;
}
}

// If the desired directory wasn't found, look in subdirectories
char subdirectory[FILENAME_SIZE + EXTENSION_SIZE + 2];
char *next_slash = strchr(remaining_path, '\\');

if (next_slash)
{
strncpy(subdirectory, remaining_path, next_slash - remaining_path);
subdirectory[next_slash - remaining_path] = '\0';

remaining_path = next_slash + 1;
delete_directory_recursive(disk, sb, current_block, subdirectory);
delete_directory_recursive(disk, sb, entry.first_block, remaining_path);
}
}
```

delete_directory

This function is the entry point for the rmdir command. It takes the following parameters:

- disk_name: The name of the file system data file.
- path: The path of the directory to be deleted.

The function opens the file system data file in "r+b" (read and write binary) mode using fopen. It reads the superblock from the file system using fread. The function then reads the FAT from the file system by seeking to the appropriate position and using fread. Finally, the delete_directory_recursive function is called with the root directory block, the file system's superblock, and the provided path.

```
void delete_directory(const char *disk_name, char *path)
{
FILE *disk = fopen(disk_name, "r+b");
if (!disk)
{
perror("Error opening filesystem");
exit(1);
}

// Read Superblock
Superblock sb;
fread(&sb, sizeof(Superblock), 1, disk);

// Read FAT
fseek(disk, sb.fat_start_block * sb.block_size, SEEK_SET);
fread(&FAT, sizeof(FAT), 1, disk);

// Delete the directory recursively
delete_directory_recursive(disk, sb, sb.root_directory_block, path);

fclose(disk);
}
```

Operation “dumpe2fs”

The dumpe2fs command allows us to inspect the file system by printing various information about its blocks and occupied blocks. It uses two functions: dump_filesystem and traverse_directory.

dump_filesystem

This function is the entry point for the dumpe2fs command. It takes the disk_name parameter, which specifies the file system data file. The function opens the file system data file in "rb" (read binary) mode using fopen. It then reads the superblock from the file system using fread and stores it in the sb variable. The function prints various information about the file system, such as the total block count, free block count, number of files and directories, and block size.

It creates an array occupied_blocks to keep track of occupied blocks within the file system.
It calls the traverse_directory function to traverse the root directory and subdirectories, marking the occupied blocks in the occupied_blocks array. After traversing the file system, it prints the information of occupied blocks and their corresponding file names.

```
void dump_filesystem(const char *disk_name)
{
disk = fopen(disk_name, "rb");
if (!disk)
{
perror("Error opening filesystem");
exit(1);
}

// Read Superblock
Superblock sb;
fread(&sb, sizeof(Superblock), 1, disk);

// Print file system information
printf("Block count: %u\n", sb.total_block_count);
printf("Free blocks: %u\n", sb.free_block_count);
printf("Number of files and directories: %u\n", sb.num_files);
printf("Block size: %u bytes\n", sb.block_size);

// Traverse the file system to identify occupied blocks
int occupied_blocks[sb.total_block_count];
memset(occupied_blocks, 0, sizeof(occupied_blocks));
```

```c
// Traverse the root directory and subdirectories
traverse_directory(sb, sb.root_directory_block, occupied_blocks);

// Print occupied block information with corresponding file names
printf("\nOccupied Blocks:\n");
for (unsigned int i = 2; i < sb.total_block_count; i++)
{
if (occupied_blocks[i])
{
// Print file name
char filename[FILENAME_SIZE + EXTENSION_SIZE + 1];
fseek(disk, i * sb.block_size, SEEK_SET);
fread(filename, sizeof(filename), 1, disk);
printf("Block %u: %s\n", i, filename);
}
}

fclose(disk);
}
```

traverse_directory

This function recursively traverses the directory entries within the file system to identify occupied blocks. It takes the following parameters:

- sb: The superblock containing information about the file system.
- current_block: The block number of the current directory being examined.
- occupied_blocks: An array that keeps track of occupied blocks within the file system.

The function seeks to the position of the current directory block within the file system using fseek. It reads each directory entry using fread and checks if it is the end of the directory entries (indicated by a filename of 0). For each directory entry, it checks if it represents a directory or a file based on the attributes. If it is a directory and not "." or "..", the function recursively calls itself to traverse the subdirectory. If it is a file, the corresponding block is marked as occupied in the occupied_blocks array. This process continues until all directory entries are processed.

