

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran

CSC415 Operating Systems

File System Project - The Ducklings

Description:

In this project, we were tasked with implementing a basic file system capable of simulating essential file operations such as `md`, `rm`, `cd`, `pwd`, and more. The first phase involved formatting the volume, which included initializing the Volume Control Block (VCB), setting up the free space management system, and creating the root directory. In the second phase, we implemented core directory-related functions such as `fs_setcwd`, `fs_getcwd`, `fs_isFile`, `fs_isDir`, `fs_mkdir`, `fs_opendir`, `fs_readdir`, `fs_closedir`, and `fs_stat`, among others. Finally, in the last phase, we implemented file-level operations and commands. Our file system project is composed of three key files that connect user interaction, file system logic, and low-level I/O:

- `fsshell.c` handles user input and command dispatching,
- `mfs.c` implements the core file system logic
- `b_io.c` provides buffered low-level file I/O functionality.

Approach:

Phase 1

The three parts of this phase were split among the team, where Ranj and Juan are in charge of the Volume Control Block, Tybo for free space, and Julia and Eugenio for the directory system.

The Volume Control Block will contain its signature, volume size, total blocks, root directory block, and the starting block of free space, along with its size and its head. The signature is here to ensure the VCB is not initialized again if it is already initialized. The volume size and total blocks are here to ensure that the volume does not extend beyond its limit, while the free space size and free space start block are here to keep track of free space. As for the root directory, this is here so that it can be called upon by any function that requires it, which also applies to the free space head.

The Free Space Structure we plan to employ is a FAT table, as we believe it will be the easiest to manage while also providing everything we need to keep track of free space. In this approach,

Ty Bohlander	ID: 922436005
Eugenio Ramirez	ID: 923053570
Julia Bui	ID: 923518762
Juan Ramirez	ID: 922495948
Ranjiv Jithendran	ID: 922694185
Github: Tybo2020	CSC415 Operating Systems

each block in the volume will have a corresponding entry in the FAT, which will indicate its current status. Whether it is free, in use, or part of a specific file's chain. This method allows for quick identification of available blocks, efficient space allocation, and straightforward tracking of file block sequences.

Our Directory System will make use of a structure called `DirectoryEntry`. This structure will contain the name of the file, its permissions, its size, the starting block, the time of its creation along with the last time it was accessed and modified, a bool to determine if the file is a directory, and another boolean that determines if it is in use. We think that the permissions, file size, start block, and boolean that determines if it is a directory are all important pieces of information needed for a file, while the three variables dealing with time and the filename are information that while not as important, still seem useful to have. As for the boolean that determines if the entry is in use, it is supposed to be an alternative way to track entry status.

Phase 2

The 11 functions needed for this phase are split among the team, where Ty is in charge of `getcwd`, `setcwd`, and `opendir`, Eugenio is in charge of `mkdir`, Julia is in charge of `fs_stat` and `closedir`, Juan is in charge of `isFile`, `isDir`, `fs_delete`, and `rmdir`, and Ranj is in charge of `loadDir`.

For `getcwd`, it will just check if the path name is valid before returning the pathname. This will be done to make sure the function returns a directory that actually exists rather than something else that could cause errors later on.

For `setcwd`, it will read the desired directory from the file system in order to move the user to that directory. This also means that the desired directory will have to be checked for its validity, as reading a directory that does not exist would cause problems. The last thing this function would need to do is update the current path for the user, which means the path to the directory should be presented in a clear way. While there is no technical reason for this requirement, showing a clean path to the user is preferable than showing a messy one.

For `opendir`, it will find the desired directory entry in the file system, making sure to check if it is a directory to avoid errors. In order to open the directory, it will load it. The loaded directory will also be put in a directory structure, which contains other information that will be needed for `closedir` and `readdir`.

Ty Bohlander	ID: 922436005
Eugenio Ramirez	ID: 923053570
Julia Bui	ID: 923518762
Juan Ramirez	ID: 922495948
Ranjiv Jithendran	ID: 922694185
Github: Tybo2020	CSC415 Operating Systems

For `mkdir`, it needs to check if the desired directory does not already exist in the file system in order to make sure the previous directory does not get overwritten. The actual creation of the directory will be done through the function `createDirectory`, while `mkdir` will just initialize, and update, the entry in the parent so that the user can go back to the directory later on.

