

# **File System Project**

**CSC 415-01**

**Group: Vile System**

**Group Members: Nathaniel Miller, Jasmine**

**Stapleton-Hart, Arianna Yuan**

**Student IDs (respectively): 922024360, 921356953,**  
**920898911**

**December 01, 2022**

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Overview</b>	<b>4</b>
Assignment Description	4
Github Project	4
File System Description	4
<b>Structure Descriptions</b>	<b>5</b>
Volume Control Block Structure	5
Directory Entry Definition	6
Metadata of files we initially desired	7
Metadata of files we ultimately implemented	7
File stats	7
<b>Teamwork</b>	<b>9</b>
Summary	9
Division of Work	10
<b>Approach</b>	<b>12</b>
Free Space Tracking	12
b_io.c	12
fsInit.c	14
Non File System Directory Functions	15
<b>Issues and Resolutions</b>	<b>17</b>
<b>Screenshots</b>	<b>18</b>
Screenshots of Program Compilation	18
Hexdump Compilation	18
Hexdump Execution	19
Execution of program	21
ls - Lists the file in a directory	21
cp - Copies a file - source [dest]	22
mv - Moves a file - source dest	23

md - Make a new directory	23
rm - Removes a file or directory	24
touch - creates a file	24
cat - displays contents of a file	24
cp2l - Copies a file from the test file system to the linux file system	25
cp2fs - Copies a file from the Linux file system to the test file system	26
cd - Changes directory	27
pwd	27
history - Prints out the history	28
help - Prints out help	29
<b>Resources &amp; References</b>	<b>30</b>

# Overview

## Assignment Description

The File System project is a group assignment in which we were asked to implement a file system of our choice. We designed the directory entry structure, the volume structure, and the free space to handle formatting the volume. Then, we implemented the interfaces for directory based operations and file operation functions.

## Github Project

Our project was created with Arianna's Github account (arianna-y).

<https://github.com/CSC415-2022-Fall/csc415-filesystem-arianna-y>

## File System Description

We chose to implement a file system similar to the FAT32 file system.

"A FAT file system volume is composed of four basic regions, which are laid out in this order on the volume:

- 0 - Reserved Region
- 1 - FAT Region
- 2 - Root Directory Region (doesn't exist on FAT32 volumes)
- 3 - File and Directory Data Region" (Microsoft Doc)

The difference between the FAT formats (FAT 12, FAT16, FAT32) is the number of bits in an entry. In FAT32, there are 32 bits in each entry.

# Structure Descriptions

## Volume Control Block Structure

```
typedef struct vcb
{
    // signature to check if vcb has been formatted yet
    long sig;
    // number of blocks in volume
    int numBlocks;
    // size of each block
    int blockSize;
    int numLBAPerBlock;
    // number of free blocks available
    int freeBlockCount;
    // block number of first free block available
    int nextFreeBlock;
    // where root dir starts
    int rootDirStart;
} vcb;
```

The Volume Control Block (VCB) structure is used in the initialization process of formatting the disk. The key aspect of this structure is the signature. The first thing that's checked when the first block is read is whether or not this signature is matched. This is so we know to format the volume to ready the file system or if it has already been written. If it does, we do not want to reformat our disk. Otherwise we would be formatting every single time we boot up.

If the signature does not match it is set (0x4E415445) and the initialization of the volume sizes are set. Then the free space and root directory initializations are called. This information is written to the 0th block.

## Directory Entry Definition

```
typedef struct directoryEntry
{
    char name[DE_NAME_MAXLEN];
    // limited to 20 characters

    // type of directory:
    // 0 is unused
    // 1 is directory
    // 2 is file
    int type;

    // large integers to hold to location (address)
and size
    uint64_t location;
    uint64_t size;

    // c time type to hold dates of creation, latest
modification,
    // latest viewing
    time_t lastModified;
    time_t lastOpened;
    time_t dateCreated;
} directoryEntry;
```

The directory system description is a collection of attributes of a file that will be useful to both the system and the user. It has the information about the directory or location of file in the file system, such as its name, whether it is a file location, its size, its last modification date, and its initial creation date.

