File System Project

Team Fiore

- Haoyuan Tan(Sunny), 918274583, CiYuan53
- Minseon Park, 917199574, minseon-park
- Yong Chi, 920771004, ychi1
- Siqi Guo, 918209895, Guo-1999

The GitHub link for your group submission
https://github.com/CSC415-Summer21/csc415-filesystem-ychi1

Understanding of the file system:
A file system is a logical collection of files on a partition or disk. A partition is a container for information. It uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there. The root directory (/) contains other files and subdirectories; and each file or directory is uniquely identified by its name, the directory in which it resides. It is self-contained and there are no dependencies between one filesystem and another. The directories have specific purposes and generally hold the same types of information for easily locating files.

A file system is designed in a way so that it can manage and provide space for non-volatile storage data. All file systems required a namespace that is a naming and organizational methodology. The namespace defines the naming process, length of the file name, or a subset of characters that can be used for the file name. It also defines the logical structure of files on a memory segment, such as the use of directories for organizing the specific files while using the file system to perform specific tasks. Once a namespace is described, metadata will be created and helps us to know about the stored data.

The data structure supports a hierarchical directory structure and is used to describe the available and used disk space for a particular block. It also has the other details about the files such as file size, date & time of creation, update, and last modified. It stores advanced information about the section of the disk, such as partitions and volumes. The advanced data and the structures that it represents contain the information about the file system stored on the drive; it is distinct and independent of the file system metadata.

Filesystems also require an Application Programming Interface (API) that provides access to system function calls that manipulate file system objects like files and directories. APIs provide for tasks such as creating, moving, and deleting files. It also provides algorithms that determine things like where a file is placed on a filesystem. Such algorithms may account for objectives such as speed or minimizing disk fragmentation.

A description of our file system:
We designed a simple file system that manages files and directories based on the understanding of the concept of a file system in Linux. This file system uses a bitmap to manage free space. Bitmap stores a binary value and it can check whether it is free space or not. We changed bitmap as 1 from 0 after writing codes in that block. This main driver offers listing directories (ls), creating directories(md), adding and removing files (md, mv), copying files (cp, cp2l, cp2fs), moving files(mv), and printing help to let users know about commands easier.

Issues we had:

1. At the very beginning, we were struggling with how to set up the directory. When we format a new volume, we always need to initialize the volume control block, freespace, and root directory. However, the root directory is relatively harder to implement since we didn't know what to put in it. So we browse the lecture videos and recordings and decide to write a simplified version where it just stays right behind the volume control block and freespace. However, a new problem arose when we tried to deal with the rest of the new functions in mfs.c. That is the freespace allocation. We were hard-coded on the root directory, but generally, we need a flexible one for other new directories. So we had to modify and create a new algorithm to look for contiguous blocks for allocating freespace. By doing that, we had a flexible way to retrieve the free blocks and write data into the volume. But this was not all, Sunny came up with a new optimization where we can keep track of the index of the first free block to save time while searching for contiguous free blocks. This can save a lot of time if we already had a lot of things allocated, so we implemented a new check after finding the contiguous free blocks and to compare the new free index and previous free index to determine the first free index.
2. While we are trying to implement the 'ls' command, there is a tricky function we need to use, which is fs_opendir(). In the beginning, we thought we could set the opened dir to the current working directory, but it felt weird and we couldn't reset it in the fs_closdir(). That was not good. So we tried to store the previous current working directory as a global variable, but we suddenly thought: why not just put the opened directory as a global variable so we can easily do all things with it across different methods. So we chose to keep track of an opened directory in the global and assign the pointer to it if we open one, and close it by freeing the pointer and setting it to NULL. This helped us to fix the problem when we called isDir() and isFile() while opening a directory.
3. We also had a problem on isDir() and isFile() while working with fs_opendir(). The main problem was that they didn't accept a path, so we had to work on checking the directory or file from the current working directory. But 'ls' wanted them to work with an opened directory! So we almost skipped this part. But once we stored the opened directory as a global variable, we now had a way to access the opened directory. Even though the main checking part was still on the current working directory, we set the current working directory to our open directory. After we finished checking, we just reset it, which was a wonderful strategy but a bit dangerous if we didn't handle it correctly.
4. We also had a small problem while working with the 'pwd' command since we didn't record the whole path for each directory. What we could do was store the current name, and call the parent. Repeat this process until we reach the root directory. This can be done by checking the root directory location where we store in the volume control block. But the hard thing was to concatenate them together with a slash. Because we were going backward not starting from the root, which meant that we needed to concatenate the name to the front. So we spent a few hours trying to figure out the logic. Finally, we decided to have a temporary string to concatenate the first for our current directory name, then we concatenated the path from children on that temporary string. So now the temporary has a correctly ordered path, and we copied back to the backward path and loop until we reached the root. Then we did

