Group File System

Team Name: Horse

GitHub Link:

https://github.com/CSC415-2025-Spring/csc415-filesystem-Karina-Krystal

Team Members:	**Student ID:**
Karina Alvarado Mendoza	921299233
Wilmaire Mejia	922846038
Anshaj Vats	923760991
Ayesha Irum	923656653

File System - Milestone 1

Description:

The file system project was developed through three progressive milestones, each that built essential components to create a complete, and functional system. Milestone 1 was the main foundation of our system. Within the milestone, it included the creation of the Volume control Block (VCB) to store critical metadata. A bitmap based free space system was implemented to keep track of allocated and available blocks. The root directory structure was initialized as well, this created a basic file system organization and to further support our files. Milestone 2 expanded our system's capabilities by integrating key functions, which allowed the user to interact with the file system through commands. These functions included fs_setcwd, fs_getcwd, fs_isFile, fs_isdir, fs_mkdir, fs_opendir, fs_readdir, fs_closedir, and fs_stat. Additionally, fs_delete and fs_rmdir later on. Milestone 3 introduced b_io functionally to support buffered file input and output. Functions like b_open, b_seek, b_read, b_write, and b_close were implemented in this portion. With these milestones, it established a complete and well functioning file system, with all 12 core functions working.

Approach:

Milestone 1:

Filesystem Initialization & Format Check

On mount, we call initFileSystem(), which begins by reading the first eight bytes of block 0 and reconstructing our magic signature. If it doesn't match, we know the volume is unformatted: we zero out a fresh VCB, record signature, total blocks, block size, the first free block, and placeholders for our FAT and root directory, then write it back to disk.

Volume Control Block & Free-Space Map

Next, initFreeSpace() computes how many blocks are needed to store one int per disk block (rounding up to the block size) and allocates an in-memory array, freeSpaceMap[]. We chain every free block into a single linked list—index i points to i + 1—and mark reserved blocks (the VCB at index 0, the FAT blocks themselves, and the last block) with the end-of-chain sentinel 0xFFFFFFFF. Writing this table out to block 1 establishes our on-disk FAT, and we stash its location and size back in the VCB.

Upon formatting an uninitialized device, initFileSystem() reads block 0 into a temporary buffer and checks the signature. If it is absent, the code zeroes out a fresh VCB, assigns the signature and geometry (totalBlocks and blockSize), and calls initFreespace(). Here, we compute how many blocks the free-space map itself will occupy:

int blocksNeeded = (sizeof(int)*numberOfBlocks + blockSize − 1) / blockSize;

Later on I made a function for this called calculateFormula because it is used so much.

At the very heart of our design sits the Volume Control Block (VCB), a small structure stored in block 0 that holds the filesystem's "super-metadata"—the magic signature, total block count, block size, and the locations and sizes of the free-space map and root directory.

To track free space, we chose an in-memory FAT (File Allocation Table) model: an array of integers, one per logical block, where each entry points to the next free block in a singly linked list, with a reserved sentinel value (0xFFFFFFFF) marking list ends or reserved blocks. In format, **every** unallocated block is simply chained into this "big" free list, and the reserved regions (the VCB itself, the FAT blocks, and the last block) are immediately marked off-limits. This contiguous, head-only allocation strategy was deliberate: it reduces both conceptual complexity and code paths, giving us **O(1)** allocation of the next free block (or chain of blocks) and a straightforward rollback path on failure.

Contiguous allocation at format (i.e., building one long run of block pointers) further simplifies our initial on-disk layout: files and directories allocated without fragmentation all appear as contiguous chains in memory, and our simple allocateblock(n) routine merely severs the first *n* entries off the free list. While this approach can suffer from fragmentation over time (and does not pack new allocations next to existing ones), it provides a clean baseline: free space is always represented by a single linked chain, and neither allocation nor deallocation requires any search beyond pointer chasing.

We bootstrap the namespace with createDirectory(…), which leverages that same contiguous allocation to carve out a root directory. By initializing just the "." and ".." entries and writing them in one sweep, we guarantee that the root is always self-contained and immediately traversable.

Milestone 2:

Structure & Parse Path:

Our second phase of our project was well built off of what we had previously. It helped us begin user level interactions with the file system using a shell interface via fsshell. Before creating any functions, we added essential and necessary features to our DE (Directory Entry) struct. Our main goal was to implement the main given functions first and add the other optimal ones. A key function we had to develop first was parsePath(). This function gave us the ability to read in a

string based path (like “/documents/../student/file.txt”) into a structured component that the system can use to find and or create files and directories. It splits a path string into separate components ((“documents” , “..”, “student”) and “file.txt”). We later fixed parsePath to be simpler for when we implemented rmdir.

Functions:

Collectively we implemented a range of directory functions that allowed us to create subdirectories, remove them, change the working directory, and list out the contents, all while updating the FAT (File Allocation Table) and directory entries accordingly. These functions shared a unified design logic.

Milestone 3:

B_io.c

In Milestone 3, I approached the b_io.c file, focusing on implementing the buffered I/O interface to provide higher-level buffered operations like b_open, b_read, b_write, b_seek, and b_close which sit on top of the low-level disk access functions. The plan is to use these functions to improve performance by minimizing direct disk reads and writes by using an in-memory buffer.

We will begin by implementing initialization and management of file control blocks using the b_init() and b_getFCB()functions. These will maintain the internal table that tracks open files and ensures that each entry is properly reset when it is not in use. In **b_open,** I will handle different file access modes based on flags like O_RDONLY, O_WRONLY, and O_CREAT. I plan to use a helper function such as parseFilePath() to locate the file or create when needed, and assign the access

logic to specific handlers for reading, writing, or creating. Once we successfully open a file, the FCB will be populated with relevant metadata and the buffer will be prepared using setupCommonFCBFields, and then finally the parent directory will be updated to reflect any changes on disk.

The plan is to implement a segmented strategy for reading in the **b_read** function that divides the operation into three distinct parts. We want to approach this in a way that minimizes unnecessary disk access and properly handles reads that will span multiple blocks. To do so, we approach this by first reading any remaining data in the buffer. If the requested read size is larger than what remains, we will then read full disk blocks directly into the caller's buffer to maximize efficiency. Lastly, if there are still bytes left to read, we will refill the buffer and copy the remaining data.

For writing, the plan is to validate the input and make sure that there is enough space available before writing. We approach this by writing data from the caller’s buffer into the file’s buffer, then updating the file’s size and modification time. We will also write the updated metadata back to disk to maintain consistency.

In the **b_close** function, the plan is to properly release all dynamically allocated memory used during the file's lifetime, including the buffer, file metadata, and the parent directory copy. My approach here includes resetting the file control block so that it can be safely reused in future operations.

Issues and Resolutions:

Issue 1:

While testing with the md command (to create a directory within our file system shell), I encountered the following error:

md: command not found.

I got this message when I entered md aaa in the hope that a new directory named aaa would be created. The command was not recognized in any way, which meant either that the command had not been registered within the shell, or the corresponding function (fs_mkdir) hadn't been properly linked.

