

# **Team No Preference**

## **Team Members:**

Jeffrey Friedrich	920095797
Brian Cheng	918757468
Michael Davichick	920196625
Ives-Christian “Jay” Jadman	920450086

## **Repository:**

<https://github.com/CSC415-Fall2021/csc415-filesystem-ljadman>

# Table of Contents:

<b>Final Assignment Description:</b>	<b>5</b>
<b>Final Report</b>	<b>6</b>
File System Description:	6
VCB (fsVCB.c):	6
Changes from Milestone 1:	6
Free Space (fsFree.c):	6
Changes from Milestone 1:	6
Directory/ Directory Entries (fsDir.c):	6
Changes from Milestone 1:	6
Directory Operations (mfs.c, fsParse.c, fsString.c):	6
mfc.h: includes all of the implemented directory functions.	6
fsParce.c: includes the parser which takes in a path and checks if it's valid up until the second-to-last directory. Depending on which function we are working with, we can do different operations with this.	6
fsString: used to modify the current working directory path. This has a function that reverses the pathname which will be used to remove the last element, first by getting the reversed first token as the last when changing to the parent directory.	6
File Operations (b_io.c, mfs.c):	6
Description: this file handles all the file operations for our file system( i.e. read , write, seek, etc.)	6
Issues:	7
Issue 1 - Tokens had additional characters appended:	7
Resolution 1:	7
Issue 2 - Command "cd .." not working properly	7
Resolution 2:	7
Successful Compilation:	8
Driver Program Description (fsshell.c):	8
Command Descriptions:	8
ls	9
Screenshot:	9
Description:	9
Test Cases:	9
Bugs:	9
cp <source> [<dest>]	10
Screenshot:	10
Description:	10
Test Cases:	10
Bugs:	10

---

mv <source> <dest>	10
Screenshot:	10
Description:	10
Test Cases:	10
Bugs:	10
md <file/pathname>	11
Screenshot:	11
Description:	11
Test Cases:	11
Bugs:	11
rm <file/pathname>	12
Screenshot:	12
Description:	12
Test Cases:	12
Bugs:	12
cp2l <file/pathname>	13
Screenshot:	13
Description:	13
Test Cases:	13
Bugs:	13
cp2fs <file/pathname>	13
Screenshot:	13
Description:	13
Test Cases:	13
Bugs:	13
cd <file/pathname>	14
Screenshot:	14
Description:	14
Test Cases:	14
Bugs:	14
pwd	14
Screenshot:	14
Description:	15
Test Cases:	15
Bugs:	15
history	15
Screenshot:	15
Description:	16
Test Cases:	16

---

Bugs:	16
help	16
Screenshot:	16
Description:	16
Test Cases:	16
Bugs:	16
<b>Milestone One Description:</b>	<b>17</b>
<b>Milestone One Submission:</b>	<b>18</b>
Coding Status:	18
Coding Write-up:	19
Compilation:	19
Make Run:	19
Hexdump (BLOCK 1: Volume Control Block) :	20
Hexdump (BLOCK 2: Root Directory ) :	21
VCB Structure Description:	22
Free Space Structure Description:	22
Directory System Description:	22
Changes to Design and Why:	22
Team Dynamics:	22
Team Approach:	23
Team Meetings:	23
Issues/Resolutions:	24
<b>Design Description:</b>	<b>25</b>
<b>Design Submission:</b>	<b>26</b>
Volume Control Block:	26
Free Space Management:	27
Directory Entry:	28
File Metadata:	28
<b>Notes (not part of our submission):</b>	<b>29</b>
<b>References:</b>	<b>30</b>

## Final Assignment Description:

What needs to be submitted (via GitHub and iLearn) (include Git URL in iLearn submission):

- All source files (.c and .h)
- Modified Driver program (must be a program that just utilizes the header file for your file system, i.e. you can not use any linux or library file routines).
- The Driver program must be named: fsshell.c
- A make file (named “Makefile”) to build your entire program
- A PDF writeup on the project that should include (**this is also submitted in iLearn**):
  - The github link for your group submission.
  - A description of your file system
  - Issues you had
  - Detail of how your driver program works
  - Screenshots showing each of the commands listed in the readme
- Your volume file (limit 10MB)
- There will also be an INDIVIDUAL report (form) to complete.

# Final Report

## File System Description:

### VCB (fsVCB.c):

Changes from Milestone 1:

No major changes have been made since milestone 1.

### Free Space (fsFree.c):

Changes from Milestone 1:

Free space information is stored in DirEntries but not on the actual blocks anymore.

### Directory/ Directory Entries (fsDir.c):

Changes from Milestone 1:

No changes to the structure of Directory Entries have been made since milestone 1. Many new functions were created to help manipulate them though (reads, writes, searches, etc).

### Directory Operations (mfs.c, fsParse.c, fsString.c):

mfc.c:

includes all of the implemented directory functions.

fsParse.c:

includes the parser which takes in a path and checks if it's valid up until the second-to-last directory. Depending on which function we are working with, we can do different operations with this.

fsString:

used to modify the current working directory path. This has a function that reverses the pathname which will be used to remove the last element, first by getting the reversed first token as the last when changing to the parent directory.

