

Komaldeep Kaur
Raymond Liu
Aleia Natividad
Collins Gichohi
Louis Houston

The Strugglers
CSC415 Operating Systems

File System Project

Team: The Strugglers

[Github Link](#)

Members

Name	Student ID	Github Name
Collins Gichohi	922440815	gsnilloC
Louis Houston	922379442	LouisHouston
Komaldeep Kaur	920198887	komalkaaur
Raymond Liu	916624142	Airray117
Aleia Natividad	922439437	leileigoose

File System Approach

Milestone 1 - Creating the VCB and Root Directory

Task Division

Our first milestone was to design/create the Volume Control Block, the Root Directory, and a free space manager. While reading the steps for Milestone 1, our group had decided that we would break up the tasks into two groups: 1) The VCB group and 2) The Root Directory and Freespace. Group 1 would only deal with the Volume Control Block since it was a harder concept for us to grasp. Group 2 would have those two tasks as we went over how to do the root directory in class, so we thought that would be easier. The way we divided tasks was via random selection—names were pulled from a hat.

Main Functions

Volume Control Block (VCB)

We had our feedback from our previous VCB design and modified a few things.

- Changing the unsigned int values we previously had to `u_int64_t` and `u_int32_t`
- Changing the signature to something easier to compare vs a char array
- Added a variable to hold the location of the FreeSpace bitmap and the location of the Root Directory.

After making those changes, We tried to understand whether or not we had to format the volume. This took some discussion, but we concluded that because it is our filesystem, if the VCB doesn't match then it's already established that the volume isn't formatted for our system. Therefore we can create a VCB struct just to check the signature but if it doesn't have the same struct as ours it already fails.

Once that was out of the way, all that was needed was to create the `initVCB` function which initializes all the values given the number of blocks and the block size. It turned out that the VCB was easier than we thought.

`initVCB`

To create a new VCB, memory needs to be allocated, then all of the VCB variables: `volumeName`, `totalBlocks`, `freeBlockCount`, `locFreeSpaceBitMap`, `locRootDir`, `blockSize`, and `Signature`. As mentioned previously, this function will only run if the VCB signature that we're loading doesn't match our file system's signature.

Free Space

To keep track of our free space, we decided to create a bitmap that would keep track of the free blocks. The free space map would be an array of bits (0s and 1s). Each bit would indicate a block with 0 being free and 1 being occupied.

If the free space map is indicated by bits in an array, we needed ways to set, check, and clear bits, so those functions were worked on first. To make sure these functions worked a debug function called printBitMap.

After creating those functions, we could then initialize the free space map by allocating the space in memory and writing those blocks to disk. The free space would be initialized when a new VCB was created, so when a known VCB is loaded into the system, we created a function to load that specific VCB's free space.

initFreeSpace

Initializes the free space map, allocating memory for the map and setting all bits to 0 (indicating all blocks as free). The function then sets the first 6 bits of the bitmap to be used because they are occupied by the VCB and free space map. Finally, it writes these blocks to disk and returns the starting block of the free space map.

allocateBlocks

The function takes the number of blocks and how many blocks should be in each extent. If the numbers are the same this will ensure contiguous allocation if possible. Similar to Professor Bierman's example, this function allocates blocks by finding consecutive free blocks to form extents. It then marks these blocks as used and returns an array of extents. The function also updates the free space map accordingly and writes it to disk.

Helper Functions

- setBit: Sets a specific bit to indicate it's used.
- clearBit: Sets a specific bit to indicate it's free.
- checkBit: Checks if a bit is used or free.
- printBitMap: Prints the free space map.

Root Directory/Directory Initialization

During this milestone, we created the function that would initialize a directory. It sets all the default metadata, including the current directory and the parent. The location of the directory and its entries was stored in extent tables, and then all of it was written to disk.

