Feras Alayoub

Jennifer Finaldi

Lilian Gouzeot

Lauren Wong

"DreamTeam"

CSC415.02

FileSystem Project Report

Github Link:

https://github.com/CSC415-Fall2020/group-term-assignment-file-system-LilianGouzeot/

Introduction to DreamTeam FileSystem

The DreamTeam filesystem is a limited demonstration of some of the functionality of a hierarchical filesystem, similar to the Linux File System. The basic core concept behind this file system is that the directory model interfaces with either our directory services interface, or our file services interface through an open file table system. These interfaces talk with the shell and the directory entry model to pass data to and between them. In order to implement this, some design choices were made. A directory entry system exists as an array structure, while the hierarchy is implemented similarly to a linked list structure. Free space is organized and managed using a bitmap to represent logical blocks that are used with a 1, and free blocks with a 0. Memory is allocated in contiguous blocks, with file memory allocation being allocated using the concept of extents as well.

Directory Entry System and Layout:

The directory system itself is composed of a series of arrays of directory entry structs. The struct definition consists of all of the data variables that can uniquely identify and describe a corresponding file or directory that it pertains to. These directories are organized in a hierarchical tree, with a max of 50 nodes per directory. In essence, this means that a directory

can only store up to 48 other directories or files or combination thereof. The most important members of the directory entry consist of the name of the directory or file that the entry points to, it's location in the parent directory, and the location of the directory/file it points to. With these three pieces of information, it allows the directory services to navigate the tree using a method very similar to linked-list operations.

Upon a directory creation, first an empty directory entry is found within the parent directory. It is then assigned any relevant values at the time, then an array of 50 directory entries is initialized into a location the free-space provides, with the second entry (. .) pointing back to the parent directory. The parent directory also holds the location to the LBA block where its new directory is actually stored.

Free Space Structure

The free space is a simple “bitmap” of booleans (so it is not actually a bitmap, sort of a pseudo bitmap.) Entries are set to 0 if they are free and 1 if they are being used. There are two main functions, one that takes in the number of desired free blocks and then returns an LBA which has many free blocks after it as needed. It also sets all of the blocks to “used” . The other takes in a starting LBA and the number of blocks to free and flips them all back to 0.

Formatting the Drive

Master Boot Record

The goal of the Master Boot Record (MBR) is to store some infomation that is mandatory for the file system to start somewhere at a fixed place, like the place of the root directory, the size of the volume, and some other information. Inside our MBR we store the location of the root directory, the location of the freespace bitmap, the name of our filesystem, the size of the volume, the size in bytes of one block, and the number of blocks inside the FileSystem.
We also store our magic number. We use FILE for a file and DIR for a directory.
The MBR is created the first time we format the volume. This MBR is in the first block of the volume. We use the MBR to see if the volume is already formatted or not, if the MBR is present

at block 0 then we know that the volume is formatted. But if at block 0 we do not found the MBR we know that the volume is not formatted.

Initialize Root Directory

Initializing the root directory is the same process and function as initializing any ordinary directory, save for the special case that its dot and dot-dot entries are calibrated differently. Ordinarily the dot entry (the first entry) points to its start location in memory, and the dot-dot entry points to its parent directory. With the root directory, however, it does not have a parent so the dot-dot and dot both point to the same location in memory. This means that when the "cd .." command is used in the shell, it would behave exactly the same as "cd .", provided the current working directory is root.

Initialize the Freespace

The free space is initialized by creating an array of booleans all initialized to 0 other than the blocks corresponding to MBR and space being used for the current bitmap which are initialized to 1. The bitmap is stored on disc, in the block immediately following the MBR, being read and written to whenever needed.

Directory Services Interface (makeremove.c)

The key to the functionality of many of the directory services routines involve being able to parse a given path, relative or absolute. Paths are navigated by splitting them into tokens and then searching each directory for the matching token by name until it is either not found or the tokens run out. The path is always searched starting from the root by appending relative paths to the current working directory. The behavior for navigating directories is slightly different depending on the last part of the path. Sometimes we look at the second to last directory, sometimes the last that is found, and sometimes the directory is not found at all. This is okay, however, if we are possibly expecting a new name like in move or make new directory. Remove, for example, will only want to look at the second to last directory since it needs to remove from that directory and not go directly into the directory that it is removing. For setting the current

working directory, “.” and “..” are special cases that get filtered out (along with any directories .. correspond to) of the path when it is being set.