the same process to concatenate the dot to represent the root directory to the very beginning of the path.

5. While we were removing a directory or file, we ran into a problem trying to get the parent directory. Because for deleting that, we don’t really clean up the data, we just tell the system that they are free and we can replace them whenever we want. To accomplish this, we must make sure we deallocate the freespace, which means mark the block to be free, and also mark it free in the entry list of the parent. To do this, first, we did some modifications on the allocateFreespace() and used it as a template for releaseFreesapce(). The problem was not hard to fix, but we needed to change both the algorithm and check for the first index. We quickly fixed this, but we actually stuck for a while with getting the parent directory. Because we need to split the string, and we only want the path before the last slash. Since C languages don't work well with strings like Java. We had a hard time figuring that out. But we found a good method, strrchr(), which can help us easily find the location of the last slash. Then we can split the path into two parts and copy them into the returned buffer so we can use them. So now we got the path to the parent directory so we could get the information of the parent. Lastly, we just loop through the entry list and find the file or directory that is marked used, and has the same name, and mark them to be free from now on. Also, we needed to decrease the number of allocated entries to avoid possible errors. And it is pretty important that we only looked for the one that was marked used because we might have some deleted directories but have the same name.
6. The last largest problem we encountered was the logic of b_io.c. But thanks to Sunny’s hard work, we made it work with reading and writing data. But we still had some small bugs on moving files. Because we didn't check if the action was successful or not. So we decided to use the read count and write count to determine. Only if both of them are 0 or above, means they actually do the movement, so we can delete the original file. We know this is not efficient, because we can just copy the data from an entry list and put it to the destination’s entry list. But we just didn’t have more energy to deal with it, since b_io can handle this job pretty easily.
7. There were still some very small problems but we didn’t remember them, but most of them were just fixing small bugs and the segmentation fault, or just some expected errors that we need to handle.

Detail of how your driver program works:
This driver program has the main function. This main function starts partitioning and executes initFileSystem() in order to read the volume and access the data. Then, the main function will run as a shell and take commands from the line. It can perform many tasks like a modern file system that is handling files in volume. We implement all commands required by this assignment and even give them some extra functionalities to make our file system more flexible.

Compilation Screenshots:

```
student@student-VirtualBox:~/csc415-filesystem-ychi1$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o mfs.o fsInit.o b_io.o fsLow.o -g -I. -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.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 6
used block index: 0 1 2 3 4 5
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 0

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 11
used block index: 0 1 2 3 4 5 6 7 8 9 10
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 6

DEBUG: fsInit.c:140 - initFileSystem(): *** VCB STATUS ***
DEBUG: fsInit.c:141 - initFileSystem(): number of blocks: 19531
DEBUG: fsInit.c:142 - initFileSystem(): block size: 512
DEBUG: fsInit.c:143 - initFileSystem(): vcb block count: 1
DEBUG: fsInit.c:144 - initFileSystem(): freespace block count: 5
DEBUG: fsInit.c:145 - initFileSystem(): first free block index: 11

Prompt >
```

