

Assignment- File System

Team - Coney Island

Akim Tarasov - 922761746
Devarsh Hirpara - 922991898
Diego Antunez - 922061514
Gerious Heishan - 922559089
Zari Haidarian - 917423031

GitHub Repo Owner: AkimT13

GitHub Link:

<https://github.com/CSC415-2025-Spring/csc415-filessystem-AkimT13.git>

Description:

Designing a file system is putting different pieces together in collaboration with one another to work, and there are a number of important features such as memory allocating, space maintaining, and tracking finding data blocks. In our situation, for this project, the job is running a volume control block, managing available space, providing shape to the directory entry structure, and announcing what types of metadata our file system will hold. Each of these components and their design and functionality will be described in great detail in the sections that follow.

VCB- The Volume Control Block or the VCB consists of metadata about volume. It has many pieces of data including a signature to confirm the volume, the total number of blocks available, as well as the total number of free blocks available. It also includes information about the free space map such as where it starts from, its size, and the root location. All the data is stored in a specific order to prevent unnecessary padding. This struct helps us track and allocate space for directories and files in our file system.

Free Space- We decided to use a simple bitmap for our free space system. Each bit represents a block. If the bit is 0 that block is free, if it is 1 then it is used. The `initFreeSpace` function is called when the vcb's function signature doesn't match and will return the starting block of the free space. We used `calloc` to allocate memory for the freespace because `calloc` initializes every bit to 0. If `malloc` was used, every char would be uninitialized so setting it to 0 when allocating prepares for eventual bit flipping. Similarly to the lectures and instructions, we chose 5 blocks to be the size of the bitmap which will manage 40 blocks. We will increase this in the future. In order to flip the bits, we used a bit mask where the first 6 bits are 1. This is then OR'd to the first byte of the freespace in order to mark the first 6 blocks as used. This is because the VCB is the first block and the freespace is the next 5 blocks. `LBAWrite()` is then called in order to have the free space map be persistent. `allocBlocks` will take in a count that represents how many blocks the caller wants to use. It will then try to find that number of consecutive blocks within the freespace. `currBlock` tracks which bit we are checking and currently tracks how many consecutive free blocks we have found. To check if a block is free, we determine the corresponding byte and bit position using $\text{byteIndex} = \text{currBlock} / 8$ and $\text{bitIndex} = \text{currBlock} \% 8$, since each byte contains 8 bits. We create a bit mask by shifting 1 by `bitIndex` positions. This results in a number where all the bits are 0 except the bit we are interested in checking. This can then be AND'd with the byte in the bitmap to see if it's free. If the result is 0, then the block is available. Once `curr` hits the desired count, we need to mark all of those bits as used and return the where that consecutive sequence starts. `LBAwrite` is then called to save the new bitmap.

Directory- The Directory struct (which we named `File`) consists of metadata about the free space. We used a bitmap to allocate memory based on the number of blocks needed to cover the disk's total blocks. Every bit in `freeSpaceMap` represents a block's availability. It has many pieces of data including the name, permissions, location, size, and date modified/created. We

also included data about whether it is a directory or not, and whether it is free or not. All the data is stored in a specific order to prevent unnecessary padding.

Approach:

Akim: The `parsepath` function takes in a pathname and populates a structure that contains important metadata that is required for many of the `fs` functions and a few of the `b_io` functions. Most of the implementation was shown during a class lecture, but I made a few modifications upon facing a few issues that will be described in the issues section. The function starts by determining if it is an absolute path or relative. You can tell if the pathname is absolute because it must start with `'/'`. It then uses `strtok_r` to split the path by its elements separated by slashes. It checks if each element is there by using `findInDirectory`. In either case, whether the parent is root or the parent is the current directory, both are already loaded into memory and can be passed in `findInDirectory`, which just iterates through the 50 possible directory entries inside of a given directory. If a final is not found it returns -2 and tells functions such as `mkdir` that they should have something new in that path. -2 is also used for the index if it's the root. If the current parent isn't the original (root or `cwd`), the previously loaded parent directory is freed to prevent memory leaks. The function fills out the passed in `pplInfo` struct and contains things such as the name, index in parent directory, pointer to the parent directories contents, and the parent starting block number which is used for LBA read or write.

