Evan Caplinger ID: 924990024
Alejandro Cruz-Garcia ID: 923799497
Alexander Tamayo ID: 921199718
Ronin Lombardino ID: 924363164
Github: RookAteMySSD CSC415 Operating Systems Fall 2025 Section 2

Assignment: File System (Final Submission)

Team #1 Victory Royal

https://github.com/CSC415-2025-Fall/csc415-filesystem-RookAteMySSD

Description

This project tasked us with creating an entire filesystem from the ground up. This was split into 3 milestones.
Milestone 1 had us creating the low-level parts of the filesystem, we created the VCB, Freespace manager, and the Root directory, as well as creating the directory entry structure.
Milestone 2 has us implementing the directory functions that were defined in mfs.h.
Milestone 3 has us finishing the file functions and ensuring that all the shell commands in fsshell.c functioned.

Plan

Milestone 1:
In this milestone we split the tasks up into 3 modules; the VCB, the Freespace, and the Root Directory as described below from the first submission:

Volume Control Block

Our volume control block will the following information:

1. A signature or “magic word”
2. Some general volume characteristics: the total number of blocks and the size of each block in bytes.
3. Information used by the free-space manager, like the number of free blocks, the index of the first free block, and the index of the last free block.
4. Block indices of the first blocks of some important structures: FAT and root directory.

When formatting the volume, we determine the block size and total size of the volume. We write the VCB to block 0, the FAT starting at block 1 (more information on how the FAT is formatted can be found later in this document), and the root directory at the block following the last block of the FAT. We want to allocate enough memory for the root directory that it can be stored contiguously at the beginning of the volume, although it may be expanded later. Thus the root directory will contain a number of empty or unused directory entries at formatting. The following is the definition of our VCB:

```
#include <stdint.h>
typedef struct {
    uint32_t signature;
```

```
    // volume characteristics
    uint32_t blockSize;
    uint32_t numBlocks;

    // free space variables
    uint32_t fatStart;
    uint32_t fatNumBlocks;
    uint32_t firstFreeBlock;
    uint32_t lastFreeBlock;

    // root dir info
    uint32_t rootStart;
    uint32_t rootSize;
} vcb;
```

In our file system initialization routine, we load the VCB from disk and check to see whether the magic number is present in the signature; if so, we assume the volume has been formatted, and if not, we initialize all values, initialize the FAT, create the root directory, and return.

Free Space System

We are using a File Allocation Table. This is a flat array of ints stored at the beginning of the volume. Each block in memory has an int in this table associated with it; this value at each block's index then points to another block number, This forms a linked list or several sublists within the array that describe the blocks that are allocated for files or the blocks that are free. The end of a file is represented by a sentinel value 0xFFFFFFFE ("EOF"). Space for the VCB and FAT will be represented by another sentinel value, 0xFFFFFFFF ("RESERVED").

Free blocks will also be represented as a linked list. The VCB will contain the index of the first free block, which will point to the next free block, and so on. The last free block will contain the EOF sentinel value.

When the drive is formatted, the first free block will be the block immediately after the end of the FAT. Each block in memory will then point to the subsequent block. The last free block will be the last block on the drive. The VCB will contain an index of lastFreeBlock, making this operation cheap. Since each entry of the FAT is a 32-bit integer, the space that the FAT occupies in bytes will be the number of blocks in the volume times 4 bytes per block.

When allocating space for a file, the free-space management system will start at the block pointed to by the firstFreeBlock value in the VCB, and then chain through the free-block linked list until it has allocated enough blocks. It will set the firstFreeBlock value in the VCB to the block *pointed to* by the last block allocated, and it will set the FAT value for the last allocated block to EOF.

When blocks are deallocated, the value in the FAT for the current last free block in the volume will be modified to point to the *first* block that was just deallocated. The value lastFreeBlock in the VCB will be modified to point to the *last* block that was just deallocated. This way, the head of the free-blocks linked list will point to a large amount of contiguous memory, limiting fragmentation within files, until we write to the end of the drive.

When initializing the FAT, the first free block will be the block immediately after the end of the FAT. Each block in memory will then point to the subsequent block. If the first free block is EOF, then we are out of disk space.

Directory System

Our directory entry contains the name of the file, an attributes value, the start block, and the length of the file in bytes. It also contains 3 timestamps for use with the ls -l command, created, accessed, and modified. The following is the Directory Entry definition:

```
#include <stdint.h>
typedef struct
{
    char name[47]; // name field; 47 bytes to eliminate internal padding
    uint8_t flags;      // bitmap for DE properties:
                        // DE_IS_USED (bit 0), DE_IS_DIR (bit 7)
    uint64_t created;   // created UNIX timestamp
    uint64_t accessed;  // accessed UNIX timestamp
    uint64_t modified;  // modified UNIX timestamp
    uint32_t size;      // size in bytes
    uint32_t location;  // start block
} DE;
```

These modules were completely implemented and submitted as part of Milestone 1.

Changes:

Our initial plan did not consider where in memory the VCB would live while the file system was active. We ended up deciding to have a single global VCB instance accessible by all modules via a retrieval function and write this to disk every time we updated it.

Our initial idea for the VCB contained some extraneous values that did not make it into the final iteration.

Milestone 2:

Similar to the divide and conquer approach we used for milestone 1, we split the commands into 4 different parts to be worked on. We had the parsePath function as one part, and then

had a part for the Key functions, Iterator functions, and miscellaneous functions. During this milestone we implemented all these functions except the fs_stat function.

In general, Evan was in charge of directory tracking and implementation of the “cd” and “pwd” commands; Alex was in charge of making and removing directories (mkdir and rmdir) and supplied an implementation of parsePath; Ronin was in charge of reading and traversing directories (opendir, readdir, closedir); and Alejandro was in charge of all other functions for this step.