ls - Lists the file in a directory

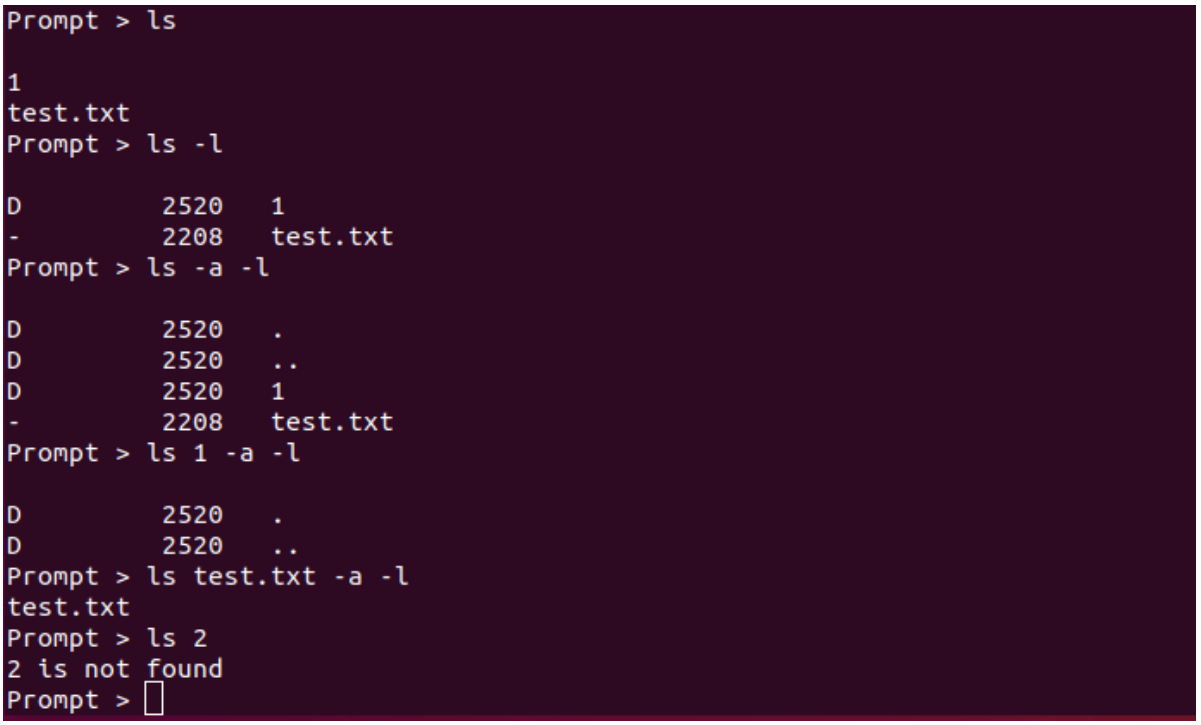
Syntax: ls [path]

- if no path is given, print entries of the current working directory
- if the path is given, 1) if it points a directory, print entries of that directory 2) if it points to a file, print the name of that file

Logic:

- finds the directory from the given path and “opens” it
- read the entry from the list of the opened directory one by one
- determine if the entry is a file or directory
- print info based on and flags

Screenshots:



cp - Copies a file

Syntax: cp sourcePathAndFilename destPathAndFilename

- it copies the data from the source file and creates the file in the destination directory from the path
- if NO name is given in the path, such as “1/”, it is designed to be failed

Logic:

- splits bot arguments into paths and file names by looking for the last slash
- “open” both directories in the file control block
- check if the directories are existed **(fail if not)**
- check if the source file name is not empty **(fail if not)**
- copies the data into the buffer from the source file by looking for the split filename
- check if the destination file name is not empty **(fail if not)**
- check if there is no same name in the destination directory **(fail if not)**
- write the data into the destination file buffer
- after all data has been copied, allocate freespace based on its size
- put in the information of the new file in the destination directory’s entry list
- write a new file and change directory into the volume

