File System Project Writeup

Team:
Innov8

Names:
Justin Ho 922266680
Junyoung Kim 920303420
Ryan Flannery 921824732
Bilguun Bayasgalan 922564900

Central Repository:
csc415-filesystem-justinh128
https://github.com/CSC415-2024-Fall/csc415-filesystem-justinh128

The plan for each phase and changes made

This is the approach and plans we made for each phase to complete our File System project, We split the work into three main milestones. Each milestone needed good planning and understanding of our skills. Our team had four members: Bill, Justin, Jun, and Ryan. First, we met as a group to talk about our backgrounds and skill levels. This helped us make a plan where each of us could work on tasks we felt most comfortable with. In our first meeting, we checked how familiar everyone was with C programming since it was important for the project. Some of us had a deeper understanding of C than others, so we made sure that no one would feel overwhelmed by the tasks. We assigned work based on what each person was good at. Some parts of the assignment were harder than others, so we needed to balance the workload properly.

For Milestone 1, we focused on setting up the file system, which included setting up the volume, free space management, and the root directory. We split into pairs to make sure not everyone was working on the same thing, which helped us avoid merge conflicts and made sure each part got enough attention. We made two teams of two: Bill and Jun worked together on fsInit and the root directory, while Justin and Ryan teamed up to work on the free space management and the Volume Control Block (VCB). Working in pairs helped us support each other without overloading anyone. It also allowed us to focus better on our tasks and avoid multiple people working on the same files. Justin and Ryan were responsible for setting up the Volume Control Block (VCB), which involved allocating memory, reading block 0, and deciding if the volume was already formatted. Bill and Jun worked on setting up the root directory. They created the directory entries and set up the "root" ("." and "..") entries properly.

Throughout Milestone 1, we kept in close communication within our pairs and between the two teams. We had regular meetings on Monday and Wednesday at 9 pm to make sure we stayed on track with each milestone, as they were only going to get tougher. After Milestone 1, we updated it to three meetings per week to stay on top of our progress and address challenges faster. We also had regular check-ins to discuss our progress, find any problems, and help each other out. We used a shared repository and worked on separate branches to avoid merge conflicts. Once a part of the milestone was done, we reviewed each other's code before merging it into the main branch. This phase helped us get familiar with the project's basics, including setting up the file system structure and initializing key parts. By working in pairs, we made sure everyone had the support they needed while splitting the work efficiently.

For Milestone 2, we focused on integrating the fsshell into our file system. This milestone wasn't going to be turned in, but it was important for making sure our system could handle basic file operations like creating directories, listing directories, printing the current working directory, and changing directories. We knew this part would be more challenging because our other classes and midterms were picking up, which made it harder to keep making consistent progress.We split up the work for this milestone by dividing the required functions among the four of us.

We assigned the tasks based on each person's strengths to make sure everyone had a clear focus and to avoid breaking the code. Bill worked on fs_opendir and fs_readdir, Justin worked on fs_setcwd, fs_getcwd, and fs_isFile, Jun handled fs_isDir, fs_mkdir, and fs_closedir, while Ryan worked on fs_stat, fs_delete, and fs_rmdir.

This milestone took us a couple of weeks to complete, mainly because it was challenging to get all the functions to work together properly. We continued our regular meetings on Monday, Wednesday, and added a new one on Sunday nights to keep up with the workload. These extra meetings helped us stay organized and allowed us to solve integration issues quickly. By the end of Milestone 2, we got all the required functions working, which was a big achievement considering how busy our schedules were. For the final stretch of the project we focused on filling in the last functions in the b_io.c file we all pitched in contributing to at least one of the functionsThis phase was tough because the biggest challenge was getting all of our code to work together smoothly. Understanding how each function worked and tying them all together was difficult. We stuck to our meeting schedule on Monday, Wednesday, and Sunday nights, which helped us stay on track and communicate well. These meetings were crucial for solving issues with our functions and making sure our changes didn't break the rest of the code.