```c
void traverse_directory(Superblock sb, unsigned short current_block, int
*occupied_blocks)
{
fseek(disk, current_block * sb.block_size, SEEK_SET);

DirectoryEntry entry;
while (fread(&entry, sizeof(DirectoryEntry), 1, disk))
{
if (entry.filename[0] == 0)
{
break; // End of directory entries
}

// Check if the entry is a file or a directory
if ((entry.attributes & 0x10) == 0x10)
{
// Directory found, recursively traverse it
if (strcmp(entry.filename, ".") != 0 && strcmp(entry.filename, "..") != 0)
{
unsigned short block = entry.first_block;
traverse_directory(sb, block, occupied_blocks);
}
}
else
{
// File found, mark its block as occupied
occupied_blocks[entry.first_block] = 1;
}
}
}
```

Operation "write"

This part is not working properly.

It creates a file within the file system using the make_file function and its helper function make_file_recursive. Here is how the file creation process works:

The make_file_recursive function reads the current directory block from the disk and loops through each directory entry within the block to find an empty entry or a matching file entry.
If an empty entry is found, a new file entry is created. The function initializes the entry structure with the necessary details such as the filename, attributes, time, and date. If a matching file entry is found, indicating that the file already exists, a message is displayed, and the function returns. If no empty entry is found, the function needs to allocate a new block for the file. It searches for an available block in the FAT and updates the FAT entries accordingly. The contents of the source file are then copied into the new file.

The function opens the source file in "rb" (read binary) mode and determines its size using ftell. It then reads the file contents in chunks and writes them to the appropriate blocks in the file system until the entire file is copied. If the remaining path contains subdirectories, the function recursively creates them by calling make_directory_recursive for each subdirectory.
The function ends when the entire file path has been traversed, and the file creation is complete.

```
void make_file_recursive(FILE *disk, Superblock sb, unsigned short
current_block, char *remaining_path, const char *source_file)
{
// Read the current directory block
DirectoryEntry entry;
fseek(disk, current_block * sb.block_size, SEEK_SET);

while (fread(&entry, sizeof(DirectoryEntry), 1, disk))
{
if (entry.filename[0] == 0)
{
// Empty entry found, create new file entry
memset(&entry, 0, sizeof(DirectoryEntry));
strncpy(entry.filename, remaining_path, FILENAME_SIZE);
entry.attributes = 0x00; // Regular file attribute

// Set time and date
time_t current_time = time(NULL);
```

```
struct tm *tm_info = localtime(&current_time);
entry.time = (tm_info->tm_hour << 11) | (tm_info->tm_min << 5) |
(tm_info->tm_sec / 2);
entry.date = ((tm_info->tm_year - 80) << 9) | ((tm_info->tm_mon + 1) << 5)
| tm_info->tm_mday;

// Write the new file entry
fseek(disk, -sizeof(DirectoryEntry), SEEK_CUR);
fwrite(&entry, sizeof(DirectoryEntry), 1, disk);

// Copy contents of source file into the new file
FILE *src_file = fopen(source_file, "rb");
if (!src_file)
{
fprintf(stderr, "Error opening source file: %s\n", source_file);
exit(1);
}

fseek(src_file, 0, SEEK_END);
unsigned int file_size = ftell(src_file);
fseek(src_file, 0, SEEK_SET);

unsigned short current_block = entry.first_block;
unsigned int remaining_bytes = file_size;

while (current_block != 0xFFF && remaining_bytes > 0)
{
unsigned int bytes_to_write = (remaining_bytes > sb.block_size) ?
sb.block_size : remaining_bytes;

fseek(disk, current_block * sb.block_size, SEEK_SET);
fread(disk, bytes_to_write, 1, src_file);

remaining_bytes -= bytes_to_write;
current_block = FAT[current_block];
}

fclose(src_file);

return; // File created
```

```c
}

if (strcmp(entry.filename, remaining_path) == 0 && entry.attributes ==
0x00)
{
printf("File already exists, no need to create it..\n");
return;
}
}