Screenshots:

```
Prompt > ls -l

D         2520   1
-         2208   test.txt
Prompt > cp test.txt

same name of directory or file existed
Prompt > cp test.txt second.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 26
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 21
Prompt > ls -l

D         2520   1
-         2208   test.txt
-         2208   second.txt
Prompt > ls 1 -l

Prompt > cp test.txt 1/first.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 31
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 26
Prompt > ls 1 -l

-         2208   first.txt
Prompt > cp 1/first.txt back.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 36
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 31
Prompt > ls -l

D         2520   1
-         2208   test.txt
-         2208   second.txt
-         2208   back.txt
Prompt >
```

mv - Moves a file

Syntax: mv sourcePathAndFilename destPathAndFilename

- it copies the data from the source file and creates the file in the destination directory from the path, then it will delete the source file
- if NO name is given in the path, such as “1/”, it is designed to be failed

Logic:

- do a “cp” on the file
- check if the file is copied by readCount and writeCount **(fail if either is negative)**
- delete the source file

Screenshots:

```
Prompt > ls -l

D         2520   1
-         2208   test.txt
Prompt > mv test.txt rename.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 26
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 21

DEBUG: mfs.c:1008 - releaseFreespace(): first free block index changes to 16

used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 21 22 23 24 25
test.txt : test.txt was removed
Prompt > ls -l

D         2520   1
-         2208   rename.txt
Prompt > mv rename.txt 1/

name should be given!!!
Prompt > mv rename.txt 1/first.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 26
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 16

DEBUG: mfs.c:1008 - releaseFreespace(): first free block index changes to 21

used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
rename.txt : rename.txt was removed
Prompt > ls -l

D         2520   1
Prompt > ls 1 -l

-         2208   first.txt
```

md - Make a new directory

Syntax: md pathAndDirectoryname

- if NO path is given, it is designed to be failed
- if the path is given, it creates the directory at the directory from the path
- if NO name is given in the path, such as “1/”, it is designed to be failed

Logic:

- makes a copy of the path to avoid modifying the original path
- split the copy of the path for the parent directory and the new directory name
- check if the new directory name is not empty **(fail if not)**
- gets the directory pointer by the split path
- check if the parent directory is existed **(fail if not)**
- check if that directory’s entry not reaching the max amount **(fail if not)**
- check if there is no same name in the directory or file **(fail if not)**
- creates a new directory with . and .. setup
- finds the first available space and puts the data in the parent’s entry list
- writes new directory and change directory back into the volume

Screenshots:

```
Prompt > ls -l

Prompt > md 1

DEBUG: mfs.c:808 - fs_mkdir(): creating new directory 1
DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 16
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 11
Prompt > ls -l

D         2520   1
Prompt > md 1/2

DEBUG: mfs.c:808 - fs_mkdir(): creating new directory 2
DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 21
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 16
Prompt > ls 1 -l

D         2520   2
Prompt > md .

same name of directory or file existed!
Prompt > md ..

same name of directory or file existed!
Prompt >
```

rm - Removes a file or directory

Syntax: rm path

- if NO path is given, it is designed to be failed
- if the path is given, it creates the directory at the directory from the path

Logic:

- check if the path points to a file or directory
- if it is a file, delete it from the parent's entry list and release its freespace
- if it is a directory, check if it is a not root directory **(fail if not)**
- do a "rm" (using recursion) on all existing entries except . and .., and then delete it from its parent's entry list and release is freespace