Akim: `fs_isfile` pretty much just checked the loaded in directory and checks the `isFile` field in the `File` struct to see if its a file. This function was more of an after thought and I ended up just checking if it was a file directly in the other `fs` functions rather than calling this one.

Akim: `fs_setcwd` is called whenever the user changes directories using `cd`. It utilizes a file called `globals.C` which holds pointers to directories and important information that is shared across the program. On start up, the current directory is set to the root. As the user changes directories, you either append the name of the directory they go into, or you pop off the last element if they go up to the parent using `../`. This behavior is enforced using a stack and the function called `fix path`. The stack can be thought of as the result of popping elements off if its `../` and pushing elements on when it's some sort of valid name.

Akim: `fs_getcwd` just returns whatever the currentWorkingDirectory is in `globals.c`.

Akim: The missing part of the freespace from Milestone 1 was the `releaseBlocks` function. It's meant to flip the used bits in the freespace bitmap back to 0 which indicates those blocks are allowed to be overridden. It's essentially the inverse of the `allocBlocks`. Instead of OR'ing bits with a bitmask I instead have to AND the portion I want flipped back to 0. It calculates which bit and byte in the freespace map corresponds to the block we want to free and uses a bitwise AND with this bitmask $\sim(1 \ll \text{bitPos})$ to clear the bit. This video and this article gave tricks on how to

deal with flipping particular bits within particular bytes.

<https://youtu.be/LZ09TOAp4Is?si=s3GJh99PyrXc3npx>

<https://leetcode.com/discuss/post/3695233/all-types-of-patterns-for-bits-manipulat-gezp/>

Akim (makeFile): Make file is similar to mkdir except that it instead creates a file. It starts by calling parse path, if -2 is returned then it will exit early to avoid duplicate file creation. To determine the correct parent directory, it uses strrchr to split it by the last slash. This allows it to reparse the parent path and corresponding directory block and File struct in memory. Then it will populate pinfo with the correct parent directory and filename. Next, the function will search for the next empty slot within the directory. Once the next available slot is found, it will use allocBlocks to allocate a single block for the file. It also initializes its other fields. The updated parent directory is written to disk with LBA write.

Zari (fs_stat): This function gets metadata about a file or directory at the specified path (such as its size, creation time, and last modification time). The information is stored in the provided fs_stat structure. My process when writing this function was to first go over the fs_stat struct and see what metadata is being saved. When I realized the data was similar to what our File struct already contained, I was able to easily create the function to save the metadata.

Zari (fs_delete): This function deletes the specified valid file by marking it as free in the file system, resetting its size, location, and modification time. My approach to creating this function was to plan out what information I needed to check to confirm the file name was valid, and then plan out what information I needed to free/clear so the file would be considered deleted.

Zari (fs_rmdir): This function removes the specified directory from the file system, assuming it is empty. Like fs_delete, it marks the directory as free and updates its metadata. My approach to creating this function was the same as my approach to creating fs_delete, once I realized that the steps to removing a file and a directory are the same.

Zari (cmd_mv): This function implements the mv (move) command. It does so by copying the contents of the source file to the destination file and deleting the original source file afterward. My process for working on this one was to first do some research and planning on how I was going to move the files. I initially tried to use b_seek but had issues with using it, so then I ended up using b_read and b_write instead to rewrite the file into the new location I wanted to move it to.

Zari (b_open): This function opens a buffered file using the given filename and access mode (O_RDONLY, O_WRONLY, O_RDWR). This one and b_close are both needed to access any file in the file system. My approach to working on this function was to look back on assignment 5 and

make sure I include all the necessary steps needed to open a file for reading or writing purposes.

Zari (b_close): This function closes the buffered file associated with the file descriptor fd. It frees any internal resources and invalidates the descriptor. This one and b_open are both needed to access any file in the file system. My approach to working on this function was the same as for b_open, which was to look back on assignment 5 and make sure I include all the necessary steps needed to close a file and free any necessary values.

