File System Project

CSC 415 - Operating System Principles

Team Spork

Robert Bierman

August 1, 2024

Group Github repository:

https://github.com/CSC415-2024-Summer/csc415-filesystem-shezgo

Plan phase:

In reality, planning started from the beginning of the semester as we learned different components of the file system individually through each class assignment. As we learned skills and built the bases for our knowledge, the planning phase began with trying to identify what exactly we'll need to do for the base of the file system - this lead to Milestone 1: formatting the volume.

In Milestone 1 we did our best to connect the information we had learned throughout the class regarding definitions and relationships between entities - namely, the volume control block, directories, and directory entries. As a lot of time was spent mulling over the definitions and reviewing past lectures, we followed the Steps for Milestone 1 document and had faith in implementing that list to the best of our ability. After doing my best on formatting the Volume Control Block and the free space bitmap, initializing the root directory was delegated to the rest of the group. Unfortunately, the group was lost, so I had reviewed past lectures and class materials to implement to the best of my ability. While at first I felt fairly confident but knew we'd probably get dinged on a couple things, I did not expect to get a grade of 3.5/30 on the first Milestone.

Next is the plan resulting from getting our feedback on M1. Immediately upon receipt, we knew that we needed to become much more comfortable with the basics than we were, and communication was had about everyone taking their time to watch all lectures and attend office hours to make sure we had the fundamentals, in order to not waste time and implement efficiently. Unfortunately, the follow through was minimal for most of the group.

This is where planning started to become less group oriented as not everyone was catching up or keeping up with the material. At some point I had realized that I would be the only person implementing, and from this point planning had changed to realizing I could only focus my own efforts on a solid foundation to build such a large project by trying to perfect milestone 1. I copiously viewed all prior lecture material and

took a large volume of notes for implementation, paying particular attention to key concepts and walkthroughs of main helper functions and directory functions. After careful consideration of the alternatives to a freespace bitmap, including linked lists and extents, I had settled that to maximize my time, a freespace bitmap would be most efficient and to reduce the scope of the project. I would have loved to implement extents but priorities called. Having a second draft of M1 took about a week - I had finished it around the time M2 was due.

I had built a list of main helper functions through watching the class material in order to support all of the functions included in the provided mfs.h file. The list of helper functions included: parsePath, findNameInDir, entryIsDir, freeIfNotNeedDir, findUnusedDE, saveDir, isNullTerminated, loadDir, a number of helper functions to navigate freespace, loading and writing global data that I would want on memory for easy access, and more.

Once I felt confident that I had main data items and functions, I started with trying to use my planning to implement fs_opendir, fs_mkdir, fs_readdir, fs_setcwd, fs_getcwd, and the other functions after. I figured as I step through the pseudocode for each, the helper function needed would reveal itself and I could implement them along the way. This is exactly what I did.

After implementing these functions to the best of my ability, the plan was to iteratively test each command starting from ls, then md, back to ls, pwd, cd, touch, and rm. Once all of these commands tested positively, then I would move onto the functions requiring use of b_io: cat, cp, cp2L, and cp2FS. . When focusing on b_io, I'd first focus on designing the functions such that they are supported by b_fcb and identify any helper functions as needed. I'd then start with b_open, b_read, b_close, b_seek, and b_write in that order.

Unfortunately, the plan needed to pivot because after completing all of the functions within mfs.h for the first pass of M2, I had debugged to the best of my ability and then ran into segmentation faults that I could not resolve by tracing code because it was happening in the middle of a return statement. This informed me that throughout my program, I wasn't allocating and navigating memory correctly, despite thinking I had planned carefully. After spending a long time trying to figure this out, I had set out to essentially redo my entire M1 foundation for a third time - rather than executing commands and trying to follow output step by step, at this point it had become more efficient to start from the initFileSystem function and follow the logic of my code to see what I was doing wrong.

The result was a complete refactoring of all data, sending walls of text to Professor for help, and an average of 3-5 hours of sleep per day (more 3 than 5). The

plan at this point had been to just push and get the infrastructure for the project correct, and make sure that persistent memory was functioning as planned.