Resolution:

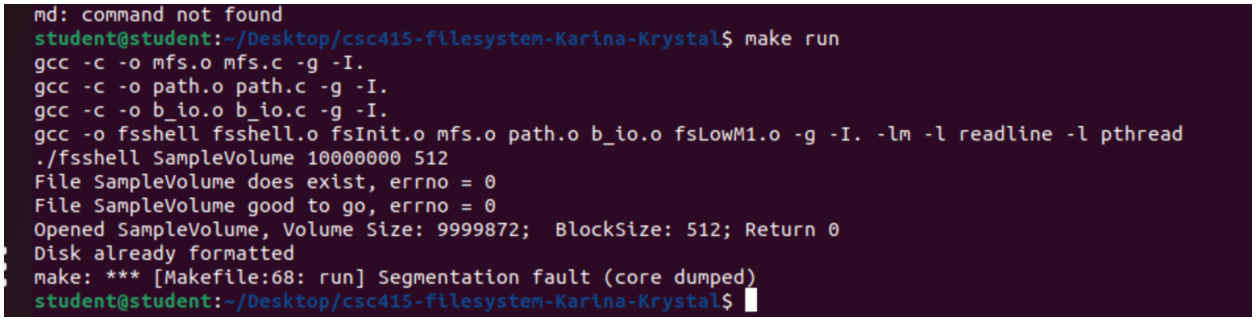
To fix this, I looked into the command registration place in fsShell.c and found that the md command handler was missing. I inserted the handler to execute the correct function (fs_mkdir) when typing md. After recompiling and rerunning, the command did just what I had hoped. The directory was created, and checking for it using ls verified it was present.

Issue 2:

After introducing path-related operations like fs_getcwd and fs_setcwd, I experienced a segmentation fault as soon as I ran the shell using make run. The file system volume was successfully loaded, but the shell crashed once disk information was printed. The terminal output was:

This meant that something in initialization, likely involving setting or getting the current working directory was attempting to access bad memory.

I found upon inspection that my fs_setcwd was directly assigning a string literal to cwdPathName, and my fs_getcwd was returning a pointer to global memory instead of the incoming buffer, causing corruption at startup or later commands like pwd.



Resolution:

To fix this, I modified fs_getcwd to return the correct buffer and null-terminate it properly. I also fixed fs_setcwd to safely update cwdPathName using strcpy or strdup, and properly allocated and freed memory. After recompiling and rerunning, the segmentation fault disappeared and all directory commands like cd and pwd worked as they should.

Issue 3:

I executed md test to create a new directory. The command was successful and did not show any errors. However, when I issued ls, the new directory was not shown. I

inspected the issue by viewing the directory block with xxd and made sure that no valid entry was written.

I was suspecting a bug in my implementation of fs_mkdir, particularly in the section that is meant to set up and write the new directory entry.

```
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md test
Prompt > ls



Prompt >
```

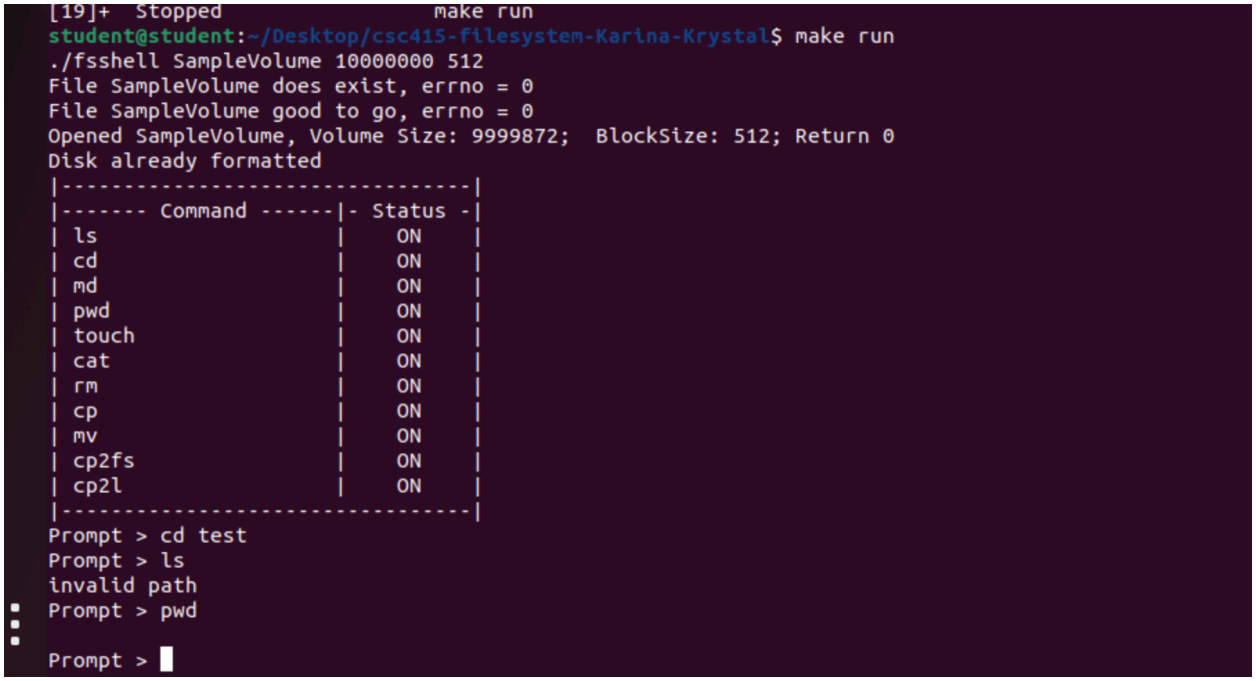
Resolution:

Once I reviewed my code, I realized that although I had created the new directory entry (DE newFolderEntry), I had failed to call initialize_directory_entry() to populate its fields such as name, isDirectory, and location. This resulted in an uninitialized structure being written to the parent directory.

Once I inserted the missing initialization, the folder was displayed correctly in the ls output, and thus the bug was resolved.

Issue 4:

After I made and switched to a directory with md test and cd test, I ran pwd and ls to verify my location and what was in it. But pwd gave an empty line, and ls returned invalid path. That indicates the current working directory was not being monitored or returned by the system properly.



Resolution:

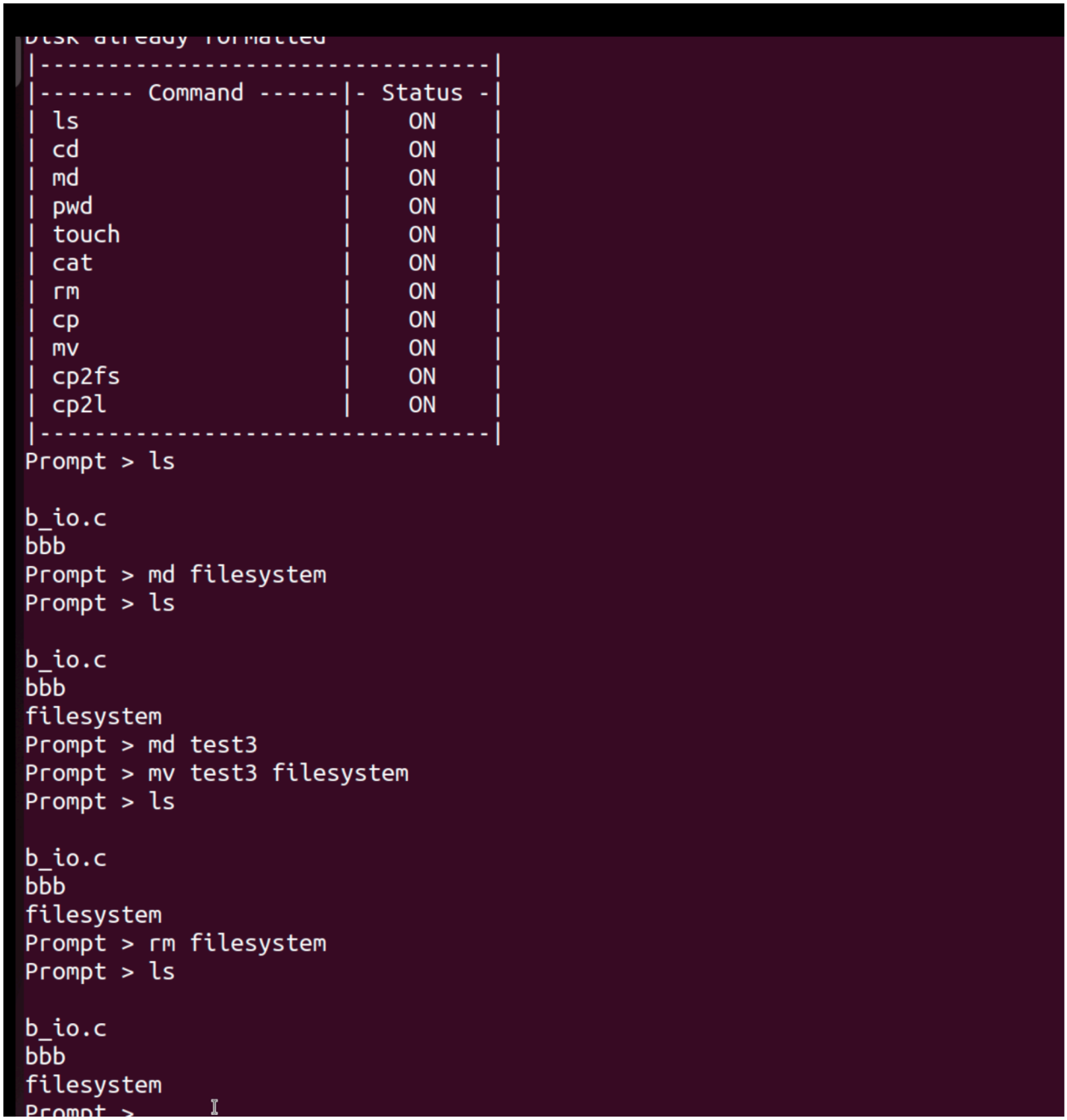
My fs_getcwd() function was returning the global pointer cwdPathName instead of the pathname buffer that it was passed. I wasn't also checking for null-termination, which can result in undefined behavior. Because of that, commands like pwd and ls depending on the current path were failing or behaving unpredictably.

I corrected the implementation like this

```
char* fs_getcwd(char* pathname, size_t size) {
    strncpy(pathname, cwdPathName, size - 1);
    pathname[size - 1] = '\0';
    return pathname;
}
```

Issue 5:

While attempting to test the rm feature, I ran into a problem where directories would seemingly be erased initially but continued to appear when I used ls to list the files. For instance, after making a directory called filesystem and placing another folder (test3) within it, I issued the rm filesystem command hoping that it would be erased. But when I ran ls a second time, the filesystem directory remained in the listing, meaning that the delete did not persist.



Resolution:

When I reviewed the code for fs_delete(), I realized that although the file entry location was correctly marked as deleted (location = -2), the new parent directory was never

written to disk. This meant that although the memory was updated, the file system itself was not. To fix this, I added a call to write_parent_directory() right after marking the entry as deleted. This made the modification saved properly to disk. After this alteration, the rm command behaved properly, and directories were being unmounted completely as wanted. This issue highlighted the importance of retaining changes after any metadata update in file system operations.

Issue 6:

I created a parent and child directory, and then used the mv child parent command to move child to parent. I then attempted to cd into the moved directory using cd parent/child. I used pwd expecting it to return the correct full path /parent/child, but it printed a jumbled string (ild/). Then after coming back via cd. and running pwd once again, no output was provided, suggesting the shell had lost its record of the correct working directory. That is, though the directory tree was now current, the internal reference to parent directory (.) inside child was not current after the move, and there was path tracking breakage.

```
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md parent
Prompt > md child
Prompt > mv child parrent
Prompt > mv child parent
Prompt > cd parent/child
Prompt > pwd
ild/
Prompt > cd ..
Prompt > wd
wd 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
Prompt > pwd

Prompt > exit
System exiting
```

Resolution:

I reviewed my fs_mv() function and realized I wasn't updating the. entry in the moved directory. After a directory is moved, its. reference should point to the new parent, not the old parent. To fix this, I changed the function to call updateDirectoryParentRef() after copying the directory entry to the destination. This utility function scans the directory

and replaces the second entry (.) with the new parent's metadata so that navigation is correct. pwd and directory traversal were both correct following this adjustment.

Issue 7:

I made two directories named dirA: one as a root-level directory and the other as a subdirectory within another directory named dirB. Then I attempted to move the root-level dirA to dirB, which could be a problem since a directory with the same name already existed there.

Resolution:

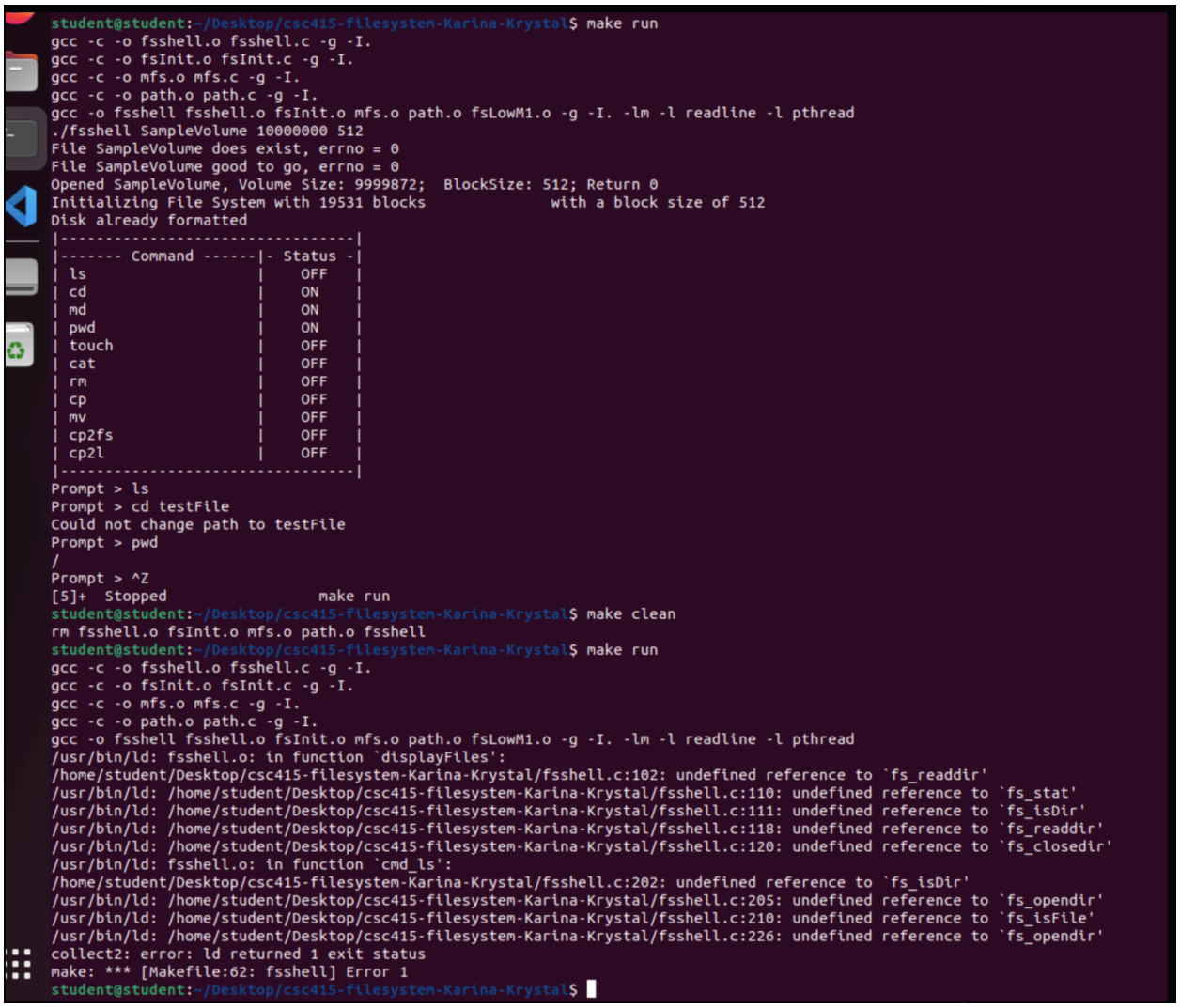
My fs_mv handled the conflict correctly. It either overwritten the existing one or mixed in well without crashing. There was only a single dirA shown in dirB after the operation, and the system was still operational, verified using ls.

```
[26]+  Stopped                 make run
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md dirA
Prompt > md dirB
Prompt > md dirB/dirA
Prompt > mv dirA dirB
Prompt > ls dirB

dirA
Prompt > ls



test
parent
mama
yyy
dirB
Prompt >
```

