

Filesystem Project Group Writeup

[Link To GitHub Repository](#)

By: 4 Threads

Student Name	Student ID Number	GitHub Username
Gregory Arruiza	922218151	ChowMeinFan
Andy Byeon	921120548	AndyByeon
Hajime Miyazaki	922402751	hMiyazaki95
Harrison Rondeau	921087437	harrison-CSC

Table of Contents

1. File System Description	2
1.1 Volume Control Block	2
1.2 Freespace Map	3
1.3 Root Directory	3
1.4 Other Blobs	3
2. Issues Faced	3
3. Functions	5
3.1 pwd	5
3.2 ls	5
3.3 cd	6
3.4 md / mkdir	6
3.5 rm	7
3.6 touch	7
3.7 cat	8
3.8 cp	8
3.9 mv	9
3.10 cp2fs	10
3.11 cp2l	10
3.12 help	11
3.13 history	11
3.14 exit	12
4. Driver Program	12
4.1 Main	12
4.2 Process Command	12
5. Build and First Run Screenshots	13

1. File System Description

The filesystem contains 4 main components:

1.1 Volume Control Block

The VCB will always be contained in the first block in the volume, and will contain: the filesystem's magic, the number of blocks, the block size, the number of entries in the root directory, the starting block of the root directory, and the starting block number of the root directory.

The magic number allows the filesystem to determine whether a volume is formatted, and then will read essential data, such as where the root directory is located. This also allows detection of changed volume parameters such as block size and block count.

1.2 Freespace Map

The freespace map is a bitmap that stores the data on whether each block in the volume is free (1) or taken (0). The size of this will vary based upon run configurations, but by default it will manage 19,531 bits, and will span 5 blocks.

1.3 Root Directory

The root directory is created when the volume is formatted. It will be tracked by the VCB, so that it can be found when loading an already existing, formatted, volume. This directory is also special because it (".") is its own parent (".."). For the purposes of the current working directory, it doesn't have a real name.

1.4 Other Blobs

Every other blob (directory or not) can be located by starting at the root directory and iterating down the path tree.

2. Issues Faced

One issue faced was how to handle the debugging of a program with so many functions. The solution was using a number of preprocessor directives in generalUtils.h. They look like:

```
// Debugging options. 0 = no extra info, 1 = extra info
#define DEBUG_MKDIR      0
#define DEBUG_RM         0
```

```
#define DEBUG_READDIR      0
#define DEBUG_SETCWD      0
#define DEBUG_LS          0
#define DEBUG_DELETE      0
#define DEBUG_CP          0
#define DEBUG_CAT         0

#define DEBUG_PARSEPATH    0
#define DEBUG_EXPANDDIR   0
#define DEBUG_MKFILE       0

#define DEBUG_OPEN         0
#define DEBUG_READ        0
#define DEBUG_WRITE       0
#define DEBUG_WRITECHECK  0
```

This allows specific debugging information specific to any number of components, or none at all to be displayed.

Another issue was how to handle the potential of needing to expand the size (both bytes and block) of a blob that is being written to. This was handled by using a helper function, **writeSizeCheck()**. This function takes in a **b_io_fd** as a parameter, and will update the relevant directory entry to indicate more bytes/work with the freespace manager to allocate more blocks as needed. This is used in both **b_write()** (when a buffer fills up) and in **b_close()** (to write any bytes that still remain in the buffer).

Another issue was how to handle the user pathname/filename input. This has to be able to support both relative paths (**touch myfile**) and absolute paths (**touch /home/student/Desktop/myfile**). It also has to be able to support trailing slashes for directory names (**cd /home/student/Desktop/**), without requiring them (**cd /home/student/Desktop**). To do this, a helper function called **splitBlobPath()** was used. This helper function handles all of this required functionality, and minimizes code reuse, as this function is called in many places throughout the code.

Another issue was how to deal with the **b_open** flag **O_RDONLY**. What makes this somewhat hard to work with is that this macro expands to 0. Therefore to detect whether this flag was set or not, the other two flags that have to do with reading/writing, **O_RDWR** and **O_WRONLY** were checked, and if neither of them were set, then it is implied that the write only flag is not set.

Another issue was how to implement **parsePath()**. We thought of different ideas like returning the directory index or the directory entry itself. The function instead took in a **fdDir** struct and **parsePath()**

used to populate the values of the structure. We would use this function to get the information needed from directory entries and apply it towards ***fs_readdir()***, ***fs_stat()***, ***fs_isFile()***, and ***fs_isDir()*** among others file system functions.

3. Functions

3.1 pwd

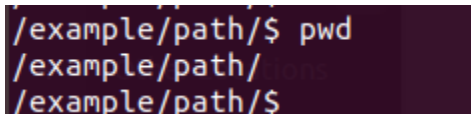
pwd is used to print the current working directory. (*Note: the current working directory is already displayed as the prompt in this file system*)

The syntax is:

pwd

Parameters will not cause an error, but will have no effect on the result.

Screenshot of usage:

A terminal window with a dark background. The prompt is "/example/path/\$". The user enters "pwd" and the output is "/example/path/". The prompt then changes to "/example/path/\$".

```
/example/path/$ pwd
/example/path/
/example/path/$
```

pwd shows the current working directory, which in this case is “/example/path/”

3.2 ls

ls is used to show the files in a directory.

The syntax is:

ls [--all/-a] [--long/-l] [--help/-h] [pathname]

The flags can be:

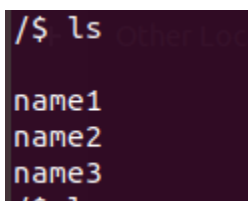
--all/-a show all entries (including “.” and “..”)

--long/-l additionally print out whether an entry is a directory, and its size

--help/-h show the valid syntax options

pathname can be used to specify one or more directories to be searched. If no pathname is provided, the current working directory will be used.

Screenshot of usage:

A terminal window with a dark background. The prompt is "/\$". The user enters "ls" and the output is "name1", "name2", "name3". The prompt then changes to "/\$ ls".

```
/$ ls
name1
name2
name3
/$ ls
```

ls shows that there are 3 non-automatically generated directory entries in the current working directory.

```
/ $ ls -a
.
..
name1
name2
name3
```

Running **ls** with the **-a** flag prints the entries from before in addition to “.” and “..”

3.3 cd

cd is used to change the current working directory.

The syntax is:

cd *pathname*

Pathname can be either a relative path (such as “folder”) or an absolute directory (such as “/new/dir/”). Trailing forward slashes are optional.

Screenshot of usage:

```
/ $ pwd
/
/ $ cd /example/path/
/example/path/ $ pwd
/example/path/
/example/path/ $
```

pwd shows that the current working directory before is root. **cd** then changed this to “/example/path/”, which is confirmed by running **pwd** once more

3.4 md / mkdir

md and **mkdir** (same functionality) are used to create a new directory.

The syntax is:

md *pathname*

mkdir *pathname*

Pathname can be either a relative path (such as “folder”) or an absolute directory (such as “/new/dir/”). Trailing forward slashes are optional.

Screenshot of usage:

```
/$ ls

/$ mkdir exampleName
/$ ls

exampleName
/$
```

First, **ls** is used to show that there are no entries (other than “.” and “..”) in this directory. Then, **md / mkdir** is used to create a new directory named “exampleName”. Then **ls** is used again to show that a directory of that name now exists.

3.5 rm

rm is used to remove a directory.

The syntax is:

rm pathname

Pathname can be either a relative path (such as “folder”) or an absolute directory (such as “/new/dir/”). **rm** only allows removing a directory with only the “.” and “..” children. To remove a directory with more children, those children must be removed first. Trailing forward slashes are optional.

Screenshot of usage:

```
/$ ls

rm_Me
/$ rm rm_Me
/$ ls

/$
```

First, **ls** is used to show that there is a directory name “rm_Me”. Next, **rm** is used to remove that directory. Finally, **ls** is run again to confirm that the directory was removed.

3.6 touch

touch is used to create a non-directory blob.

The syntax is:

touch filename

Filename can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

Screenshot of usage:

```
/ $ ls
/ $ touch example
/ $ ls
example
/ $
```

First, **ls** is used to show that there is no “example” inside of the directory. Next, **touch** is used to create that file. Finally, **ls** is run again to confirm that the file was removed.

3.7 cat

cat is used to print out the contents of a file to the console.

The syntax is:

cat filename

Filename can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

Screenshot of usage:

```
/ $ cat example
Hello World!
/ $
```

cat prints out the contents of the file named “example”, which happen to be the string “Hello World!”

3.8 cp

cp is used to copy a file from one location to another

The syntax is:

cp srcfile [destfile]

srcfile can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

destfile is optional, and is where the file should be copied to. This can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

Screenshot of usage:


```

/$ ls a
example
/$ ls b

/$ cp a/example b/example
/$ ls a
example
/$ ls b
example
/$

```

The usage of **ls** shows that the directory named “a” has a file named “example”, while there is no such file in directory “b”. **cp** then copies the file from directory “a” to directory “b”. In this case, the name “example” was kept, although this is not necessary. **ls** is then used again to show that the file now exists in directory “b”, while still existing in directory “a”

3.9 mv

mv is used to move a file from one location to another

The syntax is:

mv srcfile destfile

srcfile can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

destfile is optional, and is where the file should be moved to. This can be either a relative path (such as “myfile”) or an absolute directory (such as “/new/myfile/”). Trailing forward slashes are optional.

Screenshot of usage:

```

/$ ls a
mv_example
/$ ls b

/$ mv a/mv_example b/mv_example
/$ ls a
mv_example
/$ ls b
mv_example
/$

```

The usage of **ls** shows that the directory named “a” has a file named “mv_example”, while there is no such file in directory “b”. **mv** then moves the file from directory “a” to directory “b”. In this case, the name “mv_example” was kept, although this is not necessary. **ls** is then used again to show that the

file now exists in directory “b”. The original file in “a” no longer exists

3.10 cp2fs

cp2fs is used to copy a file from the linux filesystem into this project’s filesystem

The syntax is:

cp2fs Linuxsrcfile [destfile]

Linuxsrcfile is the path to the file in the linux file system

destfile is the name of the file in this project’s filesystem where the contents of the linux file should be copied to

Screenshot of usage:

```
/S
/$ cp2fs /home/student/Desktop/csc415-filesystem-harrison-CSC/Makefile mkfl
/$
```

cp2fs is used to copy a file in the linux file system (the makefile) to a file in this project’s file system.

ls is used to confirm that this file now exists in the project filesystem.

3.11 cp2l

cp2l is used to copy a file from this project’s filesystem to linux’s filesystem

The syntax is:

cp2l srcfile [Linuxdestfile]

srcfile is the path to the file in this project’s filesystem

Linuxdestfile is the name of the file in the linux filesystem where the contents of the source file should be copied to

Screenshot of usage:

```
/S
/$ cp2l mkfl /home/student/Desktop/newmakefile
/$
```

cp2l is used to copy a file in this project’s filesystem to the linux filesystem.

```
student@student-VirtualBox:~/Desktop$ ls -l
total 24
drwx----- 3 student student 4096 Feb  6 09:44 csc415-assignment-1-command-line-harrison-CSC
drwx----- 4 student student 4096 Feb  7 17:04 csc415-assignment2-bufferandstruct-harrison-CSC
drwx----- 3 student student 4096 Feb 27 10:06 csc415-assignment3-simpleshell-harrison-CSC
drwx----- 4 student student 4096 Mar  9 13:07 csc415-assignment-4-word-blast-harrison-CSC
drwx----- 5 student student 4096 Apr 26 10:13 csc415-filesystem-harrison-CSC
-rw-rw-r-- 1 student student 2462 Apr 26 10:14 Makefile_copy
student@student-VirtualBox:~/Desktop$
```

ls in the linux file system is used to show that **cp2l** was successful

3.12 help

help is used to print out all possible commands

The syntax is:

help

Screenshot of usage:

```
/$ 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
mkdir   Make a new directory (alt syntax)
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
/$
```

help here is showing all possible commands (with the exception of **exit**, which is handled differently internally)

3.13 history

history is used to print out the log of what commands the user has entered in this session

The syntax is:

history

Screenshot of usage:

```
/exampledir/$
/exampledir/$ history
ls
mkdir exampledir
ls
cd exampledir
touch examplefile
ls
history
/exampledir/$
```

help prints out all the commands that have been previously entered. Note that can vary, and will almost always give a different output

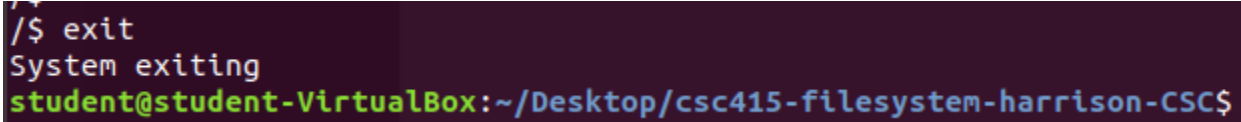
3.14 exit

exit is used to quit the file system and return to the main terminal

The syntax is:

exit

Screenshot of usage:



```
/$ exit
System exiting
student@student-VirtualBox:~/Desktop/csc415-filesystem-harrison-CSC$
```

exit gracefully closes the programs (saves everything properly) and then returns main terminal

4. Driver Program

Our driver program is **fshell.c**. It is responsible for calling for partitioning, initializing, and exiting our file system. It also handles any input and output between our user with file system commands stated earlier; **ls**, **cp**, **mv**, **md**, etc.

4.1 Main

Main contains all of our calls to build our filesystem. It firstly takes in 3 arguments and passes it to **startPartitionSystem()** to partition our file system. Volume size and block size are sent to **initFileSystem()** to create our volume control block, free space map, and root directory. We then set our current working directory to our root and start **using_history()**. After an initialization without eros, we prompt our user for input. If the command is not exit, not empty, and greater than 0, it is sent to **processcommand()**. We then wait for our user to input exit to **exitFileSystem()** and **closePartitionSystem()**.

4.2 Process Command

Process command is a helper function in charge of splitting valid user input and determining if it is a recognized command or not. User input is first split into separate arguments. We use a switch statement to place arguments into a command vector **char **cmdv**. After parsing our input, we compare the first argument of our command vector to our **dispatchtable[]**. If we find a match we call on the specified function. If there is no match we tell our user that the command isn't recognised.

5. Build and First Run Screenshots

Making the program:

```
student@student-VirtualBox:~/Desktop/csc415-filessystem-harrison-CSC$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o generalUtils.o generalUtils.c -g -I.
gcc -c -o fs-functions/fs_getcwd.o fs-functions/fs_getcwd.c -g -I.
gcc -c -o fs-functions/fs_setcwd.o fs-functions/fs_setcwd.c -g -I.
gcc -c -o fs-functions/fs_opendir.o fs-functions/fs_opendir.c -g -I.
gcc -c -o fs-functions/fs_readdir.o fs-functions/fs_readdir.c -g -I.
gcc -c -o fs-functions/fs_closedir.o fs-functions/fs_closedir.c -g -I.
gcc -c -o fs-functions/fs_isDir.o fs-functions/fs_isDir.c -g -I.
gcc -c -o fs-functions/fs_stat.o fs-functions/fs_stat.c -g -I.
gcc -c -o fs-functions/fs_mkdir.o fs-functions/fs_mkdir.c -g -I.
gcc -c -o fs-functions/fs_isFile.o fs-functions/fs_isFile.c -g -I.
gcc -c -o fs-functions/fs_rmdir.o fs-functions/fs_rmdir.c -g -I.
gcc -c -o fs-functions/fs_delete.o fs-functions/fs_delete.c -g -I.
gcc -c -o fsMapManager.o fsMapManager.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -c -o fs-functions/fs_make_nondir.o fs-functions/fs_make_nondir.c -g -I.
gcc -o fsshell fsshell.o fsInit.o generalUtils.o fs-functions/fs_getcwd.o fs-functions/fs_setcwd.o fs-functions/fs_opendir.o fs-functions/fs_readdir.o fs-functions/fs_closedir.o fs-functions/fs_isDir.o fs-functions/fs_stat.o fs-functions/fs_mkdir.o fs-functions/fs_isFile.o fs-functions/fs_rmdir.o fs-functions/fs_delete.o fsMapManager.o b_io.o fs-functions/fs_make_nondir.o fsLow.o -g -I. -lm -lreadline -l pthread
student@student-VirtualBox:~/Desktop/csc415-filessystem-harrison-CSC$
```

Running for the first time:

```
student@student-VirtualBox:~/Desktop/csc415-filessystem-harrison-CSC$ 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
Formatting volume...
Volume formatted
/$
```