Making a directory first validates that the path is valid, which also includes error handling if a user tries to create a directory with a duplicate name to another directory inside the same parent directory that the user is trying to create in. After this, space is allocated for the directory and it is initialized with relevant info, and default values for all the rest. Deleting a directory operates recursively, therefore, when a directory is deleted it subsequently removes all of its subdirectories. The user is prevented from deleting the current working directory or any parent, grandparent, etc. directories relative to the cwd. Removing a file also happens inside the same function as for removing directories, with the only difference being a check to see if what we are deleting is a file or a directory. If it is a file, we make sure to call a helper routine that will release all of the extents memory back to the freespace before resetting its directory entry. It is important to note that in both directory and file deletion, the directory entries they once occupied don't actually get deleted, they just have their values reset to initial values and set to unused, so that they may be recycled again in the future.

File Management Interface (readwrite.c)

The file management interface is responsible for handling any and all operations directly related to files and how they are handled in our filesystem. It makes use of a structure that stores relevant information about an open file, including copies of directory entry variables. This struct gets altered before, during, and after a file is opened and closed. These open files are referred to as "fd," and every fd that is allocated has its own pointer variable to one of these structures. All of these open fds are stored in an abstracted array-like fashion, using the fd integer index number to store its "element," which is just a method of identification, as the open file table isn't actually a data structure but a collection of pointer variables stored in different places in memory throughout program execution.

This file contains the "b functions," as we call them. These are functions designed to be similar to the read(), write(), open(), close(), and seek() system calls in Linux, with the main difference being that our b_read(), b_write(), etc functions are designed specifically to interface with our filesystem instead. This means that our functions need to talk to both our directory entry system, as well as our logical block array. Open will return a file descriptor just like the Linux version, creating a struct object for a particular file and filling in information about it (or creating a new entry and subsequently filling in a file descriptor struct at the same time). Close releases a file descriptor struct as well as performs a last write of the struct's write buffer, had the file been opened for write and written to. Our write function behaves by reading the caller's buffer provided in the arguments and copying the memory to the file descriptor struct's buffer, before doing an LBA write call, feeding it the newly full file descriptor buffer to write. Its important to note that write only calls LBAwrite() when it has a full buffer to output to our disk space (volume). The read function operates similarly to write, only we are reading from our logical block array, storing it in our file descriptor struct's buffer, then copying information from that buffer to the caller's buffer. Read calls do not leave anything left in the file descriptor buffer to be used upon execution of close. Our seek function takes in parameters for a file descriptor identifier, an offset, and a whence parameter that can store flags for SEEK_SET, SEEK_CUR, and SEEK_END. Inside this function is a switch case that positions the file pointer of a particular open file to a position determined by the parameters. It can set the file pointer to offset bytes from the start of the file, to the file pointers current location plus offset, or set to the end of the file.

Extents/Memory Allocation methods

It was decided to use a dynamic method of allocating memory for a file, using the concept of extents. For our extents we decided to use an array of key value pairs that represent a starting block of an extent, with the adjacent element representing the number of blocks

allocated at the previous element's address. This array is stored inside one 512 byte LBA block, using unsigned long variables for all of the elements. This means that the whole block can store a total of 64 8-byte elements, or 32 key/value pairs to represent a max of 32 extents. This array block is pointed to by a variable stored inside the directory entry struct. Initially, this LBA block of extents does not get initialized until the first time a b_write is attempted. This is to prevent wasted allocation if a user were to open a file for write, allocate extents upon opening, and then close without ever actually writing anything. Therefore, when a user first attempts to write, it calls a helper function that translates a logical index of a single extent LBA block into the actual address of that block. Upon finding out that no extents have been allocated, the directory functions rectify that by first initializing the LBA block array, then assigning it the very first value of 20 blocks. After this, it returns the first address of those 20 blocks to b_write, so that it can begin utilizing this allocated memory for a file. Once the first 20 blocks are used up, another extent composed of 40 blocks are added to the array, so each time a new extent gets added, it gets twice the amount of blocks that were allocated to the previous extent. Because our volume size is capped at 10MB, we don't even come close to using all of our array of extents.