Diego (fs_isDir): For this function, I needed to determine whether a given path represents a directory. I started by calling parsePath(), which populates a pplInfo struct containing a pointer to the parent directory and an index for the target entry. If the path was /, parsePath sets the index to -2, so I returned 1 since root is always a directory. If the index was -1, the path didn't resolve to any existing entry, so I returned 0. Otherwise, I used the index to access the correct entry in the parent directory array, and returned whether that entry's isDirectory field was set to 1. This function mainly acts as a helper for other parts of the file system to know what kind of path they're working with.

Diego (fs_mkdir): For this function, it is used to create a new directory at the specified path and link it properly. My approach was to validate the path first using parsePath(), then manually split it to extract the new directory name and resolve the parent path. After parsing the parent, I allocated a block for the new directory and initialized it with 50 empty entries, setting the . and .. entries to link the directory to itself and its parent. Then I updated the parent directory with a new file entry pointing to the new block and wrote both directories to disk. This was to make sure it was consistent with how root was initialized and allowed for better navigation with the relative paths.

Diego (fs_readdir): This function is used to return one directory entry at a time from an open directory enabling iteration through its contents. My approach was to iterate through a directory's entries starting from the position stored in fdDir. I used a while loop to scan for valid entries (free == 1), and when found, I filled the fs_dirent struct with the entry's name, type, and size before returning it. I increment the position after each read so it keeps moving forward. This made it easier to keep track of where we are when reading a directory one item at a time.

Gerious: I think this shows how important it is to keep track of free space on a disk in a file system. The code does this by using a bitmap, where each bit represents a block on the disk—0

means the block is free, and 1 means it's already used. This results in a simple and efficient way to manage storage. The `allocBlock()` function helps by finding one free block, marking it as used, and returning its block number. This is important because when the system needs to store a new or small file, it can quickly find an available space to use. Also the `releaseBlocks(start, count)` function frees up a group of blocks, making them available again. I think this is important because when a file is deleted, those blocks shouldn't stay marked as used they need to be released so the file system can reuse that space for new data.

Devarsh: For `b_write`, I implemented a buffered writing strategy that stores data in a temporary buffer within the File Control Block (FCB). When writing data, it's stored in this buffer until it's full to the defined chunk size (`B_CHUNK_SIZE`). When the buffer is full, it's written to disk using the low-level `write()` system call. This way, we reduce the number of direct disk writes and improve performance. I also ensured that partial writes and data that do not fill the entire buffer are handled correctly and stored until a future flush or file close.

Devarsh: In the `b_read` function, I implemented the read in three stages. Firstly, if there is leftover data in the FCB buffer from the previous read, it is copied into the user buffer. Then, if more data is needed, the function reads full blocks from the file directly into the user buffer, minimizing intermediate copies. Then, for any remaining bytes that don't make up a full block, the buffer is refilled from disk and the last portion copied over. This layered approach allows read efficiency as well as accommodating multiple read sizes and partial buffer usage.

Devarsh: The `b_close` operation offers cleanup and final data processing when a file is closed. Any data remaining in the write buffer is written to disk before the file is actually closed. This makes sure that no unwritten data is forfeited. I also included code to free the buffer memory and zero the FCB structure so that the file descriptor slot can be reused later on. This function plays an essential role in file system integrity and resource management.

Devarsh: For the `cmd_mv` command, I went with offering file moving and renaming functionality by doing the input argument validation and checking if the source file exists and the target does not. Since there wasn't a rename operation function directly available (`fs_rename`), I did it using a copy-and-delete method to simulate the move behavior. I then opened the source file in read mode and destination in write mode, copying data block by block using `b_read` and `b_write`. After copying, I closed both the files and `fs_delete` the source file. Appropriate error handling and cleanup were used to accommodate partial writes or failures at any point to ensure consistency and protect against data corruption.