In `initRootDirectory`, we first compute the size of the initial root directory and then we allocate memory for initializing the

root directory. First, we fill in the entries in the root directory -- we set the type to `DE_TYPE_UNUSED` and everything else to 0. Then, we fill in the first and second entries. The second directory has the same content as the first directory, except the name is `".."`.

## Metadata of files we initially desired

- File extension flag: the file extension of the file
- File name: the name of the file excluding the file extension and file path
- File type: extension type of the file
- File time: time of when the file was created
- File id: unique file identifier
- File user id: identifies the user that is associated with the file
- File size: size of the file
- Block position: origin/start of the block
- Location: memory address

## Metadata of files we ultimately implemented

- File name: `char name[20]`
- File time:
  - When file was last modifies: `time_t lastModified`
  - When file was last opened: `time_t lastOpened`
  - When file was created: `time_t dateCreated`
- File size (in bytes): `uint64_t size`
- Location: `uint64_t location`

## File stats

```
struct fs_stat
{
    off_t      st_size;           /* total size, in bytes */
```

```
    blksize_t st_blksize;          /* blocksize for file system
I/O */
    blkcnt_t  st_blocks;           /* number of 512B blocks
allocated */
    time_t    st_accesstime;       /* time of last access */
    time_t    st_modtime;         /* time of last modification */
    time_t    st_createtime;      /* time of last status change
*/
};
```

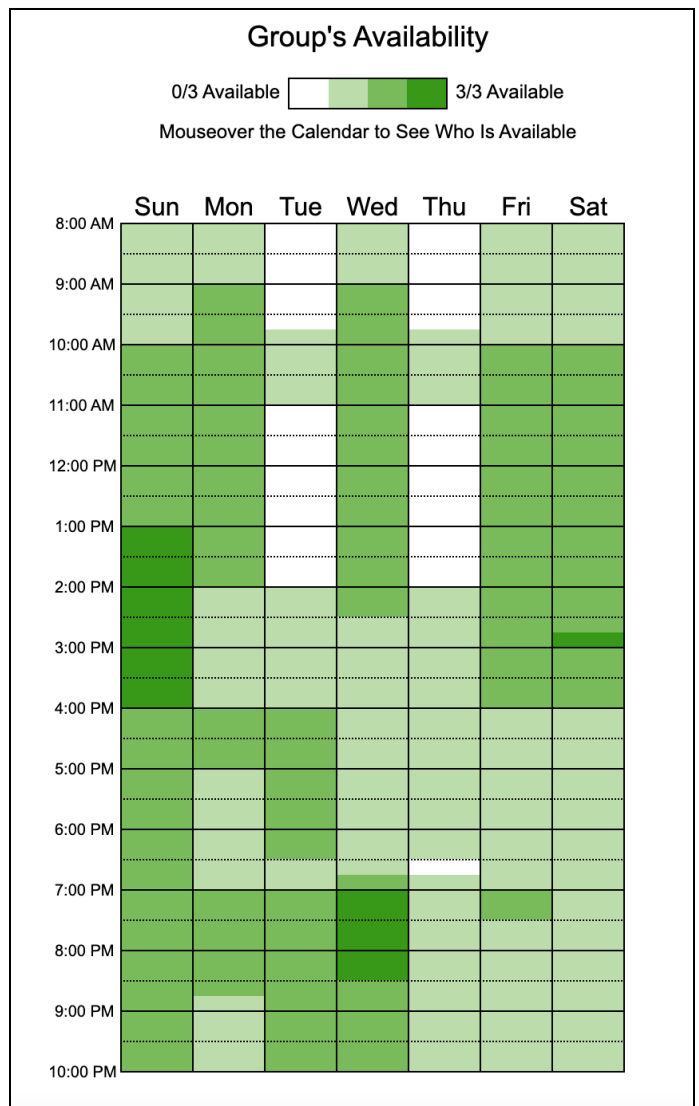


# Teamwork

## Summary

Our team met once weekly in the late evenings, usually on Mondays and Thursdays. We divided up the tasks by choosing which ones each of us felt that we understood the most. This project was extremely taxing in the aspect of scheduling and to work, because all of our group members are in multiple other groups for other classes. All members responded to requests to meet, attended meetings, and were reasonably reachable throughout the course of the project. Having one less member than most groups also negatively affected our progress, as there was one less person to contribute from the very beginning (the person who dropped never attended even the first in-class meeting).

