CSC415 Group Term Assignment - File System

Team X:
Jonathan Curimao, jonathancurimao, 921855139
Jonathan Ho, deserted, 923536806
Yashwardhan Rathore, yash03rathore, 921759459
Juan Segura Rico, velojr, 921725126

Github repository:
https://github.com/CSC415-2024-Fall/csc415-filesystem-jonathancurimao.git

General Description:

This file system project focuses on designing and implementing a custom file storage solution in C within our VM. The project is divided into three phases, guiding the development from volume formatting, to directory and file operations. Some key features include managing free space allocation, directory structures, and file metadata to ensure persistence and efficiency. Additionally, the project requires implementing advanced buffering techniques and using low-level file operations like LBA-based read/write functions.

General approach:

The approach to this file system project involves a phased implementation so we can work our way through, beginning with the creation and formatting of our volume to establish a storage base with appropriate block sizes. The first phase focuses on setting up essential data structures for tracking free space and initializing a root directory. The second phase involves implementing directory operations, allowing the creation, deletion, storing metadata and managing logical block addresses. In the final phase, we will develop file operations such as opening, reading, writing, seeking, and closing files, ensuring that buffering and free space management are handled efficiently. Throughout, we will test functionality with our shell interface, evaluating each layer and writing all steps and problems to see what's working and what isn't.

Milestone Approaches:

Milestone 1

Our approach for milestone 1 would be distributing the workload for planning and implementing the essential structures needed, which would be our volume control block, the free space map, and our directory entry. For our free space map, we decided on a bitmap, so we would need to implement the necessary functionalities present in typical bitmap structures. As for what we would implement in our volume control block to correspond with our bitmap, we would need to include the magic number to check if the volume is initialized, the block size, the total number of blocks, total number of free blocks, and the start of the free space itself. Our implementation of the directory entry would have included the attributes present in a file, such as the file name and the block location of the file.

Milestone 2

For milestone 2, we would have to methodically split the workload for each function, with our plan involving having each member assigned to at least 2 functions, with the remaining ones being left on the table for anyone to do. To avoid constant merge conflicts, we would create separate .c files to do our implementations for each function, which would all be placed in a .c file after everything is completed. We would need to make sure to maintain constant communication with each other due to how each function synergizes with one another, which means we would have to make sure our approach for one of the functions can work well with another. This also involves multiple test cases for thorough debugging and making sure every function works. We would also need to make sure to optimally implement our parsePath function, which would be essential to parsing the file system path and providing information about the parsed path. To do so, we would need to find a way to implement a check that sees if the root directory has been loaded, and if it isn't, we would need to call the loadRoot() function. We would also need parsePath to determine if the file system path starts with a '/', and if it does, we would have to set the starting point to the root directory. Otherwise, the file system would just use the current working directory. For how we would get into the parsing, we would need a loop to iterate through each token in the path, which is where strtok() may come in. In the event that the entire path is successfully parsed, the pathInfo struct would be filled out.

Milestone 3

The main goal of milestone 3 is to optimally implement and split the work for all b_io functions with the provided b_fcb struct. We would need to decide on what variables to use in the b_fcb struct, and how to use them for each of the functions. The variables would need to hold the file buffer, the current position in the buffer, the amount of valid bytes in the buffer, and the total amount of bytes read. Since we are also dealing with reading blocks, we would also include the block location, the current block offset, and the number of blocks that are allocated. Since there are five b_io functions that we would need to implement, we would all need to be assigned to one of our preferences, with one of the functions being excluded for implementation by any of the group members.

Description of Structures:

VCB

The Volume Control Block (VCB) is the main structure that stores key details about our file system. It includes a unique number called magicNumber to check if the system is already set up. The blockSize field defines the size of each block in bytes, and numberOfBlocks gives the total count of blocks in the system. To track free space we have used the freeSpace field which is where the free space map begins and helps us to manage storage efficiently. Then, rootDirStart points to the starting location of the root directory which allows us to locate and work with files from the root level of the file system. All of these fields allow the VCB to keep track of the storage layout, helping us initialize, access, and manage the file system.

FreeSpace

The free space structure utilizes a bitmap to quickly search for free blocks. This bitmap is initialized in a function that mallocs the memory needed for the bits, and it would initialize all bits to 0, since all blocks would be free initially. The freeSpace would also be accompanied by the helper functions allocateBlock() and freeBlock(), which would manage the blocks being used. Another helper function that is implemented in this structure is getBlockState, which would essentially check if a block is at a given index or not. The freeSpace would make use of our global volume control block structure for its functions.

Directory System

Our directory entry structure itself is fairly simple. It contains very simple fields such as a character array for the filename, of which it can hold 100 characters, a file location integer which stores the block that the file is held on, the file size, time created and modified, the file number, and booleans for whether the file itself is a directory or is used or not. Our create directory function, when given a number of entries, will loop through all entries and initialize these values to a "free" state, in which these entries are unused. The first two entries in the directory are always reserved for "." and "..".

Function Approaches:

B_open: This function is responsible for initializing the file within our system. It begins by checking whether the system is initialized, calling an initialization routine (b_init) if necessary. The function then retrieves a file control block (FCB) index using b_getFCB, which provides a unique identifier for the file being opened. It ensures that the associated buffer for the FCB is allocated, dynamically allocating memory if the buffer is uninitialized. Once the memory is set up, the FCB fields, such as index and buflen, are initialized to default values to prepare the file. If any error occurs during the setup process (ex: memory allocation failure), the function returns an error code (-1); otherwise, it returns the file descriptor, signaling that the file is ready to open.

B_close: This function is used to close what's associated with an open file and reset its corresponding file control block (FCB). It first validates the provided file descriptor (fd) to ensure it is within the valid range and points to an active FCB with an allocated buffer. If the validation fails, the function returns an error code (-1). Otherwise, it frees the memory allocated for the buffer and resets the buffer pointer to NULL. Then, it clears other fields in the FCB, restoring them to their default values. Finally, the function returns success (0), and our file is closed.

B_seek: The b_seek function is responsible for repositioning the file offset within an open file, allowing the user to move to a specific position for reading or writing. It begins by validating the file descriptor to ensure it references a valid open file. The function then calculates the new position based on the offset and whence parameters. The whence parameter determines the reference point for the offset which is from the beginning (SEEK_SET), the current position (SEEK_CUR), or the end of the file (SEEK_END). The calculated position is checked to ensure it stays within the file valid bounds. If the position is invalid, the function returns an error code

(-1). Otherwise, it updates the current file position in the file control block (FCB) and returns the new position which shows success.

B_read: The b_read function's main responsibility is to use the provided fcb struct to read the data from an open file. To make sure the open file is being read properly, the number of bytes is calculated by taking the minimum between the requested count and the available bytes in the buffer. After that, memcpy() is used to copy the data from the FCB buffer to the user's buffer. The FCB state gets updated, and then we return the number of total bytes read.

B_write: The b_write function needs to essentially function as the inverse of the b_read function. It should take in a user's buffer and write to an output file's buffer. To do this, we need some kind of predictive system in which we allocate blocks based on the user's needs. For example, for the first b_write call, let's allocate 100 blocks for them. On successive calls, we will double this number every time b_write is called, as they have a history of using large amounts of blocks. In order for this predictive system to function, we need to calculate exactly how many additional blocks the user is going to write, and compare that to the number of already existing and allocated blocks. If the number of existing and allocated blocks is less than the number of total predicted blocks, then of course we need to allocate more for the user.

Getcwd.c: The getcwd function retrieves the current working directory in the file system by interacting with a workingDirPtr, a pointer to an array of directory entries. It first iterates through the entries in the working directory to display the names of all non-free entries for debugging purposes. The function then checks if the current directory is the root directory, returning a pathname of "/" if true. Otherwise, it copies the name of the current working directory (stored in the global variable workingDirName) into the pathname buffer, ensuring it does not exceed the specified size and is null-terminated.

Setcwd.c: The setcwd function changes the current working directory of the file system by validating and processing the provided pathname. It starts by duplicating the input string to avoid modifying it directly and uses parsePath to check the path's validity; if invalid, it outputs an error message and returns -1. If valid, it updates the workingDirPtr to the new directory, freeing any previously allocated memory if necessary. For absolute paths, it sets workingDirName directly to the given pathname. For relative paths, it combines the current directory path with the new path, tokenizes the components, and handles navigation symbols like "." and ".." to resolve the final directory structure. The reconstructed path is stored in workingDirName, with all temporary memory safely freed to prevent any leaks.

Fs_mkdir: The fs_mkdir function is responsible for creating a new directory in the file system. It starts by checking if the input path is valid and if the system is ready to handle the creation. Then, it uses the parsePath function to locate the parent directory and verify that the directory being created does not already exist. If the directory already exists or the path is invalid then the function displays an error message and stops. Otherwise, it uses findUnusedDE to locate an available slot in the parent directory to store the new directory information. It then creates the directory using the createDir function and sets its properties like size, location, and permissions, and updates the parent directory with this new information. Then, it writes the updated parent directory back to the disk to save the changes and cleans up any temporary memory and confirms that the directory was successfully created.

Fs_stat: The fs_stat function retrieves detailed information about a specific file or directory. It begins by duplicating the input path to avoid modifying the original string. The parsePath function is then used to verify the path and locate the file or directory. If the path is invalid or does not exist then the function outputs an error message and stops. If the path is valid then the function identifies the file or directory entry in its parent directory. It then fills the fs_stat structure with details that includes the file size, the size of each block, the total number of blocks used, and timestamps for when the file or directory was last created, modified, or accessed. After collecting this information, the function cleans up any temporary memory it used and returns success which allows the system to use the gathered details as needed.

Fs_isDir: This function determines whether a given path corresponds to a directory in our file system. It duplicates the input path string using strdup to ensure the original path remains unchanged. A pathInfo structure is then initialized to store parsed information about the path, including its parent directory and the last element's name. The function calls parsePath to process the path and populate the pathInfo structure. If the parsing fails (indicated by a return value of -1 or an invalid parent index), an error message is displayed, memory allocated for pathInfo and the path copy is freed, and the function returns -1 to indicate an invalid path. If the path parsing succeeds, the function checks if the parent directory entry at the specified index is marked as a directory (isDir == true). If it is, the function frees the allocated resources and returns 1, indicating the path is a directory. Otherwise, it frees resources and returns 0, showing the path is not a directory.

Fs_isFile: This function is used to determine if a given path corresponds to a file in our system. It first duplicates the input file path to avoid modifying the original string, to ensure safe manipulation. It then relies on a helper function, isDir, to identify whether the path is a directory. IsDir returns an error (-1), the function outputs an error message, cleans up the allocated memory, and exits with an error code. If isDir confirms the path is a directory (returning 1), the function concludes the input is not a file and returns 0. Finally, if isDir indicates the path is not a directory (returning 0), isFile concludes that it is a file and returns 1.