For `fs_stat`, this function retrieves metadata about a file or directory. The function calls `parsePath()` to locate a file or directory. And if located, the function then fills the `fs_stat` struct with the size of target, how many disk blocks it uses, and when it was last accessed, modified, or created.

For `fs_closedir`, the function properly closes and free any memory that was previously opened by `fs_opendir`. `fs_closedir` ensures that all memory allocated for reading and storing the directory's contents is freed and returns an `int` back to caller upon success or failure.

For `isFile` and `isDir`, these functions will make use of the boolean `isDir` in the struct `DirectoryEntry`. These functions will also call `parsePath` to not only access the parent, which has access to `isDir`, but to make sure the input is a part of the file system. `fs_delete` will also call `parsePath` for the same reasons, and will also call `isFile` as the filename is not guaranteed to actually be a file name. As to how the file will be deleted, the memory associated with the file will be freed, allowing this entry to be used for other things. All variables associated with the file will also be set to 0 or NULL to make sure the file is not accidentally used somewhere else. `rmdir` will follow a similar process as `fs_delete`, except with the added condition of checking if every entry in the directory is empty before deleting because that is how linux deletes directories.

For `loadDir`, it will use the `fileSize` variable from the struct `DirectoryEntry` to read the requested entry from disc. The only other thing this function will need to do is update access time.

Phase 3

Like phase 2, the five `b_io` functions are split among the team, with Ty handling `b_read`, Eugenio handling `b_seek`, Julia handling `b_open`, Juan handling `b_write`, and Ranj handling `b_close`.

For `b_read`, the amount of bytes that can be read from a file will be calculated in the beginning to avoid any problems that could arise later regarding count, and it avoids the need to consider the status of count later on. As for reading bytes from a file, it will be broken into three parts.

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

The first part will move any remaining bytes in fd's buffer to the user's buffer. This handles the bytes that are left over from previous reads. The second part will directly read bytes from a file onto the user's buffer, if count was not handled in the first part. Directly reading bytes from a file avoids the need to constantly fill up fd's buffer when count is larger than fd's buffer. If count is still not reached, then the third part will fill up fd's buffer, which will then fill whatever remaining bytes are needed by the user's buffer. If the function reaches part three, then the amount of bytes remaining is less than a block, which is why fd's buffer is used for the reading.

For b_seek, it will need fd's corresponding file in order to find the buffer's exact position. The position will then be changed depending on the variable whence. However, the new position will need to be calculated in terms of blocks in order to switch fd's current block to the new one.

For b_open, the file it is trying to open needs to exist in the file system, so the function will need to find said file, unless the flag is set to create the file, in which case the function will make a new entry in the parent if possible and update it to disk. A buffer for the file will be created for b_write and b_read to use, and will load up the first block if the flag is set to read the file.

For b_write, it will follow the same logic of b_read, where it will first calculate the amount of bytes that can be written to the file, then check if fd's buffer holds any bytes, then directly write blocks into the file, and finally handle any leftover bytes for the next write. Doing b_write this way will avoid the need for multiple if statements throughout the function, and may better handle any problems that arise with count before any bytes are actually written.

For b_close, it will simply need to free the buffer associated with the file, and clear all the field's associated with the file descriptor.

Limitations:

All commands work as intended.

Issues and Resolutions:

- An issue arose in b_write, where the case of count ≤ 0 was not considered. During this change, we decided to also change the way b_write functioned. The new design would employ a while loop where fd's buffer would be filled up by buffer, and once it got full,

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

would be written to disk. This was done to avoid buffer issues that arose from the previous implementation.

- Another issue we encountered was getting our filesystem to run properly when working with relative paths. We were able to resolve this by adding a global variable that kept track of the parent directories no matter which directory you were in. Prior to this implementation, commands, such as mv, were failing because once our fs was in a directory it didn't account for what the parent directories are. With a global variable, we were able to constantly keep track of the directories.
- For our directory, a couple of the issues we encountered included how we should write the files to the disk and how we should handle memory management. We plan to use a FAT table for memory allocation, which makes it difficult to write our files using LBAWrite. Since we are not guaranteed that our blocks will be continuous, so we had to read our data in clusters. With the function implemented in the freespace.c file, our resolution is detailed as:
 1. Track currentBlock = startBlock.
 2. While we still have data to write:
 - a. Read from the newDirectory buffer.
 - b. Write up to BLOCK_SIZE bytes to currentBlock using LBAWrite.
 3. Use readFATEntry(currentBlock) to get the next block in the chain.