One module that was completed for this step included the directory tracking system, as described below:

Directory Tracking System

In order to track and switch between directories, we used a stack of directory entries to represent the current location within the directory tree. This stack has a constant size that represents the maximum depth of the directory tree. When changing to a subdirectory of the current directory, we push a new DE containing the directory entry (name, location, etc. - as defined above) for that subdirectory onto the CWD stack. When moving from a subdirectory to its parent (to “..”), we pop an element from the stack. The stack is initialized to contain only one element, which is the root directory. The elements of the subdirectory are all pointers to heap-allocated DEs which must be freed and nulled when popping an element.

This behavior is mostly implemented in fs_setcwd and its internal helper function setcwdInternal, so for further information refer to the definition of that function as well as the explanation further below in this writeup. Essentially, if given a relative path, this function will pop and push elements onto the stack as necessary; if given an absolute path, this function will do the same but first pop all elements except the root directory.

We also used a slightly modified version of the parsePath implementation given in class for many of the other aspects of this milestone. However, most other functions were largely independent without an overarching system like the CWD subsystem.

Changes:

Our initial plan did not specify which functions would be dependent on parsePath. Therefore, we had some redundant code for parsePath that we eventually decided to scrap in favor of just using parsePath to handle it.

The CWD system had to undergo some extensive debugging because the initial implementation was quite buggy. These issues are described below, but some aspects of how this system functioned were changed as a result of our debugging effort.

Milestone 3:

At this point there were only a few functions and a lot of bugfixing to complete. We split the last few functions that needed to be completed, those being fs_stat, b_open, b_close, b_read, b_write, and cmd_mv. During this milestone, we completed all the functions and got all of the commands and filesystem working. The final portion of milestone 3 consisted of debugging the functions that we had all implemented separately, during which time we were in close communication with each other to ensure that we weren't stepping on each other's toes. We all did very edge case testing, independently poking and prodding the file system to see if we could induce any errors. Because of this, we were able to identify and eliminate a large number of bugs.

This milestone contained the following system:

File Handling System

We implemented all file handling in b_io.c without any further helper functions. The definition of our FCB (file control block) is below:

```
typedef struct b_fcb {
	char * buf; //holds the open file buffer
	int index; //holds the current position in the buffer
	int buflen; //holds how many valid bytes are in the buffer
	uint32_t startBlock; // first data block (FAT head)
	uint32_t fileSize;   // Current file size in bytes
	uint32_t filePOS;     // Byte Cursor within file
	int bufBlockIdx;     // Which logical file block is cached in buf, -1
if none

	int bufDirty;          // 1 if buffer must be flushed to disk
	int flags;             // O_RDONLY, O_WRONLY, O_RDWR, O_APPEND
	uint32_t blockSize;  // VCB block size

	int inUse;             // Mark FCB used

	DE* parentDirectory;    // the parent directory loaded into memory so
that we can write to reflect access/modified times
	int directoryIndex;        // index within the parent directory

	int canRead;      // depends on flags; false if write-only
	int canWrite;     // depends on flags; false if read-only
} b_fcb;
```

All information about file handling can more or less be found below in the descriptions of our implementations of b_open, b_read, b_seek, b_close, b_write. This system was developed in collaboration between Evan and Alejandro, with Alejandro doing the bulk of the buffered I/O logic and Evan doing resource management work under Alejandro's direction.

Changes:

This milestone was relatively smooth and did not significantly deviate from our plan. Of course, debugging presented some significant challenges, but we expected that the debugging step would be challenging, and we all put in significant work to get past this step to a working FS.

FS description

Our Filesystem uses a FAT (file allocation table) and is thusly non-contiguous, using a linked list to manage blocks in use or free. Each Directory entry only needs to store the first block where it's located and our Free Space manager can handle the rest as we can iterate through the blocks to find our sentinel value. The FS modules are described above.

Shell Commands

LS	Working
CP	Working
MV	Working
MD	Working
RM	Working
CP2L	Working
CP2FS	Working
CD	Working
PWD	Working
TOUCH	Working
CAT	Working

Issues and Resolutions

Milestone One

Directory entry definition

- Types were not compiler native. Replaced uint32_t with unsigned int and uint8_t with unsigned char. We later reinstated the stdint.h types, as the provided code harness mostly used stdint.h types.
- Data was incorrectly formatted, causing unnecessary padding and an unnecessarily large directory entry. We rearranged data entries to account for this.
- Size of the file should be in bytes, not blocks.

Miscellaneous

- Learning how to get the current time in C and learning how to cast types in order to get the getCurrentTime() function to work and return the proper uint type. Initially we were casting time_t as a uint32_t; by examining the sizeof time_t, we chose to use a uint64_t to hold time instead.
- Learning that there is no scope resolution operator in C and coming up with an alternative way to set a global variable that at the time had the same name as a local variable. However the best solution was to change the name of the global variable.
- In an early version of the project, we were checking for the return value of LBAwrite() incorrectly, causing the program to fail before the volume could be written. Rather than checking that the correct number of blocks were written, we were checking for the return code 0. Fixing this resolved the issue.
- An early implementation of initFileSystem() was not properly reading the VCB from disk into the VCB buffer, and thus always detected the file system as uninitialized. We added code to read in the VCB and this solved the issue.
- allocateBlocks() was initially placing EOF one block after the end of allocated space, within the FAT. We moved the code that writes EOF inside the main loop that iterates over the free space linked-list, and this solved the problem.