When the project was first assigned, we created this when2meet form to find a weekly time for Vile System to meet (pictured right) The darkest green color represents times that all three of us are free during the week. The lighter the color, the less people are available (white being none of us are available). As you can see, there are really only two times a week that we were able to all meet up together. Additionally, we cannot all be on campus during any of these times, so almost all of the project was done



over Discord. While convenient because we were all able to work at a comfortable setup at home, sometimes teamwork is just simpler and more effective in person. Especially when it's such a complicated project with many moving parts.

We did end up persevering despite the time conflicts, but we feel that we would have a more cohesive end product if we had been on the same page more and had more consistent verbal discourse instead of relying on text-based replies through Discord.

Overall, we think that we did a good job adapting to each other and having individuals contribute what they're most capable of. For example, Arianna ended up doing a lot of work on the shell functions and the formatting early on in the project. So when it came time to write the documentation, Nate and Jasmine contributed much more in order to equalize the amount of work we were all doing and avoid burning anyone out.

## Division of Work

Item	Arianna	Nate	Jasmine
Writeup	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Document setup	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Adding screenshots of file functions, compilation, etc.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Function & interface descriptions	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Project, teamwork, approach, description	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Issues & Resolutions description	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Proofreading	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Item	Arianna	Nate	Jasmine
Hexdump explanation	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Interface implementations for shell function:	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>b_io_fd b_open (char * filename, int flags);</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<code>int b_read (b_io_fd fd, char * buffer, int count);</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<code>int b_write (b_io_fd fd, char * buffer, int count);</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>int b_seek (b_io_fd fd, off_t offset, int whence);</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<code>int b_close (b_io_fd fd);</code>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Modified Driver program (must be a program that just utilizes the header file for your file system)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
modify mfs.h for the fdDIR structure to be what your file system needs to maintain and track interaction through the directory structure.	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
file function implementations	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

# Approach

## Free Space Tracking

We will be following similar specifications and use similar free space tracking/allocation to the FAT32 file system. Our free space will be non-contiguous, so there is no need to find contiguous free blocks. To track it, we will have a File Allocation Table with each entry in the table representing a block. The entries will have a number which indicates if the block is free or not. There will also be an entry number reserved to signal end of file.

To create the table, "The file system driver implementation must scan through all FAT entries to construct a list of free/available clusters." ([Microsoft FAT specification p. 20](#)). The FAT will then be placed in the first block (after the reserved volume), followed by the root directory, and then the rest of the file/directory data. We could also create helper functions (eg. getNextFree) to find the next free block(s) with better readability. If there aren't enough free blocks when requested, an error will be thrown. We decided to use the code for the few file functions we had to create for Assignment 5 (create, read, close) from Jasmine's A5 submission as our base for b\_io.c moving forward.

## b\_io.c

The interface **b\_open** takes the string filename and an integer representing the flags, and returns the file descriptor associated with the file (if) successfully opened. The first thing to do is check if the system has been initialized (in which case **b\_init** would have already been called in which startup would've been set to 1). We get the file information for the file we want to open, and check that it's valid.

If not initialized, we call **b\_init** before moving on. This function checks that startup is set to zero (meaning the volume has not been initialized), then loops through all of the indices in the FCB array, initializing them to NULL to indicate that these blocks are free.