### File Operations (b\_io.c, mfs.c):

Description:

this file handles all the file operations for our file system: read, write, open, and seek.



## Issues:

### Issue 1 - Tokens had additional characters appended:

Functions `md` and `rm` took a string parameter that would then have to be tokenized in order to separate each directory from one another. We stored each of the directories into an array and returned a directory entry pointer to the  $n-1$  index while storing the  $n$ th entry into a buffer that was passed into the `parsePath` function. The logic behind this was that we didn't know if the  $n$ th object existed, so by returning a pointer to the  $n-1$  index (assuming it wasn't null), we could then search through that directory entry array to see if the  $n$ th index object (that resides in the buffer) existed. The problem however, was that the last token ( $n$ th index) would have additional unwanted characters appended to it. This caused problems specifically in `remove`, because the targeted file varied from what actually existed, resulting in a "file not found" error.

### Resolution 1:

This issue resulted from a lack of a null character at the end of the last token. Since this token was added to a character buffer, any previous garbage that existed in the buffer previously would append itself to the end. Our first attempt was to add a null character to each individual token, ensuring it had one attached before being set to the buffer; however this didn't result in the outcome we expected. It continued to append random characters to the end, so we changed our focus to the buffer itself. The resolution came from filling the entire buffer with null characters before passing it into the `parsePath` function; that way regardless of how long the token was, there would always be a null character that followed it. In doing so, we found that the outcome had become consistent with our expectations every time.

### Issue 2 - Command "`cd ..`" not working properly

The case, "`cd ..`", would not change the pathname correctly at times. One instance of this was when we had the path name, "`/foo/bar`." Say this was our current working directory. After calling "`cd ..`" the new path name would print out "`foobar`," deleting the slashes. To find the last element, we created a buffer for the reversed path name ("`rab/foo`"), and tokenized it with the delimiter `"/`". This way we would get "`rab`." However, tokenizing would override the value of the reversed pathname, giving us an incorrect path name.

### Resolution 2:

To solve this, we had to set the buffer to null and then concatenate the new path. The last element was stored in the token, and with this we decided to concatenate the original path name up to the last slash so that the path will change from "`/foo/bar`" to "`/foo`."



## Successful Compilation:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-Ijadman$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsFree.o fsFree.c -g -I.
gcc -c -o fsVcb.o fsVcb.c -g -I.
gcc -c -o fsDir.o fsDir.c -g -I.
gcc -c -o fsParse.o fsParse.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o fsString.o fsString.c -g -I.
gcc -c -o fsTest.o fsTest.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsFree.o fsVcb.o fsDir.o fsParse.o mfs.o fsString.o fsTest.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Documents/csc415-filesystem-Ijadman$
```

## Driver Program Description (fsshell.c):

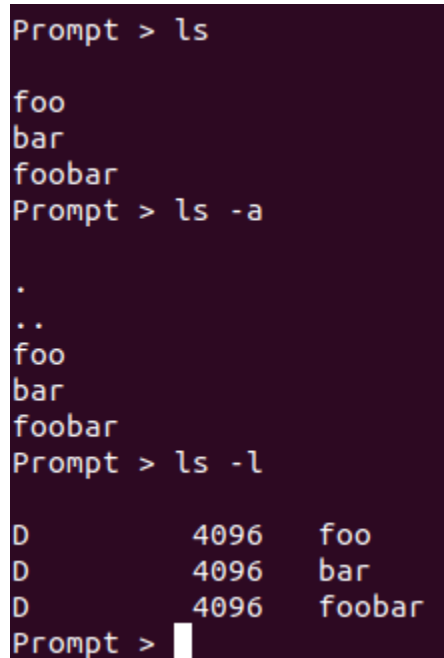
Fsshell.c is the main driver for our file system project. It houses an initial structure named `dispatchTable` which holds a shell command, as well as its associating function and description. Each command function within the `dispatchTable` calls other varying functions inside its implementation, in which we were responsible for creating. At a macro level, `fsshell.c` drives not only the shell but the entire project as a whole; it's here that each function we've written cohesively comes together to create one seamless entity.

Fsshell.c functions like `cmd_ls` rely on our implementation of `fs_isDir`, in which the return value is used in condition checks to execute the rest of its contents. While other functions like `cmd_md` are just housings that abstract the real implementation of creating a directory that exists within our `fs_mkdir`.

## Command Descriptions:

ls

Screenshot:



```
Prompt > ls
foo
bar
foobar
Prompt > ls -a
.
..
foo
bar
foobar
Prompt > ls -l
D      4096  foo
D      4096  bar
D      4096  foobar
Prompt > 
```

Description:

The ls command is used to list a directory's contents and print them out to the user. These contents exist in the form of directory entries, each consisting of either a file or a directory.

Test Cases:

Test cases for ls are executed during flag calls. As shown in the screenshot above, we've tested ls by itself alongside the -a flag and -l flag. The -a flag is supposed to show all entries and not exclude . and .. which are present. The -l flag uses a long listing format, detailing the type of directory entry (file or directory) as well as the size available to each struct.

Bugs:

RESOLVED: When we first tested ls in the root directory, the error message printed out that the program could not search the directory "/".

`cp <source> [<dest>]`

Screenshot:

We were unable to implement the correct functionality for this function.

Description:

The `cp` command is used to copy files, a group of files or a directory. It creates an exact image of that copy on disk with a different name.

Test Cases:

Bugs:

`mv <source> <dest>`

Screenshot:

We were unable to implement the correct functionality for this function.

Description:

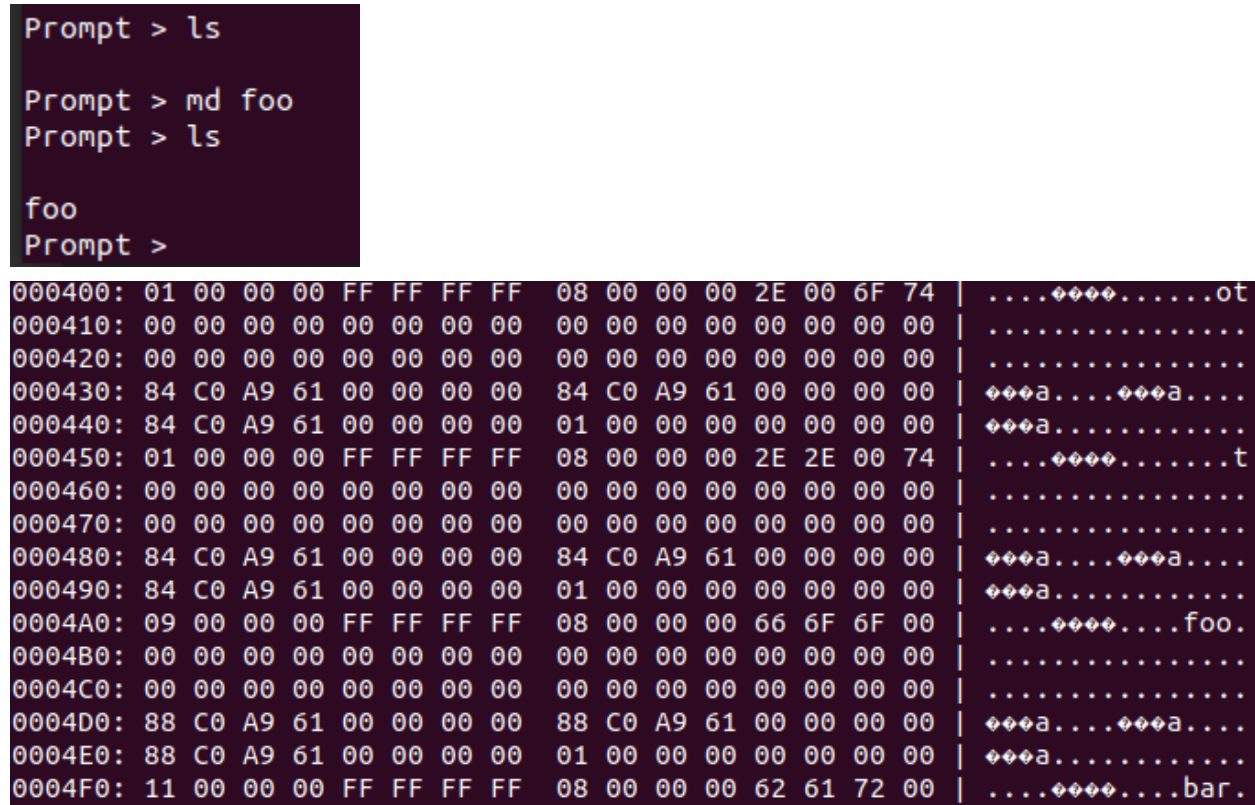
The `mv` command is used to move one or more files, or directories from one place to another in the file system. It has two distinct functions: 1) it renames the file or folder 2) it moves the selected files or directories to a different existing directory.

Test Cases:

Bugs:

md <file/pathname>

Screenshot:



```
Prompt > ls

Prompt > md foo
Prompt > ls

foo
Prompt >
```

```
000400: 01 00 00 00 FF FF FF FF 08 00 00 00 2E 00 6F 74 | .....ot
000410: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000430: 84 C0 A9 61 00 00 00 00 84 C0 A9 61 00 00 00 00 | ....a....a....
000440: 84 C0 A9 61 00 00 00 00 01 00 00 00 00 00 00 00 | ....a.....
000450: 01 00 00 00 FF FF FF FF 08 00 00 00 2E 2E 00 74 | .....t
000460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000480: 84 C0 A9 61 00 00 00 00 84 C0 A9 61 00 00 00 00 | ....a....a....
000490: 84 C0 A9 61 00 00 00 00 01 00 00 00 00 00 00 00 | ....a.....
0004A0: 09 00 00 00 FF FF FF FF 08 00 00 00 66 6F 6F 00 | .....foo.
0004B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004D0: 88 C0 A9 61 00 00 00 00 88 C0 A9 61 00 00 00 00 | ....a....a....
0004E0: 88 C0 A9 61 00 00 00 00 01 00 00 00 00 00 00 00 | ....a.....
0004F0: 11 00 00 00 FF FF FF FF 08 00 00 00 62 61 72 00 | .....bar.
```

Hex dump of the root directory showing that foo has been created in it.

Description:

The md command is used to create a new directory that is placed in the current working directory. Each new directory contains a directory entry array that will initially contain only two entries: 1) It's parent entry ( .. ) 2) Itself ( . )

Test Cases:

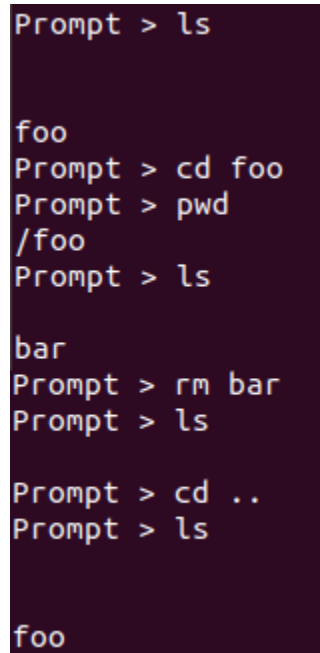
Test cases regarding our md function involve creating directories in different locations. For example, creating multiple directories within a single location, as well as traversing through these newly created directories to then create new directories within those directories as well. To ensure each of these directories have the intended execution, we list the contents of the current working directory and traverse as needed.

Bugs:

RESOLVED: As stated in Issue 1 listed above; we found that files would periodically have extra characters added to the end of string names.

rm <file/pathname>

Screenshot:



```
Prompt > ls
foo
Prompt > cd foo
Prompt > pwd
/foo
Prompt > ls
bar
Prompt > rm bar
Prompt > ls

Prompt > cd ..
Prompt > ls

foo
```

Description:

The rm command removes a file or directory given a path. If the path leads to a directory, rm will remove this directory by freeing its directory entry pointer (found in the parent's directory entry array), rewriting the disk such that the free space management knows those blocks are free, and freeing the pointer to that space.

Test Cases:

Similar to make directory, these test cases include the removal of multiple directories and files to ensure consistency. In addition to removing existing directory entries (files or directories), there's the case of attempting to remove a non existing directory entry. In this scenario, the user will be met with an error prompt, letting them know the removal was unsuccessful due to the file not being found.

Bugs:

RESOLVED: While implementing this function, we discovered that some files would not be removed because of an error in our parsePath function. Here, some tokens had added garbage at the end of their name, and because of this, rm would try to remove the directory, given the name plus the extra garbage.

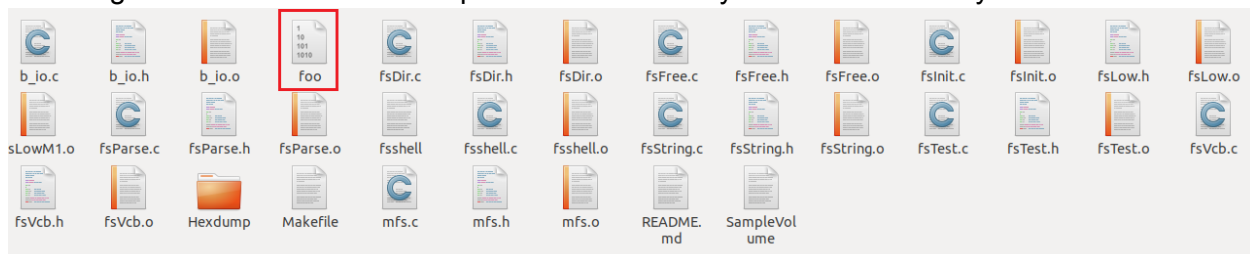
To solve this, we found that the temporary path name in our parsePath did not malloc the correct size. It was only malloced up to the size of the pathname, not leaving space for the null character. Because of this, random garbage could have been added to our tokens at any moment. After discovering this, we simply increased the malloc size by 1 for the null character.

The rm function had trouble telling when a directory was empty. Because of this, the directory would end up getting deleted even though there might be stuff inside.

cp2l <file/pathname>

Screenshot:

The image below shows that we copied the foo directory into the linux file system



Description:

The cp2l command copies a file from the test file system to the Linux file system.

Test Cases:

Bugs:

ONGOING: There's a bug regarding the write aspect when copying. There are times the file is missing chunks upon copying.

cp2fs <file/pathname>

Screenshot:

```
Prompt > ls

Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-Ijadman$ ls
b_io.c  fsDir.o  fsLow.h  fsshell  fsTest.c  Hexdump  SampleVolume
b_io.h  fsFree.c fsLowM1.o fshell.c fsTest.h  Makefile
b_io.o  fsFree.h fsLow.o  fshell.o fsTest.o  mfs.c
foo     fsFree.o fsParse.c fsString.c fsVcb.c  mfs.h
fsDir.c fsInit.c fsParse.h fsString.h fsVcb.h  mfs.o
fsDir.h fsInit.o fsParse.o fsString.o fsVcb.o  README.md
student@student-VirtualBox:~/Documents/csc415-filesystem-Ijadman$ make run
./fshell 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 > cp2fs foo
-----Directory Entry Initialization-----
-----Directory Entry Initialization Finished-----
-----
Prompt > ls
foo
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-Ijadman$
```

Description:

The cp2fs command copies a file from the Linux file system to the test file system.

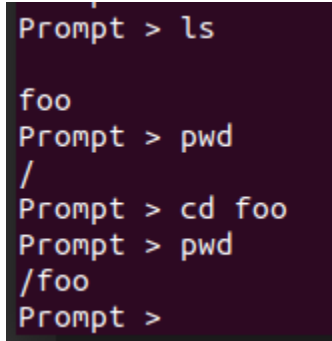
Test Cases:

Bugs:

ONGOING: The file system does not perform writes properly. We began to implement this but ran out of time.

`cd <file/pathname>`

Screenshot:

A terminal window with a dark background and light-colored text. The text shows a sequence of commands and their outputs: 'Prompt > ls' followed by 'foo' on the next line; 'Prompt > pwd' followed by '/' on the next line; 'Prompt > cd foo'; 'Prompt > pwd' followed by '/foo' on the next line; and finally 'Prompt >' on the last line.

```
Prompt > ls
foo
Prompt > pwd
/
Prompt > cd foo
Prompt > pwd
/foo
Prompt >
```

Description:

The `cd` command is used to change the current working directory. If the path does not exist, an error message will prompt to notify the user that a change could not be made. If the path does exist, the current working directory will change alongside a notification of success with the new current working directory path.

Test Cases:

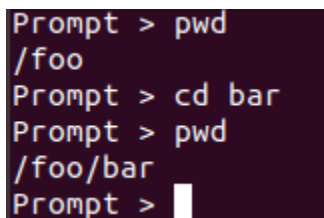
Test cases regarding `cd` were very similar to `remove`. We first created multiple paths that we then traversed through using `cd`; we followed up each `cd` with an `ls` and `pwd` to ensure the contents and our location were consistent. Like `remove`, our second conditional test case was trying to change to a non existing directory; the result of this prompts an error notifying that the change was unsuccessful due to path not found.

Bugs:

RESOLVED: One bug we ran into was in the case "`cd ../foo`". Here the `searchDirectory()` function would search the directory name `".."` and look for `"foo"` rather than the actual name of `foo`'s parent.

`pwd`

Screenshot:

A terminal window with a dark background and light-colored text. The text shows a sequence of commands and their outputs: 'Prompt > pwd' followed by '/foo' on the next line; 'Prompt > cd bar'; 'Prompt > pwd' followed by '/foo/bar' on the next line; and finally 'Prompt >' followed by a cursor on the last line.

```
Prompt > pwd
/foo
Prompt > cd bar
Prompt > pwd
/foo/bar
Prompt > 
```



#### Description:

The pwd command is used to print the current working directory. It prints the entire absolute path, starting at root ( / ), traversing until it reaches the directory the user is currently in.

#### Test Cases:

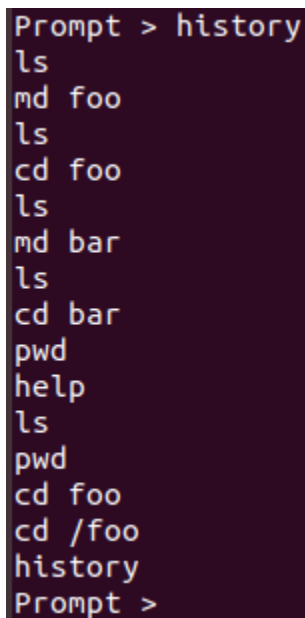
Test cases regarding pwd involved traversing through and printing multiple paths with varying depths.

#### Bugs:

No remaining bugs exist for pwd.

## history

#### Screenshot:



```
Prompt > history
ls
md foo
ls
cd foo
ls
md bar
ls
cd bar
pwd
help
ls
pwd
cd foo
cd /foo
history
Prompt >
```

#### Description:

The history command is used to list all of the previously used commands to the terminal window.

#### Test Cases:

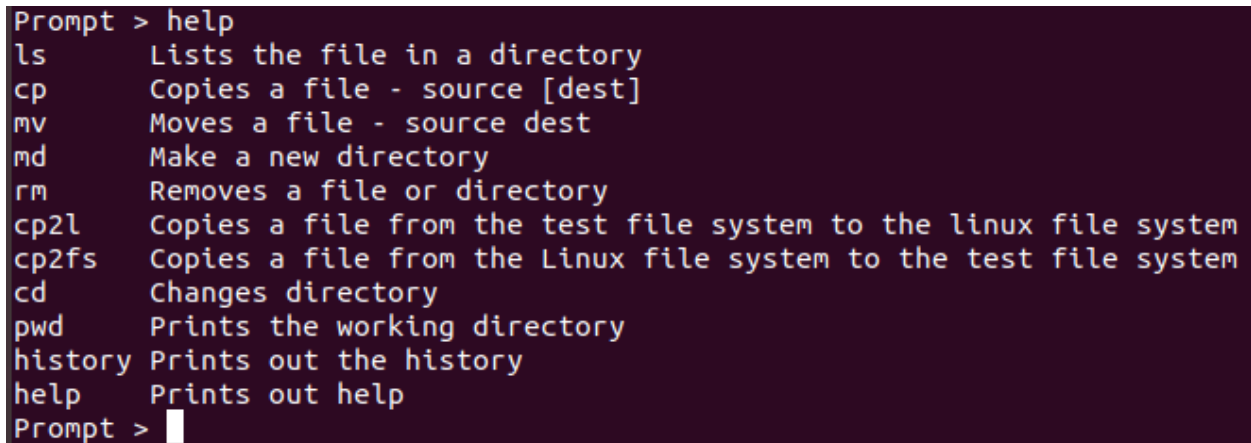
There weren't too many test cases regarding history. All we did was run multiple commands testing other features, then followed those up with a history prompt to ensure all the data was consistent.

#### Bugs:

No bugs were found during the execution of history.

#### help

#### Screenshot:



```
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
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 > 
```

#### Description:

The help command is used to list out all of the available commands that are used within the file system alongside a description of each use case.

#### Test Cases:

Similar to history, there were even less test cases used in the implementation of this function. Outside of calling the command and ensuring the data being presented was accurate, there wasn't much more to do for this section.

#### Bugs:

No bugs were found during the execution of help.

## Milestone One Description:

For this milestone of the project we want your volume "formatted".

This means that you have a volume ready for use, so:

1. The Volume Control Block (block 0) is written,
2. The Free Space management is written, this should include the initialization of the free space, and at least the allocate procedure to allocate free space (release - to place allocated blocks back into the free space pool is not a requirement of this milestone).
3. The Root Directory is written and initialized with at least the "." and ".." entries

The submission for this is a PDF that has a cover page with your team name, each team member's name, and student ID# and the github name with your groups work. The PDF should also have:

1. A dump (use the provided HexDump utility) of the volume file that shows the VCB, FreeSpace, and root directory.
2. A description of the VCB structure
3. A description of the Free Space structure
4. A description of the Directory system
5. A table of who worked on which components
6. How did your team work together, how often you met, how did you meet, how did you divide up the tasks.
7. A discussion of what issues you faced and how your team resolved them.

## Milestone One Submission:

### Coding Status:

Component:	Status:
Volume Control Block () -written -return values/error handling (optional) -fully documented (optional)	COMPLETE -COMPLETE -PARTIAL -PARTIAL
Free Space Management (fsFree) -written -initialization written -allocate written -free written (optional) -return values/error handling (optional) -fully documented (optional)	COMPLETE -COMPLETE -COMPLETE -COMPLETE -PARTIAL -PARTIAL -PARTIAL
Root Directory -written -initialized -"." entry - ".." entry -return values/error handling (optional) -fully documented (optional)	COMPLETE -COMPLETE -COMPLETE -COMPLETE -COMPLETE -INCOMPLETE -INCOMPLETE

## Coding Write-up:

### Compilation:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsFree.o fsFree.c -g -I.
gcc -c -o fsVcb.o fsVcb.c -g -I.
gcc -c -o fsDir.o fsDir.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsFree.o fsVcb.o fsDir.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$
```

### Make Run:

```
student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Creating VCB in Block 1
Initializing free space...
-----Directory Entry Initialization-----
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$
```

## Hexdump (BLOCK 1: Volume Control Block) :

```
student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$ Hexdump/hexdump.linux --count 1 --start 1 SampleVolume
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 56 43 42 00 00 00 00 00 00 00 00 00 00 00 00 00 | VCB.....
000210: 4B 4C 00 00 49 4C 00 00 00 02 00 00 00 00 00 00 | KL..IL.....
000220: 00 00 00 00 00 00 00 00 02 00 00 00 03 00 00 00 | .....
000230: 48 4C 00 00 FF FF FF FF 00 00 00 00 00 00 00 00 | HL.♦♦♦♦.....
000240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0002F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

000300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0003F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
```

## Hexdump (BLOCK 2: Root Directory) :

```

student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$ Hexdump/hexdump.linux --count 1
--start 2 SampleVolume
Dumping file SampleVolume, starting at block 2 for 1 block:
000400: 2E 00 00 00 00 00 00 00 A0 69 C8 6A 9E 55 00 00 | .....iU..
000410: 90 D4 80 07 FE 7F 00 00 00 35 26 BB 87 A8 5D A6 | d...5&]
000420: 60 D4 80 07 FE 7F 00 00 30 D4 80 07 FE 7F 00 00 | `d..d...
000430: 90 D4 80 07 FE 7F 00 00 BA 6B 67 69 9E 55 00 00 | d...kgiU..
000440: 00 02 00 00 00 00 00 00 4B 01 00 00 48 4C 00 00 | .....K...HL..
000450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000490: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0004F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

000500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000510: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000520: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000530: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000540: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000560: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000580: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000590: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0005F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

student@student-VirtualBox:~/Documents/csc415-filesystem-NoPref$

```

Root(.)

Pointer to (..)

## VCB Structure Description:

Overview: Our VCB structure holds all the meta data for our volume containing data such as the files header , total number of blocks in our volume, the number of free blocks in , size of the blocks, free space management information , and location of our root directory.

Technical: fsVcb.h serves as the interface to the volume control block. In our fsVcb.c file you will find the definition of our VCB structure as well as the implementation of all the functions provided in our fsVcb.h file.

## Free Space Structure Description:

Overview: Our free space implementation uses a linked list and stores information in two locations. One is the VCB where information about the beginning, end, and total block count of the free space is stored to memory and disk. The other is within each block where every block on disk or in memory knows what block it corresponds to in the LBA and the next block in the file or free space system (if there is one).

Technical: fs.Free.h provides the interface to the free space system including the struct definition FSM. fsFree.c stores the functions used in free space implementation. Elements of fsVcb and fsLow are used as well in its operation.

Abstraction: The functions provided by the free space system collectively create an interface that abstracts details of the blocks underneath by accepting and returning block-sized byte buffers.

## Directory System Description:

Overview: Our directory is a file structure that has the ability to hold other directories and files within. In order to maintain and manage these files, we need to populate our Directory with the appropriate variables listed below.

Technical: DirEntry is the structure we’ve implemented to replicate either a file or a directory, the only difference between the two is an unsigned char that acts like a boolean value to differentiate its type. Within its initialization, we’ve created a directory array that initializes each directory entry structure to a free state. It then writes all the blocks to disk which is managed by the Free Space.

## Changes to Design and Why:

Change 1: For our Directory Entry, we changed the name to be size 32 rather than 64.

Reason 1: We realize that we will be working with multiple files, and the average length of our file name will most likely not be around 64 characters long.

Change 2: We created a directory structure, separate from directory entry.



Reason 2: This allowed us to store an array of directory entries within a directory.

## Team Dynamics:

Team Members:	Worked On:
Jeffrey Friedrich	Write-up Structure, Free Space Structure Description, fsFree.c and fsFree.h, some VCB functions/variables concerning free space,
Brian Cheng	Initialized DirEntry, helped with debugging for writing to disk
Michael Davichick	Structured VCB and DirEntry. Helped in the initialization of both structures, troubleshooting and debugging
Ives-Christian “Jay” Jadman	Initialized volume, initialized the VCB, troubleshooting, debugging

## Team Approach:

### (how did we work together, how did we divide up the tasks)

Ives-Christian “Jay” handled the initialization of the volume and the initialization of the volume control block.

Jeff took on the free space system individually and with the rest of the group.

Mike researched and helped implement / structure the Volume Control Block and Directory Entry.

Brian implemented directory initialization and debugged errors related to LBAwrite

## Team Meetings:

Virtual/Discord - Friday, 10/15 @ 1000 - 1215

Virtual/Discord - Monday, 10/18 @ 1500 - 1845

Virtual/Discord- Thursday 10/21 @ 1100 - 1400

Virtual/Discord- Friday 10/22 @ 1200 - 1600

Virtual/Discord- Saturday 10/23 @ 1000 - 1300

Virtual/Discord -Sunday 10/24 @ 1730 - 1900

Virtual/Discord -Monday 10/25 @ 1500 - 2330

## Issues/Resolutions:

Issue1: We were unable to initialize our VCB struct with basic information into the SampleVolume correctly.

Resolution1: After going over the material we were able to correctly get our VCB struct into block 1 without issue and the hexdump reflected clean data as well. This was when we first init our file system we first read to see if the volume control block had been initialized. If not, one will be initialized. The magic numbers from our code and hexdump match, thus proving that our initialization of the VCB is correct.

Issue2: Our Directory Entry info was not able to be written to the correct location.

Resolution2: We found this error in our writeBlocks() function and found that the recentBlock variable in our FSM struct was not giving us the correct value so LBAwrite() was attempting to write to a location that does not exist. To resolve this issue, we had to initialize the remaining attributes in our volume control block, one of which was the pointer to the first free block.

Issue3: Root Directory could not be written to disk; kept getting segmentation faults during compilation

Resolution3: We debugged this issue using printf() statements and found that we failed to initialize the name of the root directory. To solve this we created the object DirEntry de to initialize the name rather than initializing it using a pointer to that struct.

## Design Description:

This is a pre-work part of your file system assignment. In this part provide the rough descriptions of

- What your Volume Control Block looks like (what fields and information it might contain)
- How you will track the free space
- What your directory entry will look like (include a structure definition).
- What metadata do you want to have in the file system

This should be a PDF.

Don't forget to include your team name, and all the team members on the document.