When a file is deleted, an extents helper function goes through every extent in the array, freeing the memory and giving it back to the free space, before setting every value in that array to zero, then giving that single LBA block back to free memory, and then committing all the changes to disk.

Defragmentation Stretch Goal

The defragmentation currently runs after every single remove or delete, which isn't very efficient but is sufficient enough for such a small file system. It is being kept this way mostly to show its functionality. The function finds the start and size of a "hole" in the free space and then goes through all the directories and file extents subtracting the size of the hole from its LBA location if the initial value is bigger than the start of the hole. Everything written on the disc is then shifted to the "left" to fill the hole. This runs recursively until no more holes are found.

Please note that if you would like to see the LBA block number of the current working directory when working with defrag, uncomment line 64 of main.c.

Issues That Presented a Challenge

If every single issue that presented a challenge were stated in this document, it would probably end up being a more detailed description of this entire filesystem. To keep things shorter, a list of issues we ran into, were aware of, tried to fix completely, but still remain bugged will be discussed. Before that happens, however, the single biggest challenge of this project was figuring out exactly what everything should do, what major components or phases are required, and how to implement each feature. It was stated by the client that the starter code did not need to be understood fully in order to complete the assignment, however, by the end of it most of the code had been dissected an analyzed enough to have a good understanding of how most of it works.

Although the shell executes basic commands from root as the current working directory (cwd), including making/removing directories, cd'ing into them, copying a file to and from linux, copying a file from one place to another, and importing/exporting large datasets (text and binary), there are some bugs that can occur. One bug, for example, can happen if a directory is created, and cwd is changed to the new directory followed by a file import, the size of the file as well as the amount of directories created inside the filesystem affect the chances of this failing. The safest file to use tends to be the smaller one, with DecOfInd.txt (Declaration of Independence) at 8120bytes tends to me much more likely to succeed when moved around the filesystem or inserted deep into the directory tree. War and Peace at 3.4MB, however, tends to crash the system a lot easier, as well as any jpeg image of the same size. The reason for this could be the fact that not all features of extent management were able to be debugged. Two functions for the extents were built but not optimized and fully debugged. One of these functions would return any wasted extent blocks after a file closes, and one of them would merge adjacent extents together as one. Had the wasted extent block return function worked, and been

utilized, that may have solved this problem. Additionally, the defragmentation function that was built in the freespace may actually have been able to consolidate used space in such a way that the file "blobs" would have more space to work with, allowing larger files to be added more reliably for cases where some directories/files were added to the system before adding something of larger scale.

A bug occurs when one of these larger files (or even smaller, depending on how many were added prior) fail to be imported correctly, causing a program crash. After the crash, booting up the program again and doing an ls -l command in the destination directory does show this file there, but with size zero. This is because there wasn't enough time to implement error checking in our file creation function that would nuke a directory entry before the program crashes. Additionally a file blob could actually be written to before the directory entry is created for it, but that would have involved a lot more overhead to modify the program so that the extent behavior would not depend exclusively on having a directory entry already in place for it. The last part of this bug is that if the user sees this corrupt file still in the directory and tries to remove it, the removal will sometimes fail, which makes sense because removal is dependant on having an uncorrupted file or directory passed to its function. Additional error checking/correction could possibly prevent this, but the fact remains that we would still not be able to remove the file.

A similar bug can occur sometimes when a user copies to linux successfully, then tries to remove the file from the directory. It can cause a program crash, and upon rebooting the program will work the second time the rm is attempted. It is expected that this is a bug having to do with the fs_open or fs_close functions, and with more time, could be explored more deeply.

There is a "feature," where if a user tries to move a file into another directory and uses a path with a trailing slash, the program will output errors as if the the move failed, however if the user goes into the directory and does an ls -l they will see that the file was indeed moved to the

correct location, though the ls output will look a bit weird. This, of course, does not happen if the user leaves out a trailing slash in the destination path.

A small bug that may not affect the workings of the filesystem, but would reduce user experience has to do with the shell prompt. Our shell prompt was modified with a little pizazz to output a generic username along with the absolute current working directory path, so that the user wouldn't have to constantly use pwd to see where they are. Most of the time this outputs without a flaw, but on rare occasions having to do with some kind of warning or error, the prompt will only output the very last character in that prompt message, which looks like "$ ". This might be from an error in the change current working directory function, or the make directory function, since it did happen in the case where the user tries to create 49 directories. Although this doesn't really change the program, it could be very annoying for the user.