At this time, this is as far as I was able to take the project in implementation by myself. The plan from here is to resolve what is happening that is incorrect in my parsePath function, which I imagine could cascade in all of the directory functions working correctly if I just get this one function to do its job now.

Description of file system:

The file system starts with fsShell.c as the main driver for the entire program. Inside of main here, the Professor Bierman's partition system is initialized followed by my initFileSystem function. This function divides into two cases in order to ensure that the data is persistent, meaning if you were to close the program and run it again, you still have your data on disk.

Case 1 is that the file system has not been mounted yet. This is checked by pulling the volume control block from the disk at its expected location of block 0, checking the signature for a hardcoded value, and if it doesn't match up, the function proceeds by creating and initializing all the data for the file system and writing the necessary items to disk.

The program takes care to initialize all allocated memory for variables to known values first to reduce unexpected behavior, and then initializes data among the allocated memory to the desired values. Global data includes a pointer to a volume control block instance, a pointer to the directory entry for root directory, a pointer to the metadata for current working directory, and a pointer to a bitmap structure which contains a pointer to the freespace bitmap to be brought to memory whenever the file is loaded for easy manipulation and reading.

In Case 2, the file system has already been mounted. This means rather than reinitializing the freespace bitmap, root directory, volume control block, or changing any data inside of the disk, the memory is left as it is and the program reads data from the disk into local memory for structures needed by the rest of the file system to execute its functions.

The main entities that are initialized for primary functions of the file system are: the volume control block which contains information about the entire volume and freespace bitmap, the freespace bitmap, and directory entries which contain metadata about directories and files.

Once the system is initialized, the user is able to navigate by using commands, each of which executes a series of directory iteration functions, which are assisted by helper functions. Upon completion of the file system, the user is able to navigate between directories, open files, write to them, overwrite them, read from them, and move both files and directories around.

Issues:

The number one issue here was being the only person to commit code on my team, and not being able to rely on my group members to correct me or assist with the basics to make it more efficient. To be on such a large scale project by myself over an accelerated semester while also being one of the leads for CSC648 has tested me physically, mentally, and spiritually in a way I've never experienced before - I've identified this as the max of my threshold.

Second - although I had felt somewhat confident about operating system concepts and principles, when it came time to implementation, I was confronted with the gaps between concepts I had learned and read about, versus implementing them and matching ideas to syntax and code. Oftentimes, I'd be stuck for hours trying to step through my bugs, only to learn that things I had taken for fact in my existing code were not fact at all.

Third - needing to keep revisiting Milestone 1 no less than 3 times before I correctly understood how core concepts tie together to form the foundation of the entire file system. I needed to take probably over 100 pages of notes to understand what's going on in my code and brain, to be able to modify as needed.

Fourth - this is such a large volume of code and I've never taken on something of this size before. Planning is crucial, but learning exactly what to consider when planning is difficult unless you have experience. This experience was a major gift that this project gave me - despite not completing, I can truthfully say I was on track, and about to reach a fully functional file system abiding by the principles I learned in this class.

Details of how each function works:

1. ***VolumeControlBlock *loadVCBtoMem(uint64_t blockSize)***

- Returns a block sized buffer with the volume control block to the caller.

2. ***int writeVCBtoDisk(VolumeControlBlock *vcb)***

- Writes the volume control block to its designated block on disk.
- Returns number of blocks successfully written.

3. ***int mapToDisk(Bitmap *bm)***
    - Writes the bitmap to disk at its designated locations. Returns number of blocks successfully written.

4. ***Int setBit(Bitmap *bm, int blockNumber)***

    -Marks a block as occupied in the freespace bitmap. Returns 1 if true, -1 if invalid.

5. ***Int clearBit(Bitmap *bm, int blockNumber)***
    - Marks a block as unused in the freespace bitmap. Returns 1 if success, -1 if failure

6. ***Int isBitUsed(Bitmap *bm, int blockNumber)***
    - Returns 1 if the block is used, 0 if unused, or -1 if invalid blockNumber.