Besides finishing the code, we also had to focus on the final write-up, which was a big part of our submission. We explained our design choices, the challenges we faced, and how we solved them. The write-up was important to show our understanding of the project and the effort we put in.

A description of your file system

Our file system relies on a set of interconnected components to organize and track data efficiently. These components include mechanisms for managing storage, maintaining metadata, and navigating directories. Each aspect of the system works together to provide a reliable framework for handling data. At its foundation, the system organizes and tracks information using structures and processes that ensure data is stored, retrieved, and maintained effectively. One of the central components of the file system is its ability to track metadata. This includes information about the storage volume, such as its size, the available space, and where important elements like directories and free space tracking are located. This metadata is stored and updated to ensure that the system has a clear understanding of how the storage is being used. The file system uses a structure to monitor which parts of the storage are in use and which are free. This structure allows the system to efficiently allocate and free storage as needed, ensuring that space is used effectively and no resources are wasted. Directories in the system are organized in a hierarchy, which makes it easier to store and find files. This hierarchical structure resembles a tree, starting with a root directory and branching out into subdirectories and files. Each directory and file has associated information, such as its name, size, and when it was last modified. This organization helps maintain a logical structure for data, enabling users to navigate and interact with it easily. Paths are used to locate files and directories within this hierarchy. The system processes these paths by breaking them into parts and following the structure step by step to reach the target. This ensures that users can work with both absolute paths, starting from the root directory, and relative paths, which depend on the current location in the hierarchy. The file system provides functionality for creating and managing files and directories. This includes creating new directories, storing data, and managing how files grow or shrink as data is added or removed. The ability to navigate the file system is another important aspect. The system allows users to move through directories, list their contents, and determine their current location within the structure. These navigation tools make it easy to explore and manage the data stored in the system, mimicking the behavior of widely used file systems. Error handling is an important feature of the file system as well. It can handle unexpected situations by exiting gracefully. The system checks for invalid inputs, such as incorrect paths or nonexistent files, and provides feedback when errors occur. This prevents issues from escalating and helps maintain the stability of the system.

Issues you had

An issue we faced was with root.c and free_space.c including the structs within the same .c file. We knew that we needed to be able to call these functions across files and pass information. So, what we did was separate the structs into separate header files. This included vcb.h, root.h, and free_space.h. Within these files we had the designated structs as well as function prototypes to ensure there were no linkage errors.

An issue we faced was initializing the free_space, the VCB, and the root directory within their own .c files. We saw that there could be issues with this, so we looked through the files and decided to initialize everything in fsInit.c. Doing it this way would make sure that every aspect initialized when the file system initialized.

While initializing the free space bitmap in free_space.c, we faced a recurring issue where the memory allocation sometimes went out of bounds due to improper allocation sizes and handling. The bitmap initially required 2442 bytes to manage all blocks, but the additional blocks for metadata were causing the total allocation to round up, occasionally leading to insufficient memory. We resolved this by carefully calculating the total memory needed, including extra blocks for metadata, and allocating memory accordingly. We implemented boundary checks to ensure no allocation exceeded defined limits, such as the set block size, and used memset() to zero out the allocated memory. Adding debug statements also allowed us to track memory locations and identify areas where overflow could occur, which we corrected by refining the allocation code and using constants for clarity.

Another issue we encountered was correctly interpreting the hexdump for different structures, such as the VCB. Given that each block contains data in hexadecimal format, understanding the structure and pinpointing each component within the dump was challenging. Also, we had to make sure each field in the VCB aligned with its correct address offset. We attempted to resolve this issue by systematically reviewing the structures and confirming each field's size and offset within the block. We also looked back on Assignment 2 for help deciphering the hexdump.

Memory management was a recurring issue, particularly with dynamic allocations and freeing of memory for directories and file data. Errors like double freeing memory or accessing null pointers were encountered. We addressed these by standardizing the use of helper functions, such as FreeDir, to manage memory consistently and by carefully tracking ownership of allocated memory throughout the code.