Fs_rmdir: This function is designed to remove directories as needed. It would have to check if the specified directory exists, and if it is empty. This would be done through a while loop that iterates through all the data in the directory. Any time other than "." or ".." breaks the loop. If the directory is indeed empty, a copy to the pathname would be created and the directory entry is then removed. If a directory is not empty and there is an attempt to remove it, an error message will occur.

Fs_closedir: This function closes the directory. When a directory is closed, it would be set to NULL. closedir works closely with rmdir by checking the isEmpty flag. If the directory is found not to be empty, the function would return an error which means that there are contents in the directory.

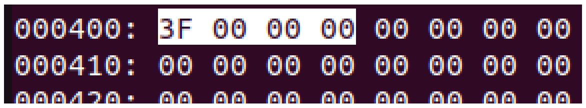
Fs_opendir: This function is designed to open a directory in the file system, with the fdDir struct representing the opened directory. The path of the directory would be parsed with the help of our parsePath function, with special cases such as empty paths, the root directory, and invalid paths being checked. If there is a directory that the file system wants to find, the target directory is reached based on the parsed path information. If found, it would be assigned to the currDir struct. The ".'', or the directory entry, is then created and stored.

Fs_readdir: The purpose of this function is to read directory entries from an opened directory. The function would return a pointer to the next valid directory entry. When going through the directory entries, we would have a loop go through them starting from a current position until it finds a valid entry or it reaches the end of the directory. When a valid entry is found, the filename is copied and its filetype is set. After all entries are processed, NULL would be returned to indicate the end of the directory.

Fs_delete: The main purpose of this function is to delete files from the file system. It would first check if the provided filename correlates with a file that is not a directory. This is done by calling fs_isFile, and if it does correlate, a copy of the filename is created using strdup() to ensure that the original filename is unchanged during deletion, which would then be freed of memory. If it fails, an error message is shown, and the function finishes with an error code.

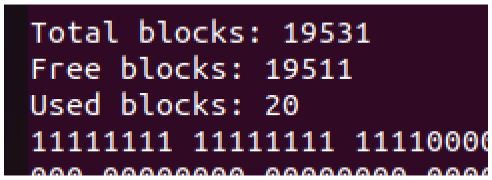
Issues and Resolutions:

The Hexdump free space map did not match the expected value of how many blocks were allocated. Our initial hexdump showed a value of 3F. Translating this to binary yielded a value of “00111111”, meaning only the first 6 blocks marked as being occupied.



```
000400: 3F 00 00 00 00 00 00 00
000410: 00 00 00 00 00 00 00 00
```

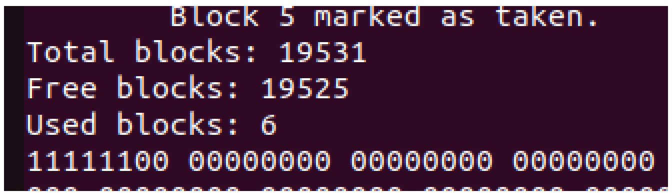
Our custom print bitmap function shows the expected output:



```
Total blocks: 19531
Free blocks: 19511
Used blocks: 20
11111111 11111111 11110000
```

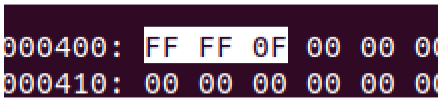
It should display 20 blocks marked, not only 6. We have 6 blocks after allocating our free space map

Upon closer inspection, we were calling LBAwrite too early. Calling our custom print function at the point of writing to disk shows that I was writing our free space map before creating and allocating the root directory. The solution is to either re-call LBAwrite at the end, or call LBAwrite only once AFTER the root directory is created.



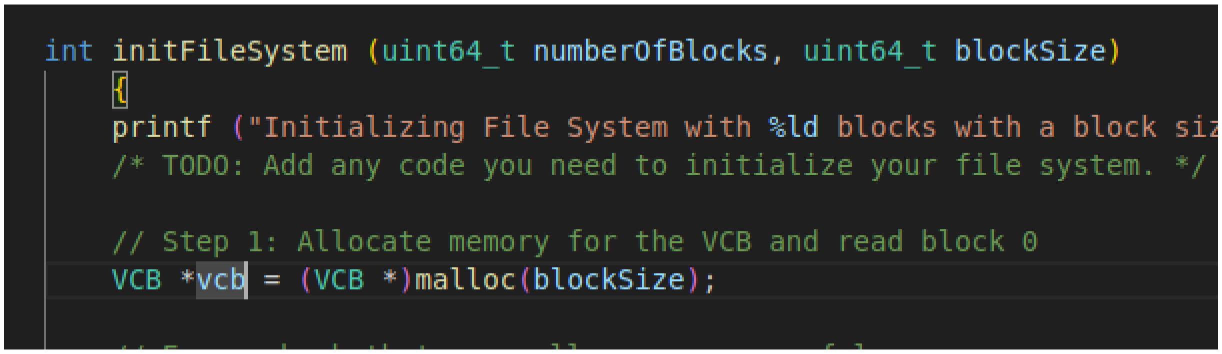
```
        Block 5 marked as taken.
Total blocks: 19531
Free blocks: 19525
Used blocks: 6
11111100 00000000 00000000 00000000
```

Correcting this simple mistake leads to the expected output: “0FFFFF”, which translates to 20 bits marked in binary.

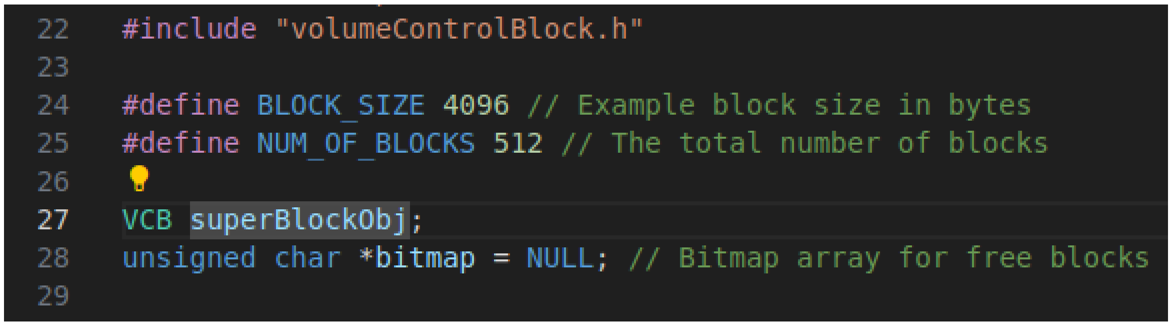


```
000400: FF FF 0F 00 00 0
000410: 00 00 00 00 00 0
```

Refactoring code: double VCB declaration

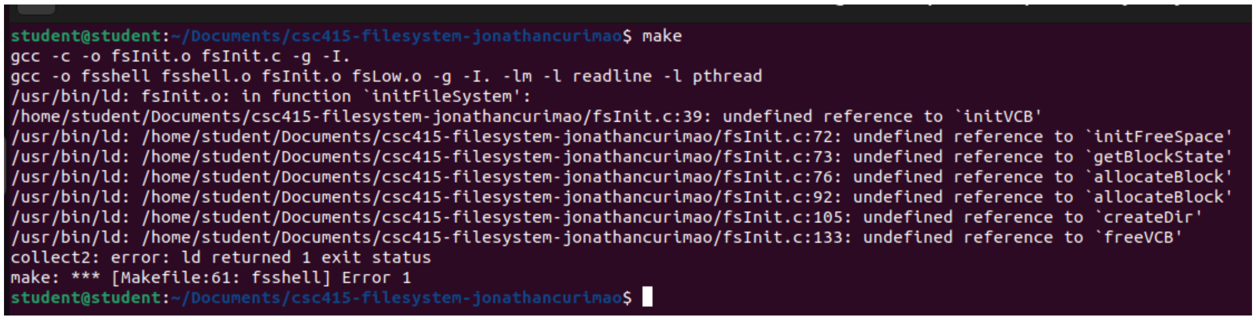


```
int initFileSystem (uint64_t numberOfBlocks, uint64_t blockSize)
    {
    printf ("Initializing File System with %ld blocks with a block siz
    /* TODO: Add any code you need to initialize your file system. */

    // Step 1: Allocate memory for the VCB and read block 0
    VCB *vcb = (VCB *)malloc(blockSize);
```



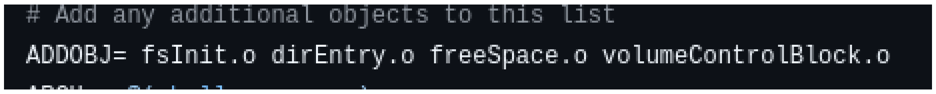
```
#include "volumeControlBlock.h"

#define BLOCK_SIZE 4096 // Example block size in bytes
#define NUM_OF_BLOCKS 512 // The total number of blocks

VCB superBlockObj;
unsigned char *bitmap = NULL; // Bitmap array for free blocks

```

We had two separate VCB objects that we were updating independently of each other. This didn’t cause any problems yet, but had potential to make things harder in the project later on. There were two solutions to this problem: either pass in the original VCB or make a singular VCB object global. Because our functions already assumed the VCB was global, I did not want to have to refactor all of our existing functions, so I opted for the latter.



```
student@student:~/Documents/csc415-filesystem-jonathancurimao$ make
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
/usr/bin/ld: fsInit.o: in function `initFileSystem':
/home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:39: undefined reference to `initVCB'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:72: undefined reference to `initFreeSpace'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:73: undefined reference to `getBlockState'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:76: undefined reference to `allocateBlock'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:92: undefined reference to `allocateBlock'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:105: undefined reference to `createDir'
/usr/bin/ld: /home/student/Documents/csc415-filesystem-jonathancurimao/fsInit.c:133: undefined reference to `freeVCB'
collect2: error: ld returned 1 exit status
make: *** [Makefile:61: fsshell] Error 1
student@student:~/Documents/csc415-filesystem-jonathancurimao$
```

We were getting compilation errors from making custom .h and .c files. The compiler would complain about references to undefined functions. After doing some research, it turns out that I had to edit the Makefile to tell the compiler to compile my external files, like so:



```
# Add any additional objects to this list
ADDOBJ= fsInit.o dirEntry.o freeSpace.o volumeControlBlock.o
```

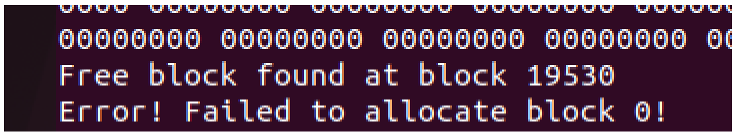
This solved the compiler issues.

parsePath function: Need to keep the root directory loaded in RAM at all times. Need to adjust the root directory to be a global variable that anyone can access. To accomplish this, write a new function: loadVCB, which simply reads our VCB off disk and into memory. After that, we can then access where the root directory starts and block size.

createDir bug: Initially, I hardcoded createDir to only create the root directory. This meant that I knew where the root directory would start, and so I hardcoded the location of the root to specifically that. However, ideally we want to be able to use createDir anywhere. So I changed createDir to call our allocateBlock function, which finds the nearest free block. Directories can now be created anywhere.

allocateBlock function rewrite: allocateBlock will find the nearest free block and allocate it. However, if we want to allocate a certain range of blocks, we need to loop allocateBlock until we have the number of desired blocks we would like. This can cause problems to arise with our file system, namely because ours uses a contiguous model. If we have a file that spans 4 blocks and we have a bitmap that looks like this: 11001100, then allocateBlock will allocate us 4 dis-contiguous blocks. We have no way of knowing whether our files are fragmented or not, so we need to rewrite our allocateBlock function to fetch us a range of contiguous blocks.

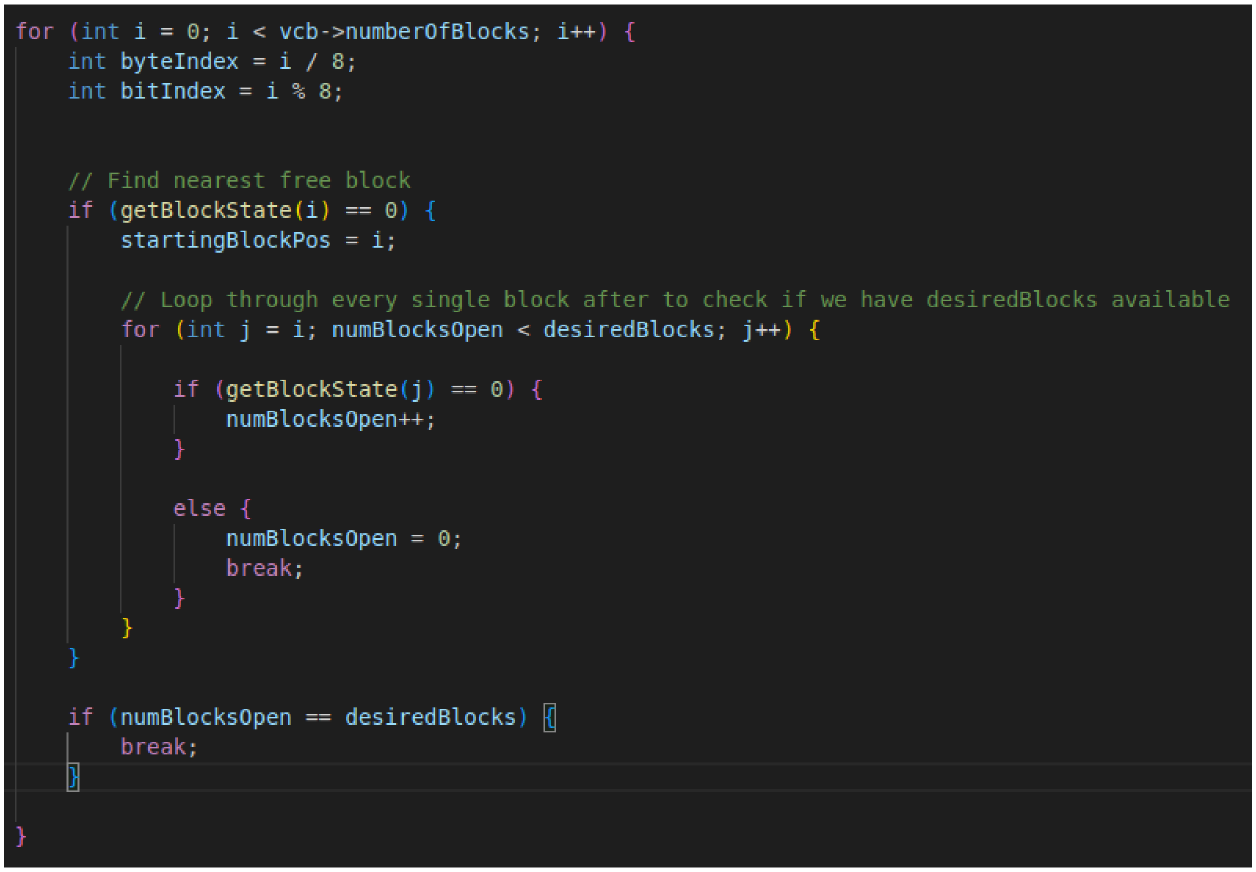
In writing a new allocateBlock function with this in mind, I had issues where the newly created function would try to fetch a block at the very end of our volume:



```
00000000 00000000 00000000 00000000 00
Free block found at block 19530
Error! Failed to allocate block 0!
```

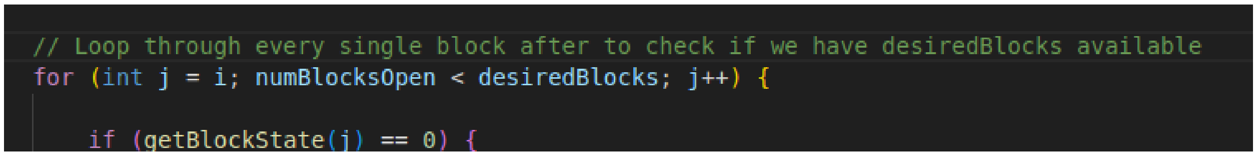
This is me trying to initialize our VCB, which should normally live at block 0. However it is finding a block at 19530!

I needed to add a special condition to break out of our loop for finding free blocks. Before, I would loop through every single block to find a free block, and get the position of the nearest closest block. However, I didn't have a condition to stop searching for free blocks once we had a sufficient number of contiguous free blocks. So I would just keep looping endlessly through the array, despite already having enough free blocks. The solution was to add a break condition to the loop, where we check if we have enough blocks like so:



```
for (int i = 0; i < vcb->numberOfBlocks; i++) {
    int byteIndex = i / 8;
    int bitIndex = i % 8;


    // Find nearest free block
    if (getBlockState(i) == 0) {
        startingBlockPos = i;

        // Loop through every single block after to check if we have desiredBlocks available
        for (int j = i; numBlocksOpen < desiredBlocks; j++) {

            if (getBlockState(j) == 0) {
                numBlocksOpen++;
            }

            else {
                numBlocksOpen = 0;
                break;
            }
        }
    }

    if (numBlocksOpen == desiredBlocks) {
        break;
    }

}
```

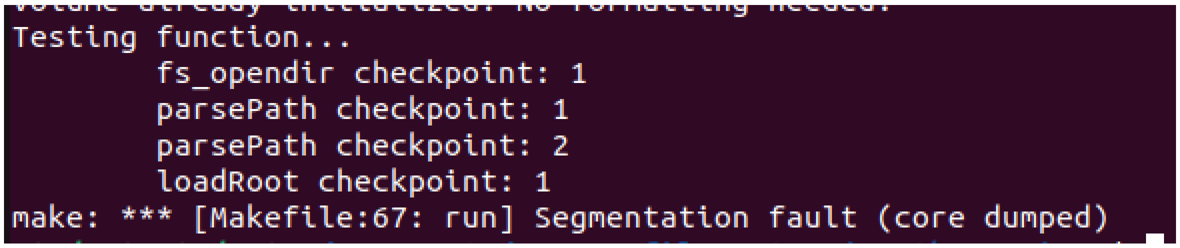
Essentially, if we've found a sufficient number of free blocks, then we can break out of looping through every single block. Another bug that popped up with our new allocateBlocks function is that I was ending up allocating 1 block more than I needed, which was because of this line:



```
// Loop through every single block after to check if we have desiredBlocks available
for (int j = i; numBlocksOpen < desiredBlocks; j++) {

    if (getBlockState(j) == 0) {
```

Previously, it used to be "numBlocksOpen <= desiredBlocks". This was problematic because if we've found 4 numBlocksOpen and our desiredBlocks is 4, then we're going to loop to find 5 numBlocksOpen. The solution was to change the equality to "<", which made numBlocksOpen match desiredBlocks.

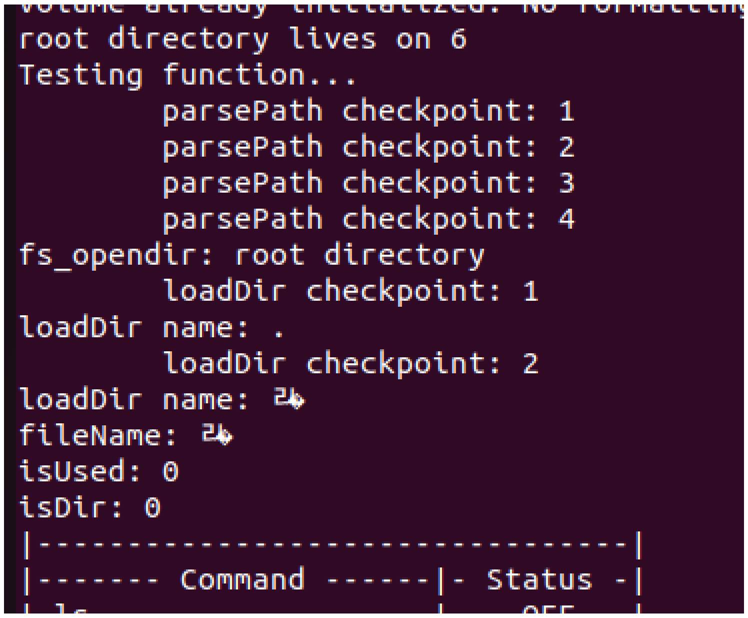
Testing fs_opendir on root: Getting crashing with segfaults



```
Testing function...
        fs_opendir checkpoint: 1
        parsePath checkpoint: 1
        parsePath checkpoint: 2
        loadRoot checkpoint: 1
make: *** [Makefile:67: run] Segmentation fault (core dumped)
```

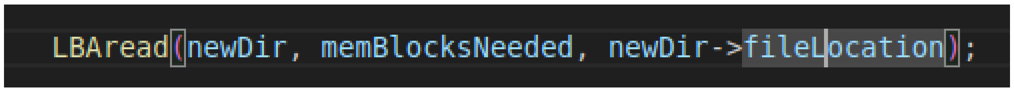
Added checkpoints to the chain of functions called by fs_opendir. Leads to loadRoot segfaulting. Our loadRoot function was trying to ask the VCB for the block size so it knows the appropriate amount of bytes to allocate in memory. The problem was that, on "fresh boots", or when the hard drive is first created, our VCB gets loaded into RAM. However, on subsequent boots, the VCB never ends up being loaded. So our loadRoot function was trying to access memory that was not available.