Issue 8:

When I tried recompilation of the project using make clean and then make run, I encountered a number of undefined reference errors. These were created at the linking stage and referred to functions like fs_stat, fs_readdir, fs_closedir, fs_opendir, fs_isDir,

and fs_isFile.

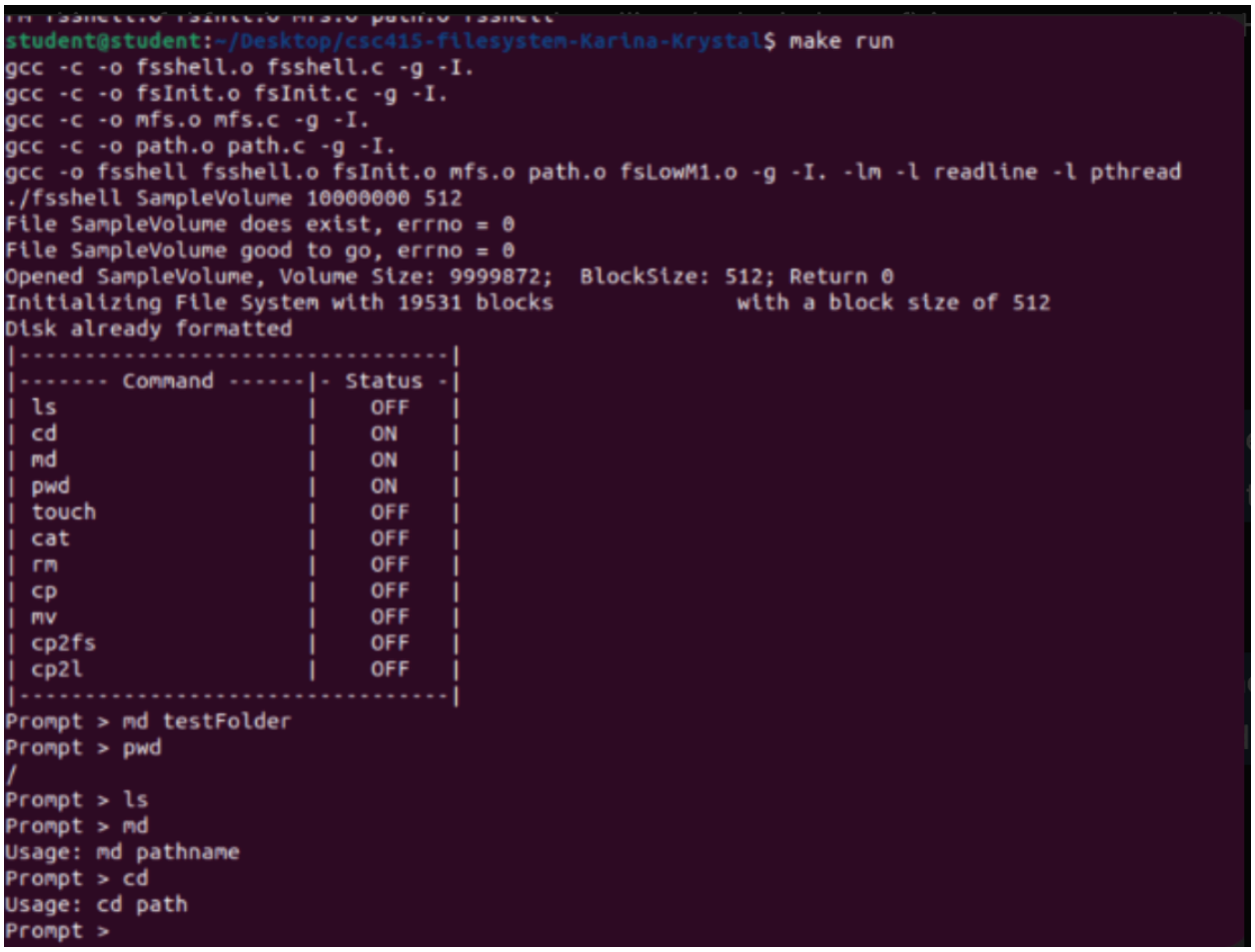


Resolution:

These errors typically happen when function declarations are included in source files (e.g., mfs.c) but are not compiled or linked properly. I resolved the issue by ensuring that all necessary object files were included in the Makefile (e.g., mfs.o) and that the function prototypes were correctly declared in the header file.

Issue 9:

When I used commands like md and cd with no arguments given (e.g., cd, md, cd path), the shell printed error messages like Usage: cd path or Usage: md pathname. This shows that the shell correctly recognizes missing parameters but requires more detailed feedback or examples to the user.