An issue we had for a while was getting our file system to compile. With all the different files, it was difficult to keep track of including each header that was needed across files. We had to be extremely careful and make sure that if a function was being called from a different file to include it as well as place the function prototypes in the header files.

At first, we attempted to parse the path each time needed in specific functions. However, this was a lot of extra work, so instead, we made a separate function for it. This allowed us to call it whenever needed. It also helped a lot to have gone over this function in class as we already had the general idea of it within our class notes.

ParsePath was also causing errors because the function initially operated on the original path string directly, which led to errors when the path string was modified during tokenization. Since strtok_r alters the input string, it causes subsequent operations that relied on the original path to fail. We fixed this by introducing a pathCopy variable, duplicating the input string to preserve the original path while allowing safe modification during parsing.

There were various challenges with functions that were initially designed as void but needed to provide meaningful feedback or error codes to their callers. For example, writeRootToVol was originally implemented as a void function, but this made it difficult to determine whether the operation succeeded or failed. By changing its return type to an integer and implementing error-checking logic, we were able to ensure that other functions could respond appropriately to issues like disk write failures.

There were a lot of errors when it came to handling edge cases. We overlooked a lot of cases like paths with unusual characters, directories with no entries, or attempts to access files that did not exist. We had to be extremely careful when writing and debugging these functions to account for these cases.

By far, the most troubling issue was the commands not giving us our desired outputs. We revised our logic for the corresponding functions called by these commands many times. However, we are still unable to figure out exactly where we are going wrong. Something that might have helped was being more organized with our project, especially since there are a lot of files. Separating functions across files might have stopped us from getting confused so often. We also should have asked for help sooner. None of the commands work the way they are supposed to.

Details of how each of your functions work

a. b_io.c

i. void b_init()
   1. Purpose
      a. Initialize file system(file control block array).
   2. How it works
      a. FCBs are “placeholders” to track open files.
      b. The function loops through fcbArray and set entries as NULL to make sure everything is free in fcbArray.

ii. b_io_fd b_getFCB()
   1. Purpose
      a. Finds and returns free FCB array element.
   2. How it works
      a. Scan fcbArray and check if buf is ULL or not. If it is, return i, but if not, return -1.
   3. Usage
      a. Call it to open file for allocation of new FD.

iii. fileInfo* GetFileInfo(char *fileName)
   1. Purpose
      a. Get the file information of given filename.
   2. How it Works
      a. Break down the filename to locate it in the file system.

iv. int createFile(char *filename)
   1. Purpose
      a. Create new file in the file system
   2. How it works
      a. It finds the parent directory by parsing the file path.
      b. Make sure there’s no file that has the same name.
      c. Find empty directory entry in parent directory and initialize with details.
      d. Returns -1 for invalid path or if the name already exist.

v. int truncateFile(fileInfo *fi)
   1. Purpose
      a. Reset the file’s size to 0.
   2. How it works
      a. Free all blocks associated with the file.
      b. Set the file size to 0.

vi. b_io_fd b_open(char *filename, int flags)
    1. Purpose
        a. Open the file and return file descriptor
    2. How it works
        a. Initialize the system.
        b. ‘GetFileInfo’ fetches file details
        c. If the file doesn’t exist, call ‘createFile’ to create.
        d. Handle modes like O_RDONLY, O_WRONLY, O_RDWR, etc.
        e. Return -1 if something fails.

vii. int b_seek(b_io_fd fd, off_t **offset, int w**hence)
    1. Purpose
        a. Adjust current position in file for reading or writing. It helps you to move “cursor” to a specific location in the file.
    2. How it works
        a. It finds the new position in the file based on the ‘whence’ and provides ‘offset’. It helps file descriptors to work, and make sure the new position is not going outside of the range, and update the current position of the file.