Evan Caplinger's Individual Issues

Issue

Functions / components: addEntryToDirectory, setcwdInternal

Description: In the initial implementation, there was no system to resize a directory once all initially allocated directory entries had been filled. Designing a system to resize a directory proved especially troublesome. There were the following issues:

- After resizing, although the "." entry within the directory itself was updated to reflect the new size, the entry within the parent directory was not – and this was used to load the directory into memory. Since the size was never updated, the newly allocated DE's would not be loaded into memory.

- When creating DE's (files and directories), the DE would be successfully added to the directory, but the entry loaded into the directory traversal system (the CWD DE) would not be updated, so in cases where one removed a file allocated beyond the *initial* size of a directory, the file system would not reflect this unless we first moved out of the CWD and back in.

Resolution: Here are resolutions for the two sub-issues described above.

- Added new code to update the entry within the parent directory.
- Added a new function, reloadCwd(), which simply automatically exited and re-entered the CWD – a somewhat hacky fix, but actually relatively clean and without much overhead. This function would then be called by any function that modified the CWD, such as createFile, createDir, deleteFile, etc.

Note that the instability of the CWD system due to not being regularly refreshed caused a myriad of other issues, not all of which are described here. For example, cd would spuriously fail, and adding new DE's to subdirectories would fail unpredictably. As far as we can tell, the fixes described above resolved all of these bugs.

Issue

Functions / components: fs_readdir

Description: This function would iterate through the directory until it either found a *used* DE *or* reached the end of the directory. However, if it reached the end of the directory, it would not check whether the DE was used or not, so attempts to delete the last entry in a directory would succeed but the *ls* command would still show the now-deleted file.

Resolution: Have fs_readdir perform the check of whether the file is used or not separately; explicitly return NULL if we reach the end of a directory without finding a used DE.

Issue

Functions / components: fs_delete

Description: The rm command failed to successfully delete files.

Resolution: The initial implementation did not use parsePath, so I updated the function to replace the directory traversal to rely on the parsePath function. This resolved the issue without too much detailed debugging work.

Issue

Functions / components: allocateBlocks

Description: When allocating blocks, the free space pointer in the VCB was being updated to point one block beyond where it should, resulting in wasted space in the FAT and a couple other bugs.

Resolution: This was based on faulty thinking of how the loop within the allocateBlocks function was working. I removed a single traversal to the next block and the issue was resolved.

Issue

Functions / components: freeBlocks

Description: When freeing blocks, we would write beyond the

Alejandro's issue's/resolutions

1. **Files reported as directories / rm refused to delete files**

Issue: fs_isFile was checking the directory bit (DE_IS_DIR), so regular files were being treated as directories and cmd_rm printed "neither a file or a directory"

Fix: Changed the logic to treat entries as files when DE_IS_USED is set and DE_IS_DIR is not set, so fs_isFile and fs_isDir now correctly distinguish files vs. Dirs, and rm can delete files

2. **Cp2fs, touch, and cat created / saw zero-byte files**

Issue: b_open, b_write, and b_close weren't fully wiring up startBlock, fileSize, filePOS, and the directory entry, so files existed but size stayed 0, and cat printed nothing

Fix: in b_open/b_write/b_close we now allocate data blocks when needed, grow/shrink the FAT chain with resizeBlocksSmart, ensureBlocksForWrite, track filePOS and fileSize correctly, flush dirty buffers, and write the updated direcotry entry back to disk

3. **Cat showed nothing even when byte count looked write**

Issue: The buffered read path didn't always load the correct logical block for the current filePOS, and EOF/partial-block handling was incosistent.

Fix: Reworked b_read to: ensure the buffer holds the correct logical block using loadBlock, copy any remaining bytes from the current block, optionally fast-path full blocks directly into the caller's buffer via the FAT, and handle the final partial block cleanly without reading past fileSize

4. **FreeBlocks spammed fileEnd is 0x... and went on forever**

Issue: The original logic walked the chain and printed fileEnd each step, with weak stopping conditions, so a corrupted or incorrectly linked FAT could cause endless loop of prints

Fix: I changed the loop to track both the current block and the last valid block, stop cleanly on FAT_EOF or FAT_RESERVED, update lastFreeBlock once, and then write the FAT and VCB back once

5. **LoadDirectory and writeFileToDisk assumed contigous blocks**

Issue: Earlier versions treated the file/directory as if it lived in a single contiguous region, instead of following the FAT chain, so non-contiguous allocations would break the reads/writes

Fix: loadDirectory now allocates numBlocks * blockSize, walks the FAT chain block-by-block with getNextBlock, and fills the buffer. WriteFileToDisk walks the logical blocks with getBlockOfFile, handles full vs partial blocks, and zero-pads the last block into a temp buffer before calling LBAwrite

6. **Fs_delete could not reliably delete nested files**

Issue: The deletion path did its own path tokenizing the directory walking, which was fragile and easy to desync from how the rest of the FS resolved paths

Fix: I rewrote fs_delete to resolve the file via ParsePath, verify it's a used, non-directory entry (not . Or ..), free its FAT chain with freeBlocks, clear the DE, and write the containing directory back writeBlocksToDisk

7. **mv copied incorrectly and didn't clean up the source**

Issue: The mv command was mixing Linux open() with b_open so it wasn't actually able to read data properly and kept throwing an error at me

Fix: I simply just changed open() to b_open()

8. **Small edge cases**

Issue: A few edge cases existed: multiple incompatible flags b_open, seeking with the wrong base, and not always syncing timestamps/size back to disk

Fix: b_open now enforces at most one of the O_RDONLY/O_WRONLY/O_RDWR and handles O_CREAT, O_TRUNC, and O_APPEND correctly. b_seek updates filePOS with bounds checks and resets the buffer. b_close flushes the buffer and syncs size/timestamps. fs_stat uses ParsePath and the real DE to fill out the file metadata accurately

Alex Tamayo Issues/Resolutions for Milestone 2/3

Issue – When overwriting an existing file, a duplicate entry was being created instead

```
Prompt > touch /file1.txt
Prompt > touch /file2.txt
Prompt > mv /file1.txt /file2.txt
Prompt > ls /

file2.txt
file2.txt  <-- duplicate

Prompt >
```

Resolution – In the process of overwriting an already existing file, destParent is freed and then I tried to load it again with destPPI.parent. I can't do that since destPPI.parent is destParent which I had JUST freed. Both destParent and sourceParent need to be reloaded when they are pointing to the same directory. So instead, I saved the location and size of parent first, then freed the memory of both destParent and sourceParent. Once freed, I could then reload parent using the location and size saved before deletion. As a precaution, I added error handling in the event of failing to overwrite or reload the directory destination.