Akim (`fs_opendir`) : This is responsible for opening a directory and preparing to read its contents. It starts by using a parse path to populate the `pplInfo` struct. It then loads it into memory using `loadDirectory`. It allocates and populates an `frDir` structure using the data of the directory.

Issues and Resolutions:

Akim: I noticed an issue where the `initFileSystem` function kept printing out “signature not matching, volume uninitialized” even though the volume had already been initialized. We could tell that the signature did in fact match because the hexdump clearly showed our signature in little endian form at the top. The `vcb` struct had the signature defined as an unsigned long, but I noticed typedefs in `mfs.h` that showed `uint32_t` integers. I decided to print out versions of the signature that were of either type and found that using `uint32` yielded the correct result. This also could have had something to do with padding of the `vcb` struct given the use of unsigned longs.

Akim: For some reason as I was testing directory creation, there could only be at max like 5 directories before the freespace system said there were no more blocks to allocate. My first thought was to check the `initFreeSpace` function and check how many blocks I was actually managing. It turns out it wasn't covering the entire volume. I fixed this by carefully checking the math of block calculation within `freeSpace.c`. In the `allocBlocks()` function I incorrectly calculated the total number of bits that represent all of the blocks of the disk. I was only doing $5 * 8$ total bits, probably for testing purposes for milestone 1, rather than the correct $512 * 8 * 5$. After changing that, I could make a bunch of directories and not run out of free space.

Zari: I had the most issues during the process of implementing the `cmd_mv()` function. My first attempt I tried using only `b_seek()` to move the file, but I was getting errors and issues with it and it wouldn't move the file. I did some research and realized that I could try rewriting the file to the correct location that I wanted to move it to, so I switched my approach to using `b_read()` and `b_write()` instead. At first that was also giving me issues as well. I got errors that my file wasn't being closed or opened properly, and I implemented more checks within the move function to exit if there are certain errors. I tried using `parsePath` to validate the file name, but then switched to the function `is_file()` instead. Eventually I was able to make it rewrite the file in the correct location and safely delete the old version.

Gerious: I noticed that some of the commands weren't working when I tried them in the terminal, and I think this shows that those commands haven't been implemented yet. The menu at the start listed which ones were ON and OFF, so it makes sense that commands like `touch`, `cat`, and `rm` are marked off. Also, This is important because it helps me understand which parts of the file system are working and which parts still need to be built. It also saved me time from thinking there was a bug or if the features are complete or not.

Diego: An issue I ran into while doing `fs_mkdir` was getting the new directory to actually link with `..` so that the relative navigation would work. At first, I was setting `..` to block 6 root for

every new directory by default, but this caused errors when creating subdirectories inside other folders meaning they would all think root was their parent. I realized that I needed to dynamically grab the correct parent block by using `currentDirectoryStartBlock` during creation. Once I fixed that and made sure `..` pointed to the real parent, it was fixed.

Diego: While testing `readdir`, I kept getting null results even though I knew the directory had entries. At first I thought the issue was with how I was looping, but it was actually that I wasn't accessing the correct in-memory copy of the directory. I first used `currentDirectory`, but since `fs_opendir` stores the directory in the `fdDir` struct, I had to use `dirp->directory` instead to fix the issue.

Akim: I found an issue where I struggled finding a way to reliably persist modified directory data in memory back into the disk. `ParsePath` returns a pointer to the parent in memory, but I wanted an easier way to figure out where to write it back on disk. So I added a `parentStartBlock` field to `pplInfo` which holds wherever the parent start block is. This addition saved a lot of headache where I would modify a directory, run `ls` to view the results, and not have the change be shown.

Akim: Something I overlooked was the fact that the vcb only gets written to disk when its first initialized. However during the running of the system, important variables of the vcb are changed such as the number of free blocks available. I made a function called `persistVcb()` that runs during moments where the vcb gets updated, ensuring that an accurate vcb gets loaded during the next session.

Devarsh: When writing the `b_write` function, I was facing an issue in which data was not writing out as it ought to when the buffer got filled up. Initially, I considered the data getting lost due to the write call itself, but after going deeper into the FCB structure, I noticed that the buffer was never written to disk unless a file got closed. I fixed this by adding logic to check if the buffer had filled up to the chunk size (`B_CHUNK_SIZE`) during writes, and if it did, causing it to be flushed out by writing to the underlying file descriptor manually. This wrote data in a piecewise manner as the buffer became full, instead of just the end.

Devarsh: I also noticed that `b_close` wasn't flushing the remaining data in the buffer before closing the file, so the last piece of data would be lost if the buffer wasn't full when the file was closed. I corrected this by adding a check in `b_close` to flush the buffer one additional time if there were any bytes remaining (`index > 0`) before freeing memory and clearing the FCB.

Devarsh: In addition to experimenting with edge cases, I also added defensive checks to both functions to check whether file descriptors were valid and initialized. This avoided crashes when a user tried writing to or closing a file that had not been opened. Both `b_write` and `b_close`

operated reliably in all of the test cases we utilized, like writing huge files, appending to a file, and closing a file in the middle of an operation, after such changes were implemented.

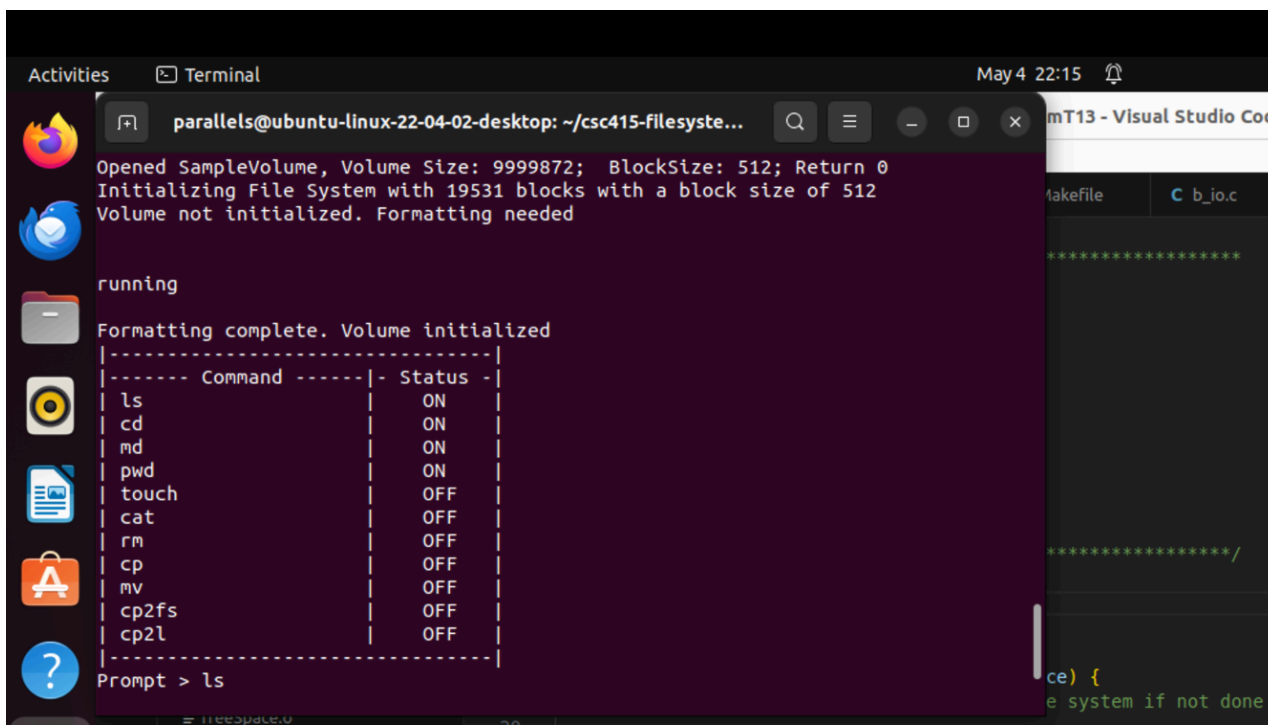
Devarsh: One issue I encountered when using the `cmd_mv` command was that we lack a built-in `fs_rotate` function in our file system, which we would be able to simply rename. I circumvented this by using a copy-then-delete mechanism. Another issue was to avoid partial writes during copying or reading errors from producing corrupted or incomplete destination files. For that, I conducted error checks immediately after each read and write, and on failure, took special care to close open file descriptors and delete an in-progress destination file to maintain file system consistency. This turned out to be an effective move command without direct support for rename.