At least the VCB and Directory Entry Structure should be in the form of a C structure definition with details of each field

## Design Submission:

### Volume Control Block:

```
type struct VCB {  
    int totalBlockCount;    // total # of blocks  
    int numFreeBlockCount;  // # of free blocks  
    int blockSize;         // block size  
    int freeFCBCount;       // free FCB count  
    FSM * root;            // pointer to root directory (0 index)  
    FSM * firstFreeSpace;   // free block pointer  
    FSM * lastFreeSpace;    // last free block pointer  
    bFCB * nextFCB;        // free FCB pointer  
}
```

## Free Space Management:

We are using a linked list. All free blocks point to the next free block with the last block pointing to NULL. File blocks point to the next block with the last block pointing to NULL. The location of the first and last free blocks as well as the total number of free blocks will be stored in the VCB. When a block is allocated, it will take from the first free block and so on until the file is allocated and then the location of the first free block will be updated. When a block is freed, the last free block will point to the first block of the new section and the last free block location will be updated. The free space will be implemented in its own separate file (fsFree.c and fsFree.h).

After volume initialized:

0	1	2	3	4	5	6	7	8
root dir first free last free totalfree =7	[]	=>3	=>4	=>5	=>6	=>7	=>8	=> \0
VCB	root dir	first free						last free

After file0 allocated (2 blocks):

0	1	2	3	4	5	6	7	8
root dir first free last free totalfree =5	[file0]	=>3	=> \0	=>5	=>6	=>7	=>8	=> \0
VCB	root dir	file 0		first free				last free

After file0 freed:

0	1	2	3	4	5	6	7	8
root dir first free last free totalfree =7	[]	=>3	=> \0	=>5	=>6	=>7	=>8	=>2
VCB	root dir		last free	first free				

## Directory Entry:

Our directory entry will contain name, location, file size, and the dates modified, created, and accessed. We have decided to use lots of metadata because files will be easier to search by metadata. However, this also makes the directory itself larger and may be more fragmented.

```
type struct DirEntry
{
    char deName[32]        .           // directory name
    int deLocation;         // current location of directory entry
    int deSize;             // size of directory entry
    time_t dateCreated;     // time variable of file creation
    time_t dateLastModified; // time variable of file modification
    time_t dateLastAccessed; // time variable of file access
    unsigned char          isDir;      //determine if file is a directory
}DirEntry;

DirEntry* getDirEntryInfo(char* DENAME);
```

## File Metadata:

```
typedef struct fileInfo {
    DirEntry entry;
    Block fileBlockOffsetArray[16]; //filled with block offsets
}fileInfo;

fileInfo* getFileInfo(char * fileName);

typedef struct bFCB {
    fileInfo * fi;
    char * buf;
    int index;
    Int remain;
    int blockIndex;
    int blockCount;
    int buflen;
} bFCB;
```

## Notes (not part of our submission):

### Volume Control Block -

- A **volume control block**, ( per volume ) a.k.a. the **master file table** in UNIX or the **superblock** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
- It has information about a particular partition ex:- free block count, block size and block pointers etc. In UNIX it is called super block and in NTFS it is stored in the master file table.

### File Control Block -

The **File Control Block, FCB**, ( per file ) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

### Volume Control Block:

Partition Size  
(root directory pointer?)  
(total blocks)  
(number of free blocks)  
(pointers to free blocks?)

### File Structure: (extends Directory Entry)

Attributes:

private size: int  
private content: string

Methods:

getSize(id): int  
deleteFile(id): int (1 or 0)  
// openFile(): int (id)  
// closeFile(id): int (1 or 0)  
// readFile(id, buffer): int (bytes read)  
// seekFile(id, pointer): int (new location/size)  
getContent(id)  
setContent(id)

### Directory Structure: (extends Directory Entry)

Attributes:

private directoryEntries: DirectoryEntry []

Methods:

deleteDirectory(id)

### Directory Entry Structure:

Attributes:

public id: int  
private name: string  
private created: date



```
private lastUpdated: date
private lastAccessed: date
private parent: Directory
private permissions: bitmask
Methods:
  getEntryName(id): string
  changeEntryName(id, string) : int (1 or 0)
  setPermissions(id, flags): int (1 or 0)
  getPATH(id): string - recursively call parent getPATH until root
  deleteDirectoryEntry(id) - recursively call delete
  getLastUpdated(id): date
  getLastAccessed(id): date
  getCreated(id): date
```

#### How is Free Space Stored/How to find Free Space:

Free space: what blocks are free, which are not

- We are using a linked list:
  - When the volume is initialized, the first open block location is stored and the last open block is stored.
  - All free blocks point to the next free block. File blocks point to

Interface Diagram:

Upper API (mfs.h):

File Operations (b\_io): b\_open, b\_read, b\_write, b\_seek, b\_close

Directory Operations(mfs): fs\_mkdir, fs\_rmdir, fs\_opendir, fs\_readdir, fs\_closedir,

fs\_getcwd, fs\_isFile, fs\_isDir, fs\_delete, fs\_stat

Structures: struct fs\_direntinfo, struct fdDir, struct fs\_stat

File System
-------------

Lower (fsLow.h)

LBA functions: LBAread, LBAwrite

Partition functions: startPartitionSystem, closePartitionSystem

## References:

- <https://www.geeksforgeeks.org/design-data-structures-algorithms-memory-file-system/>
- [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/11\\_FileSystemImplementation.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/11_FileSystemImplementation.html)
- [https://yajin.org/os2018fall/14\\_File\\_System\\_Implementation.pdf](https://yajin.org/os2018fall/14_File_System_Implementation.pdf)