Issue - When attempting to overwrite into an entry in another directory, cmd_mv would be unable to detect the location of the entry.

Resolution - I was trying to use the index in the parent directory when calling isDirectoryEmpty(). But isDirectoryEmpty() is looking for the entry count of the target directory. I refactored code in deleteEntry() to calculate the entry count.

Issue - When attempting to overwrite, I still would receive a failed to overwrite destination error. After a lot of deliberating over cmd_mv and its helper functions, I This time, it was due to touch possibly creating empty files with a location = 0, meaning that no blocks had been allocated yet. When deleteEntry() tries to free an empty files blocks, it would fail.

Resolution – I rewrote deleteEntry() to be able to properly handle empty files. With the rewrite, it now checks if location is valid before calling freeBlocks(), made sure that it doesn't return an error if freeBlocks() fails as empty files may not even have blocks to free, and always return success if a directory entry was successfully removed.

Ronin's Issues and resolutions

Issue – Allocate blocks was allocating an additional block when it was called.

Resolution – The sentinel value needed to be set during the last loop as opposed to outside it as the currentBlock value is updated inside the loop.

Issue – Resize Blocks was allocating way too many extra blocks when growing the size of the block chain passed in.

Resolution – initially the function was intended to take our new size in Blocks, however it was more logical to pass in the new size in bytes which was being done. Thus the function had to be updated to expect bytes as an input.

Issue – md, rm, and cd were not working properly in Subdirectories, newly created directories would be overridden and we could not remove or enter the newest created subdirectory without first leaving and re-entering the current directory

Resolution – This issue was caused by a mismatch of what was on disk and our cwd pointer in. memory. Evan and I were able to debug this and create a function that reloads the cwd in RAM after make directory and make file operations which resolved the issue.

Work Division Table

Below is a summary of the work done by each group member. These functions are further described in the function description section.

Group Member	Functions
Evan Caplinger	• b_open

	• b_close • writeBlocksToDisk • findEntryInDirectory • cwdBuildAbsPath • getcwdInternal • setcwdInternal • saveCwdState • restoreCwdState • initCwdAtRoot • freeCwdMemory • addEntryToDirectory • isDirectoryEmpty • initFAT • uninitFAT • loadFAT • freeBlocks • resizeBlocks (debugging) • getNextBlock • initFileSystem • exitFileSystem • getGlobalVCB • initVCB • fs_setcwd • fs_getcwd
Alejandro Cruz-Garcia	• b_open (debugging) • b_read • b_write • b_seek • writeFileToDisk • findInDirectory • loadDirectory • initFileSystem • fs_isFile • fs_isDir • fs_delete • fs_stat • cmd_mv
Ronin Lombardino	• b_close

	• getCurrentTime • allocateBlocks • resizeBlocks • getBlockOfFile • fs_mkdir (debugging) • fs_rmdir (debugging) • fs_opendir • fs_readdir • fs_closedir • fs_delete (debugging) • fs_stat
Alexander Tamayo	• createFile • createDir • parsePath • isDEaDir • fs_mkdir • fs_rmdir • cmd_mv • extractFileName • getDirectorySize • attachEntry • renameInPlace • updateDotDot • isAncestorOf

Function Descriptions

Name: b_open

Description: User-facing function. Given a filename and a number of flags, opens a file, creates it if necessary, and allocates memory so that it can be read from and written to. First, it does a number of validity checks, including whether there is even a free FCB to write to. If so, then it uses parsePath to find the file. If it doesn't exist and the user hasn't specified O_CREAT, it simply returns an error code.

It processes the flags in the following order: O_CREAT, O_TRUNCATE, O_APPEND. The flags O_WRONLY, O_RDONLY, and O_RDWR are handled simply by setting the flags in the FCB. It then returns the *index* of the allocated FCB. It sets FCB variables accordingly and allocates space for the

Who worked on it: Evan Caplinger, Alejandro Cruz-Garcia (debugging)

Name: b_read

Description: This function consists of three parts:

1) What can be filled from the current buffer which may or may not be enough
2) After using what was left in our buffer, there are still 1 or more block size chunks needed to fill out the caller's request. This represents the number of bytes in multiple block sizes.
3) A value less than blocksize which is what remains to copy to the callers buffer after fulfilling part 1 and part 2. This would always be filled with a refill of our buffer.

This function utilizes helpers such as loadBlock, getBlockOfFile, and LBAread.

Who worked on it: Alejandro Cruz-Garcia

Name: b_write

Description: This function is an interface. It writes to the buffer where your position is and it look out for flags. This function uses a helper I wrote called ensureBlockForWrite, and also uses the flushBlock that I also had wrote to flush the blocks.

Who worked on it: Alejandro Cruz-Garcia

Name: b_seek

Description: updates the file’s current read/write position based on SEEK_SET or SEEK_CUR. It validates the file descriptor, computes the new position, ensures it is non-negative, and resets the buffer state so the next rad or write reloads the correct block.