We also thought about including different permissions for specific groups of users, from just viewing(reading) to modification(writing). In the DirEntry struct, we had a char fileName

that takes in a C-string containing the file name limited by the `MAX_FILE_NAME`, which is a const value of 256. An unsigned long `fileSize` which contains the size of the directory entry. An unsigned int `fileLocation` which contains the location of the directory entry. We also added `time_t` types after one of our group members brought up the `time.h` library for our timestamp for `lastModified`, `lastAccessed`, and `timeCreated`. Lastly, we set a variable `char isDirectory` as a flag as an indicator to indicate if it is a file or a directory.

Milestone 2 - Directory Functions (`fs_` functions)

Task Division

Milestone 2 dealt with creating the file system functions that would handle the directories. These functions are listed in the `mfs.h` file. We decided to do task division a little differently this time. We used a task board website called Trello and listed all the functions that needed to be done. We let everyone decide on their own which functions they wanted to work on where their names could be assigned to a task and it would indicate the progress on the task. Once we completed the functions on our own, we came back together to integrate them together so that things did not break.

Main Functions:

parsePath

`parsePath` is used to retrieve actual directory entries when given path name. It returns a `ppInfo` struct which has the fields `parent`, `index` (index of last element), and the last element as a string. First the function checks if the path is absolute or relative. If it is absolute the start path is set to the global variable `rootDir`, if it is relative the start path is set to the global variable `cwd`. We then tokenize the `pathname` using `“/”` as a delimiter. If there is no other token after the first `“/”` then the function returns the parent as `rootDir`, the index as `-1`, and the last element as null. The function then enters a while loop that processes each token of the path, if that element of the path exists it loads the directory and continues until all tokens have been processed and directories loaded. However, if the last element does not exist the function returns a non zero integer signifying an error.

fs_isDir

The function was overall pretty simple because our `DirEntry` struct included a flag for whether or not the entry that is in scope is a directory or not. Creating and using external functions was critical with the implementation of this function since it was easy to get lost in the concepts and

thinking process when having to deal with other functions that can essentially do one part very well and can be reused. Parsing the path would get us into the pathname's exact location so going up one level we can then have access to the DirEntry of the exact location and after that it was a simple check on the flag itself.

fs_isFile

We approached this function the same way we did fs_isDir but this time we checked if the Directory Entry is a file or directory by checking the flag that was set in the DirEntry struct.

fs_setcwd

Set_cwd was weird at first, on the top level it looks like a simple function put in a path and set that path to the current dir. However, there were a lot of other errors you have to check for and you have to check the final destination's parent directory for the directory. Then you have to set the global variable of cwd to that parent's directory final destinations index. For this function creating an external function called '**findEntryInDir**' was really useful since going about the logic you can get stuck trying to find the index of the child you are looking for. But really the only two things you have to check for in case of user error is either trying to set a directory that doesn't exist or if the entry itself of the final destination is a file.

fs_getcwd

Due to our global variable of cwd in directories that works itself with parsePath and sets itself to the rootDir this function was pretty straight forward. Return the name of the path to the cwd global variable

fs_mkdir

This function is used to create a new directory using the initDirectory function we implemented in milestone 1. It begins with parse path retrieving the parent's directory entries and the other information returned from parse path. We then find the next available index inside the parents directory, this will be where the new directory will be stored. The new directory is then initialized with all its default metadata values and the first two entries holding the current directory and the parent directory info. To get the information of the directory that was just created, we LBread, the first block of the news directory which is the dot(itself) and use that entry we just read to fill in the info for the new directory inside the parent. We then write the parent back to disk with all its newly populated information.

fs_stat

fs_stat was one of the easier functions to implement. It begins by using parse path to retrieve the parent's directory entry, the pathname's index inside the parent, and the last element of the pathname. We then retrieve the information for the directory entry that was specified then populate the fs_stat struct buffer that was passed into the function. This function also handles the case of the pathname being root, and whether or not the file that was trying to be accessed even exists.

fs_delete & fs_rmdir

The implementations for these two functions are very similar. They both begin with parse path, getting all the information, and tangible directory entries it needs. We then retrieve the entry, from the parent, from the parent that we need. We check whether the entry is a directory or file. If its a file **fs_delete** will set the file name but to its default value and its extent table to NULL, this all marks the entry as unused. Then the parent is written back to disk preserving the information that we just updated. The same thing as done in **fs_rmdir** only if the entry is as a directory not a file.

fs_opendir

This function takes in a path name, then looks for the directory at the end of the path using parsePath. Once the directory is found, it'll get the first entry in the directory and return a fdDir structure (which is similar to a file descriptor) which holds information on the position in that directory of the entry, the size of the entry, and that entry's information which is in the form of a fs_dirent struct.

fs_readdir

This function takes in a fdDir. From this fdDir, we make sure that there is a next entry in that directory by checking if our current position is less than the amount of records in the directory. If there is none, we return NULL since that is the end of the directory. Otherwise, we update the entry information of fs_dirent from opendir, then return it.

fs_closedir

This function frees up the memory that we previously called malloc from opendir and readdir. This includes the fs_dirent variable and the fdDir variables. This is necessary to free up memory and reset them for the next time we call these functions.

Helper Functions

findEntryInDir

This function takes in a directory and an entry name. It then goes through the directory until it finds the entry name within the directory and returns its index, if the entry was not found it will return -1.

findNextAvailableEntry

This function takes a directory and iterates through it until it finds an index that is still marked as free so that it could be used.

deleteDirEntry

This simple helper function just sets the metadata of a specific entry back to its default values indicating that the entry is now free.

LoadDir

This function gives us the most trouble. It is meant to take in a Directory Entry and load all its entries and corresponding information off of disk. It takes the entry itself and loads the full directory that the entry is holding the metadata for. It takes the startBlock and count of the entry and reads it off disk population an empty structure that was allocated just prior. It then returns the populated structure which holds all the entries of the directory that was specified by the entry that was passed into the function.

Milestone 3 - File System Functions

Task Division

b_open

For this function, the vcb struct needed extra attributes to keep things more streamlined in terms of fd and the data itself. It was simple basing it on assignment 5 and how fd works. We approached this problem similar to if the files were books and each fd was a reader. So each book would get a reader. Obviously, since it would be the first time the reader has ever “touched” the “book” it would be essentially the same thing as initializing the “reader” or fd and then adding them to our array or “list” of readers that we have currently reading for us when we want them to read.

b_seek

For this function pretty simple only confusing part was whence, same approach a reader and book except this time with the specific offset of where to start reading the book or changing

Komaldeep Kaur
Raymond Liu
Aleia Natividad
Collins Gichohi
Louis Houston

The Strugglers
CSC415 Operating Systems

where the position or line of the book is currently depending on what type of argument of whence it given. Used a switch clause since a very minute amount of arguments and simple calculations depending on the type of whence given. Once again the additional attributes that were in our new modified vcb came in handy as I had file position or file length given to help with the whence statements. Not going to act smart I had no clue what whence meant at first.

b_close

I don't know why I love this analogy but the same thing in the group of our book readers simply tell that reader to stop reading the book and go home. Pretty straightforward except for the filename needing to be null and that is because of the pointer type same thing with the buff though.

Issues and Resolutions

Issue: Something didn't make sense when we initialized the directory and it shouldn't have taken up as many blocks as it did (reference to Milestone 1 Block 2).

Resolution: There were issues regarding the calculations in `initDirectory`. We were confused about what the contents of the directory entry were so we allocated memory for every single possible directory entry that could be created, this made our root directory extremely large. We fixed this by redoing the calculations so now the directory only allocates enough memory for the actual directories entries asked for. This was just a simple miscalculation on our part that was resolved quickly.

Issue: When testing `mkdir`, we got a segmentation fault. While debugging, we found that our `parsepath` function was not handling nested directories correctly. Going more into it, the problem was the LoadDir function where we call the extent table to calculate how many blocks we were reading from to load the directory entry. So we decided to just have a `startBlock` and `blockCount` field into the `DirEntry` struct, which would force us to work with contiguous allocation at least for the directory entries.

Now we were getting the correct start blocks and counts for the specific entries, but when we passed those values into the `LBaread()` function nothing was returned and the directory structure that should have been populated remained empty. So now this leads us to think that maybe the directory entries were not written to disk properly, however we are easily able to load the root directory off of disk.

This error causes `parsePath` to not work correctly if we are dealing with directories who do not have the root as the parent. As of not without fixing that problem we cannot make any nested directories, only directories in root because it is always loaded from the disk when the file system is started.

Resolution: We never did resolve this issue and it is a fatal flaw within our file system. I think the issue lies within the way we are writing and reading the structures off of disk. We tried to manually do it but that did not work, maybe with a little more time and some guidance we could have figured it out.

Issue: Not being able to get all the file system functions fully working. We managed to get out 5 of 12 functions working. When combining the team's code, there were some functions that were not compatible with each other.

Resolution: A lot of print statements to debug and understand each other's code helped fix at least a few functions, but not all of them.

Driver Program

The driver program starts our file system by calling the `initFileSystemMethod()`. The method takes in the number of blocks our file system will support and the size of each block. We then malloc a VCB struct that will be used to read the first block of the file system which will always be by our volume control block. We check the signature of the volume control block we just created. If the signatures match we load the free space, the root directory, and we set the current working directory to the root. If the signatures do not match we have to format the volume, this involves initializing our free space manager, the root directory and the volume control block. Once our file system is started, the driver program then waits for commands from the user and interacts with the file system as directed.

Komaldeep Kaur
Raymond Liu
Aleia Natividad
Collins Gichohi
Louis Houston

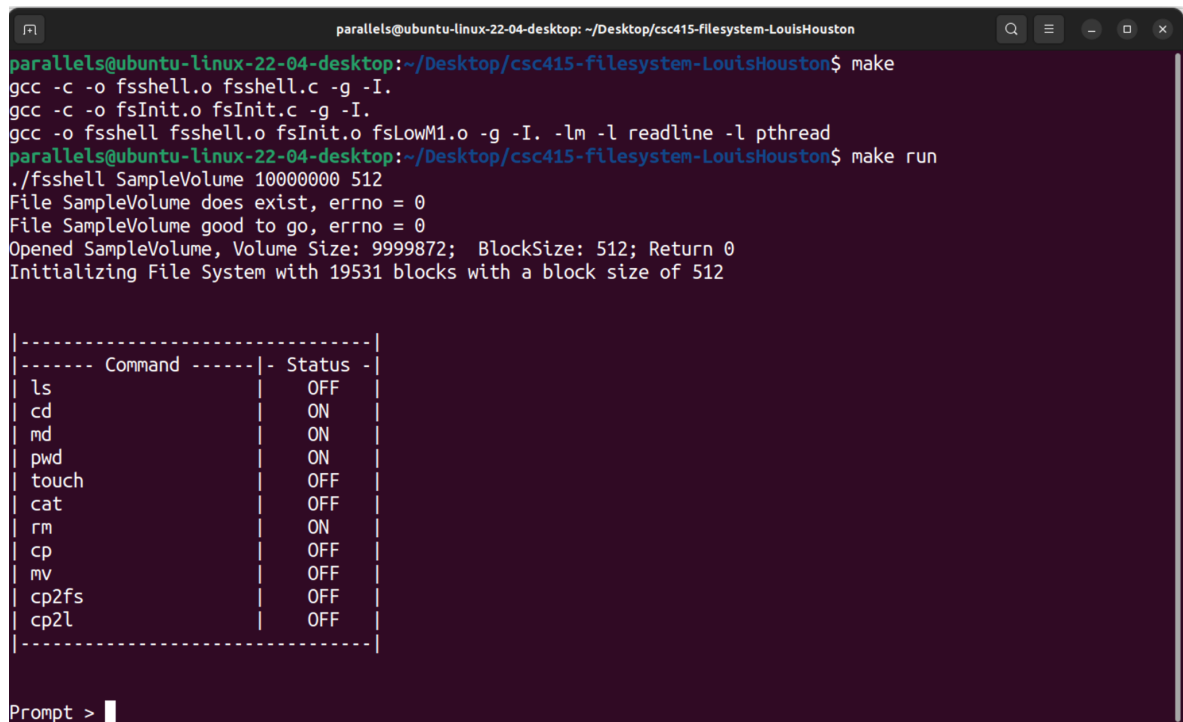
The Strugglers
CSC415 Operating Systems

List of Functions that Work

- ☐ 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 - creates a file
- ☐ cat - (limited functionality) displays the contents of a file
- ☐ 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~~

Screenshots of Commands

- Compilation

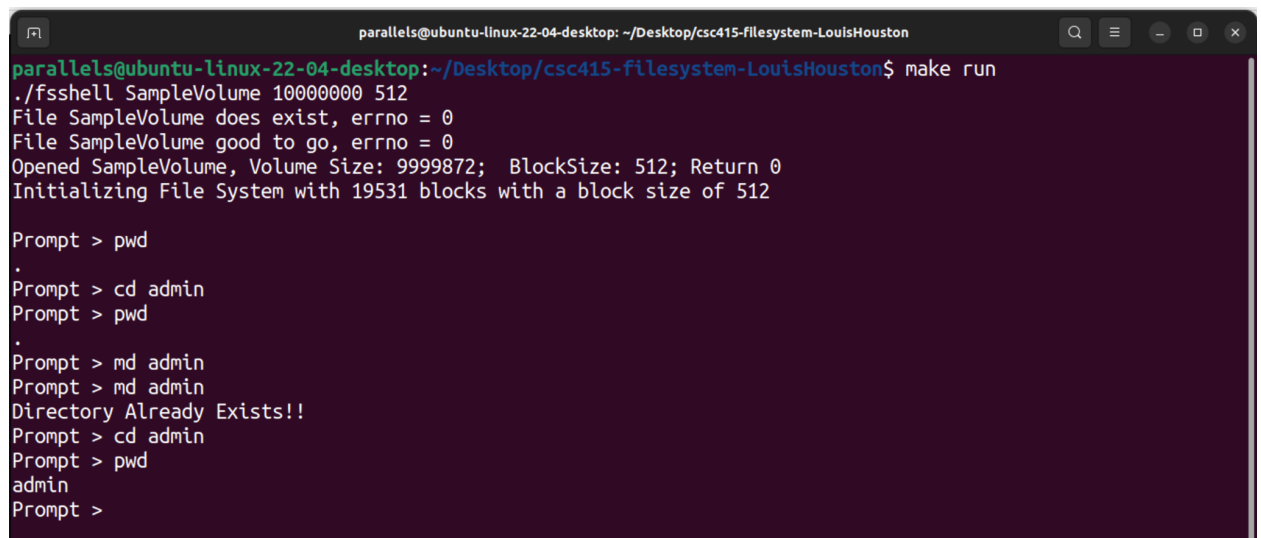


```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston
parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLowM1.o -g -I. -lm -l readline -l pthread
parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

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

Prompt > 
```

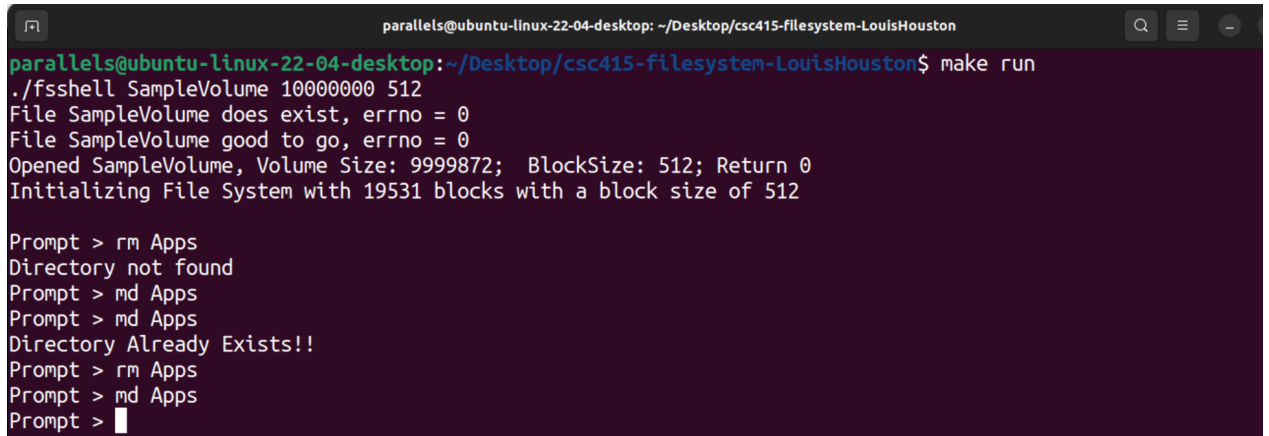
- md - Make a new directory



```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston
parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

Prompt > pwd
.
Prompt > cd admin
Prompt > pwd
.
Prompt > md admin
Prompt > md admin
Directory Already Exists!!
Prompt > cd admin
Prompt > pwd
admin
Prompt > 
```