Our loadDir function also had bugs in it. When trying to load the root directory using loadDir, I got a bunch of junk info when printing debug info on the root directory. fileName, isUsed, and isDir were all completely wrong:



```
root directory lives on 6
Testing function...
        parsePath checkpoint: 1
        parsePath checkpoint: 2
        parsePath checkpoint: 3
        parsePath checkpoint: 4
fs_opendir: root directory
        loadDir checkpoint: 1
loadDir name: .
        loadDir checkpoint: 2
loadDir name: 2�
fileName: 2�
isUsed: 0
isDir: 0
|----------------------------------|
|------- Command ------|- Status -|
```

The problem was this LBAread line:



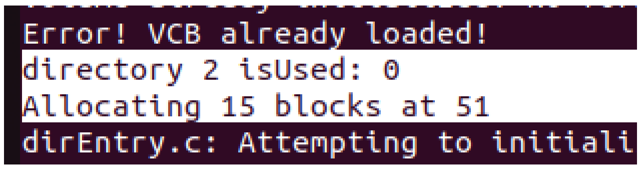
```
LBAread(newDir, memBlocksNeeded, newDir->fileLocation);
```

We were telling LBAread to read from a completely wrong location, causing random junk to be read into loadDir.

Fs_opendir crashes on loading directories not prefixed with “/”. For example, fs_opendir(“/”) and fs_opendir(“/asddg”) load fine, but fs_opendir(“test”) crashes. The problem is that, within parsePath, I was setting currWorkDir to NULL for placeholder purposes. In order to fix this crash, we had to correctly set currWorkDir to root for the moment.

Fs_mkdir has two errors: it is able to create duplicate entries, and it is also able to create entries in slots that are already occupied.

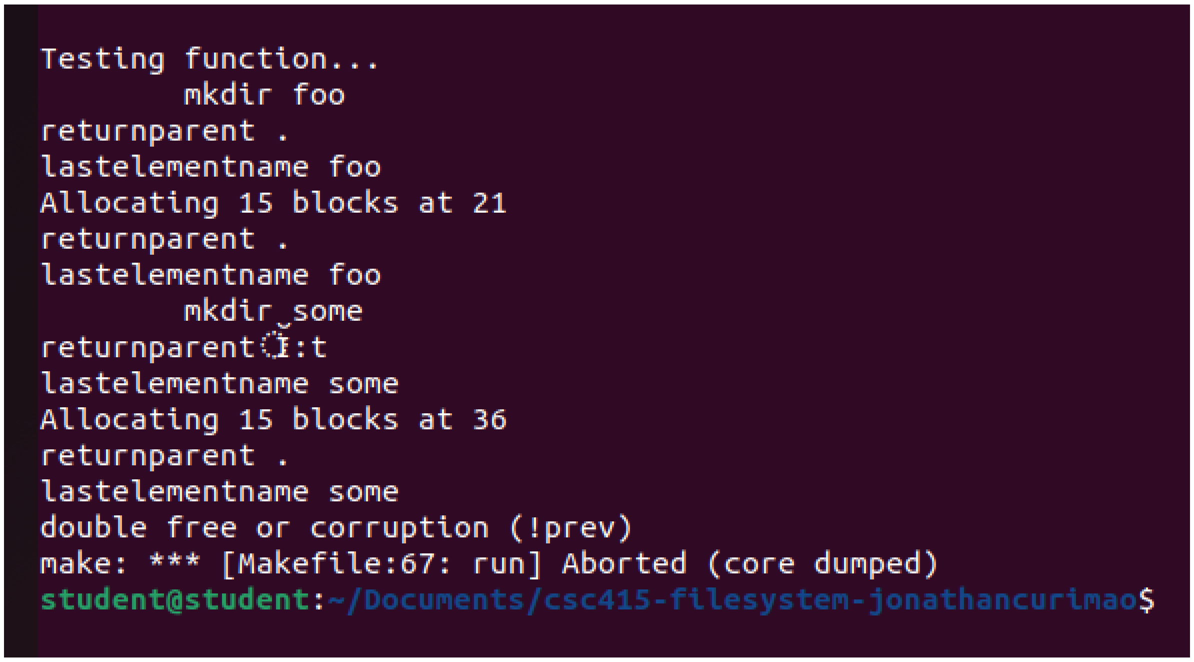
The solution was to rewrite and add some debug prints shows that, on fresh boots, occupied directory slots are being shown as unused:



```
Error! VCB already loaded!
directory 2 isUsed: 0
Allocating 15 blocks at 51
dirEntry.c: Attempting to initiali
```

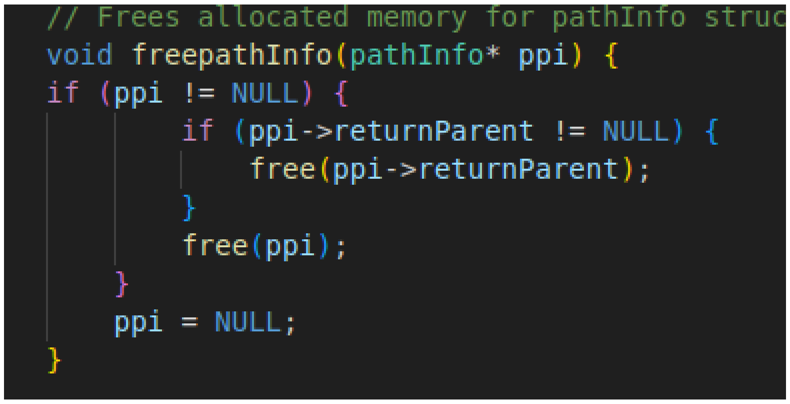
The expected output is that it should show directory 2 as “isUsed = 1”. The problem was with my underlying parsePath function. Previously, even if our function had reached the end of the string, if the last directory did not exist, parsePath would return a -1. Our fs_mkdir function checked for if parsePath returned -1 or not, and if it was -1, it would create a directory. We have corrected parsePath such that, regardless of if the last directory in the string exists or not, it will return 0 either way.

However, after that, we got another error:



```
Testing function...
        mkdir foo
returnparent .
lastelementname foo
Allocating 15 blocks at 21
returnparent .
lastelementname foo
        mkdir some
returnparent :t
lastelementname some
Allocating 15 blocks at 36
returnparent .
lastelementname some
double free or corruption (!prev)
make: *** [Makefile:67: run] Aborted (core dumped)
student@student:~/Documents/csc415-filesystem-jonathancurimao$
```

We were double freeing up pointers somewhere. In our mkdir function, we were calling a helper function to free a helper struct like so:

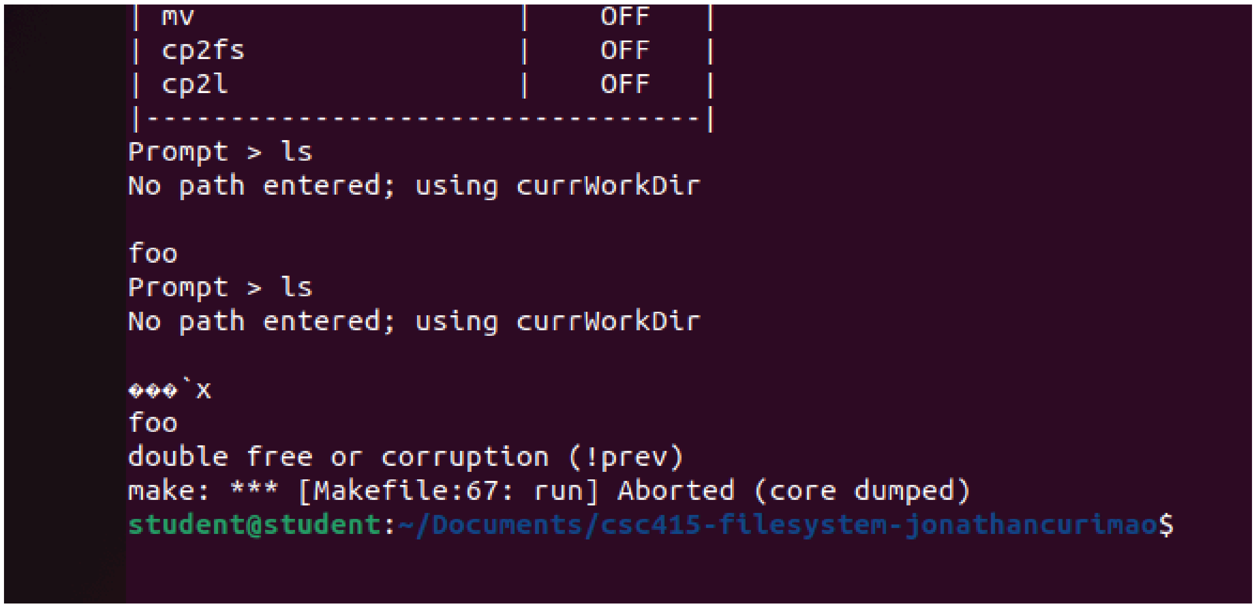


```
// Frees allocated memory for pathInfo struc
void freepathInfo(pathInfo* ppi) {
if (ppi != NULL) {
        if (ppi->returnParent != NULL) {
            free(ppi->returnParent);
        }
        free(ppi);
    }
    ppi = NULL;
}
```

Essentially, this frees the malloc'd returnParent directory entry provided by parsePath. However, our returnParent was pointing to our root directory loaded in RAM.