Who worked on it: Alejandro Cruz-Garcia

Name: b_close

Description: Finalizes an open file by flushing any dirty buffered block to disk, updating the file;s directory entry with its final size and timestamps, and writing the updated directory back to disk. It then frees the file buffer and any loaded parent directory, resets the file control block's fields, and marks it as unused. This ensures all pending writes are saved and all temp resources are released.

Who worked on it: Alejandro Cruz-Garcia and Ronin Lombardino

Name: createDir

Description: Builds a new directory table in RAM, reserves enough disk blocks via the FAT allocator, fills "." and "..", marks all remaining entries as unused (reserved capacity) and then writes the directory table to disk using the fat mapping

Who worked on it: Alejandro Cruz-Garcia

Name: createFile

Description: Once input is validated createFile will get the currentTime which will be used for timestamps that are created, modified, or accessed. Then it will create a blank directory entry, initializing all fields to zero and sets file metadata. It will then set all timestamps to the current time, and then add the entry to the parent directory, returning the index where the entry was added, or a –1 on error.

Who worked on it: Alexander Tamayo

Name: writeFileToDisk

Description: Writes a file's in-memory data back to disk blocks assigned to it in the FAT. It walks through the file's FAT chain block-by-block, writing full blocks directly and handling the final partial block with a padded temporary buffer so no garbage data is written. If any block write fails or the garbage data is written. If any block write fails or the FAT chain is invalid, the function stops and returns an error.

Who worked on it: Alejandro Cruz-Garcia

Name: writeBlocksToDisk

Description: Given a buffer of raw byte data, a location, and a number of blocks, this function will iterate over blocks within a file and write the specified number to disk by iteratively calling LBAwrite with a write block count of 1, and going to the next block by calling getNextBlock. It returns the total number of blocks written, which is either the full amount requested by the caller or some smaller number if we reach EOF.

Who worked on it: Evan Caplinger

Name: getCurrentTime

Description: This function returns the current time as an unsigned integer using C's built in t_time.

Who worked on it: Ronin Lombardino

Name: findInDirectory

Description: This function iterates through a directory and returns the index of a DE with a certain name.

Who worked on it: Alejandro Cruz-Garcia

Name: findEntryInDirectory

Description: This function is equivalent to findInDirectory, but it returns a pointer to the entry itself, rather than just the index.

Who worked on it: Alejandro Cruz-Garcia

Name: loadDirectory

Description: loads an entire directory table from disk into memory by following the FAT-mapped block chain associated with that directory. Using the directory's starting block and total size, the function determines how many physical blocks must be read, allocates a contiguous in-

memory buffer large enough to hold the directory contents, and sequentially performs block reads along the FAT chain.

Who worked on it: Alejandro Cruz-Garcia

Name: parsePath

Description: parsePath function as provided by the professor in class with some added error handling. ParsePath() breaks apart a directory entry's path into smaller components. Parses and navigates in one function. It will return parent loaded and with a name, then it will find the target's index in parent, all in a single pass.

Who worked on it: Alexander Tamayo, Robert Bierman

Name: isDEaDir

Description: Helper function to check if DE is a directory. It performs a bitwise operation and comparison. It accesses the "flags" member within the directory entry structure pointed to by "entry". Then it performs the bitwise AND operation between the aforementioned value and DE_IS_DIR which is our bitmask. The return value will non-zero if and only if the corresponding bit is also set in "entry -> flags". If the bit is 0 in the flags, the result of the AND operation will be 0.

Who worked on it: Alexander Tamayo

Name: cwdBuildAbsPath

Description: This feeds a result to the user-facing fs_getcwd function by walking along the CWD stack and adding each directory name, separating them by forward-slashes.

Who worked on it: Evan Caplinger

Name: getcwdInternal

Description: This function simply returns the top element on the CWD stack, a heap-allocated directory entry, which can then be used to load the current working directory.

Who worked on it: Evan Caplinger

Name: setcwdInternal

Description: Given a path, set the current working directory to that path by altering the CWD stack. This function will always walk the entire path provided by the user. It first does a few validity checks on the provided path, including calling parsePath on it to verify its validity. However, then the function tokenizes the string again, so that it can add all the component directories to the CWD stack. It also saves the current state of the stack so that if any parsing fails it can restore the old state of the CWD stack.

Who worked on it: Evan Caplinger

Name: saveCwdState

Description: Helper function for setcwdInternal. This creates a duplicate of the current CWD stack, with new heap allocations, so that when we alter the CWD stack through setcwdInternal, we can restore the previous state if there was a problem.

Who worked on it: Evan Caplinger

Name: restoreCwdState

Description: Helper function for setcwdInternal. This restores the backup copy of the CWD stack by freeing the current CWD stack and moving all internal pointers over from the backup stack to the primary stack.

Who worked on it: Evan Caplinger

Name: initCwdAtRoot

Description: Called when FS is initialized. It sets up the directory traversal system and sets the CWD to root.

Who worked on it: Evan Caplinger

Name: freeCwdMemory

Description: Releases all resources associated with the directory traversal system.

Who worked on it: Evan Caplinger

Name: addEntryToDirectory

Description: This function does exactly what's on the sticker: given a directory entry and a parent directory, it inserts the directory entry into the parent. It finds an unused DE in the parent, or if there is no unused DE, it resizes the parent to fit the new DE, allocating new blocks if necessary. It also rewrites the whole parent DE to disk *as well as* the DE in the parent of the parent, which will contain the new size (if necessary) and modification time of the parent DE so that everything loads properly.

Who worked on it: Evan Caplinger

Name: isDirectoryEmpty

Description: Checks whether a given already-loaded directory has no entries except for "." and ".." in it.