As far as overall limitations go, this filesystem tends to have a lot of them. One limitation is that only 48 directories/files can ever exist within a single directory, even root. Once those 48 are all used up, no more can be created from inside that particular directory. Another limitation is, of course, the size of the volume. A ten megabyte limit for this volume does greatly reduce the number of 5 megabyte photos we can store in it, which can seem problematic since most modern iphones take photos of that size. This may be a pretty decent filesystem for storing a large amount of tiny app icons, or exclusively text files, provided it was fully debugged and fully functional.

How Driver Program Works

The driver program named fsshell.c is responsible for acting as a basic shell program that gets a command from the user, processes it, and then executes the function that pertains to that particular command.

Display Files:

The routine displayFiles is a helper function for the cmd_ls routine that takes a fd_struct object and processes it to find out information about a particular directory in order to output to the user. This is the function where the actual output for ls happens. Depending on whether the ls has a second parameter or not determines how the output will look like. This function takes the dot and dot-dot entries into consideration and omits them when outputting information to the console.

List:

The cmd_ls routine stands for "list," in the context of listing information about a file or directory. It parses all of the arguments the user put in, and sets the appropriate flags for them (such as -l or -a). It then determines what we are trying to list, based on the path the user gives, such as another directory, or a file, and determines the proper course of action. If it is a directory, the directory is opened, using the directory services routine, then the displayFiles helper routine is then called to actually output the information. If the user does not specify a particular path for the ls to execute, it will simply use the current working directory and call displayFiles.

Copy:

The cmd_cp function deals with copying data from one place in the filesystem to another. It will take data from one file and write its contents to the destination file. Should a destination file not be specified, an error message will be output to the user, telling them to provide a destination. It opens up both files, the source for read-only, the destination for write-only, creat, or trunc. So far our program only utilizes the trunc functionality, meaning that if

data is copied from one file, it completely overwrites the other file starting at the beginning. One way to extend functionality of this filesystem would be to allow it to append to the end of an existing file. There is a loop in this function that feeds a buffer into the b_read function that interfaces with our filesystem. It returns a number of bytes read, according to the size that is fed in the variable BUFFERLEN. After the number of bytes is returned it then calls the b_write function that also interfaces with our filesystem to commit these changes to disk. Once the loop finishes both files are closed before returning.

Move:

The move file command has a method for moving a file or directory from one location to another. The user has to specify both a source path as well as a destination path, or this function will output an error to the console and end. Upon valid input, this function calls a helper function located in makeremove.c (directory services file). This function parses each path and creates temporary directories for each, copying information into them. It handles error checking if a user is trying to move a directory or file into a file, which shouldn't ever be possible. It also checks to see if you are trying to move something into a directory that already has an entry with that same name. It then takes steps to copy the information into the new directory entry, thus changing it's location.

Make Directory:

The cmd_md function inside fsshell is a short function that simply validates the user input and then calls a function in the directory services to make a new directory. To see how our filesystem creates new directories, please see the previous section having to do with the directory services interface.

Remove:

The cmd_rm function is responsible for validating user input, and calling one of two functions inside the directory services. One of these will remove a directory and one of them will delete a file. This function first determines based on the path which we are dealing with by

calling the directory services functions that return boolean values for if the path points to a file or a directory. If these booleans both return false, then an error is output to signal to the user that the path points to neither a file nor a directory.

Copy To Linux:

The cmd_cp2l function is responsible for copying a file from our filesystem to the linux filesystem. It does error checking and parsing of the command to obtain arguments, and then begins by opening the source file in our filesystem, opening or creating a destination file in the working folder in Linux, and then performing a loop that both reads from our filesystem and writes to linux, by a length of bytes of a buffer determined in fsshell. A buffer is filled with information from our filesystem's b_read function, and then that buffer is fed into the linux write call in order to copy it all over to the destination. The loop repeats as long as a full buffer is read from our filesystem. The very last b_read call will output a number fewer than the size of the user's buffer, allowing the loop to break. Afterwards both files are closed and the function ends.