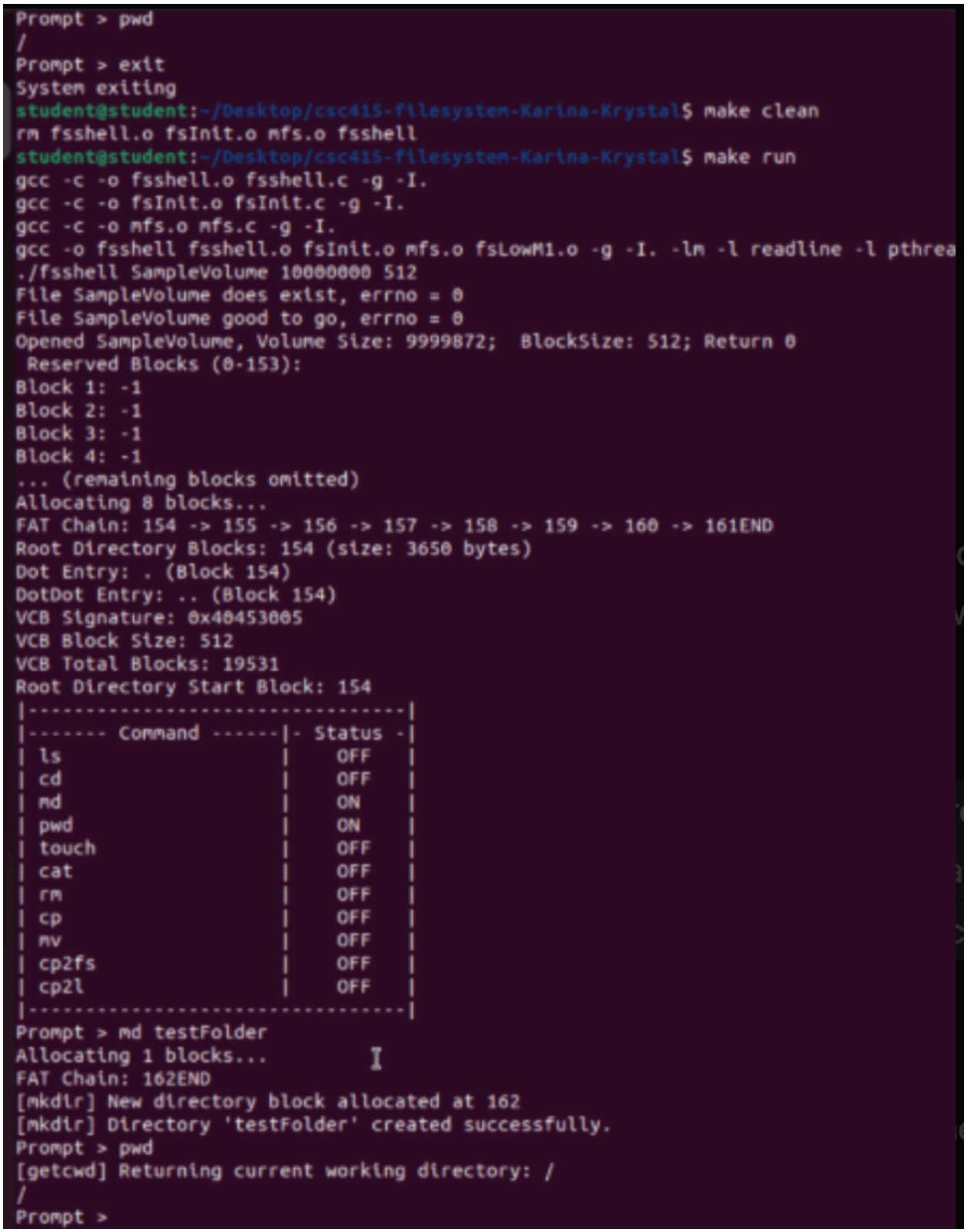
Resolution:

This issue highlights that the shell can enforce syntax on simple command use but could be better served by a more descriptive or user-friendly message format. No code was changed for this, but it ensures that input checking is working as expected.

Issue 10:

First of all, I was doubtful that the fs_mkdir() function was indeed creating directories and allocating blocks in the file system properly. Even if the command generated

"created successfully," I wanted to verify this at another level.



Resolution:

To resolve this, I re-built the project and checked the console output. It obviously shows that blocks were allocated (block 162), the folder was created successfully, and the current directory was proper (/). This confirmed that the createDirectory() and initialize_directory_entry() logic was working correctly and writing to the volume correctly.

Issue 11:

Analysis:

VCB:

```
000200: 05 30 45 40 00 02 00 00  00 00 00 00 4B 4C 00 00 | .0E@........KL..
000210: 00 00 00 00 01 00 00 00  00 00 00 00 9A 00 00 00 | ...............
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 | ................
```

From 000200 to 000207: 8 bytes is the signature. Value: 0x0000020040453005. This value will stay consistent as it is our signature for our file system.

From 000208 to 00020B
and 000210 to 000213
and 000218 to 00021B: 12 bytes is reserved space.

From 000214 to 000217: 4 bytes is status.

From 00020C to 00020F: 4 bytes is data.

From 00021C to 00020F: 4 bytes is the starting block of rootDir. Value: 0000009A.

From 000220 to 0002FF: 224 bytes is the non accessible memory. The structure reads in 512 bytes, but doesn't use them all, but it still shows it since it is part of the VCB.

FreeSpace:

000400: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000410: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000420: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000430: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000440: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000450: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000460: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000470: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000480: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
000490: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004A0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004B0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004C0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004D0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004E0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????
0004F0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ????????????????

From 000430 to 0004FF: 256 bytes is free or uninitialized space.

RootDir:

013600: **2E 00 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: **00 01 8C 0B** 00 00 00 00 **00 00 9A 00** 00 00 00 00 | ..?.......?.....
013630: 00 00 F6 4D F4 67 00 00 00 00 2E 2E 00 00 00 00 | ..?M?g..........
013640: 00 00 00 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 01 8C 0B 00 00** | ....?...
013660: 00 00 00 00 **9A 00 00 00** 00 00 00 00 **F6 4D F4 67** | ....?.......?M?g
013670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
013680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
013690: 00 00 00 00 **01 00 00 00** 00 00 00 00 00 00 00 00 | ....
0136A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0136B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....

0136C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 | ....
0136D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0136E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....
0136F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....

Analysis:
013600-013603: 00 00 00 2E is "."
013610-01361F: Empty directory entry
013620-013623: 0B 8C 01 00 is file size is 64 bytes
013628 - 01362B: 9A is location of the block which is block 154
013630- 013633: 4D F6 00 00 is time created
013634-013637:00 00 67 F4 is last access time
013638-01363B: 2E 2E 00 00 is ".."
013640-01364F: Empty directory entry
013650- 01365F: Empty directory entry followed by file size = 64 bytes
013660 - 01366F: Start block = 154, creation + access timestamps
013670-01367F :Empty directory entry
013680-01368F: Empty directory entry
013694-013697: 00 00 00 01 is 'isDirectory'

For the remaining blocks:
- Rows with all 00s are an Empty directory entry ,
- Rows with a 01 and the rest 00s are an 'isDirectory'

Screenshot of compilation:

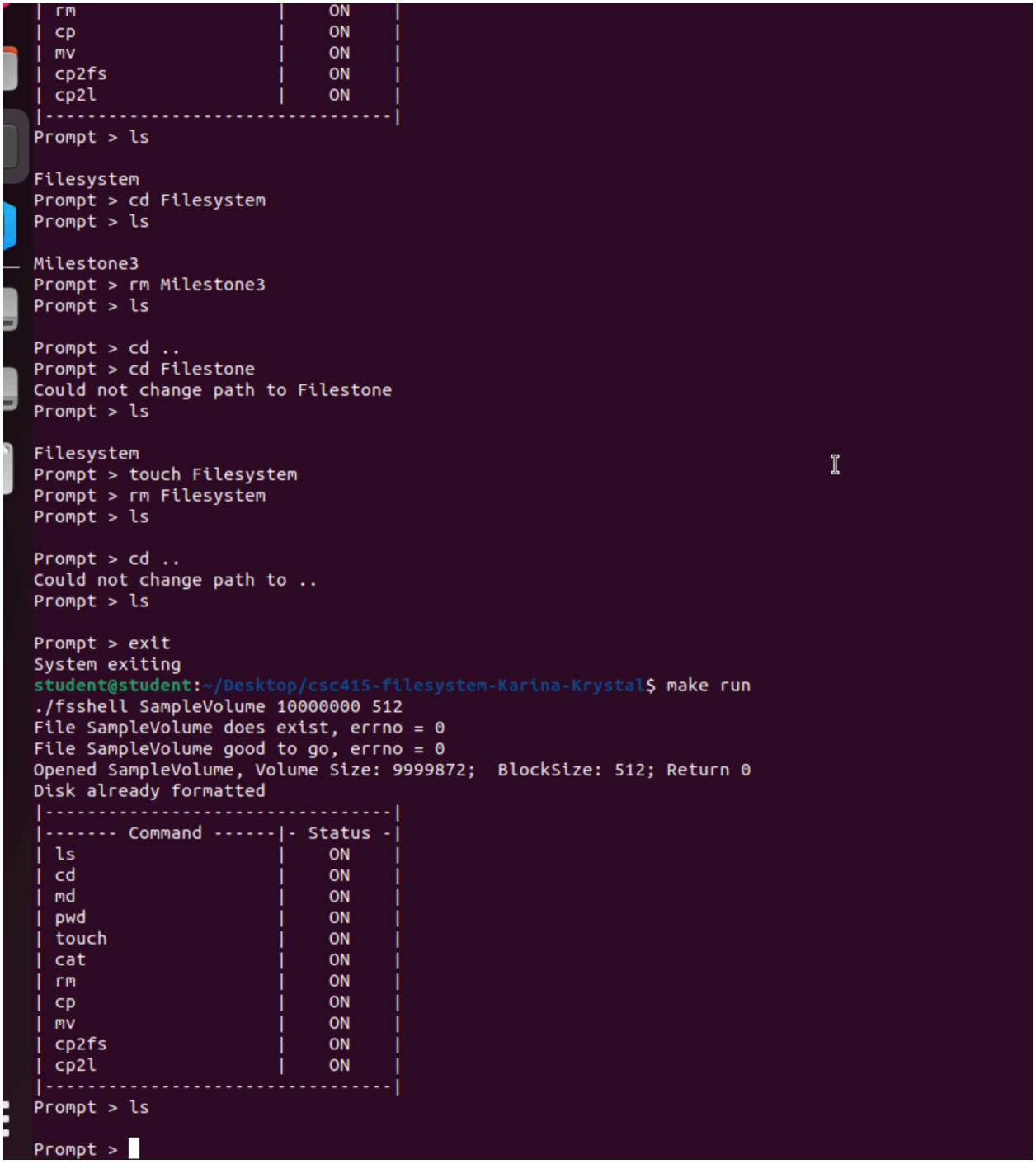
```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ make
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o path.o b_io.o fsLowM1.o -g -I. -lm -l re
adline -l pthread
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```

Screen shot(s) of the execution of the program:

```
[8]+  Stopped                 make run
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ make clean
rm fsshell.o fsInit.o mfs.o path.o b_io.o fsshell
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o path.o path.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o path.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
./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
Disk already formatted
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls

bbb
Prompt > md Milestone3
Prompt > md Filesystem
Prompt > mv Milestone3 Filesystem
Prompt > ls

bbb
Filesystem
Prompt > cd Filesystem
Prompt > ls

Milestone3
Prompt > pwd
/Filesystem/
Prompt > cd Milestone3
Prompt > pwd
/Filesystem/Milestone3/
Prompt > exit
System exiting
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$
```

```
System exiting
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ make clean
rm fsshell.o fsInit.o mfs.o path.o b_io.o fsshell
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o path.o path.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o path.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
./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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

bbb
Filesystem
Prompt > touch bbb
Prompt > rm bbb
Prompt > ls

Filesystem
Prompt > exit
System exiting
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$
```

If we want to remove parent directory


rm	ON
cp	ON
mv	ON
cp2fs	ON
cp2l	ON
Prompt > ls
Filesystem
Prompt > cd Filesystem
Prompt > ls
Milestone3
Prompt > rm Milestone3
Prompt > ls
Prompt > cd ..
Prompt > cd Filestone
Could not change path to Filestone
Prompt > ls
Filesystem
Prompt > touch Filesystem
Prompt > rm Filesystem
Prompt > ls
Prompt > cd ..
Could not change path to ..
Prompt > ls
Prompt > exit
System exiting
student@student:~/Desktop/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
------- Command ------	- Status -
ls	ON
cd	ON
md	ON
pwd	ON
touch	ON
cat	ON
rm	ON
cp	ON
mv	ON
cp2fs	ON
cp2l	ON
Prompt > ls
Prompt >

List of shell commands that work + don't/limitations:

Commands	Status
ls	working
cd	working
md	working
pwd	working
touch	working
cat	working
rm	working
cp	working
mv	working
cp2fs	working
cp2l	working

Details - How each function works

Karina:

fs_isDir: The isDir function checks whether an entry is a directory. The function takes a pointer to a directoryEntry and returns the isDir field. If the function returns a 1, then the entry is a directory. If it is not a directory it returns a 0.

fs_isFile: On the other hand, the isFile function checks if a DirectoryEntry is a file. So if the return value is 1 then the entry is a file.

b_open: The b_open function is responsible for opening a file in the file system. It takes a filename and a set of flags that tell it how the file should be opened such as with read or write permissions. First, it tries to get an available file control block using initializedFCB( ) and if that fails the function exits early. Then b_open uses a helper function to parse the path of the file and figure out where in the directory structure the file is and decides to open the file with certain permissions such as read/write/create a new file because it didnt exist, based on the flags that the function took. If none of these permissions apply or it fails then it gets cleaned up and returns an error. But once we actually open the file, the function copies the parent directory information into the file control block and then sets up the rest of the file control block fields. The parent directory entry also gets updated on the disk to match any changes that were made by opening the file. And then lastly, b_open frees any resources that were only temporary and returns the file descriptor.

b_read: The b_read function is used to read data from a file using a file descriptor, and handles partial reads and makes sure to not go beyond the file's size. First, we check if the system has been initialized, and if not, it calls b_init() to set everything up. Then it validates the file descriptor to make sure it's within range and that the buffer has been allocated. If the file was opened as write-only, it returns -1 because reading isn't allowed. Once everything is validated, it breaks the read request into smaller parts using the partsDetermined helper function, which figures out how much data can be read from the buffer, how much should be read directly from disk, and how much might need to go into the buffer again after that.

Part A copies any leftover data already in the buffer into the output buffer.

Part B reads full blocks of data directly from disk into the user's buffer and updates the current block counter. If it reaches the end of the file's blocks, it returns early.

Part C handles any leftover bytes that don't fill an entire block. It refills the buffer and then copies those remaining bytes into the user's buffer. Lastly the function returns the total number of bytes it was able to read.

b_close: The b_close function is used to close a file that was previously opened. It takes in a file descriptor and first checks if it's valid. If the descriptor is out of range or there is no buffer allocated for it, then the function returns -1 to show that there is an error. Once it passes validation, the function gets the file control

block associated with the file. If the file was opened in a mode that allows writing, the function checks if the file is empty. If it is, the space that was allocated for it can be reclaimed. But if not, it updates the block allocation to reflect the file's actual size. And finally after handling that, the function frees the memory used by the file's buffer and clears out the fileInfo pointer.

Anshaj:

fs_setcwd:

The fs_setcwd function resets the current working directory to a new one selected by the user. This is rather like using the cd command in the Linux terminal. The function does a couple of things that are key to ensuring the directory itself exists, as well as to change to a safe one.

It begins by allocating memory space to a PathParseResult structure, which splits the path (i.e., /home/folder) into its parent directory and target folder name. This is done with the parsePath function. When the path is /, the root directory, the function simply brings everything back to root by reading it into memory and setting the global cwdPathName to just /. It then returns success.

If the path is not the root, fs_setcwd verifies if the parse succeeded and if the target directory indeed exists. In case either test fails, it returns -1, i.e., the directory does not exist or is erroneous.

Next, it tries to load the target directory off disk by invoking loadDir. This is really important — if we can't load the folder (e.g., someone deleted it), or isn't even a directory (e.g., if it is a file), then again we return an error.

Once the directory is loaded and we've ensured it's valid, we update the global cwd (current working directory) to the new in-memory location. This is done using memcpy, which copies the whole directory hierarchy.

The function then writes to cwdPathName, the entire string of the path entered by the user. If the path starts with /, it is an absolute path, so we overwrite cwdPathName. If it doesn't start with /, we assume it's a relative path and just append to the current. To clean things up, cleanPath is called to make odd cases like /folder/./anotherfolder something easier.

Finally, we call fileWrite to commit any changes to disk and free up all of the memory that we've used. As long as everything went according to plan, the function returns 0 for success.

fs_opendir:

fs_opendir is used to open a directory path to read its contents subsequently, similar to ls in Linux. It starts by parsing the given path using parsePath, which helps break down the path and find the parent directory. If the path is invalid, the function immediately frees the memory allocated and returns NULL. Once the

path is confirmed, it uses getLastElement to retrieve the name of the directory we wish to open and searches for it in the parent directory using findInDir. In case a directory is not present or is not a directory, it returns NULL again. If this is valid, the function allocates and initializes a fdDir structure, which will be used to keep track of where the directory, index, and information about each entry is once we read the directory later. It sets up the starting index as 0, calculates how many blocks the directory occupies, and allocates space for holding single entries while reading. Finally, it provides this fdDir structure, which may now serve as an argument to fs_readdir to read the directory items individually.

fs_closedir:

The fs_closedir function is used to close an opened directory stream with the one previously opened using fs_opendir. It's basically the cleanup process once you've finished reading from a directory. The function checks if the provided pointer (dirp) is NULL—if it is, it prints an error and returns 0 to indicate failure. Otherwise, it frees the memory that was taken up to contain the fdDir structure and the data used to keep track of where one stands in the directory. This keeps memory leaks and the environment in general clean. Closing a directory in this fashion is crucial with any form of file system as leaving things open leads to conflicts in memory, especially when dealing with numerous directories being accessed on a session basis**.**

b_write:

The b_write function writes data into a file in the file system. When called, it takes the file descriptor, a buffer containing data to be written, and the number of bytes to be written. It starts writing from the current offset of the file and moves the offset forward as it writes more bytes. Behind the scenes, b_write ensures that the data is correctly split up and written into the correct disk blocks. If the file is now larger than it previously was, it allocates new blocks to store the extra data. This routine is useful as it does all the low-level block management but provides a straightforward interface for writing data into files for users.

b_seek:

The b_seek function is used to move the read/write position (offset) within a file. It allows the user to jump to a different part of the file, kind of like a cursor move. It comes in handy if you need to overwrite something in the middle of a file or read from any position. b_seek takes the file descriptor, an offset value, and a reference point (like the beginning of the file, the current position, or the end). Depending on this reference point, it will calculate the new offset and establish it. It's a really handy function for allowing random access to a file instead of reading or writing merely sequentially.

Free Space:

This free space management employs a straightforward effective technique based upon a free space map, actually an array of integers that identify available disk blocks. The setup of this map is done with the initFreespace procedure

during file system initialization. It calculates the number of blocks required to contain the map itself and then initializes a linked list where each available block is chained to the subsequent one. System blocks like the Volume Control Block (VCB) and free space map itself are marked by a special signature (0xFFFFFFFF) to note that they are being used. This routine also synchronizes the VCB with metadata such as where the free space map starts, how large it is, and how much free space remains.

