

File System Project - Team 42

GitHub name with group work: **nabware**

GitHub Link: <https://github.com/CSC415-2025-Spring/csc415-filesystem-nabware>

Name	Student ID
Nabeel Rana	924432311
Leigh Ann Apotheker	923514173
Ethan Zheng	922474550
Bryan Mendez	922744274

The github link for your group submission.

<https://github.com/CSC415-2025-Spring/csc415-filesystem-nabware>

The plan/approach for each phase and changes made

The goal for the first phase was to get the basic file system operational, so we initialized the free space and root directory in `fsInit.c`. This allowed us to ensure that our volume could be initialized, read from, and written to. Additionally, we defined the basic structure of our Volume Control Block (VCB) and File Allocation Table (FAT).

We walked through the logic of each command we needed to implement and broke them down into the lower-level functions required to support them. As we did this, we implemented the functions step-by-step, identifying common patterns and abstracting them into reusable helper functions. For example, functions like `getDirEntriesFromPath` and `getAbsolutePath` were introduced to streamline path handling and reduce repeated logic.

Throughout development, we added error handling like NULL checks when fetching directory entries and fixing offset miscalculations in read/write functions to ensure data alignment. These were a result of debugging feedback from our shell interface.

A description of your file system

Our File System consists of four core components, the VCB (Volume Control Block), our FAT Table (for free space management), Directory Entries, along with a buffered I/O. To interact with our File system, we've implemented a basic shell with all the required commands working.

The `SampleVolume` file acts as a virtual disk which stores the files to be created or manipulated in our File system.

The main entry point into the File system would be `fsShell.c` since that's where our `main()` function is stored. It then calls `startPartitionSystem()` which initializes the volume and `initFileSystem()` from `fsInit.c` which sets up our VCB, FAT table, and root directory. From there, we can interact with our virtual disk/sample volume through our shell (which acts as an intermediary between the filesystem and the volume).

`fsShell.c` serves as a GUI which calls functions from `mfs.c` and `b_io.c` whenever a command is entered. From `mfs.c` and `b_io.c`, whenever a LBA Operation is needed, it calls from `fsLow.h`.

To manage open files in our file system, we are using a `fcbArray` which contains File Control Blocks (FCBs). The FCB is similar to the VCB in that it contains key information/metadata but

instead of the volume, it pertains to an actual file and there can be up to a maximum of 20 FCBs.

FCBs manage the current data buffer and its position along with the current block number on disk, the flags for file access, the remaining bytes in the file, parent DEs and their location, and the file's DE information. Whenever `b_open()` is called it allocates a FCB and whenever `b_close()` is called, it releases an FCB in the `fcbArray`.

The VCB is stored in the first block of the FAT table and acts as a “master” record containing important metadata for our filesystem. The signature, volume size, block size, number of blocks, the block indexes for the root directory and FAT, along with the number of entries per directory are all stored here.

Our File Allocation Table (FAT) is how we manage free space in our filesystem. It's just a basic array where each element corresponds to a specific block in the volume. A value of -1 indicates a free block while -2 indicates the final block of a file and all other values point to the next block in the chain.

Directories are handled using a collection of Directory Entry (DE) structures which each contain the name, size, starting block index, flags, and timestamps for creation and modification.

A list of exactly which shell commands work and don't work or limitations

Command	Status
ls	Works
cd	Works
md	Works
pwd	Works
touch	Works
cat	Works
rm	Works
cp	Works, but there's a bug where file size is 0 when copying into subdirectory.

mv	Works, but there's a bug where file size is 0 when moving into subdirectory
cp2fs	Works
cp2l	Works

Issues you had

One issue we had was with the `fs_readdir` function, where the `dirEntryPosition` variable was not properly incremented after each directory entry read. This oversight caused the function to return the same entry repeatedly, instead of iterating through subsequent directory entries. We resolved this by making sure the `dirEntryPosition` is incremented on each iteration which allowed the function to correctly progress through the directory.

We had an issue with the offset calculation in the `readDirEntries` and `writeDirEntries` functions. Initially, we used `blockOffset += vcb->blockSize` to advance through the directory entries. However, this approach didn't account for the number of directory entries on each block and resulted in incorrect memory alignment. This was resolved by adjusting the calculation to `blockOffset += vcb->blockSize / sizeof(DE)`, which properly steps through each block based on the number of directory entries it contains.

We had a problem when calling `getDirEntriesFromPath` from commands like `mv` and `touch`. The return value of this function was not being checked for `NULL`, which caused segmentation faults when invalid paths were passed in. To resolve this, we added null checks following the call to `getDirEntriesFromPath`, allowing us to handle invalid paths gracefully and preventing crashes.

Another issue we had was when we tried copying files into a directory and not specifying the file name after the directory. We ran into a segmentation fault when trying to copy a file into a subdirectory which turned out to be an error value (-1) being passed in as the index to access `fcArray`. When `cmd_cp` calls `b_open`, `b_open` checks if the destination is a directory and if it is, it returns -1 and passes that value back to `cmd_cp`. However, `cmd_cp` doesn't check the return value for error handling and attempts to call `b_write` where `test_fs_fd` is -1 which leads to a segmentation fault when trying to access the `fcArray`. We fixed this by updating the destination source by calling `fs_isDir` to check if the path is a directory which would then append the source file name to the destination directory path.

One thing of note is that the segmentation fault only occurred on one of our VM instances when we ran it (we were using the same code from the latest git commit) which we found weird, but the `cmd_cp` didn't work anyways since it didn't properly copy the file into the directory. And subsequently we ran into the same issue with moving a file into a directory without specifying the file name after the directory. The fix was essentially the same as fixing the copying files with appending the file name to the end of the directory path.

Also when moving a file into a directory and specifying the file name; if you choose a file name that already exists inside the destination directory, the `cmd_mv` command doesn't move the file properly and we added an error message for that.

Details of how each of your functions work

int initRootDirectory()

This function initializes the root directory of the file system as well as initializing the current directory (".") and parent directory ("..") entries. It does so by calculating how many directory entries to allocate with `numOfEntries`, number of entries, set to 50 initially. The `requiredNumOfBytes`, or total bytes needed for allocation, is found by multiplying `numOfEntries` by the size of a single directory entry. The `requiredNumOfBlocks`, or number of blocks to store the number of bytes, is found by subtracting 1 from the `blockSize` (size of the blocks in the file system) for rounding purposes to ensure calculating for entire blocks and this is then added to the `requiredNumOfBytes` and divided by `blockSize`. At this point, the blocks needed for the initial number of directory entries, 50, have been calculated. Now, a check is performed to see if there is extra space in these blocks to use for more directory entries. This is done by multiplying `requiredNumOfBlocks` by `blockSize` (which is the total bytes in the blocks) and then subtracting this by `requiredNumOfBytes`, this equals to the unused space in the blocks. This is then divided by the size of a single directory entry, which provides the extra entries that can be included, `extraEntries`. Memory is then allocated for the directory entries and all entries are initialized to unused with the `used` field set to 0. The blocks are allocated in the file system with a call to the `allocateBlocks()` function. The current directory and parent directory entries are set up as used and as directories with their locations pointing to `starBlock`. From here, the directory entries are written to disk using `LBAWrite()`. The memory used temporarily for the directory entries is freed, the volume control block is updated with the number of directory entries, and `startBlock`, the starting block number of where the root directory is stored, is returned.

int initFreeSpace()

This function initializes the File Allocation Table (FAT), which tracks free and used blocks within the filesystem. It calculates the number of blocks needed to store the FAT based on the total number of blocks in the filesystem and allocates memory accordingly. The FAT entries are initialized so that blocks occupied by the FAT itself are properly marked, and the remaining blocks are flagged as free. After initialization, the FAT is written to disk, and the Volume Control Block (VCB) is updated to reflect the FAT's position and the first available free block.

int initFileSystem()

This function gets called as one of the first entry points into our filesystem and initializes the VCB (with the signature, volume size, block size, and number of blocks), FAT table (free space

management), and the root directory structure. It also writes the VCB back to disk at block 0. However, if a filesystem already exists (signature exists), then it just loads the existing FAT table from disk.

void exitFileSystem()

This function is called when the user enters the command “exit”. It prints a system exiting message and also frees the VCB and the FAT table, making sure we safely exit our shell.

char * fs_getcwd(char *pathname, size_t size);

The fs_getcwd function gets the pathname of the current working directory. The function takes in a pointer to a buffer where the current working directory path will be copied (pathname) and the size of the buffer in bytes (size). Ultimately, the current working directory path is stored in the cwd variable that is initialized as the root directory (“/”) outside the function. Thus, the function copies the path into the buffer. The pathname pointer that now contains the current working directory string is returned.

int fs_setcwd(char *pathname);

The fs_setcwd function sets the current working directory. The function takes in the pathname pointer as an argument. It uses the pathname to call the getAbsolutePath function to get the absolute path of the current working directory. Then, the function checks if the path exists by calling fs_isDir and to verify if it is a directory (returns -1 if not). Then, it copies the absolute path of the current working directory into the cwd variable to be stored as the now updated current working directory. Finally, it returns 0 to indicate success.

void readDirEntries(DE* dirEntries, int startBlock);

The readDirEntries function reads blocks of directory entries from disk. The function works to handle directory entries that span multiple blocks. It follows the chain of blocks using the file allocation table to read all directory entries, whether stored contiguously or dis-contiguously. The current implementation reads the current block, looks for the next block in the file allocation table, moves to that block, and repeats until the end of file. A possible improvement is to check for contiguous blocks before reading the directory entries from disk, which would allow for multiple blocks to be read at a time before handling the remaining blocks.

Essentially, the function takes in a pointer to an array where the directory entries will be stored (dirEntries) and the starting block number (startBlock) where the directory entries are stored on disk. The current block (currBlock) being read is tracked and begins at startBlock, the position in the directory entries array is tracked with blockOffset, and entriesPerBlock calculates how many directories fit within one block by dividing the blockSize variable from the VCB structure by the size of a directory entry. The block data is read into the directory entry array at the current block and uses the file allocation table to find the next block in the chain. The blockOffset is then updated for the next read by adding entriesPerBlock.

int fs_mkdir(const char *pathname, mode_t mode);

The `fs_mkdir` function makes a directory entry. It does so by taking in the `pathname` pointer and the `mode` as arguments. The function first checks if the directory entry already exists by calling the function `fs_isDir`. If it does already exist, the function provides an error message and returns -1. Next, `getAbsolutePath` is called to get the absolute path of the directory and `getDirEntriesFromPath` is called to find an unused directory entry in the parent directory. If no free directory entries are available, the parent directory entries are freed and the function returns -1. If an unused directory entry is available, it is initialized. The required number of bytes and required number of blocks are calculated so that the appropriate number of blocks can be allocated for the new directory entry by calling the `allocateBlock` function. A new directory entry is initialized in the parent directory. It is marked as a directory and as used. Memory is allocated for the new directory entries by using the required number of bytes calculated previously. The `"."` and `".."` directories are initialized for these new directory entries (child directory entries). The updated parent directory entries and child directory entries are written to disk and all allocated memory is freed. The function returns 0 to indicate success.

int fs_rmdir(const char *pathname);

The `fs_rmdir` function removes a directory from the file system. This is done through the function taking in the `pathname` pointer as a parameter. First, the function checks if the directory is empty by using the `pathname` as an argument for the function call `getDirEntriesFromPath`. If the directory is not empty, an error message is printed that states the directory cannot be removed and the directory is not empty; the function returns -1. It then loops through all entries in the directory. The `dirEntries` array of directory entries is freed. The parent directories are updated and the blocks allocated are, thus, freed. The updated parent directories are written to disk and the memory allocated for the parent directory and the parent directory entries is freed.

fdDir* fs_opendir(const char *pathname);

This function at a basic level opens a directory entry. It's similar to how `b_open` opens a file to eventually read or write to it. It takes in a pointer to a `pathname` as an argument which we first determine if that path is a relative or absolute path by passing it into `getDirEntriesFromPath()` which returns a pointer to an array of DE structures. However if that directory does not exist or cannot be accessed, it returns NULL.

Once we know that directory exists, we allocate and initialize a directory structure (`fdDir`) that functions as a handle for opening that directory. This is similar to how file descriptors (`fd`) work for opening regular files. With `fdDir` it acts as a file descriptor but for a directory. It loads directory entries into memory and creates a buffer (that `fs_readdir()` calls which reduces the need for additional disk access calls).

struct fs_diriteminfo *fs_readdir(fdDir *dirp);

After opening our directory, we then have to read and return the next DE from the opened directory in the buffer (from `fs_opendir`). First, we start from `dirp->dirEntryPosition` which we get from the previous state of the directory descriptor. It was initialized to 0 in `fs_opendir()` and we increment it by 1 each time we call `fs_readdir`.

We then search for the next valid/used DE. Since each DE has a “used” field (which indicates whether that entry contains valid data), we skip any unused entries until we find a used entry. From there, we point `di` (directory item info) to the `fs_diriteminfo` structure to populate it with information from the current DE. And then we copy the file size and name into their respective entries (`di->reclen`, `di->d_name`). Also of note, the `di` structure is already allocated from `fs_opendir()` so we are reusing the same memory location when we call `fs_readdir`.

We then increment `dirEntryPosition` so the next call starts from that position. Finally, we return a pointer to the populated `fs_diritemInfo` structure if a valid entry was found otherwise NULL if no more entries exist.

int fs_closedir(fdDir *dirp);

This function is used in conjunction with `fs_opendir()` and acts as a cleanup function for our `fdDir` that we created in `fs_opendir`. First, we check if the directory pointer (`dirp`) is NULL to avoid any segmentation faults and then we free the memory allocated for the DE array loaded from disk (which was used by `getDirEntriesFromPath()`). Subsequently, we then free the directory item info in `dirp` (which was used by `fs_readdir()`) before finally freeing the directory descriptor itself (`dirp`).

This follows/completes a UNIX directory access paradigm where we first open the directory, read the entries one by one, and then close the directory each with their respective functions. `fs_closedir()` serves as the last step where we clean up our dynamically allocated memory so we prevent any memory leaks and/or segmentation faults.

DE* getDirEntriesFromPath(const char *pathname);

This is a helper function which retrieves all DEs for a specific path. First we allocate memory based on the `numOfEntries` stored in the `vcb` (`vcb->numOfEntries`) and then we call `readDirEntries()` to load the root directory entry as the starting point. If the file path is relative, we pass in the path to `getAbsolutePath()` to retrieve the absolute path and then parse each path component into a “level”. For example, “home/user/downloads” becomes “home”, “user”, and “downloads” levels. We then traverse through each level and look for an entry matching the current path.

This helper function is critical to `fs_opendir()` to locate and open directories as it can locate parent directories of a file and also provides the functionality to navigate from the root directory to any subdirectory within. It basically converts a path like “/home/user/downloads”

into the actual DEs stored in the filesystem enabling proper functionality for commands navigating or viewing directories in our volume.

DE* getDirEntryFromPath(const char *pathname);

This is another helper function that is similar to `getDirEntriesFromPath()`, however it only returns a single DE structure that represents a single specific file or directory. We use this specific function when we only need to get metadata about a specific file or directory (whereas with `getDirEntriesFromPath()` we use it whenever we need to list the contents of a directory).

First, we convert the `pathname` passed in to an absolute path using `getAbsolutePath()` and then we get the parent path using `getParentPath()`. From there, we retrieve all the DEs from the parent directory using `getDirEntriesFromPath()` and get the last element/component from the absolute path. We then search through the parent DEs to find the entry that matches the last element and return a newly allocated DE structure containing the copy of the matched DE.

In other words, this means we are finding and returning the specific record/metadata entry of one file rather than returning all the contents inside of a directory. This is important as `fs_isFile()`, `fs_isDir()`, and `fs_stat()` uses this function to check if a file exists and to retrieve metadata.

int fs_create(char* filename);

This function creates a new file in the file system. It first checks if a file or directory with the same name already exists. If not, it retrieves the parent directory, locates an unused directory entry slot, and initializes the entry with metadata like name, timestamps, and location. A block is allocated to the file using `allocateBlocks`, and the updated directory entries are written to disk.

int fs_delete(char* filename);

This function deletes a specified file by locating its directory entry and clearing it. It first identifies the parent directory and searches for the file within it. Once found, it marks the entry as unused, frees the associated data blocks using `freeBlocks`, and writes the updated directory entries back to disk.

int fs_move(char* src, char* dest);

This function moves a file from a source path to a destination path. It checks that the source is a file and that the destination path is valid. It copies the source directory entry to the destination, updates the name, and marks the original entry as unused. Both source and destination parent directory entries are then updated and written back to disk.

int findNextFreeBlock(int startBlockNum);

This function scans the FAT starting from a specified block number to find the next available free block. It returns the index of the first free block found or -1 if none are available.

int allocateBlocks(int startBlock, int numOfBlocks);

This function allocates a sequence of blocks for a file or directory. It searches for the required number of free blocks using findNextFreeBlock and links them together in the FAT. The last block in the chain is marked as the end of the file. It updates to the FAT and VCB are written back to disk.

void writeDirEntries(DE* dirEntries, int startBlock);

This function writes a set of directory entries to disk, beginning at the specified start block. It iterates through the FAT to follow the block chain and writes each segment of the dirEntries array to the corresponding block on disk. The offset is calculated based on how many entries fit into each block.

b_io_fd b_open (char * filename, int flags);

This function opens a file for buffered I/O, returning a file descriptor. It initializes the file system on first use and checks if the file exists or needs to be created depending on flags like O_CREAT. It allocates a File Control Block (FCB), retrieves the directory entry, sets up metadata like buffer pointers, current block, and file size, and returns the descriptor.

int b_read (b_io_fd fd, char * buffer, int count);

This function reads data from a file into the provided buffer. It handles three phases: reading leftover data from the FCB buffer, reading full block-sized chunks directly into the user buffer, and reading any remaining bytes via a buffer refill. It manages block traversal using the FAT and updates internal FCB state accordingly.

int b_write (b_io_fd fd, char * buffer, int count);

This function writes data from the provided buffer into the file. Data is first written into the FCB's internal buffer. If the buffer fills up, it is flushed to disk and the FAT is updated, including allocating new blocks if needed. This process continues until all data is written. The function keeps track of the buffer index and file size.

int b_close (b_io_fd fd);

This function closes a file and deallocates its FCB. If the file was opened for writing and the buffer contains unwritten data, it flushes the buffer to disk, updates the file size, and writes the updated directory entry. It then frees the memory used for the buffer and parent directory entries, marking the FCB as available for reuse.

int fs_isFile(const char * filename); //return 1 if file, 0 otherwise

This function takes a path to a file and returns 0 if it's not a file and 1 if it is a file. It calls getDirEntryFromPath(), If getDirEntryFromPath() returns null, then we know the path doesn't exist, and we return 0. If we get back a valid DirEntry, then we check the isDirectory field. If isDirectory == 0, meaning it's a file, we set isFile to 1. If isDirectory == 1, meaning it's a Dir Entry, then we set isFile to 0. And then we return isFile.

int fs_isDir(const char * pathname); //return 1 if directory, 0 otherwise

This function takes a path to a directory and returns 0 if it's not a directory and 1 if it is a directory. It calls getDirEntryFromPath(), If getDirEntryFromPath() returns null, then we know the path doesn't exist, and we return 0. If we get back a valid DirEntry, then we check the isDirectory field. If isDirectory == 0, meaning it's a file, we set isDir to 0. If isDirectory == 1, meaning it's a directory, then we set isDir to 1. And then we return isDir.

void getAbsolutePath(char* absolutePath, const char *pathname);

This function takes a pathname and returns the absolute path into the provided absolutePath buffer. We first check if the given path starts with "/", meaning it's already an absolute path, If so, we copy the path into a temp buffer "tempPath". If it's a relative path, we copy the cwd into the temp buffer, we then append a "/", and then append the path name, to get the full path. After that, we break the full path into singular elements using strtok(). In the loop, we store each element in char* pathElements, and pathElementCount keeps track of how many elements we store in it. If the element is a "." then we decrease pathElementCount by 1, if the element is "" we ignore it. After this, we go inside a for loop that copies all the elements in pathElements and puts them all together for us, adding "/" between elements into our absolutePath buffer.

void getParentPath(char* parentPath, char *pathname);

This function takes a full path and writes the path to its parent directory into the parentPath buffer. We first copy the pathname into pathCopy since strtok() modifies the original string. We then split up the path by "/" as the delimiter, store each element in pathElements, and keep count of how many elements there are using pathElementCount. We then append a "/" to the parentPath buffer. Then we go in a for loop that runs for pathElementCount - 1 and append every element into our parentPath buffer except the last element. We also check not to add a "/" into the last element.

void getLastElementName(char * elementName, char * path);

This function takes a path and gives us the last element of the path, and writes it into elementName buffer. We first copy the pathname into pathCopy since strtok() modifies the original string. We then split up the path by “/” as the delimiter, store each element in pathElements, and keep count of how many elements there are using pathElementCount. If no elements are found (“/”), we manually set the path element to “.” If elements are found, we copy the last element from pathElements into the elementName buffer.

A detailed table of who worked on what, down to the individual function level

Name	Parts/Functions
Nabeel	fs_create, fs_delete, fs_move, findNextFreeBlock, allocateBlocks, writeDirEntries, b_open, b_read, b_write, b_close, initFreeSpace
Leigh	fs_getcwd, fs_setcwd, readDirEntries, fs_mkdir, fs_rmdir, initRootDirectory
Ethan	fs_opendir, fs_diriteminfo, fs_closedir, getDirEntriesFromPath, getDirEntryFromPath, initFileSystem
Bryan	fs_isFile, fs_isDir, getAbsolutePath, getParentPath, getLastElementName, exitFileSystem

Screen shots showing each of the commands listed in the readme

ls, cd, pwd

```
Prompt > ls -la

D          0  .
D          0  ..
-         9548  README.md
D          0  my_files
-          0  new_file
Prompt > cd my_files
Prompt > pwd
/my_files
Prompt > cd ..
Prompt > pwd
/
Prompt > 
```

Ln 2, Col 18 Spaces: 4 UTF-8 LF Plain

md

```
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
Prompt > md my_files
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
D          0  my_files
Prompt > 
```

touch

```
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
D          0  my_files
Prompt > touch new_file
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
D          0  my_files
-          0  new_file
Prompt > 
```

cat

```
Prompt > cp2fs hello_world.txt hello_world.txt
Prompt > ls -la

D          0  .
D          0  ..
-         9548  README.md
D          0  my_files
-          0  new_file
-          31  hello_world.txt
Prompt > cat hello_world.txt
Hello, World!
Nice to meet you.
Prompt > 
```

rm

```
Prompt > ls -la

D          0  .
D          0  ..
-          31  hello.txt
-         9548  copy_README.md
Prompt > rm copy_README.md
Prompt > ls -la

D          0  .
D          0  ..
-          31  hello.txt
Prompt > 
```


cp

```
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
D          0  my_files
-          0  new_file
-          31  hello_world.txt
Prompt > cp hello_world.txt hello_copy.txt
Prompt > ls -la

D          0  .
D          0  ..
-        9548  README.md
D          0  my_files
-          0  new_file
-          31  hello_world.txt
-          31  hello_copy.txt
Prompt > cat hello_copy.txt
Hello, World!
Nice to meet you.
Prompt > 
```

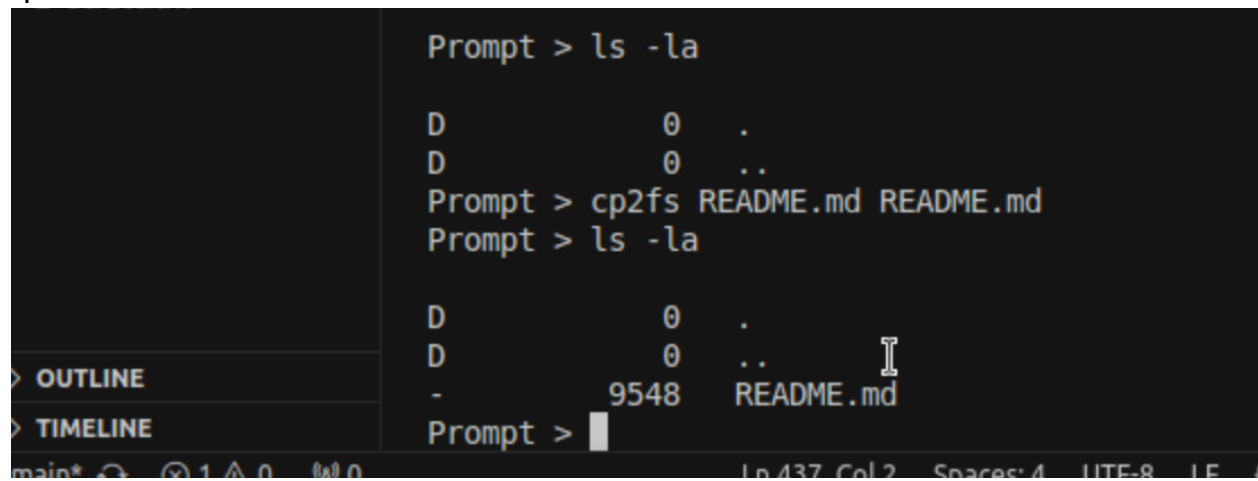
mv

```
Prompt > ls -la

D          0  .
D          0  ..
-         9548  README.md
D          0  my_files
-          0  new_file
-          31  hello_world.txt
-          31  hello_copy.txt
Prompt > mv hello_copy.txt hello_move.txt
Prompt > ls -la

D          0  .
D          0  ..
-         9548  README.md
D          0  my_files
-          0  new_file
-          31  hello_world.txt
-          31  hello_move.txt
Prompt > cat hello_move.txt
Hello, World!
Nice to meet you.
Prompt > 
```

cp2fs



```
Prompt > ls -la
D          0  .
D          0  ..
Prompt > cp2fs README.md README.md
Prompt > ls -la
D          0  .
D          0  ..
-        9548  README.md
Prompt >
```

The image shows a terminal window with a dark background and light-colored text. On the left side, there is a sidebar with two expandable sections: 'OUTLINE' and 'TIMELINE', both preceded by a right-pointing chevron. The main area of the terminal displays the output of several commands. First, 'ls -la' is run, showing directory entries for '.' and '..'. Then, 'cp2fs README.md README.md' is executed. Finally, 'ls -la' is run again, showing the same directory entries plus a new file 'README.md' with a size of 9548 bytes. The prompt 'Prompt >' is visible at the end of the last command line.

cp2l

```
mfs.o
① README.md
SampleVolume M
scratc.txt U

| cd | ON |
| md | ON |
| pwd | ON |
| touch | ON |
| cat | ON |
| rm | ON |
| cp | ON |
| mv | ON |
| cp2fs | ON |
| cp2l | ON |
|-----|
Prompt > ls

OUTLINE
TIMELINE
README.md
Prompt > 
```

```
mfs.o
NEW_README.md U
① README.md
SampleVolume M
scratc.txt U

| md | ON |
| pwd | ON |
| touch | ON |
| cat | ON |
| rm | ON |
| cp | ON |
| mv | ON |
| cp2fs | ON |
| cp2l | ON |
|-----|
Prompt > ls

README.md
Prompt > cp2l README.md NEW_README.md
Prompt > 
```

Screen shot of compilation and updated prompt with current working directory

```
student@student:~/repos/csc415-filesystem-nabware$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -o fsshell fsshell.o fsInit.o mfs.o b_io.o fsLow.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
Initializing File System with 19531 blocks with a block size of 512
|-----|
|----- Command -----| Status |
| ls                    |     ON  |
| cd                    |     ON  |
| md                    |     ON  |
| pwd                   |     ON  |
| touch                 |     ON  |
| cat                   |     ON  |
| rm                    |     ON  |
| cp                    |     ON  |
| mv                    |     ON  |
| cp2fs                 |     ON  |
| cp2l                  |     ON  |
|-----|
/ > ls

README.md
my_files
new_file
hello_world.txt
hello_move.txt
/ > cd my_files
/my_files > █
```