Copy to (our) File System:

The cmd_cp2fs method operates almost the same way as the previous, only we are copying from an existing file in Linux to our filesystem. In essence it is importing a file into our filesystem. There is basic error handling for user input. Then the method attempts to open the file in Linux. The source code failed to provide error handling for this, so extra code was added so that if the open() linux system call returns a -1 (file not found), the program will not continue to run, resulting in a crash, but will rather output an error message and return an error code from the function without doing anything else. Upon successful opening of the file, a file is created and opened in our filesystem, and a loop just like the one in cp2l continuously operates, with the difference that we are reading from linux and then writing to our filesystem. When the loop breaks from writing fewer than the buffer's length, the file is closed in our filesystem which subsequently writes out any remaining data that exists in our open file table object's buffer.

Change Directory:

The cmd_cd function first verifies that two arguments are provided; the cd command itself and a destination. Should that not be provided the function outputs an error and returns. Otherwise, it looks at the path and removes any quotation marks that may enclose it, then calls a directory services function to set the current working directory to the new path. That function then parses the path and searches all the directories until the destination is found, before updating the current working directory's struct to contain the new values.

Print Working Directory:

The cmd_pwd function in this program has been rendered mostly obsolete after adding functionality to output the current working directory as the default prompt to the user. That said, this function still exists and operates by allocating a string and storing the result of the directory services function that returns the current working directory as a string. If a string has been returned that isn't empty, it is then output to the console for the user.

History:

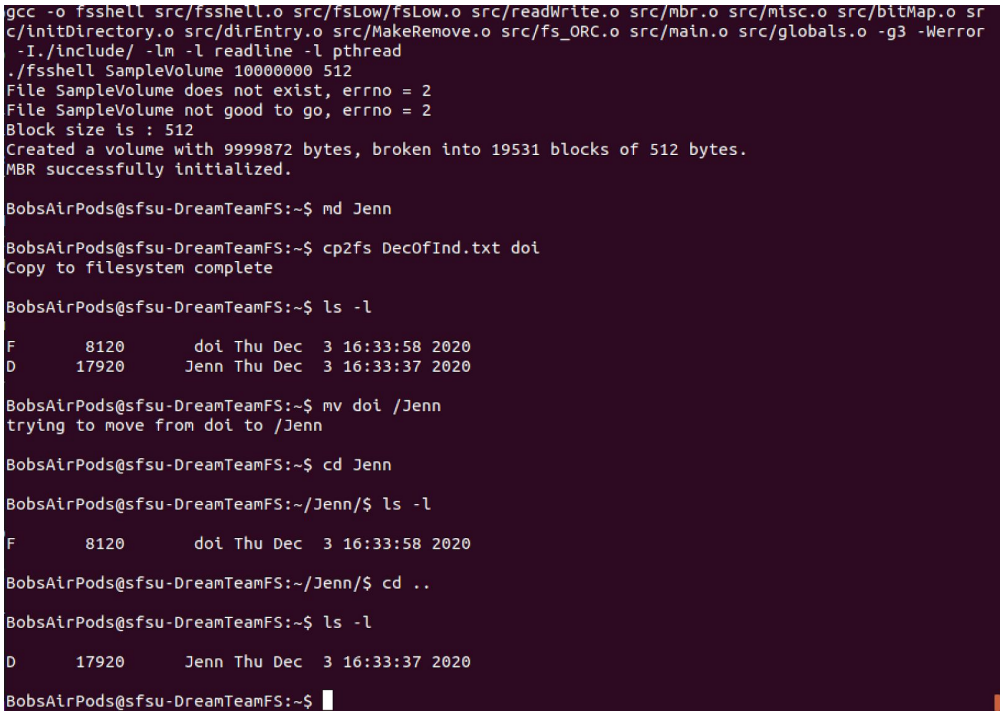
History is responsible for pulling a cached list of previous commands issued by the user and outputting them in a count-controlled loop to the user.

Help:

This function gets called if the user types "help" into the console. Once it does, a dispatch table of commands is output one at a time to the user so they can understand more about the commands of the shell and what features are available to them.