The second issue we encountered was figuring out how to work with memory management, - specifically when we should free our newDirectory. As a habit, we wanted to free our entry at

- Initializing the VCB brought no problems. Formatting, on the other hand, did bring some challenges. **(write down problems for VCB formatting)**
 1. **Heap Crash After Formatting Start:** The system crashed with a malloc assertion failure during the formatting process. This was initially due to the line in our initFileSystem function, vcb = malloc(sizeof(VolumeControlBlock)), but was later correctly fixed to be vcb = malloc(blockSize) instead. Changing this line ensured that we properly allocated enough memory for our Volume Control Block struct.

Ty Bohlander	ID: 922436005
Eugenio Ramirez	ID: 923053570
Julia Bui	ID: 923518762
Juan Ramirez	ID: 922495948
Ranjiv Jithendran	ID: 922694185
Github: Tybo2020	CSC415 Operating Systems

2. **Some merge conflicts in Github:** This occurred because some files were changed locally but the same files were updated by another teammate and already merged

- We identified a critical issue in our FAT implementation where only the first and last blocks of a file's allocated space were being properly marked in the table, leaving intermediate blocks incorrectly labeled as free. The correct implementation requires every block in a file's chain to be marked appropriately in the FAT: free blocks should be marked with a value indicating availability, each allocated block must contain the index of the next block in the chain to maintain proper linkage, and only the final block should contain an end-of-file marker (such as 0xFFFF).
- We identified a critical issue in our filesystem implementation where path strings were being modified during the tokenization process, causing subsequent operations on the same path to fail. When navigating to nested directories using `cd`, the first directory operation would work correctly, but attempting to enter a subdirectory would fail because the original path string had been altered by `strtok_r` during the initial path resolution. The correct implementation requires creating a copy of the path string using `strdup()` before tokenization, ensuring that each function call receives an unmodified version of the original path. This allows multiple operations (such as checking if a path is a directory and then changing to it) to work correctly with the same pathname, enabling proper navigation through multiple directory levels.

Functionality:

Directory Functions

createDirectory - This function takes two variables: `int initialNumEntries` and `DirectoryEntry* parent`. It makes a new directory by first allocating blocks to it, which depend on `initialNumEntries`. Then, it sets up entries `“.”` and `“..”`, pointing to the variable `parent` for `“..”`. If `parent` is `NULL`, then that means the directory created is the root directory. It initializes all other entries by setting the variable `inUse` to false. Finally, the newly created directory is written to disk.

Free Space Functions

Ty Bohlander	ID: 922436005
Eugenio Ramirez	ID: 923053570
Julia Bui	ID: 923518762
Juan Ramirez	ID: 922495948
Ranjiv Jithendran	ID: 922694185
Github: Tybo2020	CSC415 Operating Systems

initFreeSpace - This function takes two int variables: blockCount and blockSize. It makes a buffer that is the same size as a FAT entry block, which is used to mark each block as free or reserved. The number of blocks that need to be marked depend on the volume control block, which holds the size of free space. Once that is done, the buffer and the volume control block, which now holds the first block of free space, is written to disk.

allocateBlocks - This function only takes one variable of type int: numOfBlocks. After making sure this variable is valid, the function goes through the entire FAT, linking free blocks together, and throwing an error if something goes wrong. From there, the volume control block's free space size is updated and written to disk, and the startblock of this chain of free blocks is returned.

extendChain - The two input functions are headOfChain and amountToChange. Starting at headOfchain, the function iterates through the chain until reaching the last block, where it calls on allocateBlocks with amountToChange to add the new blocks. It returns the variable headOfChain.

releaseBlocks - Using int variables location and numOfBlocks, this function iterates through the chain of blocks, calling writeFATEntry to free each block of the chain. It then updates the volume control block, writing it back to disk, and returns the blocks that were released.