7. ***Int fsAlloc(Bitmap *bm, int req)***

    -Finds free contiguous blocks of memory and allocates them for the caller by setting them to be occupied for use in the bitmap. Return start block on success, -1 on failure

8. ***Int fsRelease(Bitmap *bm, int startBlock, int count)***
    - Marks designated contiguous blocks as unused. No need for deletion of data because memory can just be overwritten when needed. Returns 1 if successful, -1 if failure.

9. ***Bitmap *initBitmap(int fsNumBlocks, int blockSize, uint8_t *bm_bitmap)***

   -Initializes the bitmap or assigns a loaded bitmap if file system is already mounted. Returns the completed Bitmap structure which has the bitmap and other fields. Bitmap structure is never written to disk, just for RAM.

10. ***Uint8_t *loadBMtoMem(int blockSize)***
    - This loads the actual bitmap buffer into memory

11. ***DE *initDir(int minEntries, DE *parent, Bitmap *bm)***
    - This function initializes a directory and accounts for the special case of initializing root upon file system mounting. Return NULL if failure

12. ***DE *loadDirDE(DE *dir)***
    - This function uses a DE struct to load a directory into memory for manipulation. Return the DE* buffer or NULL if failure

13. ***DE *loadDirLBA(int numBlocks, int startBlock)***
    - This function uses an LBA address and number of blocks to load a directory into memory for manipulation. Return the DE *buffer or NULL if failure.

14. ***Int initFileSystem(uint64_t numberOfBlocks, uint64_t blockSize)***
    - This function is called to initialize the file system. It creates the volume control block, root directory, and freespace bitmap if the file system has not been mounted, but if it has already been mounted, it simply loads global data into memory for the rest of the program to access and manipulate to maintain persistent memory.

15. ***void exitFileSystem()***
    - This is used to close the file system and free any data loaded into memory.

16. int findNameInDir(DE *parent, char *name)

-Helper function that returns index of the DE with the name parameter in the parent that is passed in. Returns -1 if failed.

17. int entryIsDir(DE *parent, int deIndex)

-Checks if the directory entry is in the directory; 1 if true, 0 if false.

18. int freeIfNotNeedDir(DE *dir)

-Frees a directory only if it is not the current working directory, root, or null. Returning 1 indicates freeing the directory, 0 indicates no memory freed.

19. int findUnusedDE(DE *parent)

- Find the first unused directory entry in a parent directory entry. Return -1 if failed, return the index of the DE if success.

20. int saveDir(DE *directory)

- Write an existing directory to disk; return the number of blocks written.

21. int parsePath(char *path, ppinfo *ppi)

-Loads the parent in a path and finds if the file (the last element in the path) exists or not. Returns 0 if successful and -1 if not successful. It also populates the fields of the ppinfo struct that is passed in - the values of this struct which serve as additional return values include:

- DE * parent is a pointer to parent loaded in memory

- char * lastElement is the name of the last element in the path

- int lastElementIndex - the index of a directory entry inside of its parent; -1 if it doesn't exist

22. int fs_mkdir(const char *path, mode_t mode)

- Creates a directory in the designated path. Returns -1 if fails, 2 if it already exists, and 0 if success.

23. fdDir *fs_opendir(const char *pathname)

- Opens a directory - checks if the path is a valid directory and loads it into memory if so. Returns a fdDir struct which serves as a more detailed file descriptor, returns NULL if it fails.

24. struct fs_diriteminfo *fs_readdir(fdDir *dirp)

- Every time fs_readdir is called, it returns info about the next valid directory entry in a directory. If it has iterated through all of the directory entries in a directory, it will keep returning null for that directory.

25. int fs_closedir(fdDir *dirp)

- Free the resources created from fs_opendir. Return 1 if success, 0 if the directory doesn't exist.

26. int fs_setcwd(char *pathname)

- Sets the current working directory to the path passed in with pathname by assigning the value of the path directory to the global current working directory DE object. If the pathname is not a valid directory, it returns -1 for failure. Returns 0 on success.