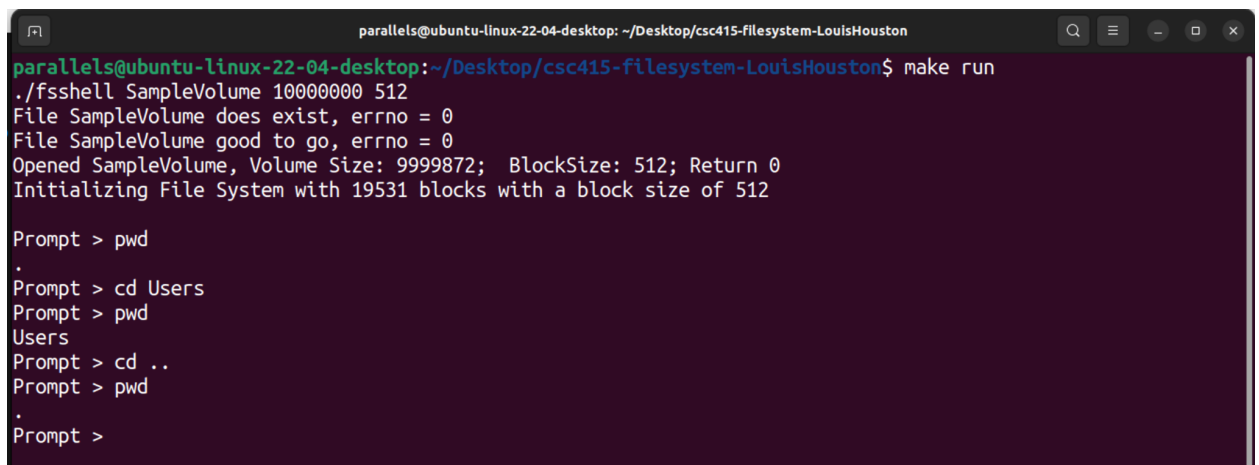
- rm - Removes a file or directory

A terminal window titled "parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston". The prompt is "parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston\$". The user enters "make run", which executes a script. The script output is: ". /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". Then the prompt changes to "Prompt >". The user enters "rm Apps", which outputs "Directory not found". The user enters "md Apps", which outputs "Directory Already Exists!!". The user enters "rm Apps", which outputs nothing. The user enters "md Apps", which outputs nothing. The prompt is still "Prompt >".

```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

Prompt > rm Apps
Directory not found
Prompt > md Apps
Prompt > md Apps
Directory Already Exists!!
Prompt > rm Apps
Prompt > md Apps
Prompt >
```

- cd - Changes directory

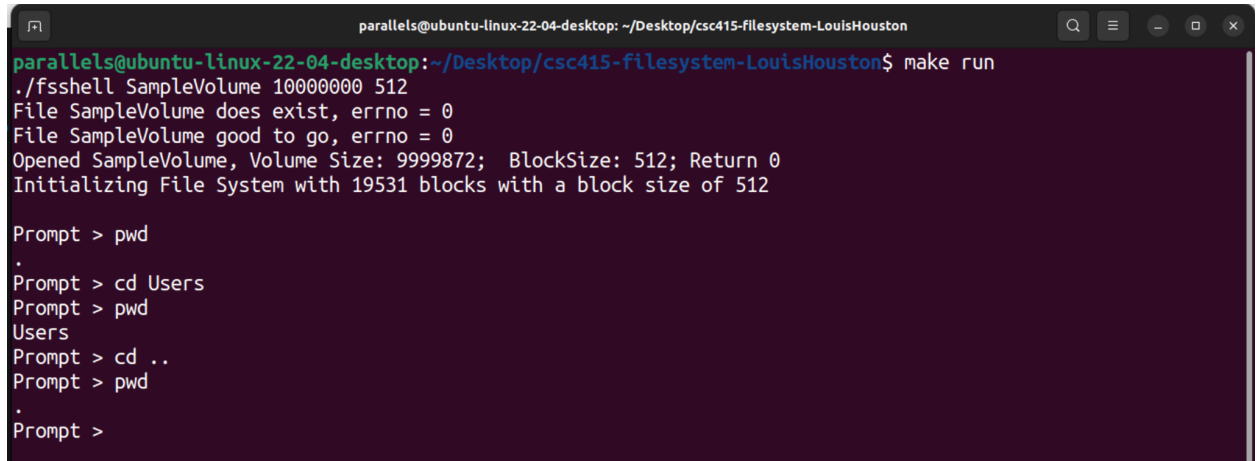
A terminal window titled "parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston". The prompt is "parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston\$". The user enters "make run", which executes a script. The script output is: ". /fsshell SampleVolume 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". Then the prompt changes to "Prompt >". The user enters "pwd", which outputs ".". The user enters "cd Users", which outputs nothing. The user enters "pwd", which outputs "Users". The user enters "cd ..", which outputs nothing. The user enters "pwd", which outputs ".". The prompt is still "Prompt >".

```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

Prompt > pwd
.
Prompt > cd Users
Prompt > pwd
Users
Prompt > cd ..
Prompt > pwd
.
Prompt >
```

“.” is a reference to the root Directory

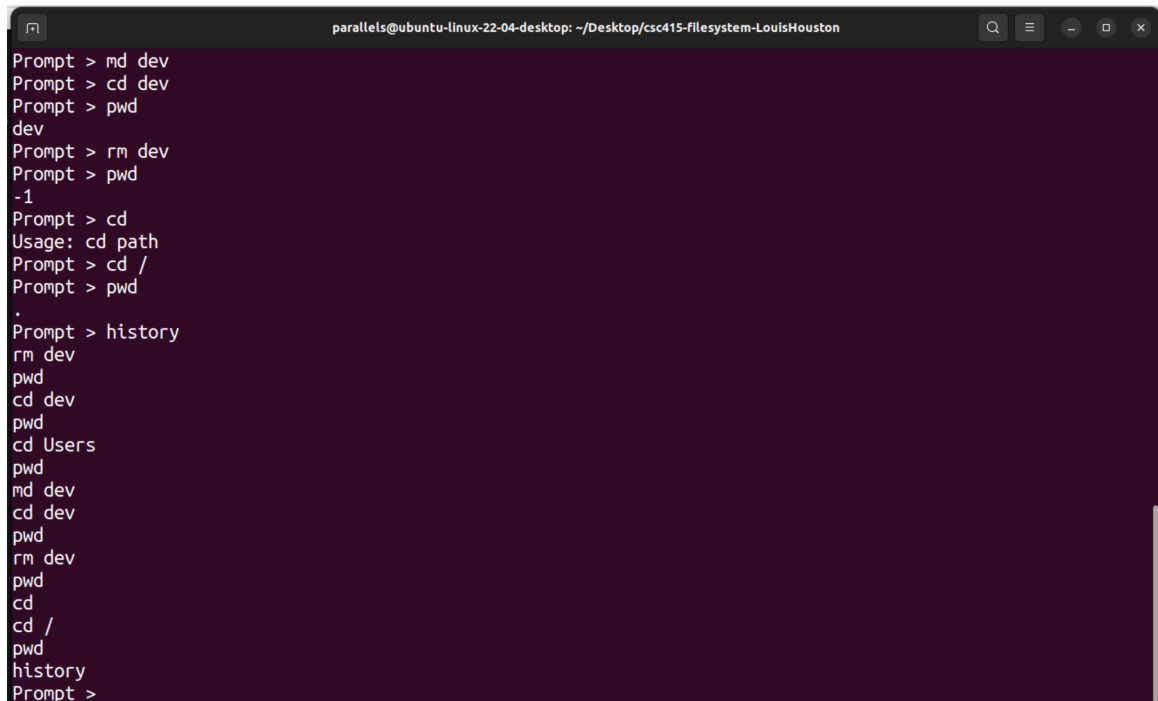
- pwd - Prints the working directory



```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston
parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

Prompt > pwd
.
Prompt > cd Users
Prompt > pwd
Users
Prompt > cd ..
Prompt > pwd
.
Prompt >
```