Limitations and future development:

One key limitation of our filesystem is the use of consecutive block allocation using a bitmap. This ensures files and directories occupy a contiguous region on the disk, simplifying the process of reading and writing. However it becomes restricted over time as more entries are written and deleted from the file system. As more blocks are freed, free space becomes scattered, making it difficult for `allocBlocks` to find a contiguous region to allocate even if there's "technically" enough blocks. There is also a hard limit on how many directory entries a directory can have, 50. A more robust system with extents would probably be able to handle an indefinite amount of blobs.

Screenshot of compilation:

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ make clean
rm fsshell.o fsInit.o freeSpace.o initRoot.o globals.o workingDirFuncs.o zari_m2
.o emptyTestFuncs.o misc.o b_io.o fsshell
student@student:~/Desktop/csc415-filesystem-AkimT13$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freeSpace.o freeSpace.c -g -I.
gcc -c -o initRoot.o initRoot.c -g -I.
gcc -c -o globals.o globals.c -g -I.
gcc -c -o workingDirFuncs.o workingDirFuncs.c -g -I.
gcc -c -o zari_m2.o zari_m2.c -g -I.
gcc -c -o emptyTestFuncs.o emptyTestFuncs.c -g -I.
gcc -c -o misc.o misc.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freeSpace.o initRoot.o globals.o workingDirFun
cs.o zari_m2.o emptyTestFuncs.o misc.o b_io.o fsLow.o -g -I. -lm -l readline -l
pthread
student@student:~/Desktop/csc415-filesystem-AkimT13$
```



Activities Terminal May 4 22:15

parallels@ubuntu-linux-22-04-02-desktop: ~/csc415-filesyste...

Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume not initialized. Formatting needed

running

Formatting complete. Volume initialized

Command	Status
ls	ON
cd	ON
md	ON
pwd	ON
touch	OFF
cat	OFF
rm	OFF
cp	OFF
mv	OFF
cp2fs	OFF
cp2l	OFF

Prompt > ls

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

ls command

```
student@student:~/Desktop/csc415-filesystem-AkinT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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

testfolder1
testfolder2
Prompt > █
```

cd command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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
test
testfile
Prompt > cd
Usage: cd path
Prompt > cd somefolder
Could not change path to somefolder
Prompt > cd test
Prompt > 
```

md command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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
test
testfile
Prompt > md newfolder
Prompt > ls
test
testfile
newfolder
Prompt > 
```

pwd command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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 > pwd
/
Prompt > cd newfolder
Prompt > pwd
/newfolder
Prompt > cd anothernewfolder
Prompt > pwd
/newfolder/anothernewfolder
Prompt >
```

touch command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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

test
testfile
newfolder
Prompt > touch newfile
Prompt > ls

test
testfile
newfolder
newfile
Prompt >
```

cat command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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 > cat myFile.txt
test data
Prompt > 
```