The interface **b\_read** returns an integer representing FILL IN . It has two parameters: the buffer we read from (char \*) and the count of things to read. We decided to use Jasmine's b\_read code from Assignment 5 and update as needed. After some error checking to ensure the file is valid,

The interface **b\_write** returns an integer that represents the file descriptor of the file that was written to. It is passed the buffer to write into, and the count of how many bytes to write.

The interface **b\_seek** can be passed a file descriptor and the offset of the file. It returns -1 if the file descriptor is incorrect.

The interface **b\_close** returns an integer which is -1 when an error occurs and zero otherwise, when passed the file descriptor. This function is responsible for closing a file, and is one of the simpler interfaces we implemented. After a quick check to make sure the file is valid and in use (-1 is returned if any of these are false), the allocated memory is freed. Then, the file control block is placed back into the unused pool of file control blocks. This is accomplished by setting the values of the FCB's block info and file info to null. The file system operates off of the assumption that free blocks are set to null, so this step (although simple) is integral to the system's functioning.

## **fsInit.c**

This file is where the initialization for our file system starts and it mirrors the aforementioned FAT system.

The first piece to build this filesystem so that it can be initialized is the structure of the directory entry. This structure describes a directory entry to have up to a 20 character name. An integer value to distinguish between its type; whether it is used, a directory, or a file (0-2 respectively). It also has a location (its address in storage) as well as its total size. Other metadata about a directory entry is that it has the dates/times for the last time it was modified, opened, and its creation.

The next step for the file system initialization is the initialization setup for the root directory. Following the guidelines the root directory is set up to have 64 initially allocated directories. For the root directory, the first entry has the . reference to itself like any other directory but the second entry, which is normally a reference to the previous entry is also self referencing because the root directory is the lowest. The blocks of the root directory are written and the starting block of where the root directory begins is returned.

What came next was the structure of the Volume Control Block. The VCB has a unique hex value for a signature that makes it unlikely to change to check whether the VCB has been formatted/written yet. Simply, it holds the number of blocks in the volume, the size of each block, the number of free blocks available, the number/position of the first free block available, as well as where the root directory starts.

After some helper functions for taking care of writing the VCB and read/writing blocks comes the allocation of free blocks. This starts with the data from the VCB which has the data of the next free block. From this, we have a function which checks if a block, in this case we check the blocks contiguously, until we get to one that is free and can therefore be allocated to be used and buffered. This continues until the requested number of blocks to be allocated is met or if there is no more free space

with which to allocate. Alternatively, we have a function to free allocated blocks which starts from a specified block and writes to the VCB that these blocks are free which will allow them to be usable without having to clear.

The next step in initializing the file system is being able to get the file info for the blocks within the FAT. A structure to hold the file block info (block size, total blocks for that file, and a reference to the block numbers on the table). After this we are able to add blocks to the FAT by allocating them and filling them with the data from the reference.

Lastly we can initialize the file system with how many blocks as well as their size. The first block of the filesystem is read/buffered to check the signature as mentioned it is reserved. The data for the Volume Control Block is set up as well as the File Allocation Table. The VCB writes to the first block and assigns the signature to ensure it isn't reformatted every boot. The FAT allocates the number of blocks to hold the table which are written to block 1 to  $((\text{numberOfBlocks} * 4) + (\text{blockSize} - 1)) / \text{blockSize} - 1$ . Next, the root directory is initialized according to the description earlier.

**mfs.h (implemented in fsInit.c)**

## Non File System Directory Functions

There are directory functions we've implemented that are not commands for use by the user in the shell but instead perform the actions that allow the use of shell commands. For example:

A helper function known as `parsePath` which takes a path reference as an argument and allows it to be broken down. It's broken down into a structure that holds both the path itself as well as the names of the locations. The reference to the names for example, holds all the directories until the last one which is either the end most directory of a file.

The file system's open directory function is made up of multiple helper functions. First our own open directory function is

called which checks the location the directory is currently in. By doing so we can ensure if it is either an absolute or relative path as well. Then, this path is saved and checked for whether or not it exists, which it is then loaded using another helper function that returns the file descriptor for the directory entry. It is then passed to the help function which finds it and returns. Ideally, all these helper functions would have their functionality included in the `parsePath` which would make the checks for things like it existing and return the information about it like its file descriptor.