Who worked on it: Evan Caplinger

Name: initFAT

Description: Only called once at FS initialization / formatting. It determines the size of the FAT, and allocates space for it. As described above, the FAT is a linked list. This function reserves blocks for the FAT and VCB, and then for all remaining blocks it points the value in the FAT to point to the next block, with EOF at the very last block of the volume. Then it writes the FAT to memory.

Who worked on it: Evan Caplinger

Name: uninitFAT

Description: Releases all resources related to free space management.

Who worked on it: Evan Caplinger

Name: loadFAT

Description: Simply reads the FAT from disk into memory.

Who worked on it: Evan Caplinger

Name: allocateBlocks

Description: This function returns the head of a chain of blocks of a given size. It does so by iterating through our freespace chain and creating a new chain which is to be returned. It will also update the head of our freespace block.

Who worked on it: Ronin Lombardino

Name: freeBlocks

Description: Starting at a certain location, walks along all blocks until we eventually reach EOF and frees them. Updates the firstFreeBlock variable in the VCB to point to the first of the blocks that we freed, and the lastFreeBlock variable to point to EOF of this file, thereby adding these blocks to the end of the free-block list. Then it writes the VCB and FAT back to disk.

Who worked on it: Evan Caplinger

Name: resizeBlocks

Description: This Function determines if the requested new size is more or less blocks then the current amount of blocks in the chain starting from the given starting block, it then calls the correct function (allocateBlocks or freeBlocks) with the difference in size. This function needs to iterate through the chain to determine the current size of the chain.

Who worked on it: Ronin Lombardino, Evan Caplinger

Name: getBlockOfFile

Description: Given a file and an index *n* within it, this function walks along the FAT and returns the location on disk of the *n*th block of the file.

Who worked on it: Ronin Lombardino

Name: getNextBlock

Description: A more lightweight version of getBlockOfFile that simply returns the next block of the block given as an argument, or EOF if that block is the last block. Created to reduce overhead of looping through entire file in getBlockOfFile.

Who worked on it: Evan Caplinger

Name: initFileSystem

Description: Initializes the file system on startup. First, it loads the VCB into memory and checks whether the file system is already initialized by checking the VCB signature. If not, it calls the function that formats the FS, initVCB. Then it initializes the CWD tracking system by calling initCwdAtRoot.

Who worked on it: Evan Caplinger, Alejandro Cruz-Garcia

Name: exitFileSystem

Description: Culmination function that calls a couple other cleanup functions; mostly just handles resource releasing. Also prints a goodbye message for the user.

Who worked on it: Evan Caplinger, Robert Bierman

Name: getGlobalVCB

Description: We have one instance of the VCB in memory for the entire FS. This function just returns a pointer to that instance.

Who worked on it: Evan Caplinger

Name: initVCB

Description: Initializes the VCB when the FS is created for the first time. Gives the VCB its magic word (signature). Takes values like block size and number of blocks, and formats the VCB appropriately. It also serves as a culmination function that calls functions for both FAT initialization and root directory initialization. Finally, it writes the newly formatted VCB to disk.

Who worked on it: Evan Caplinger

Name: fs_mkdir

Description: Our file systems functionality for making directories. Should function similarly to Unix mkdir. Use parsePath to extract parent path and new directory name and check name length. It then navigates to parent directory, checks if entry already exists, returning an error if so, otherwise it will allocate space for a new directory. Initializes the new directory, updating parent directory and metadata.

Who worked on it: Alexander Tamayo, Ronin Lombardino(Debugging)

Name: fs_rmdir

Description: Our file systems functionality for removing directories. Should function similarly to UNIX rmdir. Parses and validates the path, ensuring that it does not remove “.” or “..” and is not in the root directory. It will navigate to the target directory to remove and load its entries. It checks if directory is empty (should only contain “.” and “..”) and return an error otherwise. It will check if it’s the current working directory to prevent deletion. When removing the directory, it will also remove it from the parent directory, free allocated blocks and update the metadata

Who worked on it: Alex Tamayo, Ronin Lombardino(Debugging)

Name: fs_opendir

Description: This function takes in a directory path, then after validating that it is a valid directory it creates a fdDir struct and populates it with the Directory's info as well as creating a fs_diriteminfo struct and populating it with the info from the first Directory Entry of the Directory.

Who worked on it: Ronin Lombardino

Name: fs_readdir

Description: This function updates the fs_diriteminfo struct from a given fdDir struct with the info of the next Directory Entry of the Directory fdDir represents, and returns that fs_diriteminfo struct or NULL if there are no more Directory Entires in the Directory

Who worked on it: Ronin Lombardino

Name: fs_closedir

Description: This function just safely frees the fdDir struct and it's sub memory allocations preformed in opendir.

Who worked on it: Ronin Lombardino

Name: fs_getcwd

Description: So as to not work too much in mfs.c, this function was a thin wrapper to an internal function inside fsDirectory.c, getcwdInternal; see that function for further details.

Who worked on it: Evan Caplinger

Name: fs_setcwd

Description: As fs_getcwd above, this function also just wrapped setcwdInternal in fsDirectory.c.

Who worked on it: Evan Caplinger

Name: fs_isFile

Description: Calls ParsePath() to break the input path into the parent directory and the index of the target entry inside that directory. If parsePath reports the doesn't exist, fs_isFile returns false. If the target entry exists, the function checks the entry's flags field. Any required temporary memory (such as the parent directory buffer) is freed before returning

Who worked on it: Alejandro Cruz-Garcia

Name: fs_isDir

Description: Derermines whether a provided path represents a directory. Like fs_isFile, it begins by calling ParsePath to locate the parent directory and the corresponding index of the target entry. If parsing fails or no entry exists at the location, the function returns false. When the entry does exist, the function checks the directory entry's flag bits and returns true only if the item is both used and marked with the directory flag. This ensures that operations such as cd, ls, and directory creation/removal can reliably distinguish directories from regular files.

Who worked on it: Alejandro Cruz-Garcia

Name: fs_delete

Description: Removes a file or an empty directory from the file system while maintaining consistency of both directory entries and the FAT structure. It uses ParsePath to find the parent directory and the target entry, returning an error if the entry does not exist. If the entry is a directory, the function loads its contents and verifies that it is empty (excluding "." and "..") before allowing deletion. For regular files, or empty directories, any allocated FAT blocks are freed through freeBlocks, and the directory entry is marked unused. Finally, the updated directory is written back to disk, ensuring the file system remains structurally valid after deletion.

Who worked on it: Alejandro Cruz-Garcia, Ronin Lombardino(Debugging)

Name: fs_stat

Description: Retrieves metadata about a file or directory and stores it in an fs_stat structure for higher-level commands such as ls –l, cat, and other utilities that require size or timestamp information. It uses ParsePath to locate the target entry and verifies that it exists. Once the entry is found, the function extracts the file size, computes the number blocks occupied, and copies the creation, modification, and last access timestamps from the directory entry into the output structure.

Who worked on it: Ronin Lombardino, Alejandro Cruz-Garcia

Name: cmd_mv

Description:
Responsible for handling 4 main scenarios: renaming within the same directory, moving to different directory, moving into directory, and overwriting existing file. It will first parse and validate the source and destination paths, then determines which scenario applies. For directories, check for circular moves and update "..". For same directory, rename in place. For different directories, detach from source, attach to destination. For overwriting, delete old file first, then move.

Several helper functions were made for cmd_mv

extractFileName – Finds the last '/' in the path and returns everything after it, if no '/' exists, returns the entire path

getDirectorySize - Reads the size of a directory from its "." entry which contains the directory's size in bytes. This function reads just the first block to get that size.

detachEntry – This is used when moving a file/directory. It removes it from the parent but keep its blocks allocated so we can attach it somewhere else.

attachEntry – Finds an empty slot in the parent directory and adds the entry, it is used when moving a file/directory to a new location.

renameInPlace – Renames an entry within the same directory. It is used when source and destination are in the same directory.

updateDotDot – Updates the ".." entry in a moved directory. When a directory is moved, its ".." entry must be updated to point to the new parent. This is to make sure that "cd .." works correctly after move operations.

isAncestorOf – Check if one directory is an ancestor of another. It will move up the tree from target directory using ".." until a potential ancestor is found, root is reached, or it hits max iterations (in which case it will return an error)

Who worked on it: Alexander Tamayo, Alejandro Cruz-Garcia

Meetings

October 13th – December 3rd, we were all active over Discord text chat, consistently providing updates regarding our work or things we found during testing or questions we had about the assignment.

October 17th, we all met via Discord voice chat to discuss and finalize the parts of the File System Design that we had worked on in class and throughout the week.

October 20th, we all met after class to discuss what parts of the assignment we would do for milestone 1.

October 23rd, we all met on Discord voice chat to discuss progress on milestone 1, primarily a check in.

October 28th, we all met via Discord voice chat to discuss any debugging that needed to be done, and any pieces that weren't working for milestone 1, as well as discussing the writeup.

October 31st, Evan, Alejandro, and Ronin met for a quick meeting discussing some more issues that were seen in testing milestone 1.

November 1st – 3rd, we all worked on the writeup and were consistently discussing it over Discord.

November 3rd, we all met after class to discuss what we would do for milestone 2, how we would approach it, and how tasks should be divided.

November 3rd – December 3rd, we all met after class every Monday and Wednesday to check in and discuss any updates regarding our parts of the work, bugs or issues we saw, or things we needed help with.

November 11th, Evan, Alejandro, and Ronin met over discord to discuss milestone 3, and how we would divide up the remaining work that needed to be done.

November 29th & November 30th, Evan and Alejandro discussed some of the final testing and debugging they were doing over discord.

December 2nd, Evan and Ronin met over Discord voice chat to fix some final bugs that had appeared and do some last minute testing.

Analysis:

Screen shot of compilation:

```
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$ make clean
rm fsshell.o fsInit.o b_io.o fsDirectory.o fsVCB.o fsFreeSpace.o mfs.o fsshell
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o fsDirectory.o fsDirectory.c -g -I.
gcc -c -o fsVCB.o fsVCB.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -o fsshell fsshell.o fsInit.o b_io.o fsDirectory.o fsVCB.o fsFreeSpace.o mfs
.o fsLowM1.o -g -I. -lm -l readline -l pthread
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$
```

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

```
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$ make clean
rm fsshell.o fsInit.o b_io.o fsDirectory.o fsVCB.o fsFreeSpace.o mfs.o fsshell
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o fsDirectory.o fsDirectory.c -g -I.
gcc -c -o fsVCB.o fsVCB.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -o fsshell fsshell.o fsInit.o b_io.o fsDirectory.o fsVCB.o fsFreeSpace.o mfs
.o fsLowM1.o -g -I. -lm -l readline -l pthread
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
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    |
|---------------------------------|
Prompt > ls

b.txt
```

```
Prompt > ls

b.txt

Prompt > md newDir
writing to disk, location = 165, blocksNeeded = 2
Prompt > ls

newDir
b.txt

Prompt > cd newDir
Prompt > md newerDir
writing to disk, location = 171, blocksNeeded = 2
Prompt > ls

newerDir

Prompt > cd newerDir
Prompt > pwd
/newDir/newerDir/
Prompt > cd ..
Prompt > pwd
/newDir/
Prompt > cd ..
Prompt > cat b.txt
ABC
Prompt > touch a.txt
```