27. int isNullTerminated(char *str, size_t len)

- Checks if a string is null terminated. Returns 1 if success, 0 if failure.

28. char *fs_getcwd(char *pathname, size_t size)

- Copies the current working directory path into the user's buffer passed in as a parameter, pathname. Return the pathname if success, NULL if error.

29. int fs_isFile(char *filename)

- Return 1 if the path passed in by caller as filename is a file, 0 if not a file.

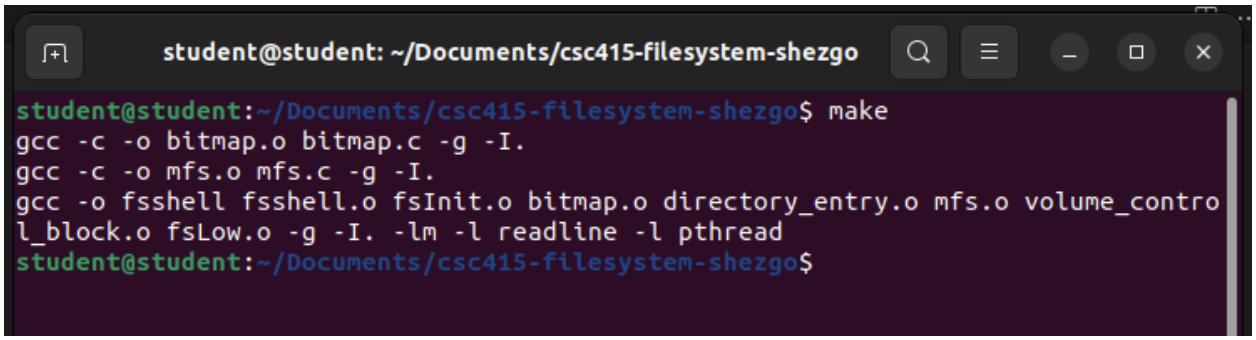
30. int fs_isDir(char *pathname)

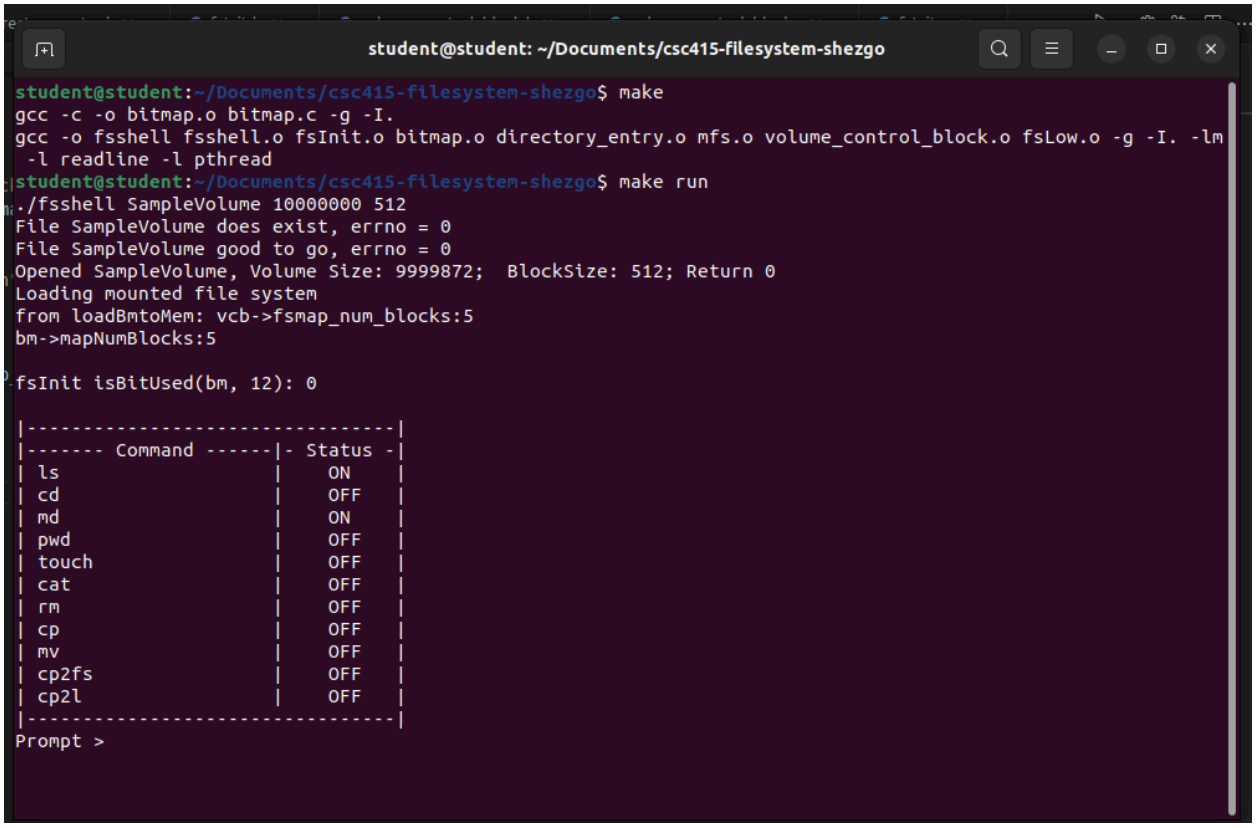
- Return 1 if the path passed in by caller as pathname is a directory, 0 if failure.