Inversely, the `close directory` function frees all the memory used to perform the `open` function.

The other non command directory function is the `read directory` function. The approach for this function is that once the file descriptor for the directory is passed in there is a pointer of entries that is filled by copying the data.



# Issues and Resolutions

One technical issue was that when maintaining the FAT, we neglected to make sure that the next free block wrote to the VCB. This messed up the organization system and threw everything off, and was hard to debug. What was happening is when the `nextFreeBlock` was updated, it did not update outside of the block. When it exited outside of the program, it lost all the data. Hence, the record was not saved so it used the incorrect data. This issue was more easily resolved than found because we did not immediately recognize it as causing the problem. Once we realized, we resolved the issue by calling `writeVCB()` and it worked.

Another technical issue we had was that at first, the read buffer and write buffer were not shared, so when you make a copy of itself, there will be an error. For example, if we were to run `cp test1.h`, it would make a copy of itself. If they were not shared, the program would run into issues.

The biggest issue we had was finding times that we were all able to meet and work together. We all feel that working concurrently (in person or on call together) is the easiest way to stay on the same page and avoid confusion. Every person in our group is in multiple other classes with group projects (with due dates around the same time every milestone, of course!). And those groups for other classes were established long before we were assigned our File System groups—so we all had multiple team meetings a week already and the possible times to meet were slim. The fact that we all already had time and contribution responsibilities to other teams made finding a time for us to meet regularly very difficult, especially three times a week. Especially since this was the last group project to be set up, we all had already made commitments to most of the time slots in our week. And this was only with 3 people in our group. Though we wish we had a fourth person so that we could have gotten farther in this project, it seems like an extra person would've been an even greater hindrance on our ability to meet as a team.

# Screenshots

## Screenshots of Program Compilation

```
student@student-VirtualBox:~/Fall2022/CSC415/csc415-filesystem-arianna-y$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Fall2022/CSC415/csc415-filesystem-arianna-y$
```

## Hexdump Compilation

```
student@student-VirtualBox:~/CLionProjects/csc415-filesystem-arianna-y/Hexdump$ ./hexdump -h
USAGE: hexdump --file <filename> [--count num512ByteBlocks] [--start start512ByteBlock] [--help] [--version]
student@student-VirtualBox:~/CLionProjects/csc415-filesystem-arianna-y/Hexdump$ ./hexdump --file ../SampleVolume --count 1 --start 0
Dumping file ../SampleVolume, starting at block 0 for 1 block:
```

## Hexdump Execution

```
Dumping file ../SampleVolume, starting at block 0 for 1 block:

000000: 43 53 43 2D 34 31 35 20 2D 20 4F 70 65 72 61 74 | CSC-415 - Operat
000010: 69 6E 67 20 53 79 73 74 65 6D 73 20 46 69 6C 65 | ing Systems File
000020: 20 53 79 73 74 65 6D 20 50 61 72 74 69 74 69 6F | System Partitio
000030: 6E 20 48 65 61 64 65 72 0A 0A 00 00 00 00 00 00 | n Header.....
000040: 42 20 74 72 65 62 6F 52 00 96 98 00 00 00 00 00 | B treboR.00.....
000050: 00 02 00 00 00 00 00 00 4B 4C 00 00 00 00 00 00 | .....KL.....
000060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000070: 52 6F 62 65 72 74 20 42 55 6E 74 69 74 6C 65 64 | Robert BUntitled
000080: 0A 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0000F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0001F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
```