// No empty entry found, need to allocate a new block for the file
unsigned short next_block = 0;
for (unsigned short i = 2; i < FAT_ENTRIES; i++)
{
if (FAT[i] == 0)
{
next_block = i;
FAT[current_block] = next_block;
FAT[next_block] = 0xFFF;
break;
}
}

if (next_block == 0)
{
fprintf(stderr, "No space left in directory.\n");
exit(1);
}

// Create new file entry
memset(&entry, 0, sizeof(DirectoryEntry));
strncpy(entry.filename, remaining_path, FILENAME_SIZE);
entry.attributes = 0x00; // Regular file attribute
entry.first_block = next_block;
entry.size = 0; // Set initial size to 0

// Set time and date
time_t current_time = time(NULL);
struct tm *tm_info = localtime(&current_time);
```

```
entry.time = (tm_info->tm_hour << 11) | (tm_info->tm_min << 5) |
(tm_info->tm_sec / 2);
entry.date = ((tm_info->tm_year - 80) << 9) | ((tm_info->tm_mon + 1) << 5)
| tm_info->tm_mday;

// Write the new file entry
fseek(disk, current_block * sb.block_size, SEEK_SET);
fwrite(&entry, sizeof(DirectoryEntry), 1, disk);

// Copy contents of source file into the new file
FILE *src_file = fopen(source_file, "rb");
if (!src_file)
{
fprintf(stderr, "Error opening source file: %s\n", source_file);
exit(1);
}

fseek(src_file, 0, SEEK_END);
unsigned int file_size = ftell(src_file);
fseek(src_file, 0, SEEK_SET);

current_block = entry.first_block;
unsigned int remaining_bytes = file_size;

while (current_block != 0xFFF && remaining_bytes > 0)
{
unsigned int bytes_to_write = (remaining_bytes > sb.block_size) ?
sb.block_size : remaining_bytes;

fseek(disk, current_block * sb.block_size, SEEK_SET);
fread(disk, bytes_to_write, 1, src_file);

remaining_bytes -= bytes_to_write;
current_block = FAT[current_block];
}

fclose(src_file);

// Create the remaining subdirectories recursively
char subdirectory[FILENAME_SIZE + EXTENSION_SIZE + 2];
```

```c
char *next_slash = strchr(remaining_path, '\\');

while (next_slash)
{
strncpy(subdirectory, remaining_path, next_slash - remaining_path);
subdirectory[next_slash - remaining_path] = '\0';

make_directory_recursive(disk, sb, next_block, subdirectory);