31. Int fs_stat(const char *path, struct fs_stat *buf)

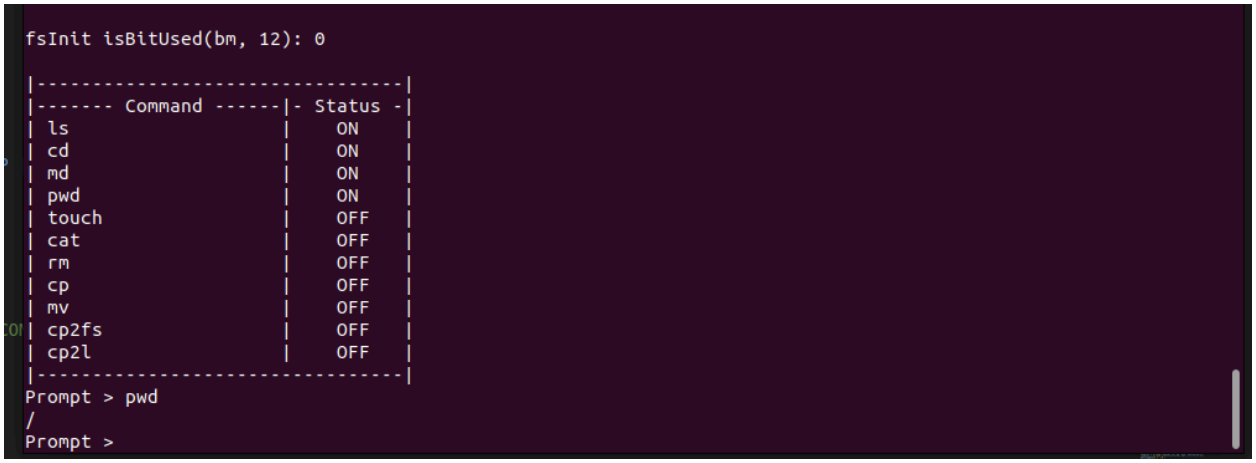
- Copies information about a directory entry and the file system to an fs_stat struct.