Screenshots:

cp2l - Copies a file from the test file system to the Linux file system

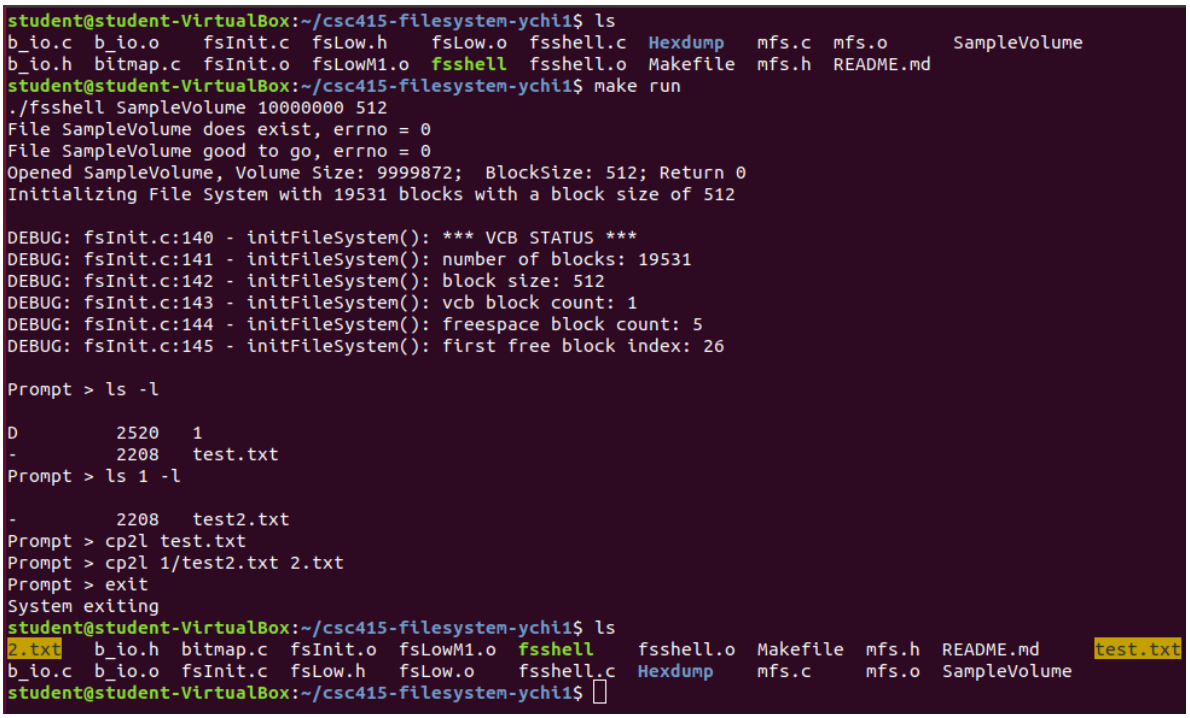
Syntax: cp2l sourcePathAndFilename [destPathAndFilename]

- if NO path is given, it uses the current working directory and source name
- if the path is given, it copies the data from the source file and creates the file in the destination directory from the path
- if NO name is given in the path, such as “1/”, it is designed to be failed

Logic:

- same logic as “cp”, but use Linux write() instead of b_write()

Screenshots:



cp2fs - Copies a file from the Linux file system to the test file system

Syntax: cp2fs sourcePathAndFilename [destPathAndFilename]

- if NO path is given, it uses the current working directory and source name
- if the path is given, it copies the data from the source file and creates the file in the destination directory from the path
- if NO name is given in the path, such as “1/”, it is designed to be failed

Logic:

- same logic as “cp”, but use Linux read() instead of b_read()

Screenshots:

```
Prompt > ls -l

D         2520   1
Prompt > ls 1 -l

Prompt > cp2fs test.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 21
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 16
Prompt > ls -l

D         2520   1
-         2208   test.txt
Prompt > cp2fs test.txt second.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 26
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 21
Prompt > ls -l

D         2520   1
-         2208   test.txt
-         2208   second.txt
Prompt > cp2fs test.txt 1/first.txt

DEBUG: mfs.c:80 - allocateFreespace(): first free block index changes to 31
used block index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
DEBUG: mfs.c:91 - allocateFreespace(): returning block index: 26
Prompt > ls 1 -l

-         2208   first.txt
Prompt >
```

cd - Changes directory

Syntax: cd path

Logic:

- get directory pointer from the path
- check if the directory is existed (fail if not)
- frees the original current working directory
- set the current directory to the directory we got