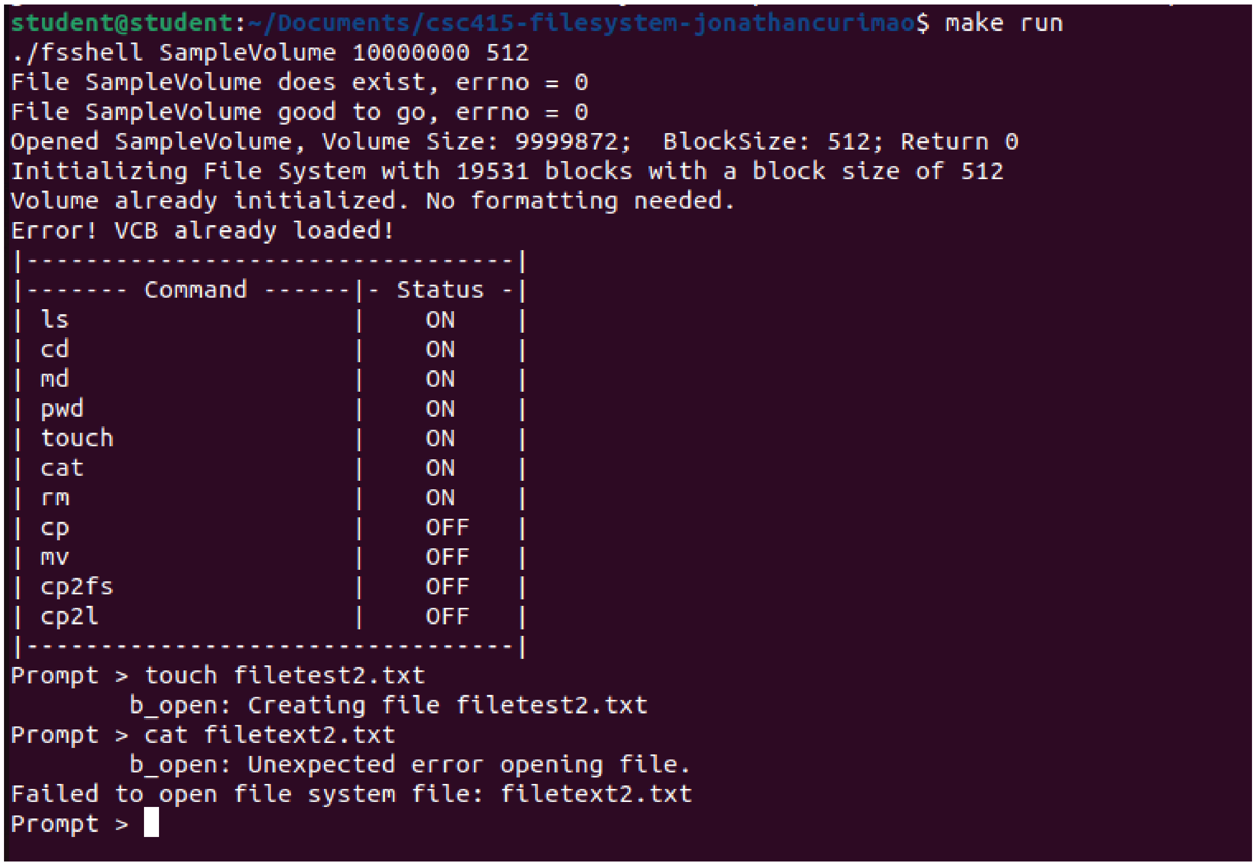
When trying to use the ls function in the shell, I would get no output. After some debugging, I discovered that it was because calling ls with an empty path automatically appends a "D" to the path. I had to add a special case in my fs_opendir function to handle if the path was equal to "D", and use the current working directory.

we would also get crashes when calling ls twice:



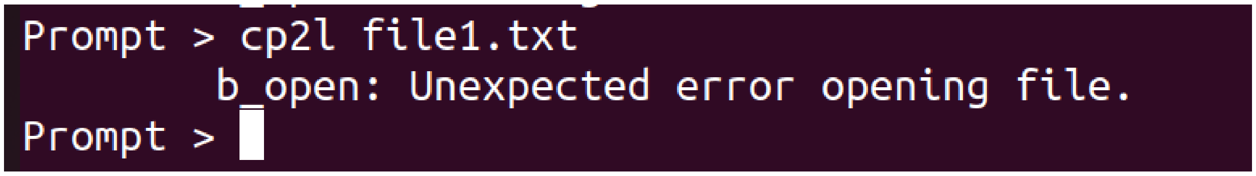
```
| mv                         |    OFF   |
| cp2fs                      |    OFF   |
| cp2l                       |    OFF   |
|--------------------------------|
Prompt > ls
No path entered; using currWorkDir

foo
Prompt > ls
No path entered; using currWorkDir

���`x
foo
double free or corruption (!prev)
make: *** [Makefile:67: run] Aborted (core dumped)
student@student:~/Documents/csc415-filesystem-jonathancurimao$
```

This is because ls calls fs_closedir. In my original fs_closedir function, we would free the directory pointer; this was dangerous because I could inadvertently end up freeing my root or current working directory from memory, and calling ls again would attempt to free already freed memory.



```
student@student:~/Documents/csc415-filesystem-jonathancurimao$ 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
Volume already initialized. No formatting needed.
Error! VCB already loaded!
|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|-------------------------------|
Prompt > touch filetest2.txt
        b_open: Creating file filetest2.txt
Prompt > cat filetext2.txt
        b_open: Unexpected error opening file.
Failed to open file system file: filetext2.txt
Prompt > 
```

Another problem that we had was our cat command not functioning properly when accessing a valid file.



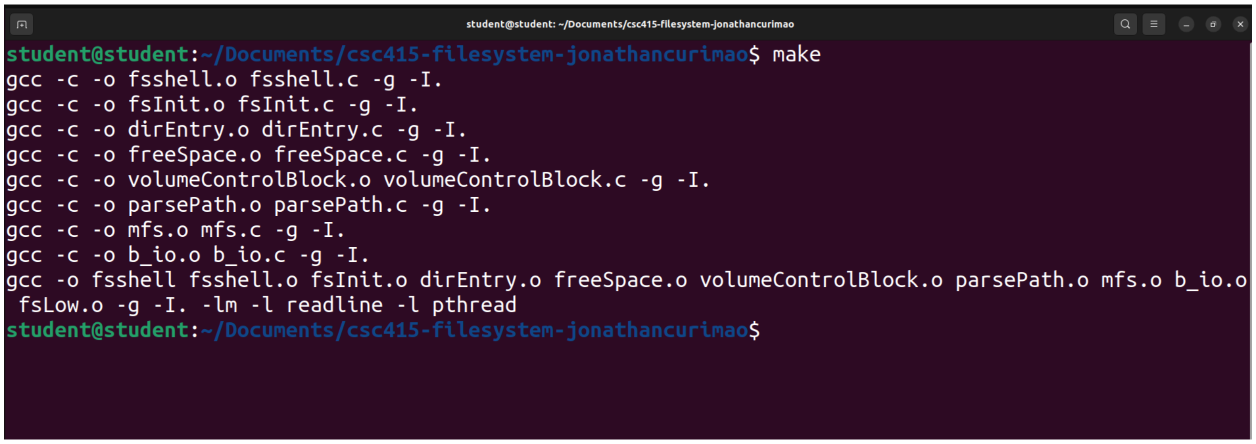
```
Prompt > cp2l file1.txt
        b_open: Unexpected error opening file.
Prompt > 
```

Here, our cp2l also isn't functioning properly, returning an error in our b_open.

Another function that does not work correctly is "ls -l [pathname here]". Running an ls -l on the current directory would return expected results, but running ls -l on a path such as "/foo" would yield incorrect results for the file size of directories.

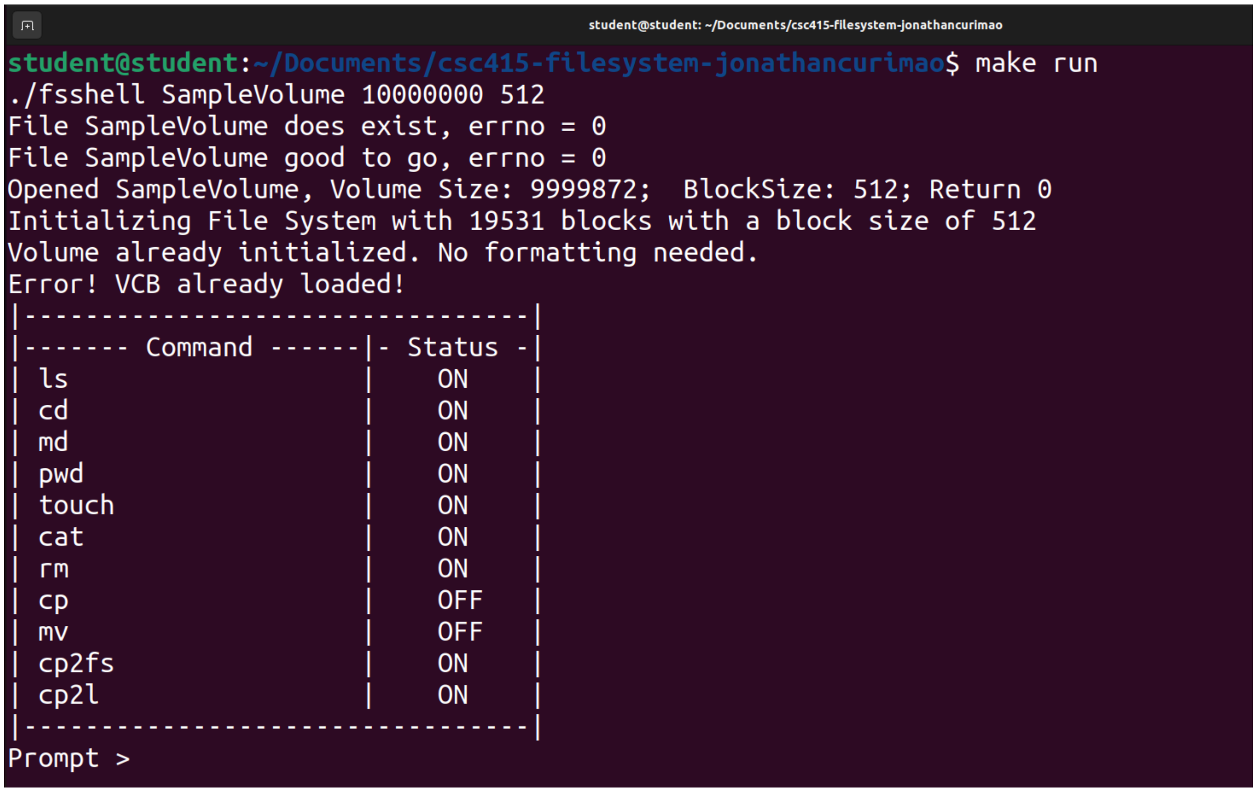
Touch does not function correctly. Currently, it allocates space for a file in memory, but that file is never actually written and saved to disk. This is because the directory containing the file is never actually updated with the proper directory entries and rewritten back to disk, meaning that the file does not persist from run to run, or even directly after being created.