isBlockFree - This function takes a variable blockNum and checks if it's valid. If it is, it checks calls readFATEntry to see if the entry is free. The function returns true if the entry is free, false otherwise.

readFATEntry - With int blockNum as its only input, this function first validates blockNum before finding its position in the FAT. Once found, a buffer is allocated, reading the FAT block where blockNum lies and returning it using the variable FatEntry entry. It returns the variable BLOCK_RESERVED if an error occurs.

writeFATEntry - This function's inputs are int blockNum and FATEntry entry. After validating blockNum, the function allocates a buffer to read the corresponding FAT block from disk. Then, the exact position is blockNum is found, updated, and written back into disk. It returns true when successful, false otherwise.

getFATEntryPos - This function takes an int variable blockNum, and returns its byte position.

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

File System Functions

fs_mkdir - Taking `const char *pathname` and `mode_t mode`, this function first calls `parsePath` to ensure `pathname` is valid. Then, it checks if the path does not already refer to an existing file or directory. Afterward, the function calls `createDirectory`, checking if it was successful, and then claims a free spot in the parent directory if available. The new directory entry is then initialized in the parent and written into disk, returning 0 to represent success.

fs_rmdir - This function takes a `pathname` and checks if it exists, if it is a directory, and if it is empty. If all this is true, it calls `releaseBlocks` to free the memory tied to this directory and clears all variables related to itself. It returns 0 on success.

fs_opendir - The input of this function is a `pathname`. After checking if the path exists and is a directory, it calls `loadDir` and then initializes the directory structure.

fs_readdir - This function takes in a pointer to variable `fdDir`, making sure it exists. It then goes through its entries to find one that has been initialized. From there, this function copies the entry's name and type, returning it through a structure called `fs_direntinfo`

fs_closedir - Taking the same variable as `fs_readdir`, and making sure it exists, this function frees the structure that was returned in `fs_readdir` and the input variable. It returns 0 on success.

fs_getcwd - The inputs of this function are `char *pathname` and `size_t size`. After making sure both inputs are valid, this function simply copies the current directory onto `pathname` through a global variable `currentWorkingPath`. It also adds a null terminator to `pathname` before returning it.

fs_setcwd - Taking a `pathname` as a parameter, this function makes sure the path exists and is a directory before checking if the path is for the root directory. It then allocates, and reads, space for a new directory block. It then copies the `pathname` to a global variable `currentWorkingPath`. It returns 0 if successful.

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

fs_isFile - This function takes a filename as a parameter, first checking if filename contains a "." to see if it is a file. If it does not, it calls parsePath to locate the filename, and then calls isDEaDir to check if filename is a file. It returns 1 if filename is a file, 0 otherwise.

fs_isDir - Much like fs_isFile, this function takes a pathname as a parameter, calls parsePath to locate it, and then calls isDEaDir to check if pathname is a directory. It returns 1 if pathname is a directory, 0 otherwise.

fs_delete - This function takes a filename and calls parsePath before calling fs_isFile. Once found, the blocks of memory associated with the file are freed, and all its values are set to 0. It returns 0 on success.

parsePath - With char* pathname and ppinfo* ppi as inputs, this function splits pathname by the "/" character. It first figures out if pathname is absolute or relative, ie if "/" is the first element in pathname. It then iterates through pathname, keeping track of the parent, where the directory entry's position is located in the parent, and the parent's name. These pieces of information are given back to the caller through ppi.

findInDirectory - This function takes in two variables: DirectoryEntry* parent and char* token. It iterates through the parent, checking if any child has the same name as token. It returns the location of the file with the same name as token on success.

loadDir - This function takes a pointer, called targetDir, to struct DirectoryEntry and calculates the number of blocks needed to load it. From there, a buffer is created to read off the required blocks, and the lastAccessed variable from targetDir is updated. It returns the buffer.

isDEaDir - This function checks the variable isDir in its parameter, targetDir, to see if the entry is a directory. It returns 1 if the entry is a directory, 0 if entry is a file.

fs_stat - The parameters for this function are char* path and struct fs_stat *buf. After checking that both parameters exist, this function allocates memory for a pointer to struct ppinfo and calls parsePath to ensure path exists. From there, it uses the pointer to fill all the values in buf, returning 0 on success.