viii. int b_write(b_io_fd fd, char * buffer, int count)
    1. Purpose
        a. Writes data to a file
    2. How it works
        a. The function is a placeholder. It returns 0. It should handle writing and managing disk blocks.

ix. int b_read(b_io_fd fd, char *buffer, int cnt)
    1. Purpose
        a. Reads data from a file into a buffer
    2. How it works
        a. It validates the file descriptor and opens the file.
        b. It buffers from the current in-memory buffer by reading full blocks from the disk, and sometimes fills the buffer from a refilled buffer if necessary.
        c. Returns -1 for invalid FDs or read errors.

x. int b_close(b_io_fd fd)
    1. Purpose
        a. Close file and release resources
    2. How it works
        a. Flush any pending writes. Free buffer and FCB entry. Return -1 for invalid FDs or when the file is not opened.

b. freespace.c

i. uint32_t initializeFreeSpace(uint32_t bitmap_blocks, uint32_t block_size)
   1. Purpose
      a. Initialize the free space bitmap to check how disk blocks are used, and mark if the blocks are free or reserved.
   2. How it works
      a. It allocates memory to free space bitmap, and it costs one bit per each block to show if the block is free or occupied.
      b. Initialize all bits to 0 with ‘memset’
      c. Check the first 6 blocks as ‘occupied’ for system use.
      d. Writes the bitmap to disk for LBAwrite.
      e. Return 1 for pass, 0 for failure

ii. uint32_t allocateBlock()
   1. Purpose
      a. Allocate the first available free block by updating the bitmap.
   2. How it works
      a. It starts with iterating through the total number of blocks. It checks every bit in the bitmap to find the first free block.
      b. Then, it returns the block number or 0 if there are no free blocks.

iii. void freeBlock(uint32_t blockNumber)
   1. Purpose
      a. Frees a block that is already allocated. It marks the block back as ‘available’(0) in the bitmap.
   2. How it works
      a. It checks if the ‘blocknumber’ is smaller than ‘totalBlocks’ to stay inside the proper range. It identifies the byte and bit index that matches with the block. Then, it clears the bit in the bitmap using a bitwise AND with negation of the target bit mask. It doesn’t return any value but updates the bitmap in memory.

c. fsInit.c

i. int initFileSystem(uint64_t volume_size, uint64_t block_size)
   1. Purpose
      a. Initializes the file system using important factors such as volume control block, free space bitmap, and root directory.
   2. How it works
      a. Start with checking a few cases that will cause errors. Use ‘startPartitionSystem’ to verify the partition system using a predefined volume name. It returns an error if it fails to start. Allocate memory for VCB and reads the first block to check if the volume is already formatted. If necessary, initialize a new file system and format the volume. It clears the VCB and populates it. It initializes the free space bitmap with ‘initializeFreeSpace’. After that, use ‘RootDirecInit’ to set up the root directory, then use ‘writeRootToVol’ to write about that on the disk. If the VCB signature is valid, it loads an existing file system and prints it out.

ii. void exitFileSystem()
   1. Purpose
      a. Shut down the file system and clean up the resources.
   2. How it works
      a. Print out that the system is exiting.
      b. Call ‘closePartitionSystem’ to properly close the partition and release any associated resources.

d. root.c

i. void rootDirecInit(struct RootDirectory *rootDirStruct, uint32_t rootBlockNum)
   1. Purpose
      a. Initializes the root directory structure, including special entries and default values for all directory entries.
   2. How it works
      a. Allocate memory for the global variable, ‘rootDir’. Set the root isDir, location, size, etc. Initialize . and .. entry. Set all remaining entries to default values(empty and unused). After that, copy the initialized structure into the global ‘rootDir’ pointer and set the global variable, ‘curDir’ to the root directory. Eventually, print out the success message upon completion.