Screenshot of compilation:



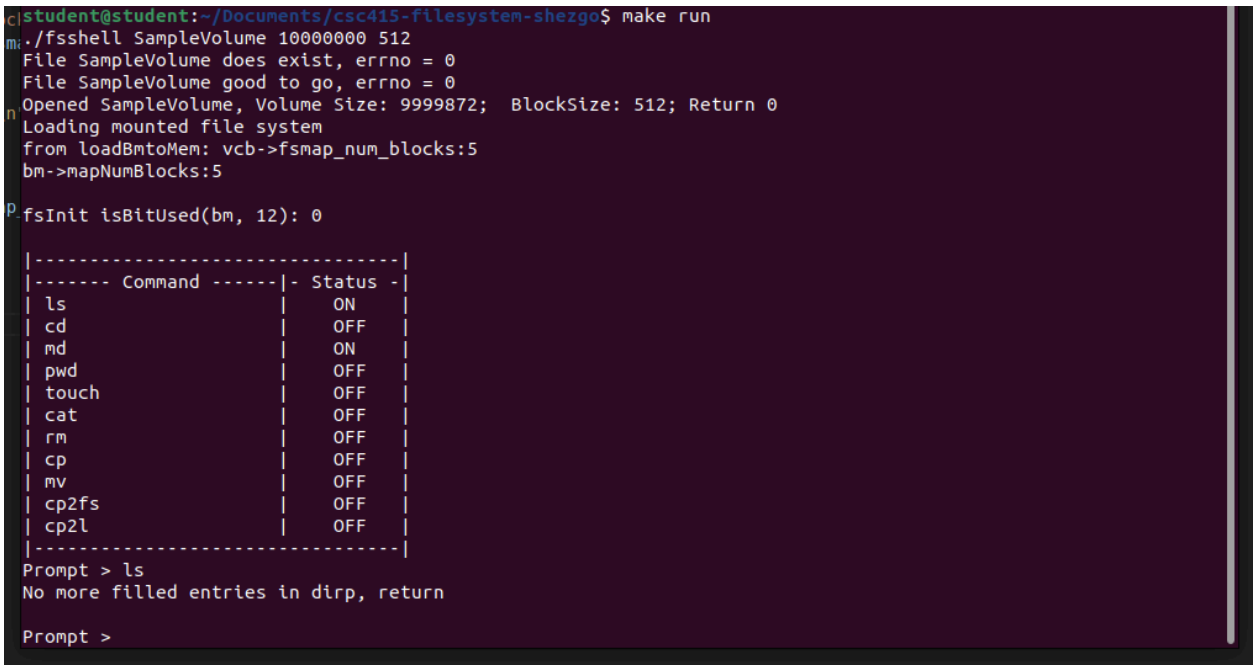


Pwd:



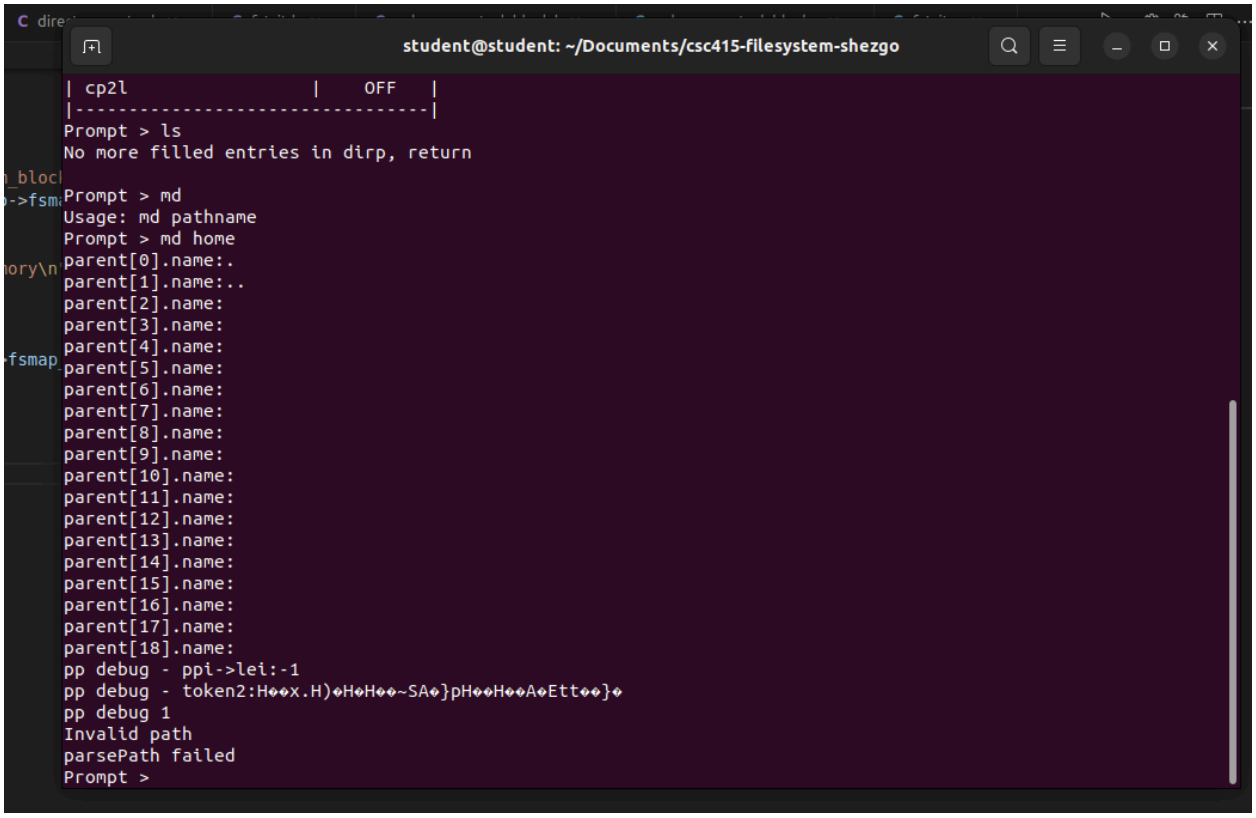
```
fsInit isBitUsed(bm, 12): 0

|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    OFF   |
| cat                  |    OFF   |
| rm                   |    OFF   |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|--------------------------------|
Prompt > pwd
/
Prompt >
```

Ls:

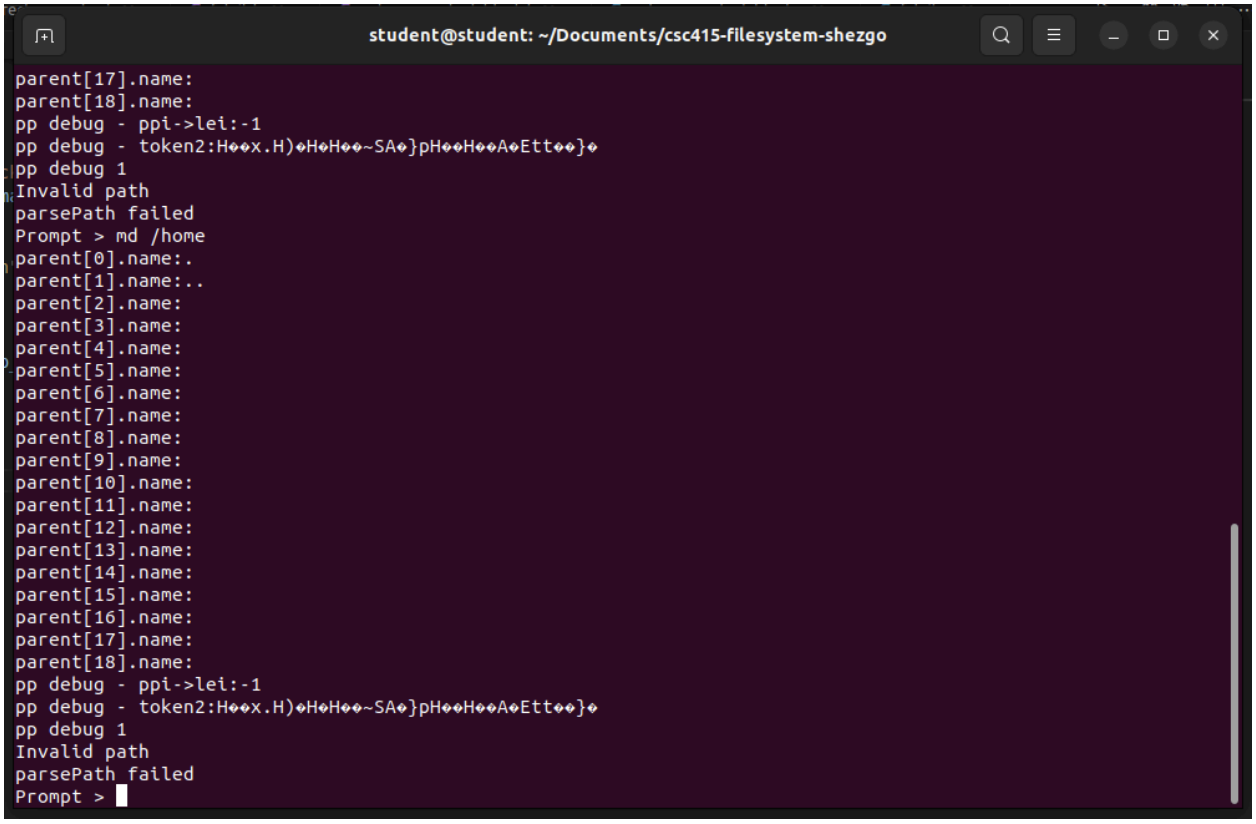


```
student@student:~/Documents/csc415-filesystem-shezgo$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Loading mounted file system
from loadBmtoMem: vcb->fsmap_num_blocks:5
bm->mapNumBlocks:5

fsInit isBitUsed(bm, 12): 0

|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    OFF   |
| md                   |    ON    |
| pwd                  |    OFF   |
| touch                |    OFF   |
| cat                  |    OFF   |
| rm                   |    OFF   |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|--------------------------------|
Prompt > ls
No more filled entries in dirp, return

Prompt >
```