remaining_path = next_slash + 1;
next_slash = strchr(remaining_path, '\\');
}

// Create the last subdirectory
make_directory_recursive(disk, sb, next_block, remaining_path);
}
```

The ***make_file function*** serves as the entry point for creating a file within the file system. It takes the following parameters:

- disk_name: The name of the file system data file.
- path: The path to the file being created.
- source_file: The source file from which to copy the contents of the new file.

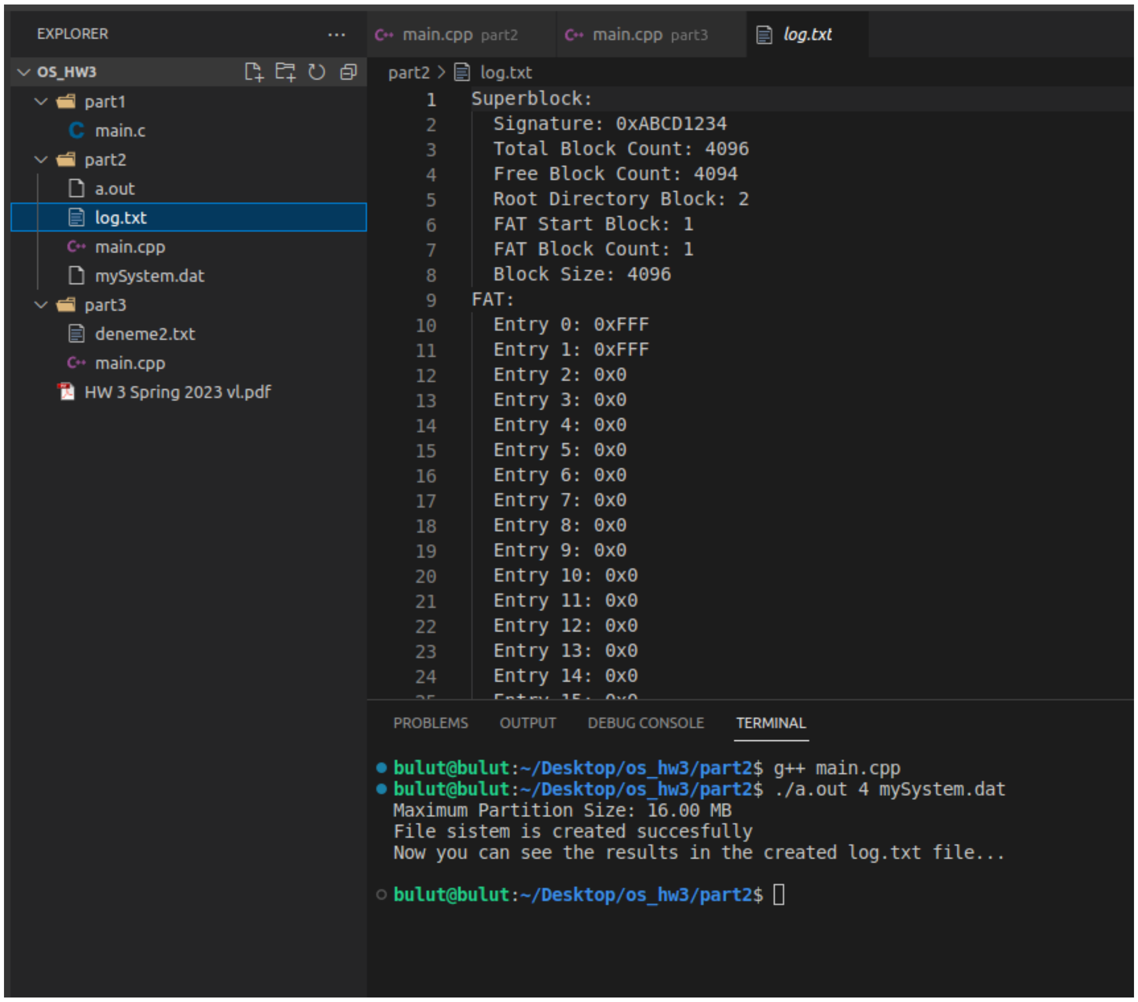
The function opens the file system data file in "r+b" (read and write binary) mode using fopen. It reads the superblock from the file system using fread and stores it in the sb variable. The function reads the FAT from the disk, seeking to the appropriate position using fseek and reading the data using fread. The function then calls make_file_recursive to create the file, passing the necessary parameters.

```c
void make_file(const char *disk_name, char *path, const char *source_file)
{
disk = fopen(disk_name, "r+b");
if (!disk)
{
perror("Error opening filesystem");
exit(1);
}
// Read Superblock
Superblock sb;
fread(&sb, sizeof(Superblock), 1, disk);

// Read FAT
fseek(disk, sb.fat_start_block * sb.block_size, SEEK_SET);
fread(&FAT, sizeof(FAT), 1, disk);

// Create the file recursively
make_file_recursive(disk, sb, sb.root_directory_block, path, source_file);

fclose(disk);
}
```

I couldn't implement the remain parts.

Tests

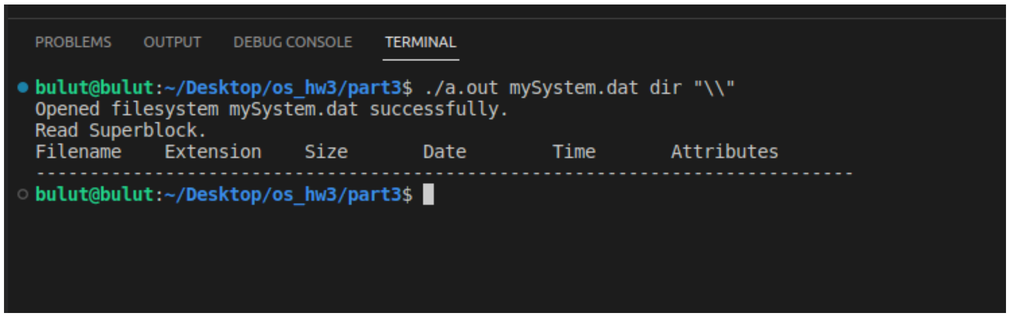
Creating a File System



```
Superblock:
  Signature: 0xABCD1234
  Total Block Count: 4096
  Free Block Count: 4094
  Root Directory Block: 2
  FAT Start Block: 1
  FAT Block Count: 1
  Block Size: 4096
FAT:
  Entry 0: 0xFFF
  Entry 1: 0xFFF
  Entry 2: 0x0
  Entry 3: 0x0
  Entry 4: 0x0
  Entry 5: 0x0
  Entry 6: 0x0
  Entry 7: 0x0
  Entry 8: 0x0
  Entry 9: 0x0
  Entry 10: 0x0
  Entry 11: 0x0
  Entry 12: 0x0
  Entry 13: 0x0
  Entry 14: 0x0
```

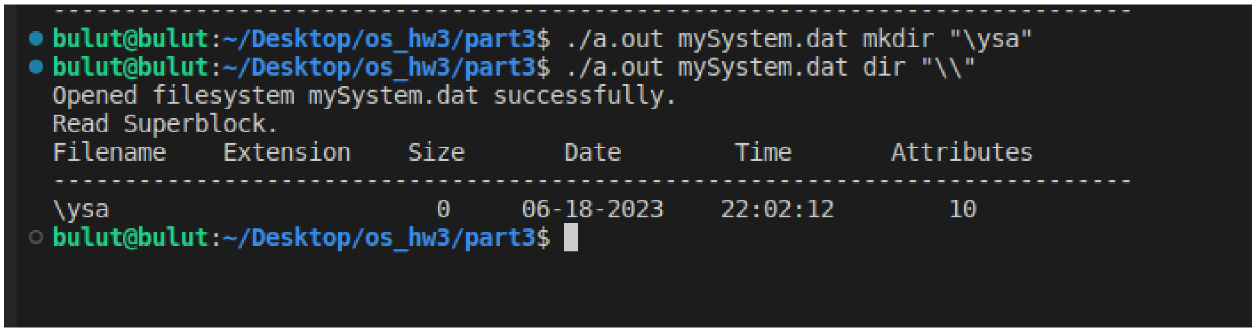
```
bulut@bulut:~/Desktop/os_hw3/part2$ g++ main.cpp
bulut@bulut:~/Desktop/os_hw3/part2$ ./a.out 4 mySystem.dat
Maximum Partition Size: 16.00 MB
File sistem is created succesfully
Now you can see the results in the created log.txt file...

bulut@bulut:~/Desktop/os_hw3/part2$
```

Dir command

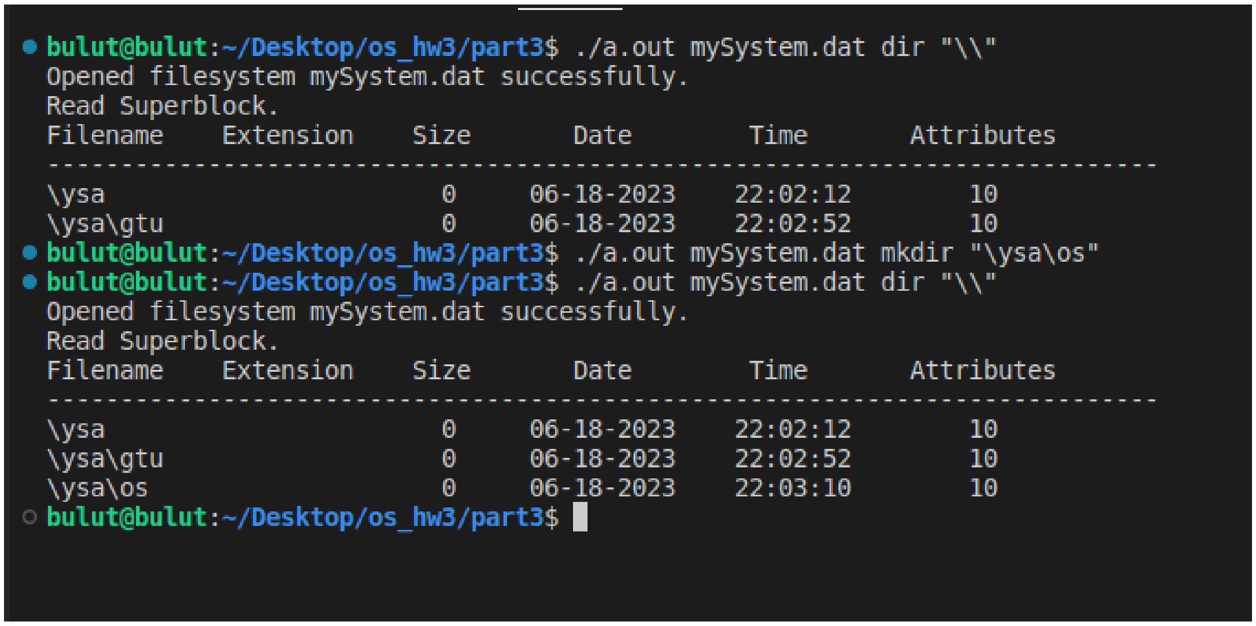


```
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
Opened filesystem mySystem.dat successfully.
Read Superblock.
Filename     Extension     Size        Date        Time        Attributes
----------------------------------------------------------------------------
bulut@bulut:~/Desktop/os_hw3/part3$
```

mkdir command

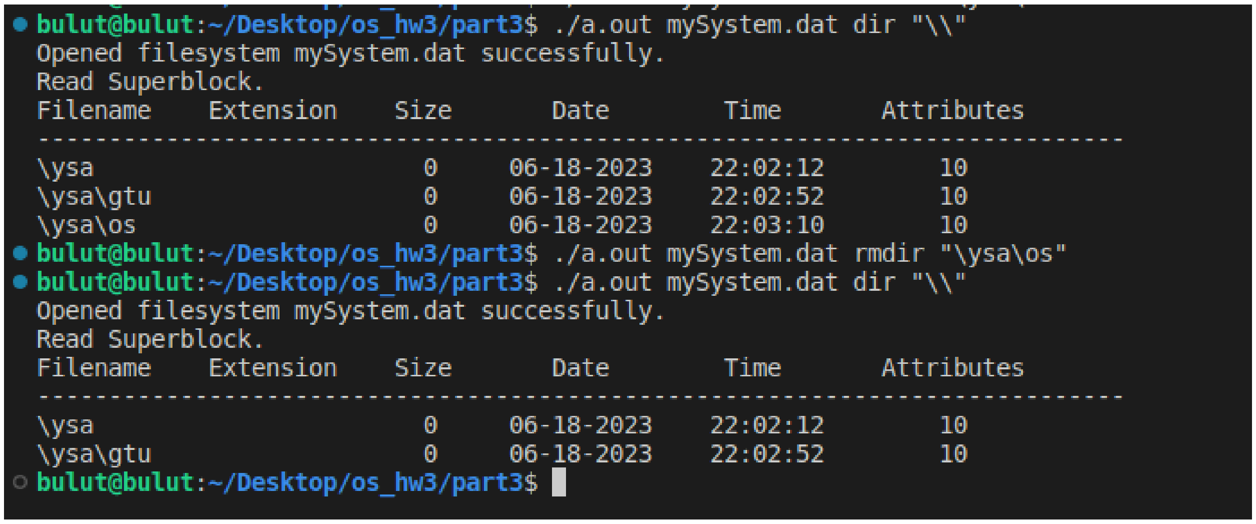


```
  ----------------------------------------------------------------------------
● bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat mkdir "\ysa"
● bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
  Opened filesystem mySystem.dat successfully.
  Read Superblock.
  Filename     Extension     Size       Date         Time        Attributes
  ----------------------------------------------------------------------------
  \ysa                       0      06-18-2023    22:02:12        10
○ bulut@bulut:~/Desktop/os_hw3/part3$ 
```