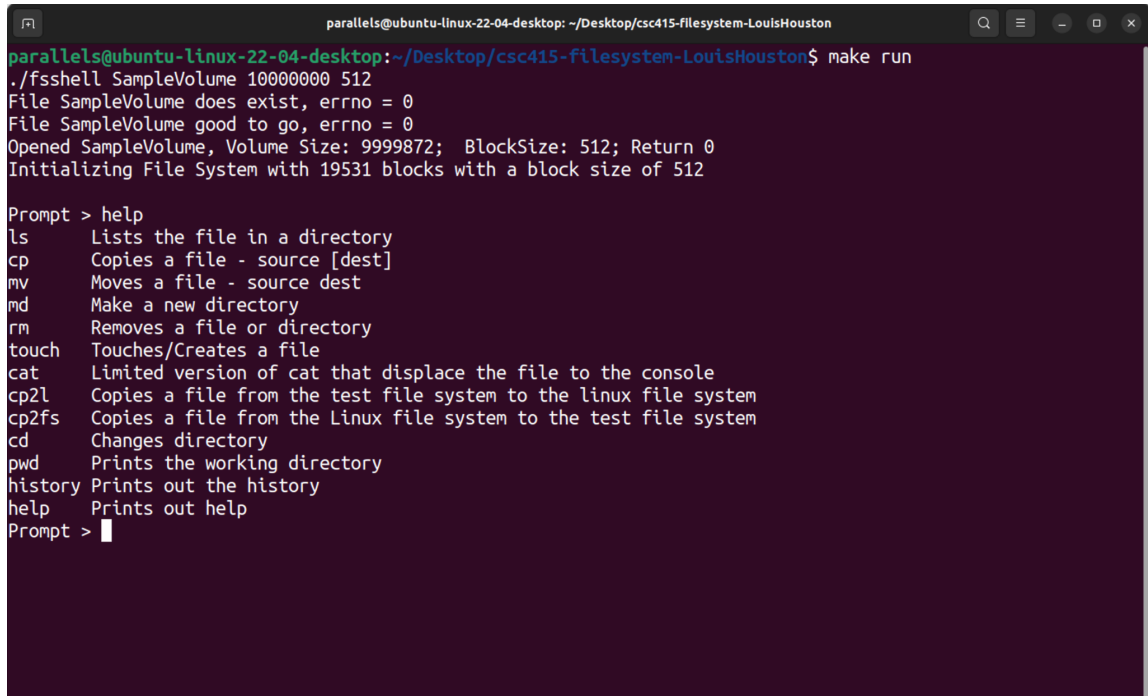
- history - Prints out the history



```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston

Prompt > md dev
Prompt > cd dev
Prompt > pwd
dev
Prompt > rm dev
Prompt > pwd
-1
Prompt > cd
Usage: cd path
Prompt > cd /
Prompt > pwd
.
Prompt > history
rm dev
pwd
cd dev
pwd
cd Users
pwd
md dev
cd dev
pwd
rm dev
pwd
cd
cd /
pwd
history
Prompt >
```

- help - Prints out help



```
parallels@ubuntu-linux-22-04-desktop: ~/Desktop/csc415-filesystem-LouisHouston
parallels@ubuntu-linux-22-04-desktop:~/Desktop/csc415-filesystem-LouisHouston$ 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 19531 blocks with a block size of 512

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 > 
```