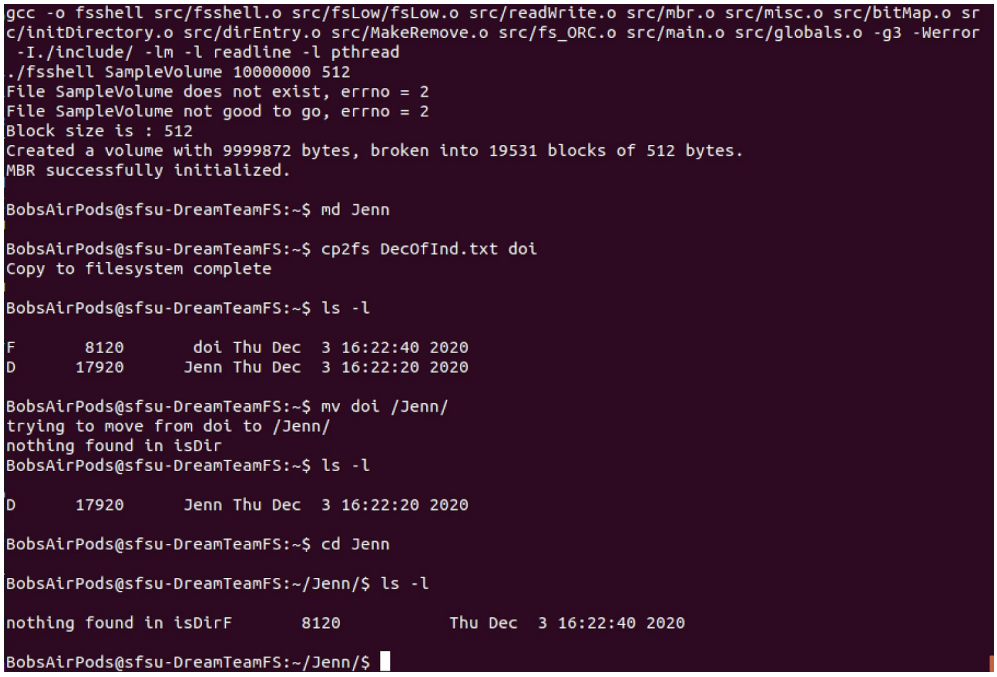
ScreenShots of Output

Example of move working with trailing slash left out:



```
gcc -o fsshell src/fsshell.o src/fsLow/fsLow.o src/readWrite.o src/mbr.o src/misc.o src/bitMap.o sr
c/initDirectory.o src/dirEntry.o src/MakeRemove.o src/fs_ORC.o src/main.o src/globals.o -g3 -Werror
 -I./include/ -lm -l readline -l pthread
./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.
MBR successfully initialized.

BobsAirPods@sfsu-DreamTeamFS:~$ md Jenn

BobsAirPods@sfsu-DreamTeamFS:~$ cp2fs DecOfInd.txt doi
Copy to filesystem complete

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

F       8120       doi Thu Dec  3 16:33:58 2020
D      17920      Jenn Thu Dec  3 16:33:37 2020

BobsAirPods@sfsu-DreamTeamFS:~$ mv doi /Jenn
trying to move from doi to /Jenn

BobsAirPods@sfsu-DreamTeamFS:~$ cd Jenn

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ ls -l

F       8120       doi Thu Dec  3 16:33:58 2020

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ cd ..

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920      Jenn Thu Dec  3 16:33:37 2020

BobsAirPods@sfsu-DreamTeamFS:~$
```

Example of move with trailing slash:



```
gcc -o fsshell src/fsshell.o src/fsLow/fsLow.o src/readWrite.o src/mbr.o src/misc.o src/bitMap.o sr
c/initDirectory.o src/dirEntry.o src/MakeRemove.o src/fs_ORC.o src/main.o src/globals.o -g3 -Werror
 -I./include/ -lm -l readline -l pthread
./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.
MBR successfully initialized.

BobsAirPods@sfsu-DreamTeamFS:~$ md Jenn

BobsAirPods@sfsu-DreamTeamFS:~$ cp2fs DecOfInd.txt doi
Copy to filesystem complete

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

F       8120       doi Thu Dec  3 16:22:40 2020
D      17920      Jenn Thu Dec  3 16:22:20 2020

BobsAirPods@sfsu-DreamTeamFS:~$ mv doi /Jenn/
trying to move from doi to /Jenn/
nothing found in isDir
BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920      Jenn Thu Dec  3 16:22:20 2020

BobsAirPods@sfsu-DreamTeamFS:~$ cd Jenn

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ ls -l

nothing found in isDirF       8120           Thu Dec  3 16:22:40 2020

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$
```