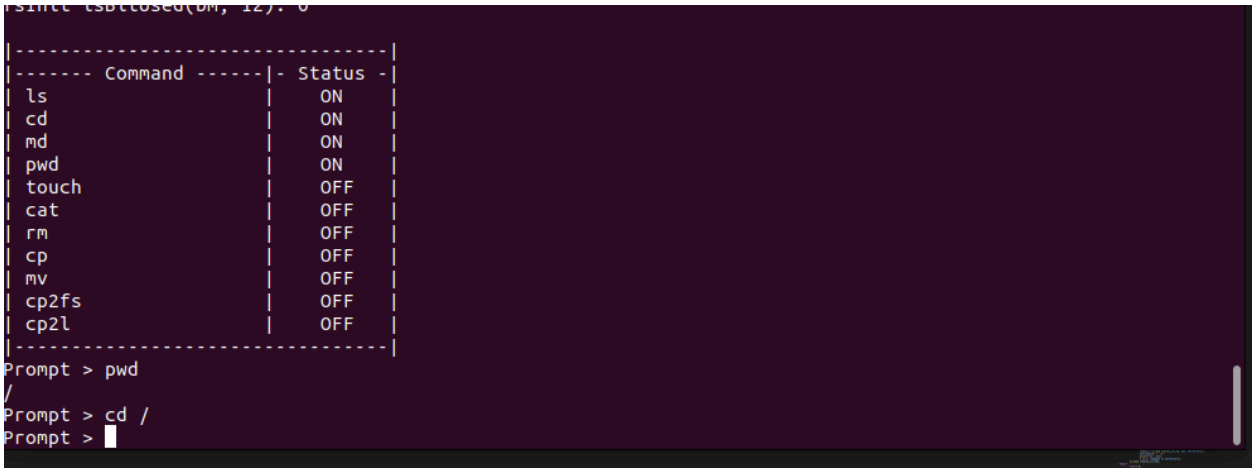
Md home + debug statement for readdir, successfully iterating through DEs in parent:



Md /home + debug statement for readdir, successfully iterating through DEs in parent



Cd /:



Cd + debug statement for readdir, successfully iterating through DEs in parent: