

CSC 415 Operating Systems File System Group Project

Section 2 Fall 2023

Professor: Robert Biermann
Created by: The Byte Builders

Members:	
Miguel Antonio Logarta	Student ID: 923062683
Mohammed Deeb	Student ID: 922875899
Abdarrahman Ayyaz	Student ID: 922859909
Avitaj Singh Bhangoo	Student ID: 921303783

Github Repository Link:

<https://github.com/CSC415-2023-Fall/csc415-filesystem-Miguel-Antonio-Logarta>

Table of Contents

1. Introduction
2. Getting Started (How to compile the project)
3. How our file system works
 - a. The Volume Control Block (VCB)
 - b. The File Allocation Table (FAT)
 - c. Directory entries
4. An overview of our header and source files
 - a. fsLow.h and fsInit.c
 - b. partition.h
 - c. b_io.h
 - d. mfs.h
 - e. debug.h
 - f. fsshell.c
5. Issues, setbacks, and compromises we encountered
6. Screenshots of our program working
7. Conclusion

Introduction

In this group project, the assignment was to create our own virtual file system using C. One of the limitations imposed on us as a challenge was that we were only allowed to use the provided functions `LBAread()` and `LBAwrite()` as our only interface between our file system and the disk. We were also not allowed to use any linux or library file functions since we were supposed to implement those features ourselves.

Our file system is based on the FAT file system, which uses linked-list allocation to store files and directories within the partition. We also decided to add a twist to this approach by adding a free-space linked list to our system. This twist adds a nice advantage by making the task of searching for free space a lot more convenient and quick.

Our chosen file system also allowed us to store our directories and files in a tree-like fashion. Since we have linked-list allocation, there is also theoretically no limit to how many subdirectories and files a directory can contain.

Getting Started (How to Compile the Project)

Thanks to Makefile that was provided to us, running the project is simple.

Type the command:

```
make run
```

Into the terminal to compile the project and run it automatically.

To clean the project after you are done, just type the command:

```
make clean
```

To remove all the object files in the build folder.

We also modified the Makefile to support other features that you might find useful.

<code>make vrun</code>	Run the project with Valgrind, a memory leak checker tool
<code>make gdb</code>	Run the project with gdb
<code>make debug</code>	Run the project in debug mode. This prints out debug statements and other diagnostic data when running the project.
<code>make vdebug</code>	Run the project in debug mode with Valgrind. This prints out debug statements and also shows potential memory leaks while running the program.

How our File System Works

The Volume Control Block (VCB)

The Volume Control Block is located on LBA block 0 of our file system. We decided to put the VCB there for easy access whenever we need to read it from the disk. We also decided to only make it 64 bytes large since the VCB didn't need to carry a lot of information. It only stores necessary information for formatting our partition.

```
/*
    Volume Control Block: 64 bytes
    This struct is responsible for storing information about a partition
*/
#pragma pack (1)
typedef struct VCB_s {
    uint64_t magic_signature;    // Unique number to identify the partition
    uint64_t volume_size;       // Size of volume in bytes
    uint64_t block_size;        // Minimum block size of partition in bytes
    uint64_t num_blocks;        // Total number of blocks in partition
    uint64_t FAT_start;         // Starting lba position of FAT
    uint64_t FAT_length;        // Number of lba blocks the FAT takes up
    uint64_t DE_start;          // Starting lba block of our root directory
    uint64_t DE_length;         // Number of lba blocks our root directory
                                // takes up
} VCB;
```

When we initialize the volume, this is the first thing we read. If the signature matches, then we have the correct partition. If the magic signature is NULL, then we format a new partition. We also keep track of the starting block locations of the

File Allocation Table (FAT), as well as the location of our root directory, which is written right after the FAT.

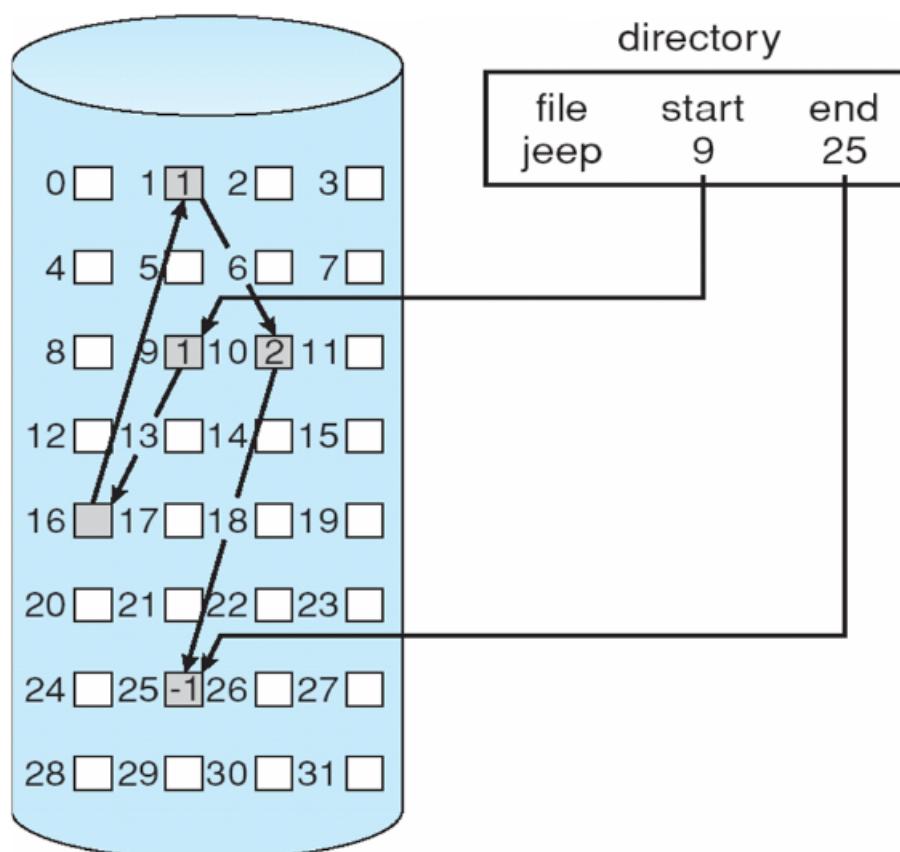
The File Allocation Table (FAT)

When we were deciding what type of file system to use, we initially went for a free space bitmap, where each bit maps to an LBA block in the disk. However, we soon realized that as we added and removed files from the directory, the space became more fragmented with lots of wasted space in between each file. The free space bitmap was compact in terms of how much memory it took up in the disk, but it did not fulfill our requirements. What we wanted was to have the ability to break our file into multiple pieces and keep track of them at the same time. The FAT file system was our next best option since it addressed these issues.

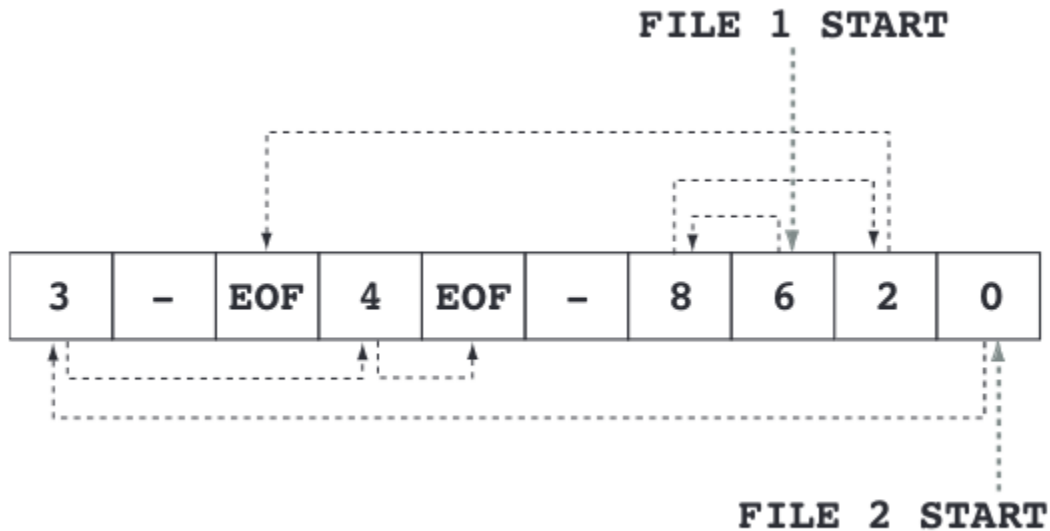
Our next best bet was the FAT File system. When we allocate space for a file, it takes up multiple blocks of memory in the file system. These blocks are not required to be contiguous in memory and can be fragmented into multiple areas of the disk.

After writing the VCB to the disk, we allocate space for our File Allocation Table (FAT). Our FAT table is represented as an array of FAT blocks. Each block is 4 bytes long. Since we are using linked-list allocation, each block contains 3 fields: An “in_use” field (1 bit), a “eof” field (1 bit), and a “next_lba_block” field (30

bits). When a block of memory is taken, “in_use” is set to 1. When we read a file, we read the first block that belongs to the file, and if the “eof” bit is not set (end of file), we read the next block. The address to the next block is stored in the “next_lba_block” field. We encounter the last block when we reach a block where the “eof” bit is set to 1.



Example of how a file would be stored in a FAT file system



Each file is stored like a linked list where each block would have an address to the next block

Here is our implementation:

```
/*  
    File Allocation Table Block: 4 bytes  
    This struct holds information for each block of space in our file system  
    Address is 30 bits long, which allows us to address up to 2^30 lba  
blocks  
*/  
#pragma pack (1)  
typedef struct FAT_block_s {  
    unsigned int in_use: 1;  
    unsigned int end_of_file: 1;  
    unsigned int next_lba_block: 30;  
} FAT_block;  
  
extern FAT_block *g_FAT;  
extern FAT_block *freeSpaceList;
```

In addition to our FAT table, we also added a way to keep track of all of our available free space in our system. Since the FAT table is represented as an array,

searching for free space would take $O(n)$ in the worst case scenario. For example, imagine we have a 10MB volume with a minimum block size of 512B, The amount of blocks we would need to address it would take up around 19,000 FAT Blocks. That could take a while to search. If we cached the addresses of all the available free space, we could reduce the search time from $O(n)$ to $O(1)$.

To do this, we scan the entire disk for free space when we boot the file system. We then cache all the free space in memory, which in this case is through the variable, `freeSpaceList`.

Directory Entries

In our file system, every file and directory is represented by a directory entry. Here is our implementation:

```
/*
    Directory Entry: 64 bytes
    This struct is how we store files and subdirectories in the file system.
    In a FAT file system, files and directories are stored in the same way.
    The only way to differentiate them is with the is_directory bit.
    If is_directory == 0, read it like a normal file
    Else if is_directory == 1, treat the directory entry's associated lba
        block like a sequence of more directory entries
*/
#pragma pack (1)
typedef struct directory_entry_s {
    unsigned char is_directory;
    char name[31];
    uint64_t block_location;
    size_t file_size;
    time_t date_created;
    time_t last_modified;
```

```
} directory_entry;
```

The directory entry has multiple variables that keep information about the file. To differentiate whether a directory entry represents a directory or a file, we use the `is_directory` variable. This variable is 8 bits long. Initially, we had plans to convert this variable into something we can store file attributes to, but that idea was discarded. We also have the file size and the starting block location for the file.

One of our limitations was the name field. It only allows 31 characters including the null terminator. This was a compromise to make the directory entry fit into 64 bytes.

For our file structure, we decided to go with a tree-based file structure. Like Linux, we have one root directory and every file and subdirectory is connected to root in a tree-like structure. When we read a directory entry, we check if the `is_directory` variable is set to true. If it is a directory, we read the rest of the block as an array of directory entries which contain pointers to more subdirectories and files.

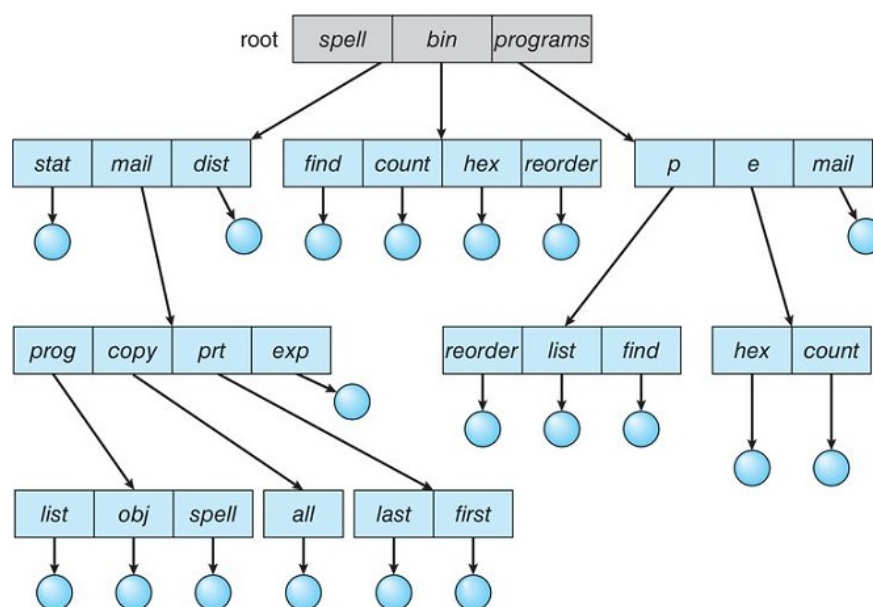


Figure 13.9 Tree-structured directory structure.

Here is a hexadecimal dump of our volume partition. This is what's contained in our root directory:

013600:	01 2F 00 00 00 00 00 00	00 00 00 00 00 00 00 00	./.....
013610:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
013620:	9A 00 00 00 00 00 00 00	40 01 00 00 00 00 00 00	@.....
013630:	96 9E 65 65 00 00 00 00	96 9E 65 65 00 00 00 00	ee.....ee.....
013640:	01 2E 2E 00 00 00 00 00	00 00 00 00 00 00 00 00
013650:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
013660:	9A 00 00 00 00 00 00 00	40 01 00 00 00 00 00 00	@.....
013670:	96 9E 65 65 00 00 00 00	96 9E 65 65 00 00 00 00	ee.....ee.....
013680:	01 48 6F 6D 65 00 00 00	00 00 00 00 00 00 00 00	.Home.....
013690:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0136A0:	9B 00 00 00 00 00 00 00	00 01 00 00 00 00 00 00@.....
0136B0:	96 9E 65 65 00 00 00 00	96 9E 65 65 00 00 00 00	ee.....ee.....
0136C0:	01 44 65 73 6B 74 6F 70	00 00 00 00 00 00 00 00	.Desktop.....
0136D0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0136E0:	9C 00 00 00 00 00 00 00	80 00 00 00 00 00 00 00@.....
0136F0:	96 9E 65 65 00 00 00 00	96 9E 65 65 00 00 00 00	ee.....ee.....

The 64 bytes is a directory entry that has the name “/” which is our root directory.

What follows are other subdirectories within the root directory. These subdirectories are .., Home, Desktop, and Notes.

FreeSpace:

We track free space using FAT (file allocation table). Which is a list of

FAT_blocks:

```
#pragma pack (1)
typedef struct FAT_block_s {
    unsigned int in_use: 1;
    unsigned int end_of_file: 1;
    unsigned int next_lba_block: 30;
} FAT_block;
```

Each FAT_block struct represents a block in the volume and tracks the information shown in the image above. When a block is Free in_use is set equal to 0 and next_lba_block is set to the next free block in the volume. This list is initialized when the volume is created and stored on the disk. And it is updated every time a block is freed or used. When the file system runs, it checks if the volume is already initialized. If the volume is already initialized it reads the table from the disk and stores it in ram for easy access. Moreover, an integer firstFreeBlock holds the value of the first Free block in the table and it is initialized when the program runs and it is updated when a block is used or freed. The list is written into the disk every time

it is modified so that in case the program shuts down, the updates will be saved on disk.

Creating a File:

To create a file you must pass the Create flag to `b_open`. You can either pass a path (with the file name being the last element of the path), or pass only the file name.

In case only the file name was passed, the file will be created at the current working directory. If a path was passed, `b_open` will trace through the directories in the path using `opendir()` and create the file there. To create the file, a new DE will be created and stored in the directory that the file will be stored in. A new free block will be also assigned for the file using our $O(1)$ time complexity freespace system. Later when the user writes into the file, the file will expand as the user needs.

The following is an example of how the root dir looks like when the testfile.txt DE is added to it

5274	013600:	01 2F 00 00 00 00 00 00	00 00 00 00 00 00 00 00	./.....
5275	013610:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5276	013620:	9A 00 00 00 00 00 00 00	C0 00 00 00 00 00 00 00	0.....0.....
5277	013630:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5278	013640:	01 2E 2E 00 00 00 00 00	00 00 00 00 00 00 00 00
5279	013650:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5280	013660:	9A 00 00 00 00 00 00 00	C0 00 00 00 00 00 00 00	0.....0.....
5281	013670:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5282	013680:	00 74 65 73 74 66 69 6C	65 2E 74 78 74 00 00 00	.testfile.txt...
5283	013690:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5284	0136A0:	9B 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	0.....
5285	0136B0:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5286	0136C0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5287	0136D0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5288	0136E0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5289	0136F0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

- Yellow is the DE of the “.” current dir (root)
- Green is the DE of the “..” parent dir (root)
- Light blue is the DE of the new file created (testfile.txt)

Opening a file:

Opening a file creates a new fcb for that file and stores it in an array of fcbs. The fcb needed metadata of the file. And it keeps track of the users interaction with the file (using index and current block)

```
typedef struct b_fcb {
    /** TODO add al the information you need in the file control block */
    char *buf; // holds the open file buffer
    int index; // holds the current position in the buffer
    int buflen; // holds how many valid bytes are in the buffer
    int location;
```

```
int currentBlock;  
int fileSize;  
int flag;  
} b_fcb;
```

Writing to a File:

Opening a file with the flags O_WRONLY or O_RDWR allows the user to call the b_write function, with the help of the file FCB b_write keeps track of where the user is writing in the file. It uses a new Free block whenever the current block is filled using the FreeSpaceList.

The following is an example of writing 750 bytes of numbers counting starting from 1 with a “-” between the numbers ()

```

013800: 31 2D 32 2D 33 2D 34 2D 35 2D 36 2D 37 2D 38 2D | 1-2-3-4-5-6-7-8-
013810: 39 2D 31 30 2D 31 31 2D 31 32 2D 31 33 2D 31 34 | 9-10-11-12-13-14
013820: 2D 31 35 2D 31 36 2D 31 37 2D 31 38 2D 31 39 2D | -15-16-17-18-19-
013830: 32 30 2D 32 31 2D 32 32 2D 32 33 2D 32 34 2D 32 | 20-21-22-23-24-2
013840: 35 2D 32 36 2D 32 37 2D 32 38 2D 32 39 2D 33 30 | 5-26-27-28-29-30
013850: 2D 33 31 2D 33 32 2D 33 33 2D 33 34 2D 33 35 2D | -31-32-33-34-35-
013860: 33 36 2D 33 37 2D 33 38 2D 33 39 2D 34 30 2D 34 | 36-37-38-39-40-4
013870: 31 2D 34 32 2D 34 33 2D 34 34 2D 34 35 2D 34 36 | 1-42-43-44-45-46
013880: 2D 34 37 2D 34 38 2D 34 39 2D 35 30 2D 35 31 2D | -47-48-49-50-51-
013890: 35 32 2D 35 33 2D 35 34 2D 35 35 2D 35 36 2D 35 | 52-53-54-55-56-5
0138A0: 37 2D 35 38 2D 35 39 2D 36 30 2D 36 31 2D 36 32 | 7-58-59-60-61-62
0138B0: 2D 36 33 2D 36 34 2D 36 35 2D 36 36 2D 36 37 2D | -63-64-65-66-67-
0138C0: 36 38 2D 36 39 2D 37 30 2D 37 31 2D 37 32 2D 37 | 68-69-70-71-72-7
0138D0: 33 2D 37 34 2D 37 35 2D 37 36 2D 37 37 2D 37 38 | 3-74-75-76-77-78
0138E0: 2D 37 39 2D 38 30 2D 38 31 2D 38 32 2D 38 33 2D | -79-80-81-82-83-
0138F0: 38 34 2D 38 35 2D 38 36 2D 38 37 2D 38 38 2D 38 | 84-85-86-87-88-8

013900: 39 2D 39 30 2D 39 31 2D 39 32 2D 39 33 2D 39 34 | 9-90-91-92-93-94
013910: 2D 39 35 2D 39 36 2D 39 37 2D 39 38 2D 39 39 2D | -95-96-97-98-99-
013920: 31 30 30 2D 31 30 31 2D 31 30 32 2D 31 30 33 2D | 100-101-102-103-
013930: 31 30 34 2D 31 30 35 2D 31 30 36 2D 31 30 37 2D | 104-105-106-107-
013940: 31 30 38 2D 31 30 39 2D 31 31 30 2D 31 31 31 2D | 108-109-110-111-
013950: 31 31 32 2D 31 31 33 2D 31 31 34 2D 31 31 35 2D | 112-113-114-115-
013960: 31 31 36 2D 31 31 37 2D 31 31 38 2D 31 31 39 2D | 116-117-118-119-
013970: 31 32 30 2D 31 32 31 2D 31 32 32 2D 31 32 33 2D | 120-121-122-123-
013980: 31 32 34 2D 31 32 35 2D 31 32 36 2D 31 32 37 2D | 124-125-126-127-
013990: 31 32 38 2D 31 32 39 2D 31 32 40 2D 31 32 41 2D | 128-129-130-131-

```


5334	013990:	31 32 38 2D 31 32 39 2D	31 33 30 2D 31 33 31 2D	128-129-130-131-
5335	0139A0:	31 33 32 2D 31 33 33 2D	31 33 34 2D 31 33 35 2D	132-133-134-135-
5336	0139B0:	31 33 36 2D 31 33 37 2D	31 33 38 2D 31 33 39 2D	136-137-138-139-
5337	0139C0:	31 34 30 2D 31 34 31 2D	31 34 32 2D 31 34 33 2D	140-141-142-143-
5338	0139D0:	31 34 34 2D 31 34 35 2D	31 34 36 2D 31 34 37 2D	144-145-146-147-
5339	0139E0:	31 34 38 2D 31 34 39 2D	31 35 30 2D 31 35 31 2D	148-149-150-151-
5340	0139F0:	31 35 32 2D 31 35 33 2D	31 35 34 2D 31 35 35 2D	152-153-154-155-
5341				
5342	013A00:	31 35 36 2D 31 35 37 2D	31 35 38 2D 31 35 39 2D	156-157-158-159-
5343	013A10:	31 36 30 2D 31 36 31 2D	31 36 32 2D 31 36 33 2D	160-161-162-163-
5344	013A20:	31 36 34 2D 31 36 35 2D	31 36 36 2D 31 36 37 2D	164-165-166-167-
5345	013A30:	31 36 38 2D 31 36 39 2D	31 37 30 2D 31 37 31 2D	168-169-170-171-
5346	013A40:	31 37 32 2D 31 37 33 2D	31 37 34 2D 31 37 35 2D	172-173-174-175-
5347	013A50:	31 37 36 2D 31 37 37 2D	31 37 38 2D 31 37 39 2D	176-177-178-179-
5348	013A60:	31 38 30 2D 31 38 31 2D	31 38 32 2D 31 38 33 2D	180-181-182-183-
5349	013A70:	31 38 34 2D 31 38 35 2D	31 38 36 2D 31 38 37 2D	184-185-186-187-
5350	013A80:	31 38 38 2D 31 38 39 2D	31 39 30 2D 31 39 31 2D	188-189-190-191-
5351	013A90:	31 39 32 2D 31 39 33 2D	31 39 34 2D 31 39 35 2D	192-193-194-195-
5352	013AA0:	31 39 36 2D 31 39 37 2D	31 39 38 2D 31 39 39 2D	196-197-198-199-
5353	013AB0:	32 30 30 2D 32 30 31 2D	32 30 32 2D 32 30 33 2D	200-201-202-203-
5354	013AC0:	32 30 34 2D 32 30 35 2D	32 30 36 2D 32 30 37 2D	204-205-206-207-
5355	013AD0:	32 30 38 2D 32 30 39 2D	32 31 30 2D 32 31 31 2D	208-209-210-211-
5356	013AE0:	32 31 32 2D 32 31 33 2D	32 31 34 00 00 00 00 00	212-213-214.....
5357	013AF0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 5358

As you can see, the file has expanded after using 1 block and started writing on the next free block. In this case the next free block was the block right after it

```
Writing to file...
Writing to file -> Writing 1 block to 155
Updating Free Space List -> Writing block 153 to 1
Writing to file -> Writing 1 block to 156
```

An Overview of our Header and Source Files

If you look into our src folder, you'll find a number of header and source files that each serve an important purpose to our file system.

- A. fsLow.h and fsInit.c - This file contains routines that are responsible for starting up the volume partition and also provides an interface to directly read and write to the virtual file system. Important functions in here are LBAread() and LBAwrite() which interact with our volume. InitFilesystem() is also another function that is responsible for formatting the volume. It writes the VCB, FAT, and the root directory.
- B. partition.h and partition.c - This file contains routines that provide an interface to reading and writing both the VCB and FAT to and from the disk. It also contains the data structure for the VCB, FAT, and directory entry. Global variables g_vcb and g_FAT are declared here. They serve to keep the VCB and FAT in memory so that we do not have to read it from the disk every time we need to read or change a value.
- C. b_io.h and b_io.c - Our buffered I/O library. These files are responsible for reading and writing data into the volume. It creates an abstraction layer that handles reading and writing multiple blocks of memory into the volume. For example, if we have a file that is split between multiple fragmented parts of disk space, b_io will search and combine all these memory blocks into one buffer for the file system to read.

- D. mfs.h and mfs.c - My File System. This is where most of our code for the file system is located. It contains implementations of basic file system interaction. It allows you to create directories/files, navigate the file system, read directories/files, and delete directories/files.
- E. debug.h and debug.c - This is our debugging library. It contains macros and functions that would only compile when the debug flag is set. For example, `debug_print()` prints out whatever string you give it when you're running in debug mode, but doesn't print anything when you're running the release version of the program.
- F. fsshell.c - Our driver program. This is our shell that the user interacts with to use our file system.

Issues, Setbacks, and Compromises We Encountered

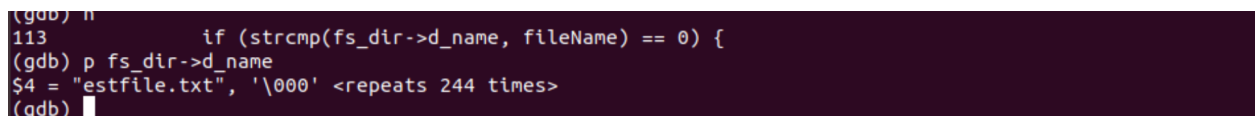
Of course, making our own version of a file system did not come without its own set of difficulties. There were many challenges that we as a team had to face while creating it. Even though we had the concepts down, it was difficult to translate what we had on paper to actual code.

Initially, writing the VCB, FAT, and Root to the partition when formatting the volume was a challenge. We wondered why writing to LBA block 0 didn't overwrite the first 512 bytes of the volume. It took us a while to figure out that the

first 512 bytes were reserved for the boot sector, which we were not allowed to access. After that, we had to figure out how to use `LBAwrite()` to write our data structures to the volume.

Next, writing code for both `mfs.c` and `b_io.c` was a difficult task to get started on. A lot of the functions were tightly coupled and relied on each other to work. One solution we had was to manually write test files to the volume, then work on each function one by one until it no longer relied on the test files.

One of the issues we encountered was a failure of opening files. After the file was created successfully, `b_open` could not locate that file in the directory it was saved in. To debug that, we made sure that we were reading from the right location in memory. In this case it was block 154 (root dir), since the file we were reading was stored in the root directory. After that, we made sure that the reading process was working correctly. Lastly, we checked if the name comparison works correctly when looking for the DE that matched the file we are looking for. The following image shows what was causing the issue:



```
(gdb) n
113         if (strcmp(fs_dir->d_name, fileName) == 0) {
(gdb) p fs_dir->d_name
$4 = "estfile.txt", '\000' <repeats 244 times>
(gdb)
```

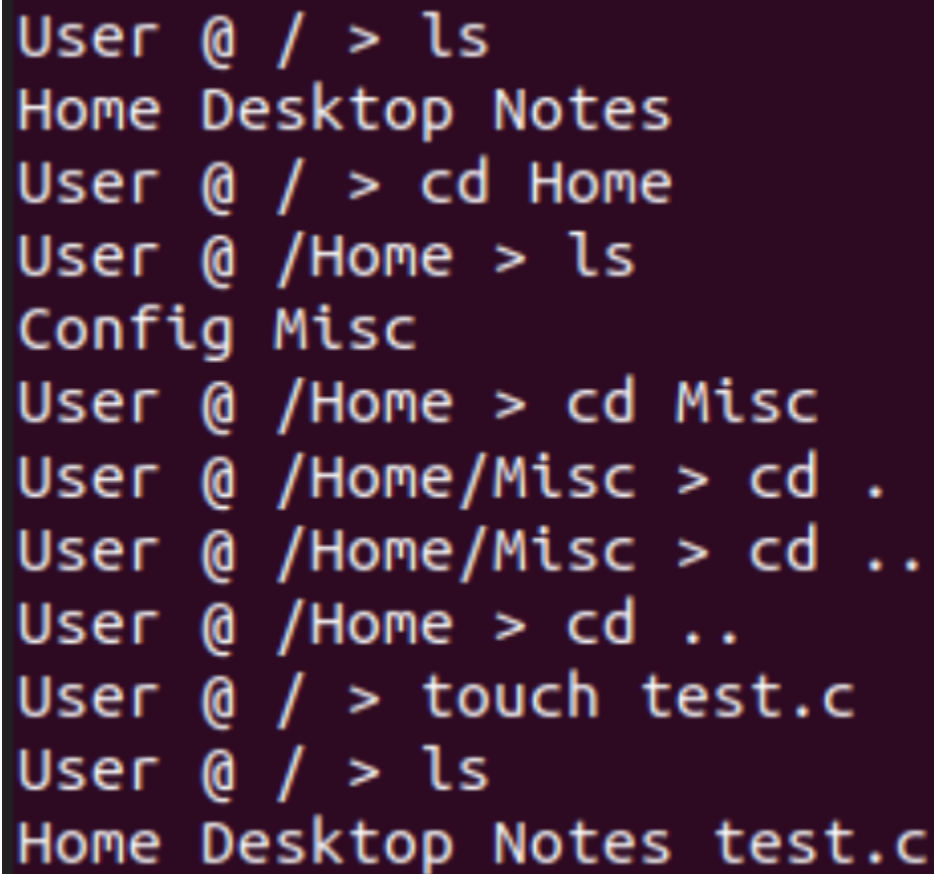
The file name stored in the file's DE was incorrect. Considering that the file's DE was read from the disk, the issue could either be in reading the file's DE

from the disk or writing the file's DE into the disks. With more debugging, we found that the issue was in writing the DE into the disk. When the file's DE was copied into the directory's buffer, it was copied starting from one byte before from where it is supposed to be copied. This was simply solved by removing the -1 from the line:

```
memcpy(dirBuffer + fdD->directory->file_size -1, DE, sizeof(directory_entry));
```

Screenshots of our Program Working

This is the “ls” command working:

A terminal window with a dark purple background and light blue text. The text shows a series of commands and their outputs. The commands are: 'ls' at the root, 'cd Home', 'ls' in the Home directory, 'cd Misc', 'cd .' in the Misc directory, 'cd ..' to return to Home, 'cd ..' to return to the root, 'touch test.c' at the root, and 'ls' at the root. The outputs are: 'Home Desktop Notes' for the first 'ls', 'Config Misc' for the second 'ls', and 'Home Desktop Notes test.c' for the final 'ls'.

```
User @ / > ls
Home Desktop Notes
User @ / > cd Home
User @ /Home > ls
Config Misc
User @ /Home > cd Misc
User @ /Home/Misc > cd .
User @ /Home/Misc > cd ..
User @ /Home > cd ..
User @ / > touch test.c
User @ / > ls
Home Desktop Notes test.c
```

This is the “ls -la” command working:

```
User @ / > ls -la
D          384B .
D          384B ..
D          256B Home
D          128B Desktop
D          256B Notes
-           0B test.c

User @ / > 
```

This is the “pwd” command working:

```
User @ / > ls
Home Desktop Notes test.c
User @ / > cd Home
User @ /Home > ls
Config Misc
User @ /Home > cd Config
User @ /Home/Config > pwd
/Home/Config
User @ /Home/Config > 
```

The following is an example of writing 750 bytes of numbers counting starting from 1 with a “-” between the numbers ()

```
013800: 31 2D 32 2D 33 2D 34 2D 35 2D 36 2D 37 2D 38 2D | 1-2-3-4-5-6-7-8-
013810: 39 2D 31 30 2D 31 31 2D 31 32 2D 31 33 2D 31 34 | 9-10-11-12-13-14
013820: 2D 31 35 2D 31 36 2D 31 37 2D 31 38 2D 31 39 2D | -15-16-17-18-19-
013830: 32 30 2D 32 31 2D 32 32 2D 32 33 2D 32 34 2D 32 | 20-21-22-23-24-2
013840: 35 2D 32 36 2D 32 37 2D 32 38 2D 32 39 2D 33 30 | 5-26-27-28-29-30
013850: 2D 33 31 2D 33 32 2D 33 33 2D 33 34 2D 33 35 2D | -31-32-33-34-35-
013860: 33 36 2D 33 37 2D 33 38 2D 33 39 2D 34 30 2D 34 | 36-37-38-39-40-4
013870: 31 2D 34 32 2D 34 33 2D 34 34 2D 34 35 2D 34 36 | 1-42-43-44-45-46
013880: 2D 34 37 2D 34 38 2D 34 39 2D 35 30 2D 35 31 2D | -47-48-49-50-51-
013890: 35 32 2D 35 33 2D 35 34 2D 35 35 2D 35 36 2D 35 | 52-53-54-55-56-5
0138A0: 37 2D 35 38 2D 35 39 2D 36 30 2D 36 31 2D 36 32 | 7-58-59-60-61-62
0138B0: 2D 36 33 2D 36 34 2D 36 35 2D 36 36 2D 36 37 2D | -63-64-65-66-67-
0138C0: 36 38 2D 36 39 2D 37 30 2D 37 31 2D 37 32 2D 37 | 68-69-70-71-72-7
0138D0: 33 2D 37 34 2D 37 35 2D 37 36 2D 37 37 2D 37 38 | 3-74-75-76-77-78
0138E0: 2D 37 39 2D 38 30 2D 38 31 2D 38 32 2D 38 33 2D | -79-80-81-82-83-
0138F0: 38 34 2D 38 35 2D 38 36 2D 38 37 2D 38 38 2D 38 | 84-85-86-87-88-8

013900: 39 2D 39 30 2D 39 31 2D 39 32 2D 39 33 2D 39 34 | 9-90-91-92-93-94
013910: 2D 39 35 2D 39 36 2D 39 37 2D 39 38 2D 39 39 2D | -95-96-97-98-99-
013920: 31 30 30 2D 31 30 31 2D 31 30 32 2D 31 30 33 2D | 100-101-102-103-
013930: 31 30 34 2D 31 30 35 2D 31 30 36 2D 31 30 37 2D | 104-105-106-107-
013940: 31 30 38 2D 31 30 39 2D 31 31 30 2D 31 31 31 2D | 108-109-110-111-
013950: 31 31 32 2D 31 31 33 2D 31 31 34 2D 31 31 35 2D | 112-113-114-115-
013960: 31 31 36 2D 31 31 37 2D 31 31 38 2D 31 31 39 2D | 116-117-118-119-
013970: 31 32 30 2D 31 32 31 2D 31 32 32 2D 31 32 33 2D | 120-121-122-123-
013980: 31 32 34 2D 31 32 35 2D 31 32 36 2D 31 32 37 2D | 124-125-126-127-
013990: 31 32 38 2D 31 32 39 2D 31 32 40 2D 31 32 41 2D | 128-129-130-131-
```



```

5334 013990: 31 32 38 2D 31 32 39 2D 31 33 30 2D 31 33 31 2D | 128-129-130-131-
5335 0139A0: 31 33 32 2D 31 33 33 2D 31 33 34 2D 31 33 35 2D | 132-133-134-135-
5336 0139B0: 31 33 36 2D 31 33 37 2D 31 33 38 2D 31 33 39 2D | 136-137-138-139-
5337 0139C0: 31 34 30 2D 31 34 31 2D 31 34 32 2D 31 34 33 2D | 140-141-142-143-
5338 0139D0: 31 34 34 2D 31 34 35 2D 31 34 36 2D 31 34 37 2D | 144-145-146-147-
5339 0139E0: 31 34 38 2D 31 34 39 2D 31 35 30 2D 31 35 31 2D | 148-149-150-151-
5340 0139F0: 31 35 32 2D 31 35 33 2D 31 35 34 2D 31 35 35 2D | 152-153-154-155-
5341
5342 013A00: 31 35 36 2D 31 35 37 2D 31 35 38 2D 31 35 39 2D | 156-157-158-159-
5343 013A10: 31 36 30 2D 31 36 31 2D 31 36 32 2D 31 36 33 2D | 160-161-162-163-
5344 013A20: 31 36 34 2D 31 36 35 2D 31 36 36 2D 31 36 37 2D | 164-165-166-167-
5345 013A30: 31 36 38 2D 31 36 39 2D 31 37 30 2D 31 37 31 2D | 168-169-170-171-
5346 013A40: 31 37 32 2D 31 37 33 2D 31 37 34 2D 31 37 35 2D | 172-173-174-175-
5347 013A50: 31 37 36 2D 31 37 37 2D 31 37 38 2D 31 37 39 2D | 176-177-178-179-
5348 013A60: 31 38 30 2D 31 38 31 2D 31 38 32 2D 31 38 33 2D | 180-181-182-183-
5349 013A70: 31 38 34 2D 31 38 35 2D 31 38 36 2D 31 38 37 2D | 184-185-186-187-
5350 013A80: 31 38 38 2D 31 38 39 2D 31 39 30 2D 31 39 31 2D | 188-189-190-191-
5351 013A90: 31 39 32 2D 31 39 33 2D 31 39 34 2D 31 39 35 2D | 192-193-194-195-
5352 013AA0: 31 39 36 2D 31 39 37 2D 31 39 38 2D 31 39 39 2D | 196-197-198-199-
5353 013AB0: 32 30 30 2D 32 30 31 2D 32 30 32 2D 32 30 33 2D | 200-201-202-203-
5354 013AC0: 32 30 34 2D 32 30 35 2D 32 30 36 2D 32 30 37 2D | 204-205-206-207-
5355 013AD0: 32 30 38 2D 32 30 39 2D 32 31 30 2D 32 31 31 2D | 208-209-210-211-
5356 013AE0: 32 31 32 2D 32 31 33 2D 32 31 34 00 00 00 00 00 | 212-213-214-....
5357 013AF0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
5358

```

As you can see, the file has expanded after using 1 block and started writing on the next free block. In this case the next free block was the block right after it

```

Writing to file...
Writing to file -> Writing 1 block to 155
Updating Free Space List -> Writing block 153 to 1
Writing to file -> Writing 1 block to 156

```

The following is an example of how the root dir looks like when the testfile.txt DE is added to it

5274	013600:	01 2F 00 00 00 00 00 00	00 00 00 00 00 00 00 00	./.....
5275	013610:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5276	013620:	9A 00 00 00 00 00 00 00	C0 00 00 00 00 00 00 00	0.....0.....
5277	013630:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5278	013640:	01 2E 2E 00 00 00 00 00	00 00 00 00 00 00 00 00
5279	013650:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5280	013660:	9A 00 00 00 00 00 00 00	C0 00 00 00 00 00 00 00	0.....0.....
5281	013670:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5282	013680:	00 74 65 73 74 66 69 6C	65 2E 74 78 74 00 00 00	.testfile.txt...
5283	013690:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5284	0136A0:	9B 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	0.....
5285	0136B0:	1A D7 67 65 00 00 00 00	1A D7 67 65 00 00 00 00	.0ge.....0ge...
5286	0136C0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5287	0136D0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5288	0136E0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
5289	0136F0:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

- Yellow is the DE of the “.” current dir (root)
- Green is the DE of the “..” parent dir (root)
- Light blue is the DE of the new file created (testfile.txt)

Conclusion

In conclusion, our team learned a lot from making our own implementation of a file system. It was difficult, but we learned a lot. We definitely underestimated the challenge of trying to implement a FAT file system, but we are proud of what we were able to achieve with the limited time frame we had. Overall it was a nice project that truly expanded our knowledge in C, FAT file systems, and Linux.