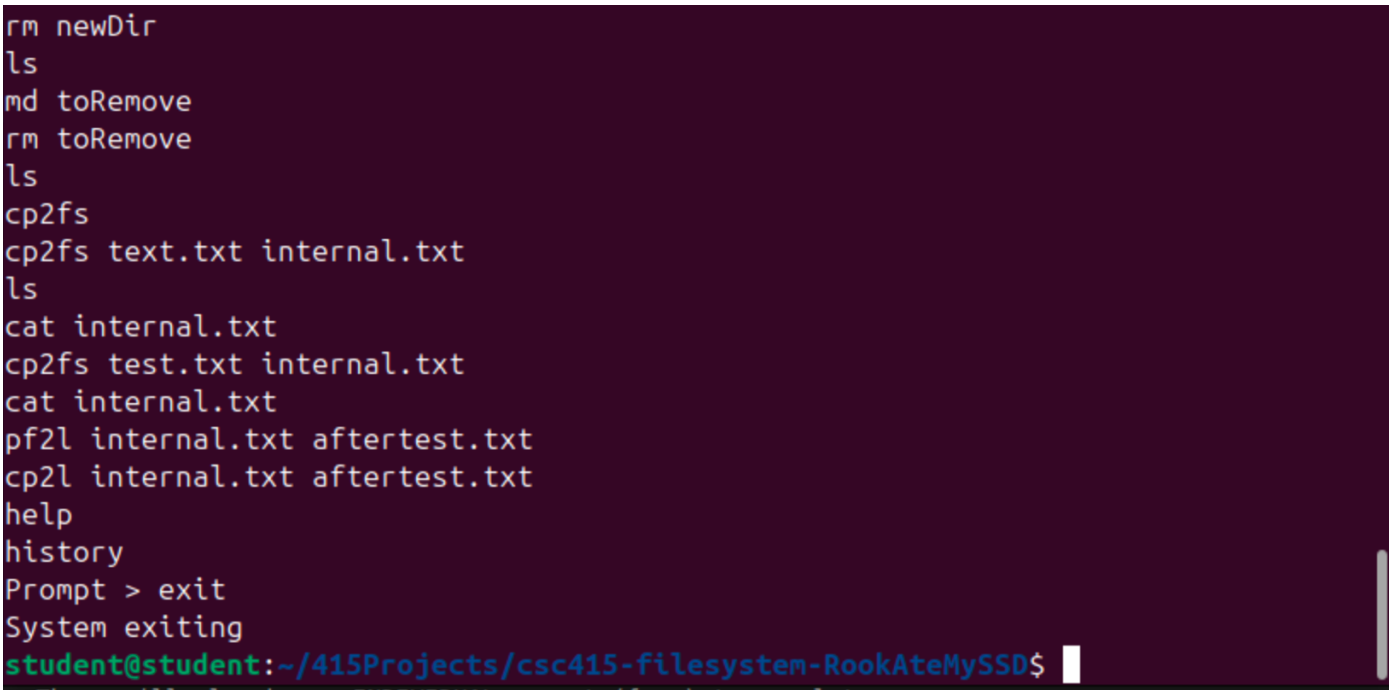
```
Prompt > touch a.txt
Prompt > ls

newDir
b.txt
a.txt

Prompt > cp b.txt newDir/newb.txt
Prompt > ls

newDir
b.txt
a.txt

Prompt > mv b.txt /newDir/newerDir/b.txt
fileEnd is 0xa9 -> next 0xfffffffe
Prompt > ls

newDir
a.txt

Prompt > cd newDir
Prompt > ls

newerDir
newb.txt

Prompt > cat newb.txt
ABC
Prompt > cd /newDir/newerDir
Prompt > ls
```

```
Prompt > ls

b.txt

Prompt > cat b.txt
ABC
Prompt > pwd
/newDir/newerDir/
Prompt > cd ../..
Prompt > pwd
/
Prompt > ls

newDir
a.txt

Prompt > rm a.txt
Prompt > ls

newDir

Prompt > rm newDir
started rmdir
wasn't empty
Prompt > ls

newDir

Prompt > md toRemove
writing to disk, location = 169, blocksNeeded = 2
Prompt > rm toRemove
```

```
Prompt > rm toRemove
started rmdir
fileEnd is 0xa9 -> next 0xaf
fileEnd is 0xaf -> next 0xfffffffe
Prompt > ls

newDir

Prompt > cp2fs
Usage: cp2fs Linuxsrcfile [destfile]
Prompt > cp2fs text.txt internal.txt
Prompt > ls

newDir
internal.txt

Prompt > cat internal.txt
Prompt > cp2fs test.txt internal.txt
Prompt > cat internal.txt
ABC
Prompt > pf2l internal.txt aftertest.txt
```

```
Prompt > pf2l internal.txt aftertest.txt
pf2l is not a regonized command.
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
Prompt > cp2l internal.txt aftertest.txt
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 >
```

```
Prompt >
Prompt >
Prompt > history
ls
md newDir
ls
cd newDir
md newerDir
ls
cd newerDir
pwd
cd ..
pwd
cd ..
cat b.txt
touch a.txt
ls
cp b.txt newDir/newb.txt
ls
mv b.txt /newDir/newerDir/b.txt
ls
cd newDir
ls
cat newb.txt
cd /newDir/newerDir
ls
cat b.txt
pwd
cd ../..
pwd
ls
rm a.txt
ls
rm newDir
```


rm newDir
ls
md toRemove
rm toRemove
ls
cp2fs
cp2fs text.txt internal.txt
ls
cat internal.txt
cp2fs test.txt internal.txt
cat internal.txt
pf2l internal.txt aftertest.txt
cp2l internal.txt aftertest.txt
help
history
Prompt > exit
System exiting
student@student:~/415Projects/csc415-filesystem-RookAteMySSD$