```
● bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
  Opened filesystem mySystem.dat successfully.
  Read Superblock.
  Filename     Extension     Size       Date         Time        Attributes
  ----------------------------------------------------------------------------
  \ysa                       0      06-18-2023    22:02:12        10
  \ysa\gtu                   0      06-18-2023    22:02:52        10
● bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat mkdir "\ysa\os"
● bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
  Opened filesystem mySystem.dat successfully.
  Read Superblock.
  Filename     Extension     Size       Date         Time        Attributes
  ----------------------------------------------------------------------------
  \ysa                       0      06-18-2023    22:02:12        10
  \ysa\gtu                   0      06-18-2023    22:02:52        10
  \ysa\os                    0      06-18-2023    22:03:10        10
○ bulut@bulut:~/Desktop/os_hw3/part3$ 
```

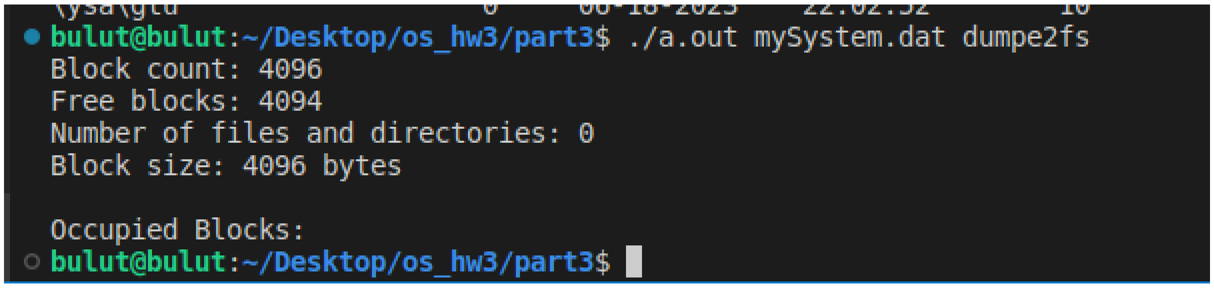
rmdir command



```
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
Opened filesystem mySystem.dat successfully.
Read Superblock.
Filename    Extension    Size       Date         Time       Attributes
----------------------------------------------------------------------------
\ysa                      0     06-18-2023    22:02:12        10
\ysa\gtu                  0     06-18-2023    22:02:52        10
\ysa\os                   0     06-18-2023    22:03:10        10
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat rmdir "\ysa\os"
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
Opened filesystem mySystem.dat successfully.
Read Superblock.
Filename    Extension    Size       Date         Time       Attributes
----------------------------------------------------------------------------
\ysa                      0     06-18-2023    22:02:12        10
\ysa\gtu                  0     06-18-2023    22:02:52        10
bulut@bulut:~/Desktop/os_hw3/part3$ 
```

dumpe2fs command

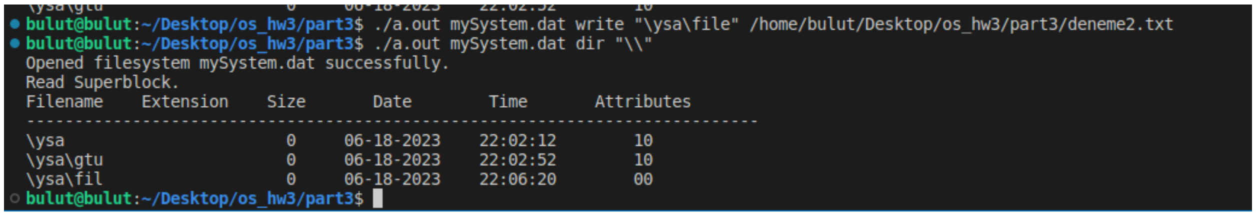
This command is not working properly.



```
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dumpe2fs
Block count: 4096
Free blocks: 4094
Number of files and directories: 0
Block size: 4096 bytes

Occupied Blocks:
bulut@bulut:~/Desktop/os_hw3/part3$ 
```

write command

Here it creates a file but it doesn't copy the content of given file.



```
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat write "\ysa\file" /home/bulut/Desktop/os_hw3/part3/deneme2.txt
bulut@bulut:~/Desktop/os_hw3/part3$ ./a.out mySystem.dat dir "\\"
Opened filesystem mySystem.dat successfully.
Read Superblock.
Filename    Extension    Size       Date         Time       Attributes
----------------------------------------------------------------------------
\ysa                      0     06-18-2023    22:02:12        10
\ysa\gtu                  0     06-18-2023    22:02:52        10
\ysa\fil                  0     06-18-2023    22:06:20        00
bulut@bulut:~/Desktop/os_hw3/part3$ 
```

I couldn't implement the other functions such as read and del.