Screenshots of Compilation:



student@student: ~/Documents/csc415-filesystem-jonathancurimao

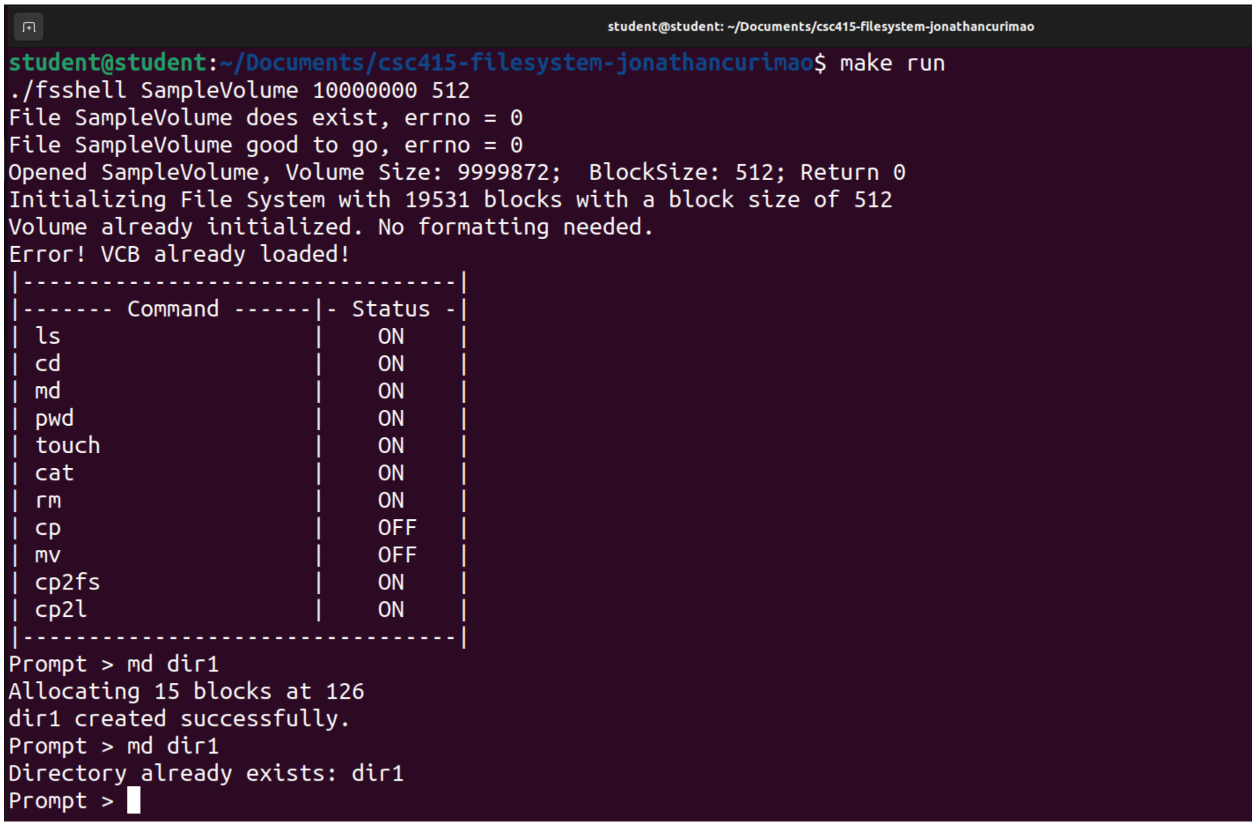
```
student@student:~/Documents/csc415-filesystem-jonathancurimao$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o dirEntry.o dirEntry.c -g -I.
gcc -c -o freeSpace.o freeSpace.c -g -I.
gcc -c -o volumeControlBlock.o volumeControlBlock.c -g -I.
gcc -c -o parsePath.o parsePath.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o dirEntry.o freeSpace.o volumeControlBlock.o parsePath.o mfs.o b_io.o
 fsLow.o -g -I. -lm -l readline -l pthread
student@student:~/Documents/csc415-filesystem-jonathancurimao$
```



```
student@student: ~/Documents/csc415-filesystem-jonathancurimao
student@student:~/Documents/csc415-filesystem-jonathancurimao$ 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
Volume already initialized. No formatting needed.
Error! VCB already loaded!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt >
```

Screenshots of Working Commands:

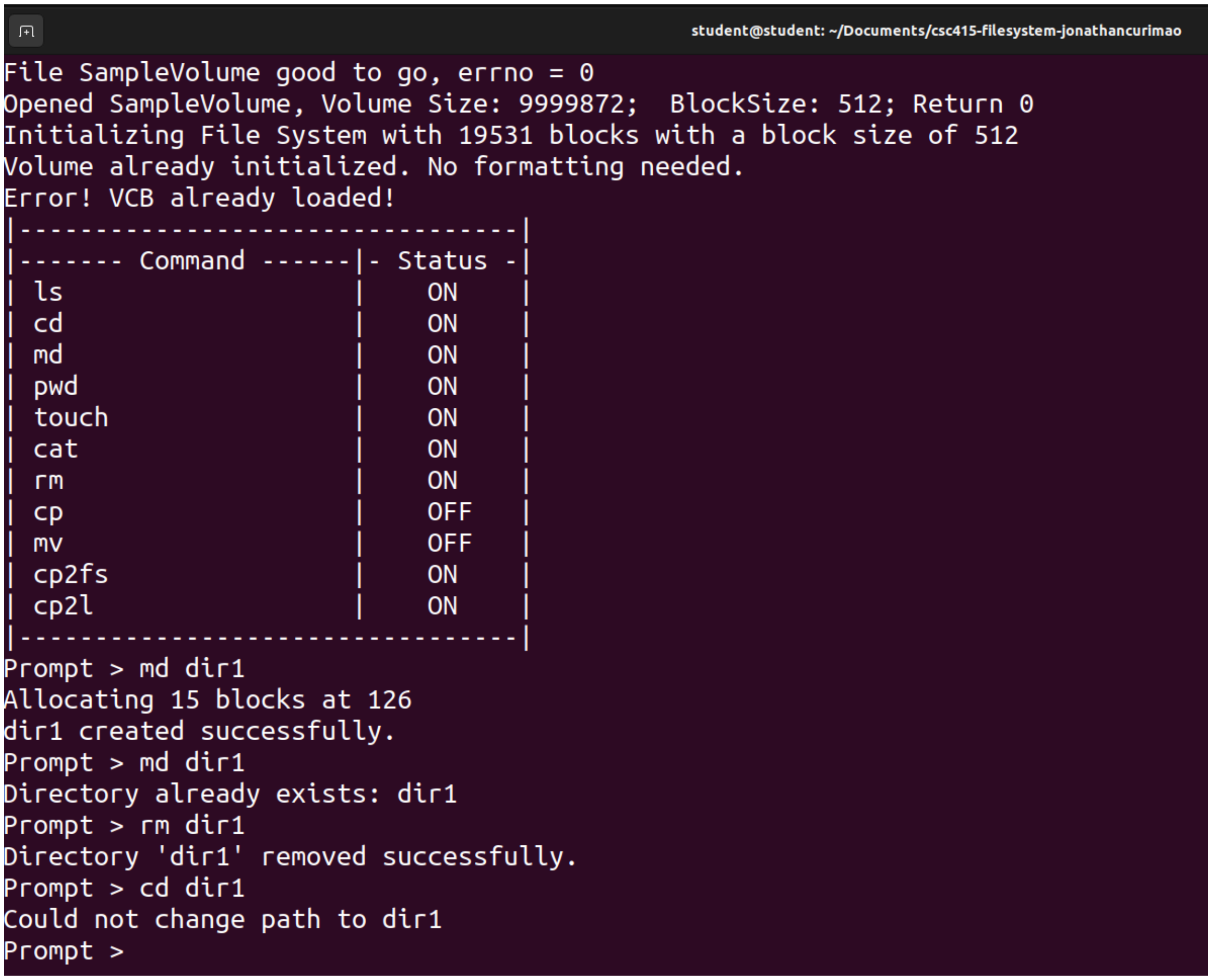
<u>md</u>



student@student: ~/Documents/csc415-filesystem-jonathancurimao

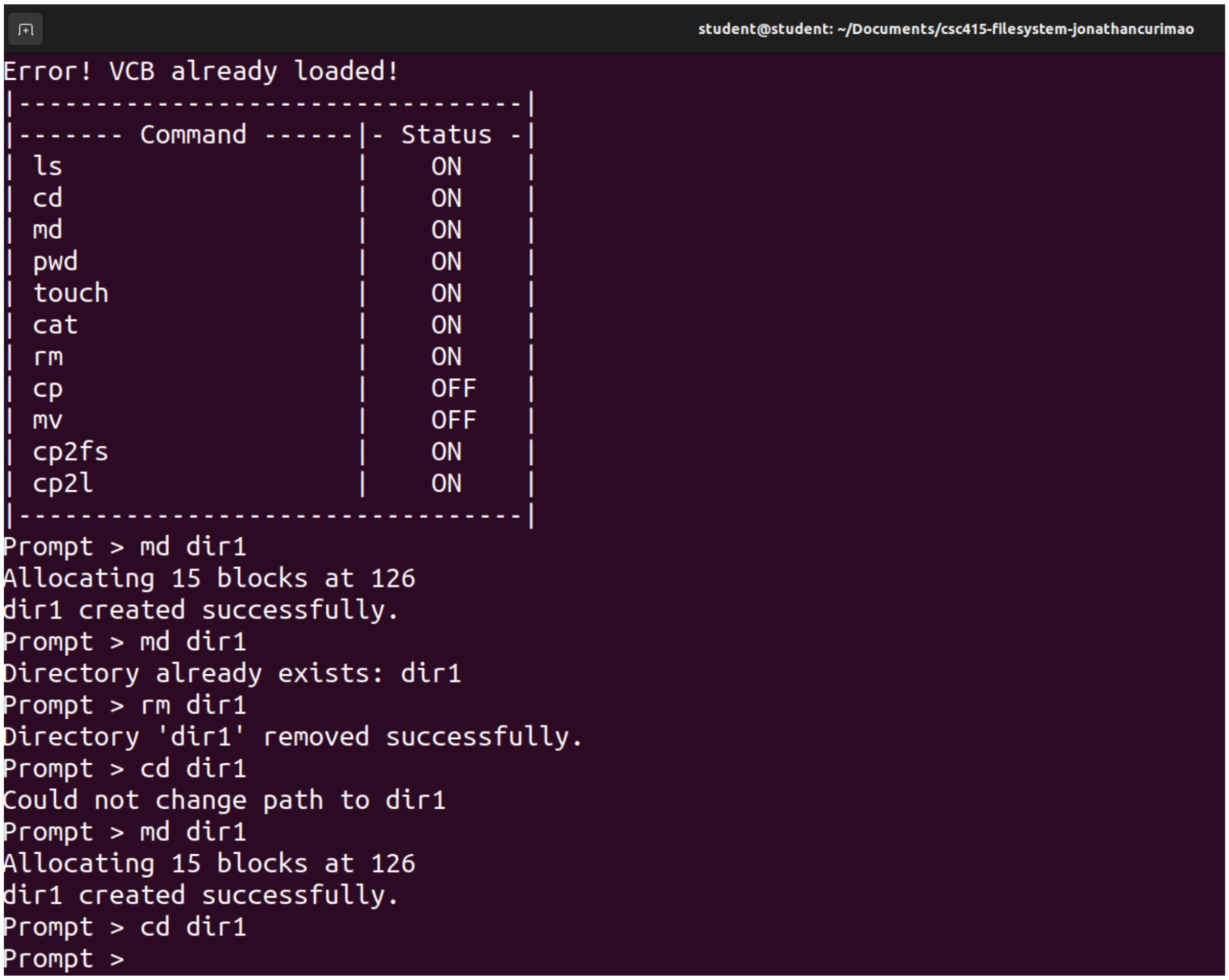
```
student@student:~/Documents/csc415-filesystem-jonathancurimao$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already initialized. No formatting needed.
Error! VCB already loaded!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > 
```