When a file or directory needs space, the allocateblock routine is called. It starts in the first free block (tracked in the VCB) and follows the chain of free blocks to allocate the specified amount. It puts the VCB on the new starting point for the next allocation and marks the end of the present allocation with 0xFFFFFFFF to indicate the end of the chain. This is easy to realize where the space allocated begins and ends.

On the other hand, whenever a file is deleted or no longer needs its space, the returnFreeBlocks function is used and gives back the blocks to the system. It traverses the block chain to find the last one of the sequence and connects it with the front of the free space list. Therefore, returned blocks can be reused in future allocations, an aspect that reduces wasted space. All this adds up to help the file system reuse disk space efficiently without compromising its count of available blocks or overwriting data.

Root Directory:

The root directory is the origin of the whole file system and is set up as the system boots up for the first time. In such a design, it's accomplished in the initFileSystem() and createDirectory() functions. When the disk is formatted for the first time, initFileSystem() sets up the Volume Control Block (VCB) and calls createDirectory() with a NULL parent to let it know that this new directory will become the root. Inside createDirectory(), a buffer is reserved to hold all entries in a directory. The two entries are treated specially at first: the " .\".\"" entry refers to the current directory (in this case, the root itself), and the "." entry refers to the root, since the root has no parent. They are helpful at navigation when the user goes forward and backward among directories. The position and extent of the root directory are logged in the VCB so that they can be accessed easily afterward. After it is installed, the root directory is stored to disk and then loaded into memory so that calls like fs_setcwd("/") can point to it as the initial working directory. This installation leaves a good starting directory from which other directories and files can branch off.

Ayesha:

fs_mkdir:

The fs_mkdir method creates a new directory at the given path by going through numerous steps with painstaking care to ensure correctness. It initially calls parse_path to break down the whole path into useful parts to it—i.e., it gets both a reference to the parent directory where the new folder will be placed and the name of the new folder (referred to as lastElementName). This parsing is crucial because a path /a/b/c needs to be interpreted as "create directory c in directory b," so we need to check b exists and parse c. If the path is malformed or part can't be read, the function immediately panics with an error.

After the parent directory is known, the function looks for an available slot with find_and_validate_slot. This guarantees that there is physically space in the parent directory hierarchy to insert the new record and avoids overwriting data by mistake. If the directory is already full or if the name is already in use, it fails early. The next call is to createDirectory to allocate real disk space for the new folder. This method does the low-level work of block allocation and setting up the basic structure (such as the \".\" and \".\" links). If this distribution fails—i.e., because of a full disk—the function cleans and returns an error.

After the space has been reserved, the function then completes a new directory entry (DE struct) with information—name, directory flag, timestamps, and the starting block address—by calling initialize_directory_entry. The entry is then written into the newly found free slot within the parent directory. As a final step to ensure the modification is permanent, write_parent_directory is called to write the modified parent directory onto disk.

Throughout the entire process, the purpose continues to be safety-oriented. When something goes awry—no matter whether during path parsing, slot validation, space allocation, or writing—it enters a cleanup phase that frees any memory that it may have allocated (for example, parent directory buffer or parse results) before it returns failure. It returns the outcome of the most recent disk write upon successful run. The construction prevents memory leaks, illegal states, and part-way updates from impacting the filesystem even on failures.

fs_getcwd:

The fs_getcwd function is used to get the path of the present working directory in our own file system. It does not invoke the Linux standard getcwd() system call since we internally maintain a global variable called cwdPathName. This always has the full path of wherever the user currently happens to be in the file system, e.g., /folder/subfolder. When someone calls fs_getcwd, we merely copy that cached path into the pathname buffer using strncpy so it does not overflow. It also returns the same path afterwards, which can be useful if someone would like

to use the result directly. This operation is convenient when, say, the shell needs to show the location of the user or if code needs to build a whole path to read/write files. It's a necessary but light-weight helper that makes path management easier.

fs_delete:

The fs_delete function deletes a file from the custom file system safely by erasing its metadata and storage. It begins by parsing the filename path provided by calling parse_path, which returns a structure containing the parent directory and index of the file to be deleted. The parsing is done to ensure the function has a correct understanding of both what file and where in the filesystem tree it is to be deleted. If parsing fails or the file doesn't exist (i.e., lastElementIndex < 0), the function returns an error immediately, indicating the file doesn't exist. Having located a good file entry, the function performs a very critical validation step: it checks if the entry is a directory. If so, the function refuses to delete it with fs_delete, since directories must be deleted using fs_rmdir to ensure safety and emptiness checks.

Having assumed that the entry is a file, the function now determines whether it occupies any storage space (i.e., size > 0) and calls returnFreeBlocks to release any disk blocks assigned. This is done to prevent resource leaks in the filesystem. If this block cleanup activity fails, it returns an error again. If everything goes smoothly, the function flags the file as deleted by setting its

location to -2, a convention the codebase follows for representing an invalid or deleted entry. It then calls write_parent_directory to write the updated parent directory (without the deleted file) back to disk. All memory that was dynamically allocated during the process is freed before the function returns to avoid memory leaks. Overall, fs_delete provides a cautious, step-by-step deletion process that ensures only valid files are deleted and that file system consistency is preserved after deletion.

fs_rmdir:

The fs_rmdir function removes a directory in the file system, but it will only be successful if the directory already exists and is not empty. In other words, just like the rmdir command works in Linux: you cannot delete a folder unless the folder is empty.

The function starts by calling parse_path, which analyzes the path provided (like /docs/notes) and breaks it into components. This allows the function to find the parent directory and identify the target folder (notes) that the user wants to delete. If this parsing fails (for example, if the path doesn't exist), the function exits early and returns an error.

It then checks whether the target is a directory using the isDirectory flag. If it is not a directory-for example, if it is a regular file-it returns an error again, this time because files should be deleted using fs_delete, not fs_rmdir.

Then it populates the contents of that folder with loadDir. This fills the folder's entries into memory from disk. Before deleting, the function checks if the folder is empty, that is, contains only "." and "..". This is done through isEmpty() helper. If there is even a single actual file or subfolder within, deletion is not allowed to avoid data loss.

If the folder isn't empty, returnFreeBlocks() is used to release the blocks that were being used on disk by that directory. Then the function marks the directory as "deleted" by setting its location field to -2. This is a special value used in your project to indicate a file or folder is deleted.

Then, the new parent directory is written to disk with write_parent_directory, and all memory utilized (for parsing and loading) is released to prevent leaks. If all goes well, the function returns success (typically 0); otherwise, -1 is returned to indicate failure.

Wilmaire:

Volume Control Block (VCB):

The Volume Control Block (VCB) is a data structure used to store data about the file system. It holds the essential information needed for the operating system to understand the layout and structure of the disk. While in most systems the VCB resides in block 0, in our implementation it is purposely placed in block 1. The VCB contains various variables like our magic number (signature), block size, free space, root directory, and more. During initialization, the system reads block 1 and checks for the magic signature. If it's incorrect or not present, it signals the file system that it is unformatted.

fs_stat:

This function acts as a metadata inspector, its sole purpose is to extract and return essential information about a file or directory. This function, fs_stat, begins by using parsepath() to break down the provided pathname and locate its directory entry that contains the target file or directory. Once this happens, it identifies the final element and locates it within its parent directory using findInDir(). After verifying this, it constructs a structure and populates it with metadata from the directory entry. This function is not involved in actual data access, it is purely read, but it is critical for any features that depend on it.

fs_readdir:

This function serves as the iterator of the system, it allows the file system to traverse through a directory's contents, one entry at a time. When a directory is opened, the function is repeatedly called to get each valid entry, skipping unused

or deleted records along the way. The function begins by calculating how many blocks the directory uses and then loads the entire directory into memory using fileRead(). It then maintains an index (dirp->index) that tracks where we are in the stream and skips the entries marked with the sentinel “-2L.” It’s essential for commands like ls, and its integration with structures like fdDir and DE.

Contribution Table:

Component	Contributor
VCB	Wilmaire
Root Directory	Anshaj + Wilmaire
Free Space	Anshaj + Wilmaire
fs_mkdir	Ayesha
fs_setcwd	Anshaj
fs_getcwd	Ayesha
fs_isFile	Karina
fs_isDir	Karina
fs_opendir	Anshaj
fs_readdir	Wilmaire
fs_closedir	Anshaj
fs_stat	Wilmaire
fs_delete	Ayesha
fs_rmdir	Anshaj+Ayesha
b_open	Karina + Anshaj
b_write	Anshaj
b_read	Karina
b_seek	Anshaj
b_close	Karina
Write Up	Everyone

Screenshots showing each of the commands listed in the readme:

ls

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o path.o path.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o path.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

b_io.c
bbb
filesystem
Prompt >
```