Example of a 5MB photo successfully importing/exporting, with persistence:

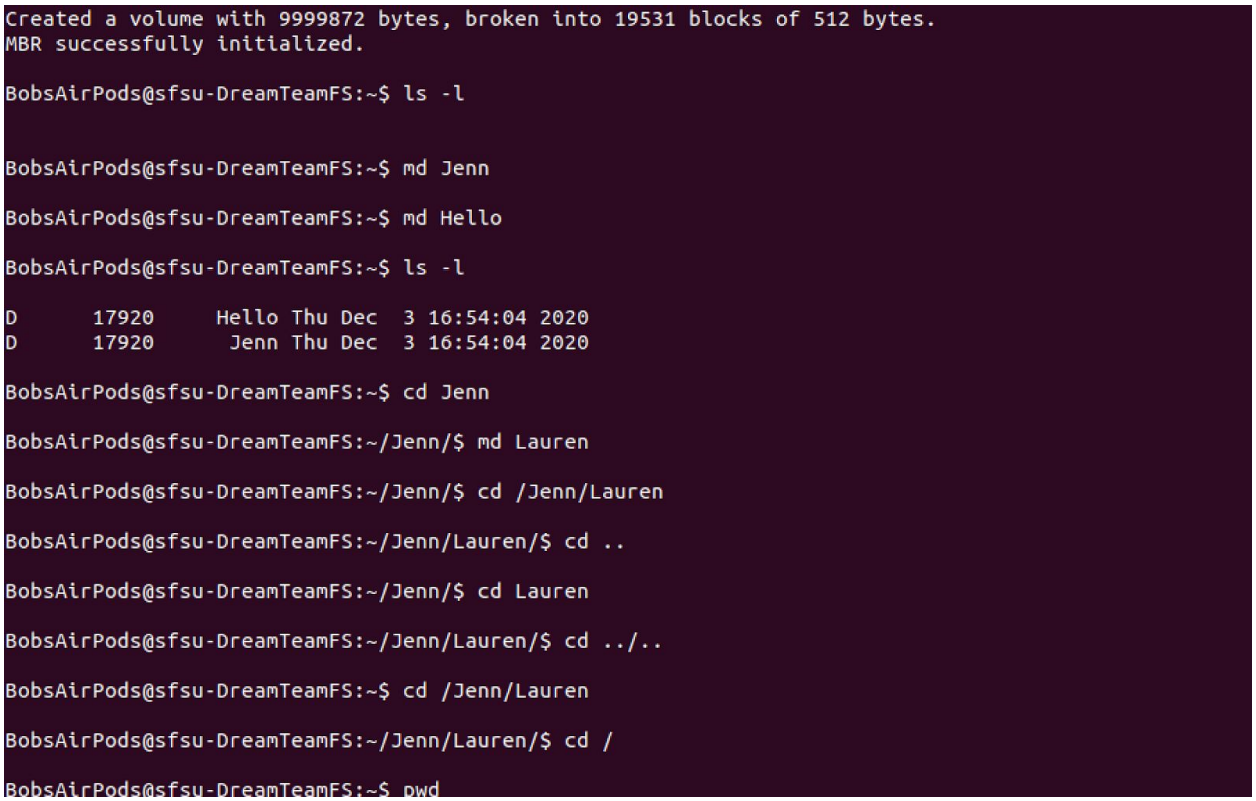
```
student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assignment-file-system-LilianGouzeot$ make run
gcc -c -o src/fsshell.o src/fsshell.c -g3 -Werror -I./include/
gcc -c -o src/fsLow/fsLow.o src/fsLow/fsLow.c -g3 -Werror -I./include/
gcc -c -o src/readWrite.o src/readWrite.c -g3 -Werror -I./include/
gcc -c -o src/mbr.o src/mbr.c -g3 -Werror -I./include/
gcc -c -o src/misc.o src/misc.c -g3 -Werror -I./include/
gcc -c -o src/bitMap.o src/bitMap.c -g3 -Werror -I./include/
gcc -c -o src/initDirectory.o src/initDirectory.c -g3 -Werror -I./include/
gcc -c -o src/dirEntry.o src/dirEntry.c -g3 -Werror -I./include/
gcc -c -o src/MakeRemove.o src/MakeRemove.c -g3 -Werror -I./include/
gcc -c -o src/fs_ORC.o src/fs_ORC.c -g3 -Werror -I./include/
gcc -c -o src/main.o src/main.c -g3 -Werror -I./include/
gcc -c -o src/globals.o src/globals.c -g3 -Werror -I./include/
gcc -o fsshell src/fsshell.o src/fsLow/fsLow.o src/readWrite.o src/mbr.o src/misc.o src/bitMap.o src/initDirectory.o src/
.o src/fs_ORC.o src/main.o src/globals.o -g3 -Werror -I./include/ -lm -l readline -l pthread
./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.
MBR successfully initialized.

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l


BobsAirPods@sfsu-DreamTeamFS:~$ cp2fs 5megs.jpg 5m
Copy to filesystem complete

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

F    5048537        5m Fri Dec  4 11:02:04 2020

BobsAirPods@sfsu-DreamTeamFS:~$ md Jenn

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

F    5048537        5m Fri Dec  4 11:02:04 2020
D      17920      Jenn Fri Dec  4 11:01:41 2020

BobsAirPods@sfsu-DreamTeamFS:~$ mv 5m Jenn
trying to move from 5m to Jenn
```

```
trying to move from 5m to Jenn

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920      Jenn Fri Dec  4 11:01:41 2020

BobsAirPods@sfsu-DreamTeamFS:~$ cd Jenn

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ ls -l

F    5048537        5m Fri Dec  4 11:02:04 2020

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ exit
student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assignment-file-system-LilianGouzeot$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
The Disk Is Already Format to a DreamTeamFS (The best file system of all time) ಠ_ಠ

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920      Jenn Fri Dec  4 11:01:41 2020

BobsAirPods@sfsu-DreamTeamFS:~$ cd Jenn

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ ls -l

F    5048537        5m Fri Dec  4 11:02:04 2020

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ cp2l 5m 5m2.jpg
Copy to Linux complete

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ rm 5m
Removing file...
Defrag initiated...
Defragmentation complete...

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/$ ls -l


student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assignment-file-system-LilianGouzeot$
```