Rm


student@student: ~/Documents/csc415-filesystem-jonathancurimao

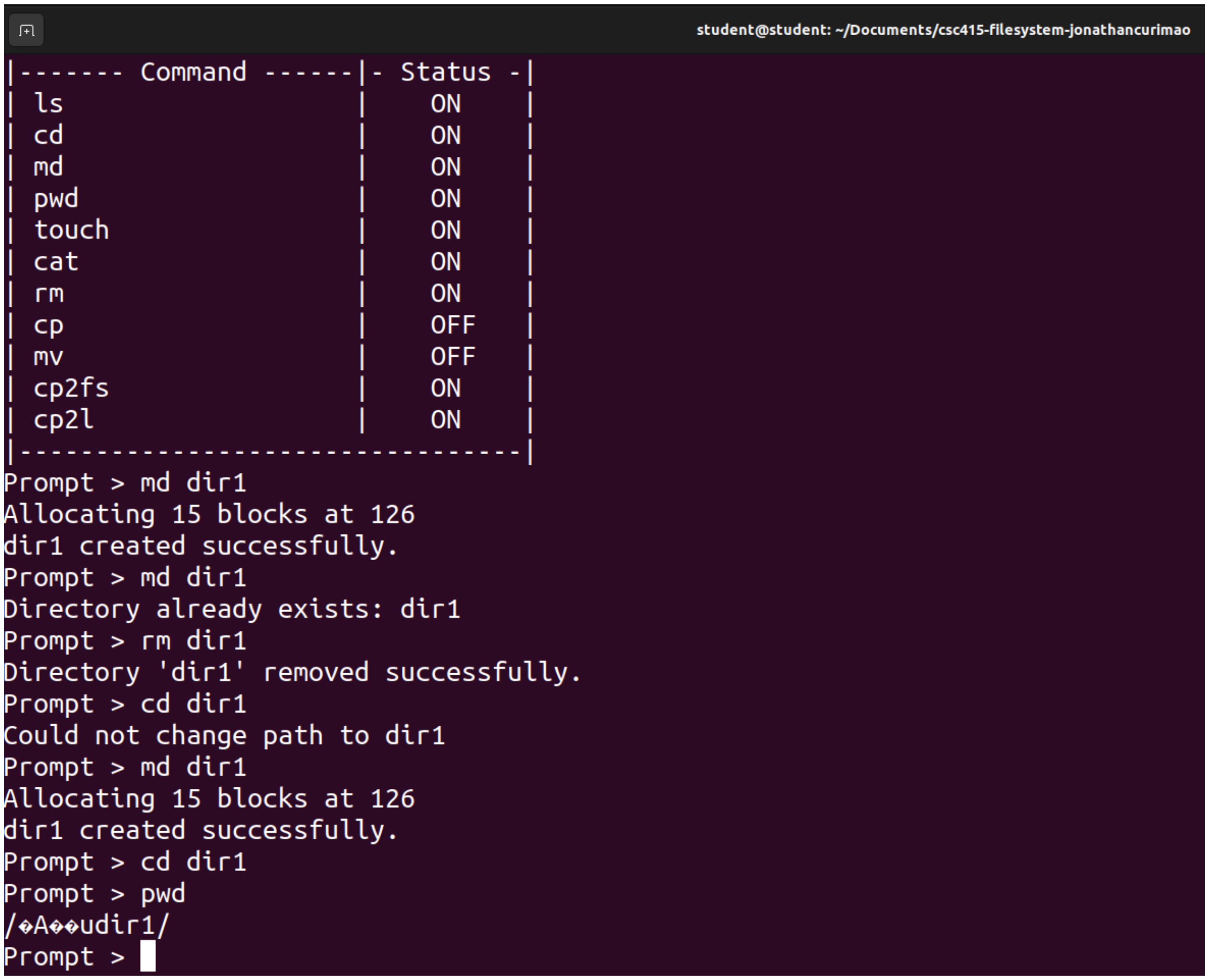
```
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already initialized. No formatting needed.
Error! VCB already loaded!
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt >
```

cd


student@student: ~/Documents/csc415-filesystem-jonathancurimao

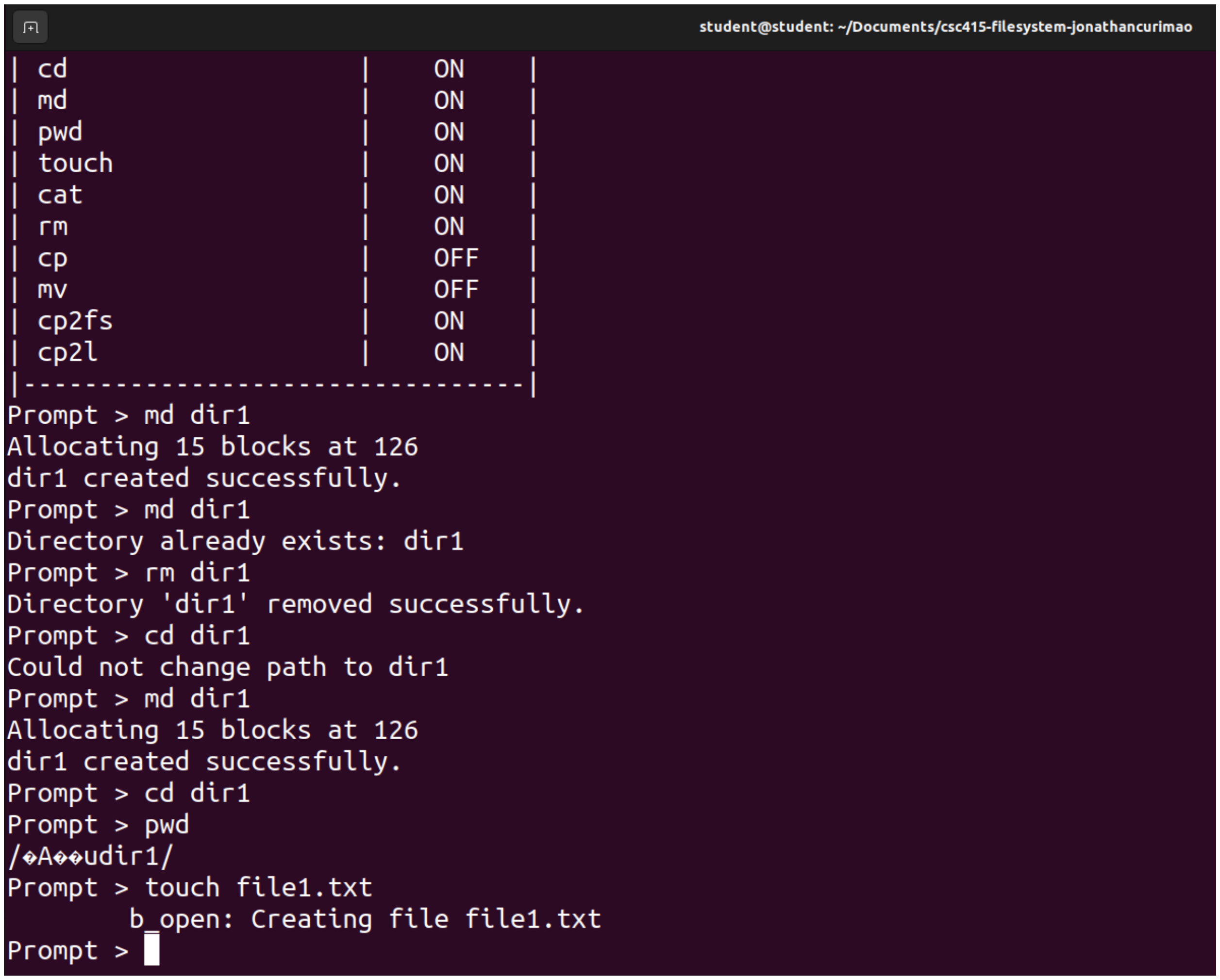
```
Error! VCB already loaded!
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt >
```

pwd


student@student: ~/Documents/csc415-filesystem-jonathancurimao

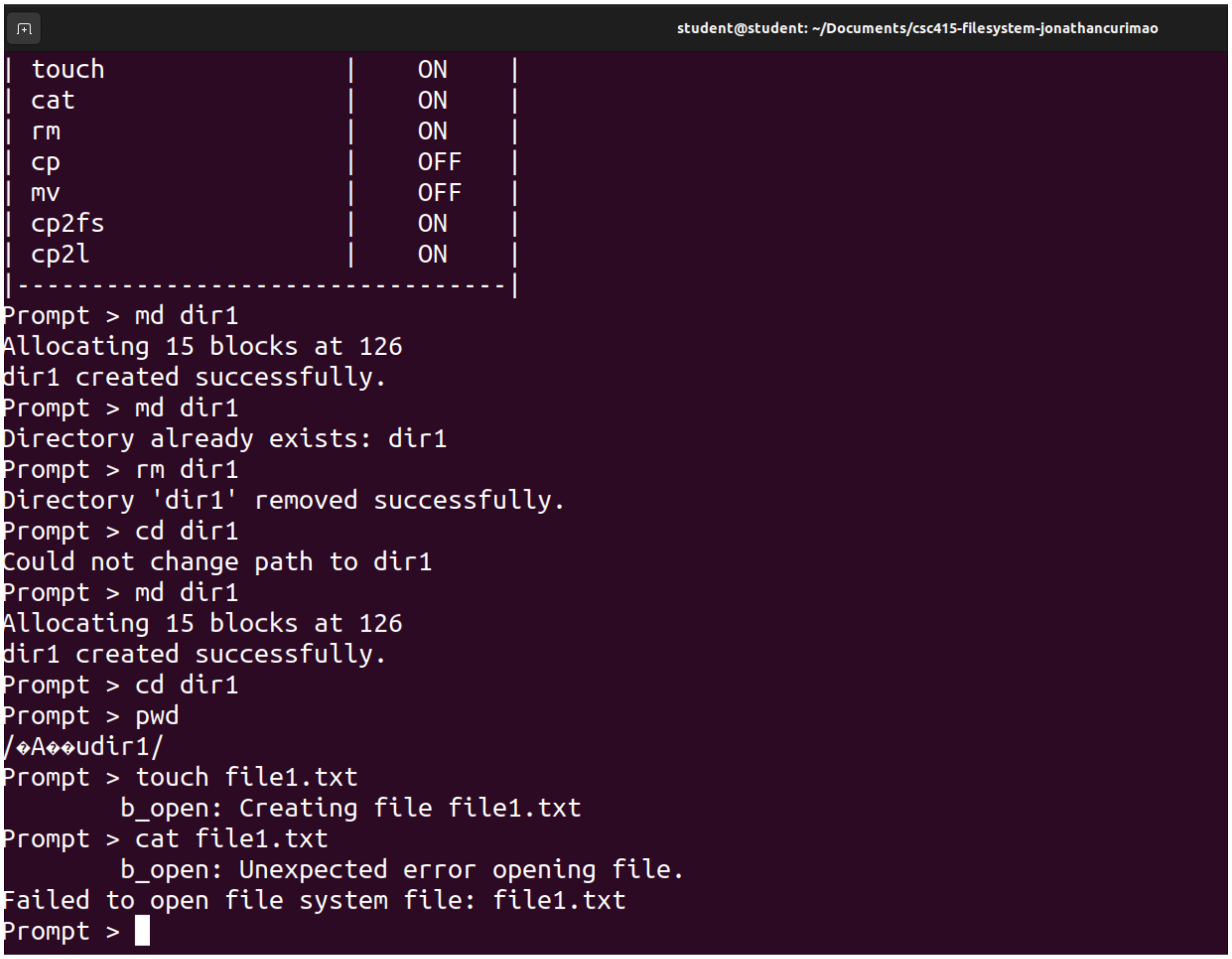
```
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt > pwd
/�A��udir1/
Prompt > 
```

touch



```
student@student: ~/Documents/csc415-filesystem-jonathancurimao
| cd                      |     ON     |
| md                      |     ON     |
| pwd                     |     ON     |
| touch                   |     ON     |
| cat                     |     ON     |
| rm                      |     ON     |
| cp                      |     OFF    |
| mv                      |     OFF    |
| cp2fs                   |     ON     |
| cp2l                    |     ON     |
|------------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt > pwd
/�A��udir1/
Prompt > touch file1.txt
        b_open: Creating file file1.txt
Prompt >
```

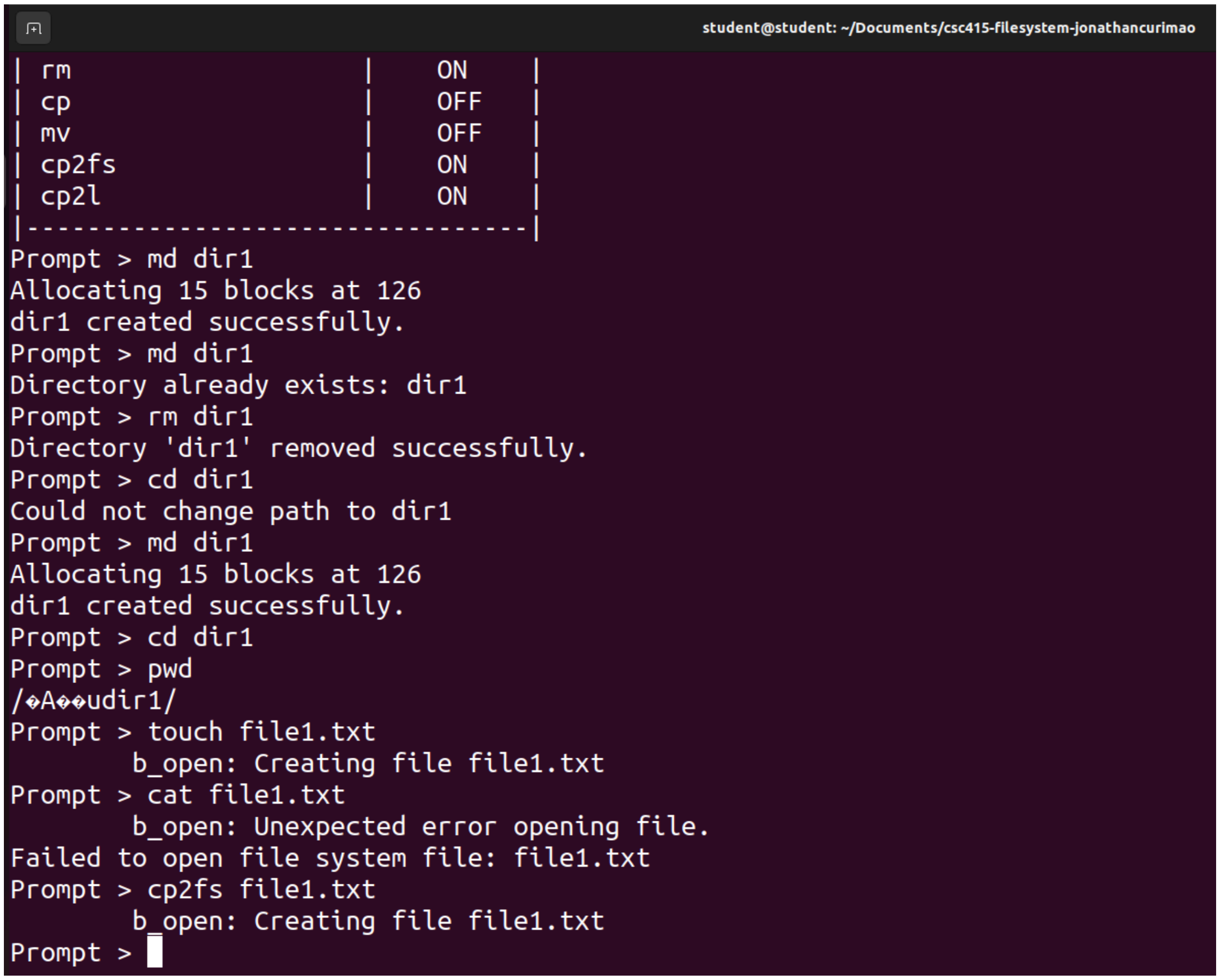
cat



student@student: ~/Documents/csc415-filesystem-jonathancurimao

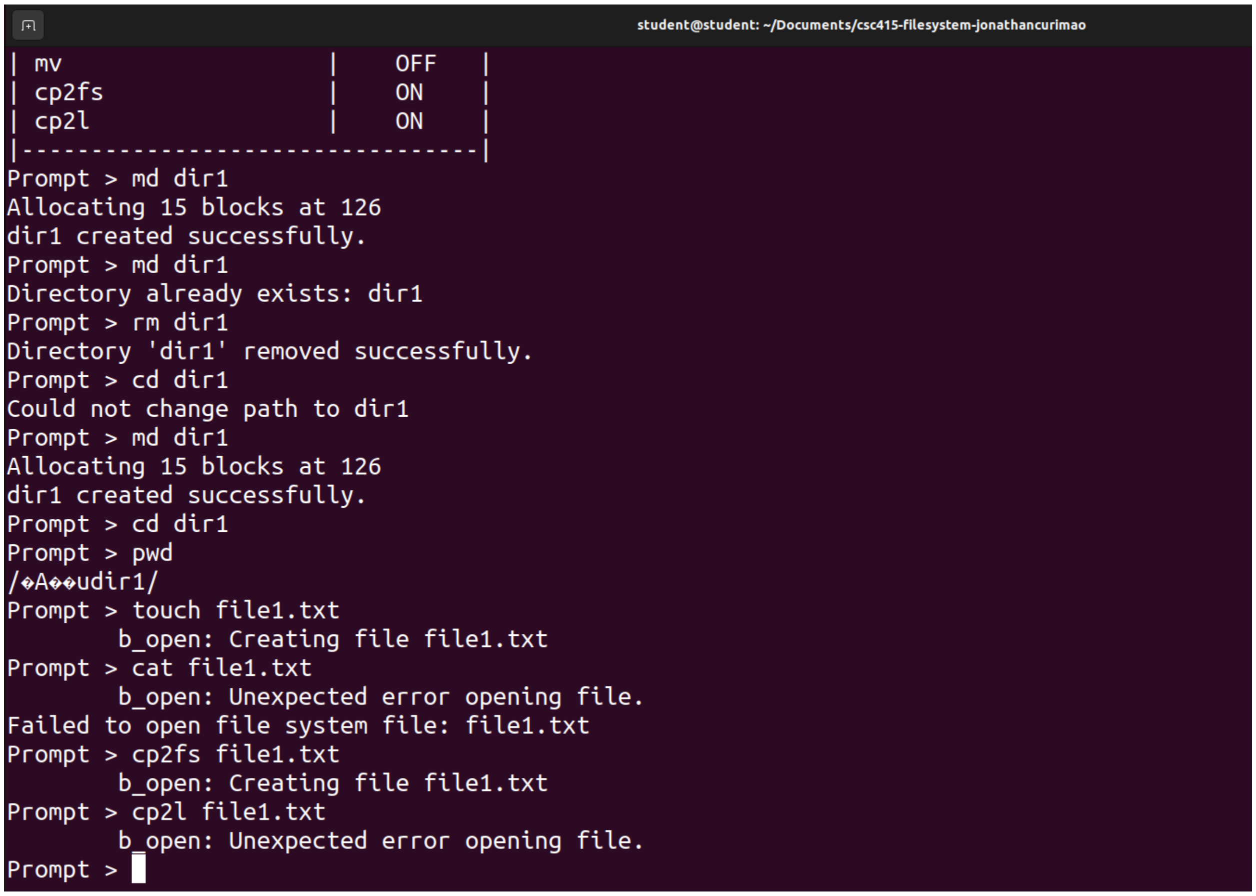
```
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt > pwd
/�A��udir1/
Prompt > touch file1.txt
        b_open: Creating file file1.txt
Prompt > cat file1.txt
        b_open: Unexpected error opening file.
Failed to open file system file: file1.txt
Prompt > 
```

cp2fs


student@student: ~/Documents/csc415-filesystem-jonathancurimao

```
| rm                    |    ON    |
| cp                    |    OFF   |
| mv                    |    OFF   |
| cp2fs                 |    ON    |
| cp2l                  |    ON    |
|----------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt > pwd
/�A��udir1/
Prompt > touch file1.txt
        b_open: Creating file file1.txt
Prompt > cat file1.txt
        b_open: Unexpected error opening file.
Failed to open file system file: file1.txt
Prompt > cp2fs file1.txt
        b_open: Creating file file1.txt
Prompt > 
```

cp2l


student@student: ~/Documents/csc415-filesystem-jonathancurimao

```
| mv                   |    OFF    |
| cp2fs                |    ON     |
| cp2l                 |    ON     |
|---------------------------------|
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > md dir1
Directory already exists: dir1
Prompt > rm dir1
Directory 'dir1' removed successfully.
Prompt > cd dir1
Could not change path to dir1
Prompt > md dir1
Allocating 15 blocks at 126
dir1 created successfully.
Prompt > cd dir1
Prompt > pwd
/�A��udir1/
Prompt > touch file1.txt
        b_open: Creating file file1.txt
Prompt > cat file1.txt
        b_open: Unexpected error opening file.
Failed to open file system file: file1.txt
Prompt > cp2fs file1.txt
        b_open: Creating file file1.txt
Prompt > cp2l file1.txt
        b_open: Unexpected error opening file.
Prompt >
```

Contributions:

Jonathan Curimao	**freeSpace, fs_isDir, fs_isFile, b_read, and the approaches for these functions**
Jonathan Ho	**Directory functions, readdir, closedir, opendir, b_write, small bit of b_open, general debugging + unifying functions**
Juan Segura Rico	**freeSpace, fs description, setcwd, getcwd, b_open, b_close, and approaches for these + isDir and isFile**
Yashwardhan Rathore	**volumeControlBlock (VCB), fs_mkdir, fs_stat, b_seek, a bit of b_write and approaches for these functions.**