This is a hex dump representing the blocks in the file system.  
The above hex dump represents the first (or zeroth) block which

is the professor's reserved space (which you can tell from the Robert BUntitled and the fact that he told us the blocks were organized in this way). Most of this block is empty, only the first fourth or so has data in it.

```
student@student-VirtualBox:~/CLionProjects/csc415-filessystem-arianna-y/Hexdump$
./hexdump --file ../SampleVolume --start 1 --count 1
Dumping file ../SampleVolume, starting at block 1 for 1 block:

000200: 45 54 41 4E 00 00 00 00 4B 4C 00 00 00 02 00 00 | ETAN....KL.....
000210: 01 00 00 00 B1 4B 00 00 CD 00 00 00 99 00 00 00 | ....0K...0...0...
000220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

000300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
```

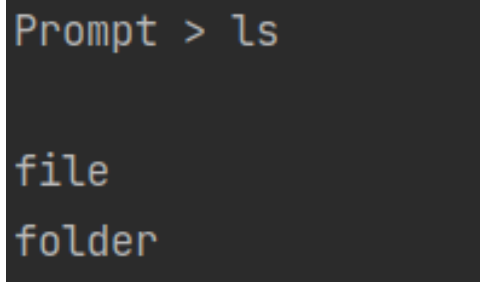
When you run the hex dump with the run option --start 1 instead of --start 0, you can see our volume control block.

The variables stores in the VCB are the signature, numBlocks, blockSize, numLBAPerBlock, freeBlockCount, nextFreeBlock, and rootDirStart (in order). The hex dump shows that these fields are filled; the formatting signature (sig) is stored in the first 8 bytes since it is a long. We can tell that it is being

correctly stored because 45 54 41 4E is the same as what the buffer is set to: **fsVCB.sig = 0x4E415445** (fsInit.c line 374).

## Execution of program

**ls** - Lists the file in a directory

A terminal window with a dark background. The prompt 'Prompt >' is followed by the command 'ls'. Below the command, the output 'file' and 'folder' is displayed on two separate lines.

```
Prompt > ls  
  
file  
folder
```

The **ls** command prints all files/directories contained in the current directory.

## **cp - Copies a file - source [dest]**

```
Prompt > cat destinationFile
Prompt > cp file destinationFile
Prompt > cat destinationFile
# CSC415 Group Term Assignment - File System

This is a GROUP assignment written in C. Only one person on the team needs to submit the project.

Your team have been designing components of a file system. You have defined the goals and design,
defined the directory entry structure, the volume structure and the free space. Now it is time to implement
your file system.

While each of you can have your own github, only one is what you use for the project to be turned
in. Make sure to list that one on the writeups.

The project is in three phases. The first phase is the "formatting" the volume. This is further
described in the steps for phase one and the phase one assignment.

The second phase is the implementation of the directory based functions. See Phase two assignment
.
```

When we first cat destinationFile, it is empty. Then we copy file (which has the README of this assignment in it) into destinationFile. The second call of cat shows that the README has been successfully copied from file to destinationFile.

## mv - Moves a file - source dest

```
student@student-VirtualBox:~/Fall2022/CSC415/csc415-filesystem-arianna-y$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls

arianna
test3
test1.h
test2.h
test3.h
newfile1.c
fileTest1
Prompt > mv test3 newfile3.h
Prompt > ls

arianna
newfile3.h
test1.h
test2.h
test3.h
newfile1.c
fileTest1
Prompt > 
```

## md - Make a new directory

```
Prompt > ls

file
folder
Prompt > md folder2
Prompt > ls

file
folder
folder2
```

## **rm - Removes a file or directory**

```
Prompt > ls  
  
file  
folder  
Prompt > rm file  
Prompt > ls  
  
folder
```

## **touch - creates a file**

```
Prompt > ls  
  
folder  
Prompt > touch file  
Prompt > ls  
  
file  
folder
```

## **cat - displays contents of a file**

When we first cat file, it is empty. Then we copy README.md (of this assignment) from my Linux vm into the test file. The second call of cat shows that the README has been successfully copied from README.md to file. Only the first few lines are included for brevity's sake.

```
Prompt > cat destinationFile  
# CSC415 Group Term Assignment - File System  
  
This is a GROUP assignment written in C. Only one person on the team needs to submit the project.  
  
Your team have been designing components of a file system. You have defined the goals and designe  
d the directory entry structure, the volume structure and the free space. Now it is time to imple  
ment your file system.  
  
While each of you can have your own github, only one is what you use for the project to be turned  
in. Make sure to list that one on the writeups.
```