ii. void writeRootToVol(const char *volumeFile, struct RootDirectory *rootDirStruct)
   1. Purpose
      a. It writes the root directory structure to the volume file on the disk
   2. How it works
      a. It opens the volume file in read/write binary mode (rb+). Then, set the pointer to the position after the volume control block with ‘fseek’ function. Write the root directory structure with ‘fwrite’. Close the file to ensure all the changes are saved. If any of the file cannot be open, print an error message.

e. Mfs.c

i. bool isUsed(DE *entry)
   1. Purpose
      a. Check if a directory entry(DE) is in use.
   2. How it works
      a. Verifies the ‘entry’ is not ‘NULL’. Checks if the ‘name’ is empty or not. Then, return true if both conditions are met, otherwise ‘false’.

ii. int FindInDir(DE *parent, char *name)
   1. Purpose
      a. Searches file or directory by name
   2. How it works
      a. Check that ‘parent’ and ‘name’ are not ‘NULL’.Go through the entries in the parent directory. For each entry, check if it is used and matches the given name from ‘strcmp’. Return the matching entry or -1.

iii. DE *LoadDir(DE *dirEntry)
    1. Purpose
        a. It loads a directory's entries from disk into memory
    2. How it works
        a. It validates that 'dirEntry' is a directory, and allocates memory for the entries. Then, read the directory's data using 'LBAread' from its disk location. Then, return the loaded directory or NULL if there's an error.

iv. void FreeDir(DE *dirEntry)
    1. Purpose
        a. Frees memory that are already allocated for the entries
    2. How it works
        a. Check if the given pointer is NULL to avoid errors. Then, it frees the allocated memory and prints a success message.

v. int ParsePath(const char *path, DE **retParent, int *index, char **lastElementName)
    1. Purpose
        a. Parses a file path to identify its parent directory, the target index, and the last element name
    2. How it works
        a. It checks the path and determines whether it's absolute or relative. Tokenize the path into components using '/' as a delimiter. Check every directory and load each level with 'LoadDir'. Eventually, return the parent directory, target index, and last element name or an error if path is invalid.

vi. char * fs_getcwd(char *pathname, size_t size)
    1. Purpose
        a. It retrieves the current working directory path.
    2. How it works
        a. It traverses from the current directory to the root. It makes the path in reverse order. Finally, it copies the result into the given 'pathname' buffer and makes sure it fits within the specified size.

vii. int fs_setcwd(char *pathname)
    1. Purpose
        a. Change the current working directory
    2. How it works
        a. Parses the provided path with 'ParsePath' function. Then, verify the resolved target is a directory. Then, update the 'curDir' to point to the new directory.

viii. int fs_isFile(char *filename)
    1. Purpose
        a. Check if the provided path matches to a file
    2. How it works
        a. It parses the path using ‘ParsePath’. Check if the target entry exists and is not a directory. Then, returns 1 if it is a file and 0 if it is a directory, and -2 if the path is invalid.

ix. fdDir * fs_opendir(const char *pathname)
    1. Purpose
        a. Opens a directory for reading
    2. How it works
        a. It parses the path to locate the directory, then verifies if the target is a directory and loads its entries. After that, allocate and initialize a fdDir structure to manage directory streams.

x. struct fs_diriteminfo *fs_readdir(fdDir *dirp)
    1. Purpose
        a. It is used for retrieving the next valid directory entry from a ‘fdDir’. It is part of directory navigation.
    2. How it works
        a. Check if ‘fdDir’ pointer is NULL. If so, return null and signal it’s an invalid input.
        b. If it’s not NULL, compare ‘dirp->dirEntryPosition’ with ‘dirp->numEntries’ to determine if the directory has reached its end. If so, return ‘NULL’, then indicate no more entries. After that, loop through the directory entries in ‘dirp->directory’. For each entry, check if the entry is valid , then populate the ‘fs_diriteminfo’ structure with details, then return the populated structure.
        c. If all entries have been checked and no valid entries remain, return NULL.

xi. int fs_isDir(const char *path)
    1. Purpose
        a. Check if the provided path matches to a directory
    2. How it works
        a. Uses ‘ParsePath’ to resolve the path and confirm if the target is a directory.

xii. int fs_mkdir(const char *path)
    1. Purpose
        a. Create a new directory
    2. How it works
        a. Check the path doesn't exist, parent directory exist, then allocate a new directory inode and add it to the parent directory

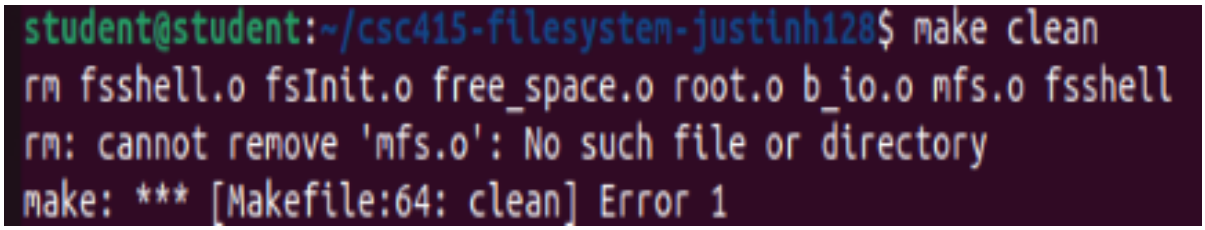
xiii. int fs_closedir(Fs_dir_t *dr)
    1. Purpose
        a. Closes an open directory stream
    2. How it works
        a. Free the memory with 'fdDir' structure

xiv. int fs_stat(const char *path, struct fs_stat *buf)
    1. Purpose
        a. Retrieves metadata for a file or directory
    2. How it works
        a. Parse the path to find the target, then populate 'fs_stat- with details.

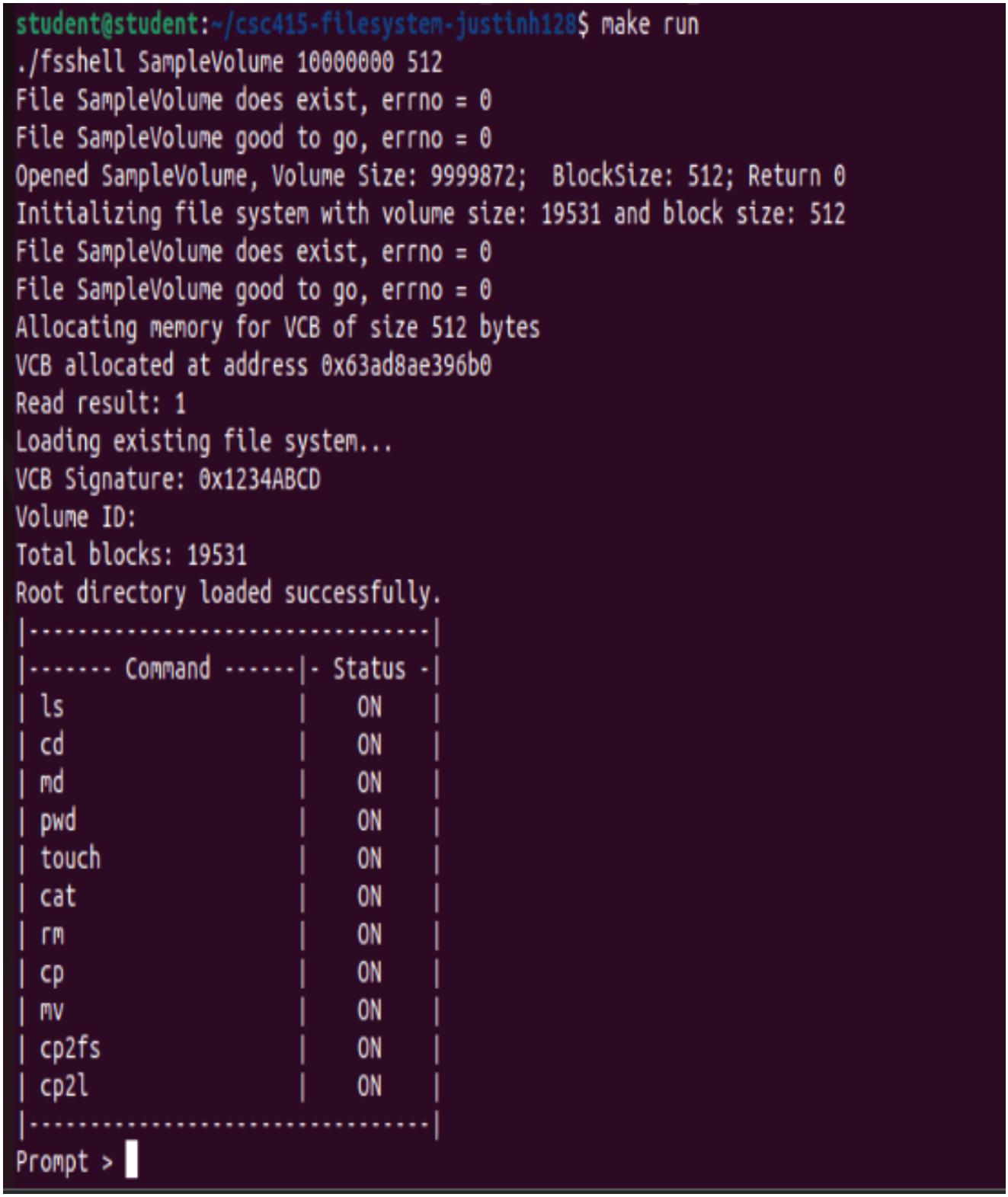
xv. int fs_delete(char *filename)
    1. Purpose
        a. Deletes a file from the file system
    2. How it works
        a. Check if the file exists with 'fs_stat'. If not, print 'file not found'. If it exists, clear its entry in the parent directory by overwriting zero.