md

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

b_io.c
bbb
filesystem
Prompt > md test1
Prompt > md test2
Prompt > ls

b_io.c
bbb
filesystem
test1
test2
Prompt >
```

mv and pwd

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

b_io.c
bbb
filesystem
test1
test2
Prompt > mv test1 test2
Prompt > ls

b_io.c
bbb
filesystem
test2
Prompt > cd test2
Prompt > ls

test1
Prompt > cd test1
Prompt > pwd
/test2/test1/
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```

touch

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls

b_io.c
bbb
filesystem
test2
Prompt > touch test.txt
Prompt > ls

b_io.c
bbb
filesystem
test.txt
test2
Prompt > rm test.txt
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```

rm

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls

b_io.c
bbb
filesystem
test2
Prompt > rm bbb
Prompt > ls

b_io.c
filesystem
test2
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```

cp2fs and cp2l and cat

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ make
make: 'fsshell' is up to date.
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > cp2fs mfs.h
Prompt > ls

b_io.c
mfs.h
filesystem
test2
Prompt > cat mfs.h
/**************************************************************
* Class::  CSC-415-0# Spring 2024
* Name::
* Student IDs::
* GitHub-Name::
* Group-Name::
* Project:: Basic File System
*
* File:: mfs.
h
*
* Description::
*       This is the file system interface.
*       This is the interface needed by the driver to interact with
```

```
// Key directory functions
int fs_mkdir(const char *pathname, mode_t mode);
int fs_rmdir(const char *pathname);

// Direct
ory iteration functions
fdDir * fs_opendir(const char *pathname);
struct fs_diriteminfo *fs_readdir(fdDir *dirp);
int fs_closedir(fdDir *dirp);

// Misc directory functions
char * fs_getcwd(char *pat
hname, size_t size);
int fs_setcwd(char *pathname);   //linux chdir
int fs_isFile(char * filename); //return 1 if file, 0 otherwise
int fs_isDir(char * pathname);          //return 1 if directory, 0 otherwis
e
int fs_delete(char* filename);  //removes a file
int fs_mv(const char* startpathname, const char* endpathname); //moves a file
int calculateFormula(int i, int j);
char* getLastElement(const char* pa
th);


// This is the strucutre that is filled in from a call to fs_stat
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_crea
tetime;         /* time of last status change */

        /* add additional attributes here for your file system */
        };

int fs
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```

```
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ ls
b_io.c  b_io.o        fsInit.c  fsInit.o  fsLowM1.o  fsshell    fsshell.o  Makefile  mfs.h  path.c  path.o     SampleVolume
b_io.h  dEFunctions.c  fsInit.h  fsLow.h   fsLow.o    fsshell.c  Hexdump    mfs.c     mfs.o  path.h  README.md
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ 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
Disk already formatted
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md test.txt
Prompt > ls

b_io.c
mfs.h
filesystem
test.txt
test2
Prompt > cp2l test.txt
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Karina-Krystal$ ls
b_io.c  b_io.o        fsInit.c  fsInit.o  fsLowM1.o  fsshell    fsshell.o  Makefile  mfs.h  path.c  path.o     SampleVolume
b_io.h  dEFunctions.c  fsInit.h  fsLow.h   fsLow.o    fsshell.c  Hexdump    mfs.c     mfs.o  path.h  README.md  test.txt
student@student:~/Documents/csc415-filesystem-Karina-Krystal$
```