Ty Bohlander	ID: 922436005
Eugenio Ramirez	ID: 923053570
Julia Bui	ID: 923518762
Juan Ramirez	ID: 922495948
Ranjiv Jithendran	ID: 922694185
Github: Tybo2020	CSC415 Operating Systems

fs_mv - Takes in parameters `const char *srcPath` and `const char *destPath`. The function first copies the metadata of the source file into the destination directory and then removes the original entry via `parsePath()`. Then the fs locate a free entry in the destination directory and copy over the source file's metadata, and update the filename if called. The original source entry is marked as unused and both directories are written back to disk to persist the changes.

normalizePathName - This function takes a path string as input and resolves relative path components to create an absolute path. It starts by determining if the path is absolute (beginning with '/') or relative, then processes each directory component using `strtok`. When encountering ".", the function maintains the current position; when encountering "..", it removes the last directory component by finding the last slash and truncating; for regular directory names, it appends them to the resolved path. The function ensures proper formatting by handling edge cases like the root directory and empty paths, ultimately returning a normalized absolute path that can be used for filesystem operations.

b_io functions

b_open - The inputs of this function are filename and flags. After making sure filename and flag are valid, this function calls `parsePath` to find the file, creating it and updating the parent if flag is for creation. The buffer of this file is then initialized, and filled up if flag is set to read. The last thing this function handles is if flag is `O_append`, where it positions the buffer to the end of file. It returns a file descriptor `fd` on success.

b_seek - This function has three input variables: `fd`, `offset`, and `whence`. After making sure `fd` is valid and in use, this function gets the directory entry associated with `fd` and finds the current position in the file. It then moves to a new position determined by `whence` and `offset`. From there, it updates `fd`'s block number and index, and returns the new position.

b_write - With parameters `buffer`, `count` and `fd`, this function makes sure `fd` is valid and open. From there it runs a while loop that fills up `fd`'s buffer with `buffer`, writing to disk when `fd`'s buffer becomes full. It then updates the file size associated with `fd`, and updates the parent where the file resides. It returns the number of bytes written upon success.

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

b_read - With the same parameters as **b_write**, this function first checks if **fd** is a valid file descriptor and is open. It then calculates how many bytes it needs to read into the buffer by checking if the file has enough bytes for count. The function then checks if **fd**'s buffer has any bytes left, and is moved to the buffer if true. If count is not reached, the function directly fills buffer with the required blocks. If count is still not reached, the function refills **fd**'s buffer and fills buffer. It returns the number of bytes read upon success.

b_close - This function takes in one variable: **fd**, and after making sure it is a valid and used file descriptor, it clears the buffer, freeing it and all variables tied to **fd**. The function returns 0 upon success.

Contributions:

Ty Bohlander	Eugenio Ramirez	Julia Bui	Juan Ramirez	Ranjiv Jithendran
initFreeSpace allocateBlocks extendChain releaseBlocks isBlockFree readFATEntry writeFATEntry getFATEntryPos parsePath fs_setcwd fs_getcwd fs_opendir loadDir findInDirectory b_read b_write b_open documentation	b_seek fs_mkdir createDirectory b_open normalizePathname fs_rmdir fs_setcwd documentation	b_open fs_stat fs_closedir createDirectory fs_move Memory leak documentation	fs_rmdir fs_isFile fs_isDir fs_delete b_write fs_setcwd documentation	b_close initFilesystem

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

Screen shot of compilation:

1. Make:

```
student@student:~/csc415-filesystem-Tybo2020$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o createDirectory.o createDirectory.c -g -I.
gcc -c -o freeSpace.o freeSpace.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 createDirectory.o freeSpace.o mfs.o b_io.o fsL
owM1.o -g -I. -lm -l readline -l pthread
student@student:~/csc415-filesystem-Tybo2020$
```

2. Make run:

```
student@student:~/csc415-filesystem-Tybo2020$ 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
Volume 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 > exit
System exiting.
student@student:~/csc415-filesystem-Tybo2020$
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

Screenshots of README commands:

1. ls command:

```
student@student:~/Desktop/csc415-filesystem-Tybo2020$ 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
Volume 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

example
testFile.txt
Prompt >
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

2. cp command:

```
student@student:~/Desktop/csc415-filesystem-Tybo2020$ 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
Volume 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 > touch original.txt
Allocating 1 blocks, starting search at block 864
Successfully allocated 1 blocks starting at block 864
Prompt > ls

example
testFile.txt
original.txt
Prompt > cp original.txt copy.txt
Allocating 1 blocks, starting search at block 865
Successfully allocated 1 blocks starting at block 865
Prompt > ls

example
testFile.txt
original.txt
copy.txt
Prompt > █
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

3. mv command:

```
Prompt > cd ..  
Prompt > mv dir1/testfile1.txt dir2/testfile1.txt  
Prompt > ls  
  
example  
testFile.txt  
original.txt  
copy.txt  
dir1  
dir2  
Prompt > cd dir2  
Prompt > ls  
  
testfile1.txt  
Prompt >
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

4. md command:

```
student@student:~/Desktop/csc415-filesystem-Tybo2020$ 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
Volume 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

example
testFile.txt
original.txt
copy.txt
dir1
dir2
Prompt > md dir3
Size of DirectoryEntry: 144 bytes
Allocating 9 blocks, starting search at block 893
Successfully allocated 9 blocks starting at block 893
Prompt > ls

example
testFile.txt
original.txt
copy.txt
dir1
dir2
dir3
Prompt > █
```


Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

5. rm command:

```
Successfully allocated 1 blocks starting at block 893
Prompt > ls

example
testFile.txt
original.txt
copy.txt
dir1
dir2
dir3
Prompt > rm dir3
Prompt > ls

example
testFile.txt
original.txt
copy.txt
dir1
dir2
Prompt >
```

6. Touch command:

```
dir2
Prompt > cd dir2
Prompt > ls

testfile1.txt
Prompt > touch testfile2.txt
Allocating 1 blocks, starting search at block 893
Successfully allocated 1 blocks starting at block 893
Prompt > ls

testfile1.txt
testfile2.txt
Prompt >
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

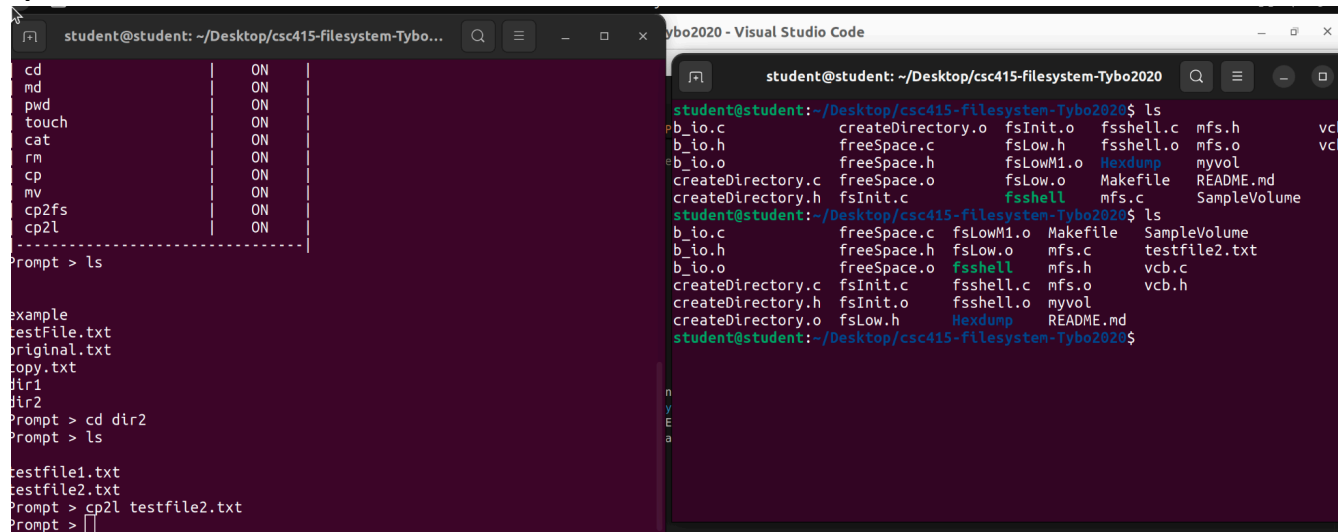
ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

7. Cat command:

```
Successfully allocated 1 blocks starting at  
Prompt > ls  
  
testfile1.txt  
testfile2.txt  
Prompt > cat testfile2.txt  
Prompt >
```

8. Cp2l command:



```
student@student: ~/Desktop/csc415-filesystem-Tybo...  
cd          ON  
md          ON  
pwd         ON  
touch       ON  
cat         ON  
rm          ON  
cp          ON  
mv          ON  
cp2fs       ON  
cp2l        ON  
-----  
Prompt > ls  
  
example  
testfile.txt  
original.txt  
copy.txt  
dir1  
dir2  
Prompt > cd dir2  
Prompt > ls  
  
testfile1.txt  
testfile2.txt  
Prompt > cp2l testfile2.txt  
Prompt >
```

```
student@student: ~/Desktop/csc415-filesystem-Tybo2020  
student@student:~/Desktop/csc415-filesystem-Tybo2020$ ls  
b_io.c      createDirectory.o  fsInit.o  fsshell.c  mfs.h      vcb  
b_io.h      freeSpace.c       fsLow.h   fsshell.o  mfs.o      vcb  
b_io.o      freeSpace.h       fsLowM1.o Hexdump    myvol        
createDirectory.c freeSpace.o       fsLow.o   Makefile   README.md    
createDirectory.h fsInit.c          fsshell   mfs.c      SampleVolume  
student@student:~/Desktop/csc415-filesystem-Tybo2020$ ls  
b_io.c      freeSpace.c  fsLowM1.o  Makefile   SampleVolume  
b_io.h      freeSpace.h  fsLow.o    mfs.c      testfile2.txt  
b_io.o      freeSpace.o  fsshell    mfs.h      vcb.c  
createDirectory.c fsInit.c     fsshell.c  mfs.o      vcb.h  
createDirectory.h fsInit.o     myvol      Hexdump    README.md  
createDirectory.o fsLow.h        
student@student:~/Desktop/csc415-filesystem-Tybo2020$
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

9. C2fs command:

```
Prompt > cp2f testfile2.txt
Prompt > cp2fs copy.txt
Allocating 1 blocks, starting search at block 894
Successfully allocated 1 blocks starting at block 894
Prompt > ls
testfile1.txt
testfile2.txt
copy.txt
Prompt >

Command 'lz' not found, but can be installed with:
lo apt install mtools
ident@student:~/Desktop/csc415-filesystem-Tybo2020$ ls
.o.c      createDirectory.o  fsLow.h    Hexdump    README.md
.o.h      freeSpace.c        fsLowM1.o  Makefile   SampleVolume
.o.o      freeSpace.h        fsLow.o    mfs.c      testfile2.txt
y.txt     freeSpace.o        fsshell    mfs.h      vcb.c
ateDirectory.c  fsInit.c        fsshell.c  mfs.o      vcb.h
ateDirectory.h  fsInit.o        fsshell.o  myvol
ident@student:~/Desktop/csc415-filesystem-Tybo2020$
```

10. Cd command:

```

|-----|
Prompt > ls

example
testFile.txt
original.txt
copy.txt
dir1
dir2
Prompt > cd dir1
Prompt > ls

Prompt > pwd
/dir1
Prompt >
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

11. Pwd command:

```
Prompt > pwd
/dir1
Prompt > md dir4
Size of DirectoryEntry: 144 bytes
Allocating 9 blocks, starting search at block 895
Successfully allocated 9 blocks starting at block 895
Prompt > cd dir4
Prompt > pwd
/dir1/dir4
Prompt >
```

12. History command:

```
/dir1/dir4
Prompt > history
ls
cd dir1
ls
pwd
md dir4
cd dir4
pwd
history
Prompt >
```

Ty Bohlander
Eugenio Ramirez
Julia Bui
Juan Ramirez
Ranjiv Jithendran
Github: Tybo2020

ID: 922436005
ID: 923053570
ID: 923518762
ID: 922495948
ID: 922694185

CSC415 Operating Systems

13. Help command:

```
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 >
```