What happens when a user tries to make 49 entries:

```
BobsAirPods@sfsu-DreamTeamFS:~$ md fourtyfour

BobsAirPods@sfsu-DreamTeamFS:~$ md fourtyfive

BobsAirPods@sfsu-DreamTeamFS:~$ md fourtysix

BobsAirPods@sfsu-DreamTeamFS:~$ md fourtyseven

BobsAirPods@sfsu-DreamTeamFS:~$ md fourtyeight

BobsAirPods@sfsu-DreamTeamFS:~$ md fourtynine
Directory is full ( ͡° ͜ʖ ͡°)

$ exit
student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assign
ment-file-system-LilianGouzeot$
```

Change/Remove directory functionality with persistence:



```
/

BobsAirPods@sfsu-DreamTeamFS:~$ cd /Jenn/Lauren

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/Lauren/$ pwd
/Jenn/Lauren/

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/Lauren/$ exit
student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assignment
lianGouzeot$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
The Disk Is Already Format to a DreamTeamFS (The best file system of all time) ಠ_ಠ

BobsAirPods@sfsu-DreamTeamFS:~$ cd /Jenn/Lauren

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/Lauren/$ pwd
/Jenn/Lauren/

BobsAirPods@sfsu-DreamTeamFS:~/Jenn/Lauren/$ ls -l


BobsAirPods@sfsu-DreamTeamFS:~/Jenn/Lauren/$ cd /

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920     Hello Thu Dec  3 16:54:04 2020
D      17920      Jenn Thu Dec  3 16:54:04 2020

BobsAirPods@sfsu-DreamTeamFS:~$ rm Hello
Removing directory...
Successful removal...

BobsAirPods@sfsu-DreamTeamFS:~$ ls -l

D      17920      Jenn Thu Dec  3 16:54:04 2020

BobsAirPods@sfsu-DreamTeamFS:~$ exit
student@student-VirtualBox:~/Desktop/CSC415/FileSystem Project/group-term-assignment
lianGouzeot$
```

Defragmentation:

You can see how boop is originally at block 130 but moves to 95 after beep is deleted, test.txt is still able to copy meaning all the extents have been adjusted correctly.

Note: To get the block outputs above the prompt, uncomment line 64 in main.c