xvi. int fs_rmdir(const char *pathname)
    1. Purpose
        a. Removes a directory from the file system.
    2. How it works
        a. It verifies if the directory exists, then removes its entry and free the resources.

Screenshots showing each of the commands listed in the readme



```
student@student:~/csc415-filesystem-justinh128$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o free_space.o free_space.c -g -I.
gcc -c -o root.o root.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -o fsshell fsshell.o fsInit.o free_space.o root.o b_io.o mfs.o fsLow.o -g -I. -lm -l readline -l pthread
```


student@student:~/csc415-filesystem-justinh128$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing file system with volume size: 19531 and block size: 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Allocating memory for VCB of size 512 bytes
VCB allocated at address 0x63ad8ae396b0
Read result: 1
Loading existing file system...
VCB Signature: 0x1234ABCD
Volume ID:
Total blocks: 19531
Root directory loaded successfully.
|--------------------------------|
------- 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 >

```
student@student:~/csc415-filesystem-justinh128$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing file system with volume size: 19531 and block size: 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Allocating memory for VCB of size 512 bytes
VCB allocated at address 0x56ad80e396b0
Read result: 1
Loading existing file system...
VCB Signature: 0x1234ABCD
Volume ID:
Total blocks: 19531
Root directory loaded successfully.
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md /test
Directory memory freed successfully.
Prompt > ls
Directory memory freed successfully.
free(): double free detected in tcache 2
make: *** [Makefile:67: run] Aborted (core dumped)
```

```
Loading existing file system...
VCB Signature: 0x1234ABCD
Volume ID:
Total blocks: 19531
Root directory loaded successfully.
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > help
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
seek    Adjust file position: <file> <offset> <whence>
Prompt > pwd
/
Prompt > touch
Usage: touch srcfile
Prompt > touch hello
Prompt >
```