rm command

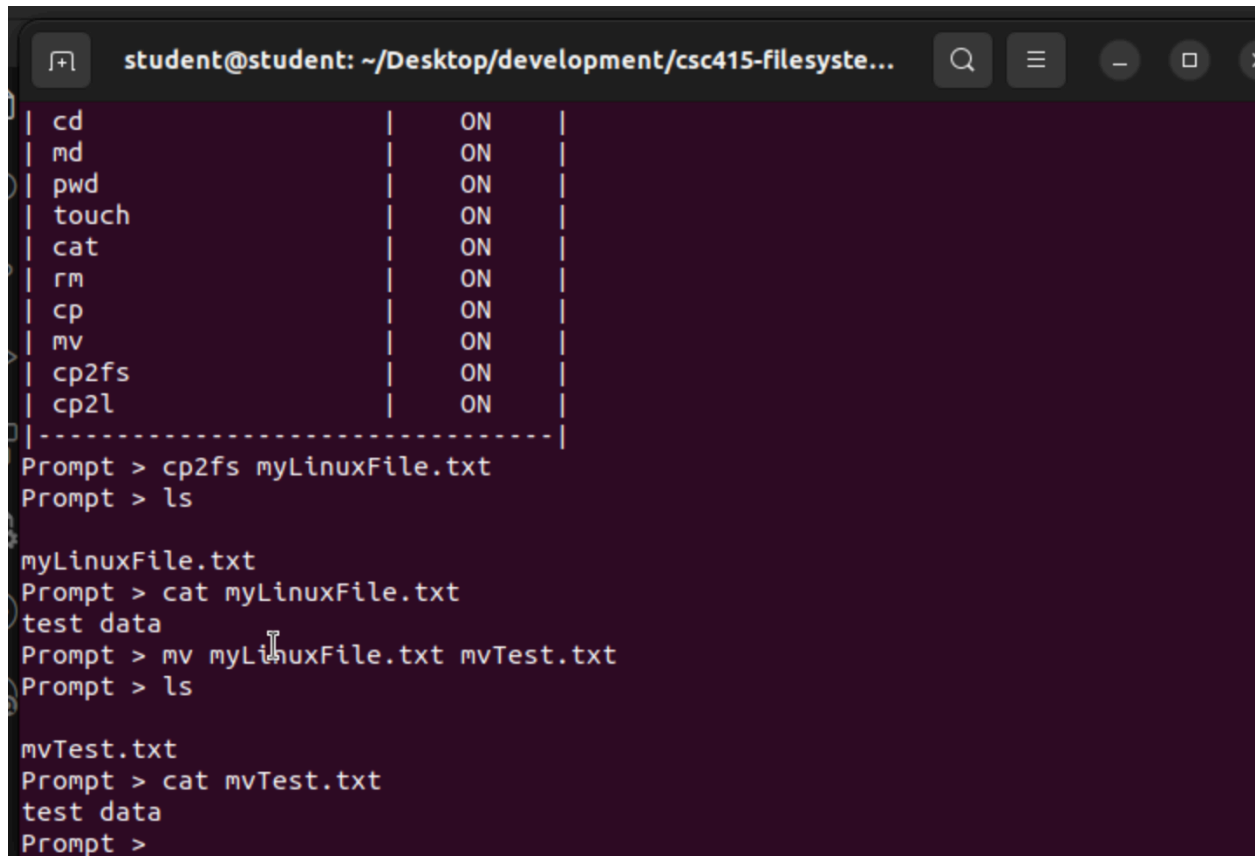
```
student@student:~/Desktop/csc415-filesystem-AkimT13$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already initialized
loading fsm
Loading free space map into memory|-----|
|----- 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
test
testfile
newfolder
newfile
myFile.txt
Prompt > rm testfile
Prompt > ls
test
newfolder
newfile
myFile.txt
Prompt >
```

cp command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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

test
newfolder
newfile
myFile.txt
Prompt > cp myFile.txt newfile
Prompt > cat newfile
test data
Prompt > cat myFile.txt
test data
Prompt > 
```

mv command



A terminal window titled "student@student: ~/Desktop/development/csc415-filesyste..." with a search bar and window controls. The terminal shows a list of commands and their status (ON) separated by vertical dashed lines. Below this, a series of commands are entered at a "Prompt" shell:

```
| cd | ON |  
| md | ON |  
| pwd | ON |  
| touch | ON |  
| cat | ON |  
| rm | ON |  
| cp | ON |  
| mv | ON |  
| cp2fs | ON |  
| cp2l | ON |  
-----|  
Prompt > cp2fs myLinuxFile.txt  
Prompt > ls  
  
myLinuxFile.txt  
Prompt > cat myLinuxFile.txt  
test data  
Prompt > mv myLinuxFile.txt mvTest.txt  
Prompt > ls  
  
mvTest.txt  
Prompt > cat mvTest.txt  
test data  
Prompt >
```

cp2fs command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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

folder1
myFile.txt
Prompt > cp2fs linuxToFS.txt
Prompt > ls

folder1
linuxToFS.txt
myFile.txt
Prompt >
```

The image shows a terminal window and a code editor. The terminal window displays the output of a formatting process and a list of commands and their statuses. The code editor shows the contents of a file named myLinuxFile.txt.

Terminal Window:

```
student@student: ~/Desktop/development/csc415-filesyste...
Formatting complete. Volume initialized
-----
----- Command ----- | - Status - |
| ls                     |           |
| cd                     |           |
| md                     |           |
| pwd                    |           |
| touch                  |           |
| cat                    |           |
| rm                     |           |
| cp                     |           |
| mv                     |           |
| cp2fs                  |           |
| cp2l                   |           |
-----
Prompt > cp2fs myFile.txt
Prompt > ls
myFile.txt
Prompt > cat myFile.txt
test data
Prompt > cp2l myFile.txt myLinuxFile.txt
Prompt >
```

Code Editor:

myLinuxFile.txt u x C zari_m3.c 7 C b_io.c C b_io.c (Working Tree) C mfs.h 1

```
myLinuxFile.txt
1 test data
2
```

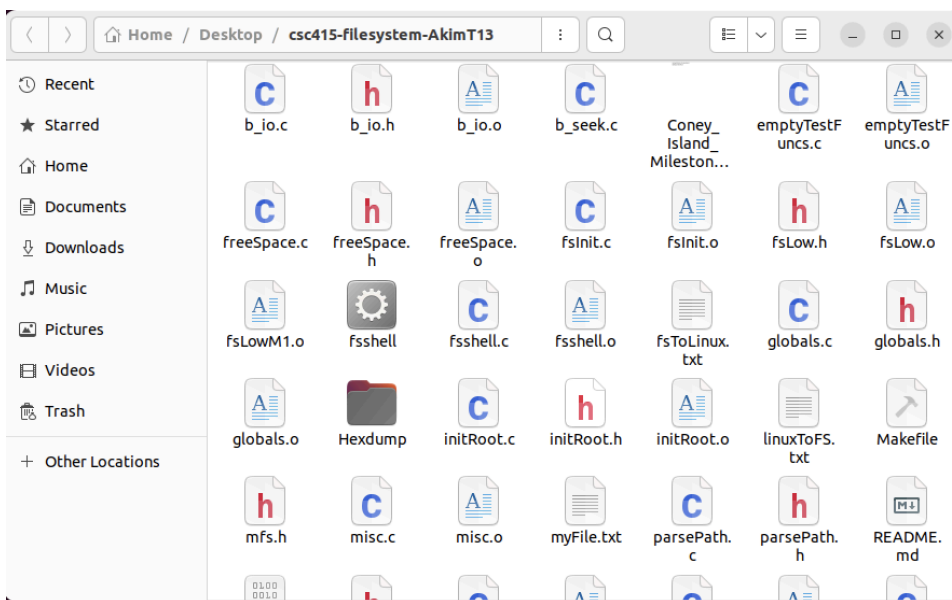
Terminal Window (Bottom):

```
md
pwd
touch
cat
rm
cp
```

cp2l command

```
student@student:~/Desktop/csc415-filesystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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

folder1
linuxToFS.txt
fsToLinux.txt
myFile.txt
Prompt > cp2l fsToLinux.txt
Prompt >
```



help and exit commands

```
student@student:~/Desktop/csc415-filessystem-AkimT13$ 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
loading fsm
Loading free space map into memory|-----|
|----- 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 > help
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 > exit
System exiting
student@student:~/Desktop/csc415-filessystem-AkimT13$
```