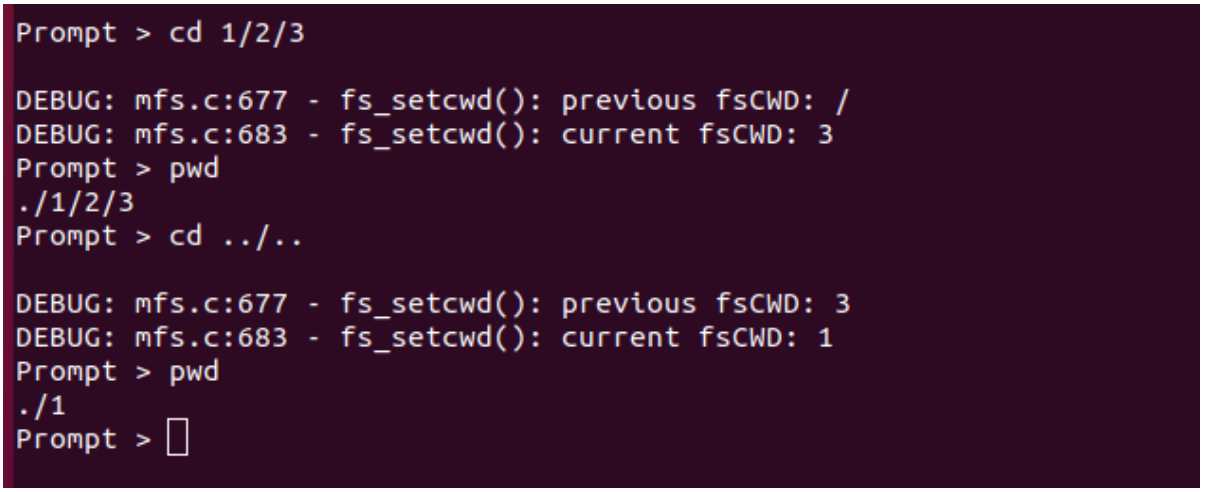
pwd - Prints the current working directory

Syntax: pwd

Logic:

- get the current directory name
- concatenate the name to the back of the parent's directory name
- continue until reaches the root directory
- put . to the beginning to represent root and print it

Screenshots:



Description of core functions

fs_mkdir(): It initializes a new directory with the given name, and puts its information into the entry list of its parent directory.

fs_rmdir(): It removes the directory directed by the path. If the directory is empty then this function will not succeed.

fs_opendir(): It opens a directory from the directory name and stores the data. Returns a pointer to that directory data. The related entry index is positioned at index 0.

fs_readdir(): It returns a pointer to an entry data from the opened directory entry list. It returns NULL on reaching the end of the entry list or if an error occurred.

fs_closedir(): It closes the directory by deallocating the stored data of the opened directory in memory. And it also resets the opened directory pointer to NULL and index to 0.

fs_getcwd(): A temp buffer is created, and malloced. Then, make a copy of the current working directory. Loops backward by getting to the parent directory until the root directory to get a full path in order to print

fs_setcwd(): It finds the directory based on the path, and deallocate the original cwd then set the directory if found as the new cwd.

fs_isFile(): It will look for the parent directory and check its entry list on used entries. If there is an entry has the same name, it returns true.

fs_isDir(): It will look for if the directory exists based on the path and return that result.

fs_delete(): It goes to the parent directory from the path, and checks if the file exists. Then it marks the space as free in parent directory's corresponding index and release its freespace.

b_open(): The function builds a connection between a file and a file descriptor. It initializes an open file descriptor that refers to a file. We allow the file descriptor to be used by other I/O functions to refer to that file.

b_read()**:** It reads data from the volume from the start location of the netry. It reads up to the buffer size of bytes or just remaining bytes in the data from the file into the buffer in the fcb.

b_write(): It writes up the whole passed in buffer into our buffer in fcb, and we concatenate the data so that they are actually connected in the memory.

b_close(): It closes the file descriptor so that it no longer refers to any file. Any it free the data that was malloc() in the fcb that was associated with.

Structure:

```
#define MAX_NAME_LENGTH 256
struct fs_diriteminfo  //directory item info
{
        unsigned short d_reclen; /* length of this record */
        unsigned char fileType;
        unsigned char space;                    // determine this entry is free or used
        uint64_t entryStartLocation;            // LBA of the entry, either a file or directory
        uint64_t size;                          // the exact size of the file occupies
        char d_name[MAX_NAME_LENGTH]; /* filename max filename is 255 characters */
};

typedef struct  // directory file
{
        unsigned short d_reclen;                /*length of this record */
        uint64_t directoryStartLocation;        /*Starting LBA of directory */
        unsigned short dirEntryAmount;          // amount of undeleted entries
        char dirName[MAX_NAME_LENGTH];          // name of this directory
        struct fs_diriteminfo entryList[MAX_AMOUNT_OF_ENTRIES];
        // unsigned short dirEntryPosition;     // we keep it as global value
} fdDir;

typedef struct // volume control block
{
   uint64_t magicNumber;
   uint64_t blockSize;
   uint64_t numberOfBlocks;          // also represents bitmap length
   uint vcbBlockCount;               // also represents freespace start location
   uint freespaceBlockCount;         // used for check when it is not the first run
   uint64_t firstFreeBlockIndex;     // used for check when it is not the first run
   uint64_t rootDirLocation;         // can be calculated by adding the other two counts
} vcb;

typedef struct b_fcb  // file control block
{
        int fd;                         // holds the systems file descriptor
        char *buf;                      // holds the open file buffer
        uint64_t index;                 // holds current index of the buffer
        uint64_t buflen;                // holds how many valid bytes are in the buffer
        fdDir *parent;                  // holds the parent directory of the file
        char *trueFileName;             // holds the true file name not the path
        unsigned short detector;        // holds the functionality of the method
} b_fcb;
```

DEBUG

There are mainly three debug related functions: dprintf(), ldprintf(), eprintf()

dprintf() and ldprintf() are showing specific information we set up and they can show which line, which file and which function. ldprintf() stands limit debug. It just hides the debug information without removing it. Buf if we want to see all debug information, just go to mfc.h and comment out the definition of LIMIT_DEBUG. Other than that, as coders, we can easily switch the debug info by simply adding or removing the 'l' which is pretty neat while testing.

eprintf() just stands for error printf(). This is only for unexpected error printing, such as fail to malloc(). It is also a good practice to have them in the code even though we didn't see any of them get triggered. But when we look back at the code, we know which part might have errors needed to handle and which errors are expected, and which are not. It is similar to the idea of try and catch.

```
#define FS_DEBUG          // comment out to completely remove all debug info
#define LIMIT_FS_DEBUG // comment out to show all debug info

// comment out FS_DEBUG to print NO debug info
#ifdef FS_DEBUG
#define dprintf(fmt, args...) fprintf(stdout, "\nDEBUG: %s:%d - %s(): " fmt, \
                                                    __FILE__, __LINE__, __func__, ##args)
#else
#define dprintf(fmt, args...) // do nothing
#endif

// ldprintf() can hide by simpling adding "l" to original dprintf
// comment out LIMIT_FS_DEBUG to print ALL debug info
#ifdef LIMIT_FS_DEBUG
#define ldprintf(fmt, args...) // do nothing
#else
#define ldprintf(fmt, args...) fprintf(stdout, "\nDEBUG: %s:%d - %s(): " fmt, \
                                                     __FILE__, __LINE__, __func__, ##args)
#endif

// print error which shows unexpected error
// all check with eprintf() normally get a NULL, so we can skip free()
#define eprintf(fmt, args...) fprintf(stderr, "\nERROR: %s:%d - %s(): " fmt, \
                                                    __FILE__, __LINE__, __func__, ##args)
```