## cp2l - Copies a file from the test file system to the linux file system

```
student@student-VirtualBox:~/Fall2022/CSC415/csc415-filesystem-arianna-y$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
./fsshell SampleVolume 100000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > ls -l
D          4096  arianna
D          4096  test3
-          3467  test1.h
-          3467  test2.h
-          3467  test3.h
Prompt > cp2fs b_io.c newfile1.c
lPrompt > ls -l
D          4096  arianna
D          4096  test3
-          3467  test1.h
-          3467  test2.h
-          3467  test3.h
-         18468  newfile1.c
Prompt > cp2l newfile1.c
Prompt > cat newfile1.c
/*****
* Class:  CSC-415-01 Fall 2021
* Names:  Jasmine Stapleton-Hart
*         Arianna Yuan
*         Nathaniel Miller
* Student IDs:
*         921356953
*         920898911
*         922024360
* GitHub Name: arianna-y
*
* Group Name: Vile System
* Project: Basic File System
*
* File: b_io.c
*
* Description: Basic File System - Key File I/O Operations
*****/
```

## **cp2fs - Copies a file from the Linux file system to the test file system**

When we first cat file, it is empty. Then we copy README.md (of this assignment) from my Linux vm into the test file. The second call of cat shows that the README has been successfully copied from README.md to file.

```
Prompt > cat file
Prompt > cp2fs README.md file
Prompt > cat file
# CSC415 Group Term Assignment - File System

This is a GROUP assignment written in C. Only one person on the team needs to submit the project

Your team have been designing components of a file system. You have defined the goals and design
d the directory entry structure, the volume structure and the free space. Now it is time to impl
ment your file system.

While each of you can have your own github, only one is what you use for the project to be turned
in. Make sure to list that one on the writeups.

The project is in three phases. The first phase is the "formatting" the volume. This is further
described in the steps for phase one and the phase one assignment.

The second phase is the implementation of the directory based functions. See Phase two assignmen
.

The final phase is the implementation of the file operations.

To help I have written the low level LBA based read and write. The routines are in fsLow.o, the
ecessary header for you to include file is fsLow.h. You do NOT need to understand the code in fs
ow, but you do need to understand the header file and the functions. There are 2 key functions:
```

## cd - Changes directory

```
Prompt > ls  
  
file  
folder  
Prompt > cd folder  
Prompt > ls  
  
anotherfolder
```

The `cd [directory]` command changes the current directory. The first `ls` shown is in the root directory, then the directory is changed to `./folder` for the second `ls`.

## pwd

```
Prompt > pwd  
/folder  
Prompt > cd ..  
Prompt > pwd  
/
```

The `pwd` command prints the name of the current working directory. Here are two examples: one `pwd` called inside a secondary folder (`/folder`) and one called in the root directory (`/`).

## history - Prints out the history

```
Prompt > ls

folder
Prompt > cd folder
Prompt > md innerfolder
Prompt > ls

anotherfolder
innerfolder
Prompt > history
ls
cd folder
md innerfolder
ls
history
```

The commands `ls`, `cd folder`, `md innerfolder`, and `ls` are called. They are printed in order when `history` is called.

## help - Prints out help

```
Initializing File System with 19531 blocks with a block size of 512
Prompt > h
h is not a regonized command.
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
```

Help prints a list of shell commands for the user. This help list is also printed when an invalid/unknown command is entered.

## Resources & References

Provided code skeleton: [https://classroom.github.com/a/l9EJkgK\\_](https://classroom.github.com/a/l9EJkgK_)

Microsoft Extensible Firmware Initiative FAT32 File System  
Specification FAT: General Overview of On-Disk Format Version  
1.03, December 6, 2000 Microsoft Corporation