CSC 415 File System Project

Summer 2023

Coffee On The Rocks		
Members	**Student Id**	**Github Id**
Matthew Bush	921619696	fattymatty15
Oscar Galvez	911813414	gojilikeog
Hyok In Kwon	922373878	hkwon4
Jacob Lawrence	922384785	liftcodesleep

Repo Link:

https://github.com/CSC415-2023-Summer/csc415-filesystem-liftcodesleep

File System Structure

VCB Structure

The Volume Control Block contains the information to locate the file system structures contained within, as well as certain metadata for the files stored within. Ours will contain:

- unique_volume_ID - Magic number - Used to test if volume needs reformatting
- total_block will store the total amount of blocks within the VCB
- free_block_map contains where the bitmap begins
- block_size will hold the specified block size
- root_location holds the starting block number for the root directory
- The rest of the VCB contained metadata, such as a max path length and a name length for a file
- bytes_in_use is declared to determine the offset required to write to a block in the correct position. Meant to be incremented when a new direntry is declared, de-incremented when a new direntry is deleted.
- blocks_in_use is meant be changed when new block has been used - Currently does not account for partial block usage - To be used in calculateOffset function
- Direntry_size is used to reference the size of a direntry struct. Will assign in the initialization function.

VCB class contained two functions:

- initVCB - Initializes the volume control block if it is determined that the unique_volume_ID does not match the one in the SampleVolume, or if no SampleVolume exists.
- calculateOffset - Function is meant to calculate the current byte offset in a block to use in conjunction with memcpy to write data in the correct position of the buffer.

Free Space Structure

The free space structure is managed with a bitmap. The struct, called blockmap, comprises an int * array blocks that keeps track of the map, and a size_t size member to keep track of the size. Each bit in the map represents the free state of a corresponding block on the disk. The first 6 bits are necessarily set because the first 6 blocks contain the VCB and the map itself. The following functions were created to interact with the bitmap:

- init_free_space(int block_count, int block_size)
  - Initializes the free space bitmap
- load _free_space(int block_count, int block_size)

    - Used to get the free space bitmap for other functions
- Extent *allocate_blocks(int blocks_required, int min_per_extent)
    - Find the empty blocks within the bitmap and allocate the blocks.
    - This is the primary function used by the directory entries to find available space to create their extents
- release_blocks(int start, int count)
    - Releases blocks to indicate new free space. Required function to delete data

Directory System

The directory system is handled by an array of structs called direntry. Each direntry contains the following information:

- Char name[100]
    - holds the name of the directory entry
- Unsigned long size
    - contains the size of the directory entry
- Struct Extent extents[3]
    - contains the location of the entry using extents
- Unsigned long time_created, time_last_modified, time_last_accessed
    - Each contains time data that will be converted to epoch time
- Char isFile
    - The flag that indicates if the directory entry indicates a file

The directories are created by the function init_dir(int minEntries, direntry * parent) which takes a minimum value of unused entries to be created and a specified parent directory that the newly created directory would be related to. If the value passed into parent is "NULL," then the root directory is created.

The location of directory entries is managed by extents using an array of Extent structs that contain:

- Unsigned int start_loc
    - The starting block number
- Unsigned int amount_after_start
    - How many blocks the data occupy from the start_loc

Our system of extents incorporates the concept of keeping track of disjointed contiguous spaces using and the bitmap as an efficient way of handling changes to our file system. While the directory entries themselves only start with an array of 3 extents, the extent manager is designed to create secondary and tertiary extents if necessary. The functions that are used to manage the extents are:

- Void extent_init(Extent *extent)
    - Creates the extent
- Void extent_free(Extent *extent)
    - Frees the memory occupied by the extent
- Int extent_get_block_num(Extent *extent, uint index)
    - Returns the specified block number within the extent
- Unsigned int extent_append(Extent *extent, uint block_number, uint count)
    - Adds a contiguous number of blocks to an extent
- Unsigned int extent_remove_blocks(Extent *extent, uint block_number, uint count)
    - Removes blocks from an extent
- Unsigned int extent_get_total_blocks(Extent *extent)
    - Getter function for the total number of blocks within extent
- Void extent_print(Extent *extent)
    - Printer function for extent contents

Work Division

Task	**Performed By**
Directory Entry	Hyok In Kwon
VCB	Oscar Galvez
Free Space Management	Jacob Lawrence
Extents	Matthew Bush
Path Parsing Functions	Hyok In Kwon
Getting/Setting Current Directories	Oscar Galvez
Making/Opening Directories	Hyok In Kwon
Reading/Closing Directories	Matthew Bush
File Checking/File Status Functions	Jacob Lawrence
parsePath revision	All
I/O Operations Functions	All
Code Cleanup	Jacob Lawrence

Testing/Debugging	Matthew Bush, Jacob Lawrence
WriteUp Editor	Hyok In Kwon

Group work Evaluation

For M2 we followed my idea and divided the work by dividing the directory functions that were recommended about evenly. This was probably not a great approach because different functions required fairly different amounts of work ranging from trivial to fairly involved, and many of them required a shared functionality that was assigned to one person when the entire group would need it to proceed. All in all the work was done but a more nuanced approach to how it was delegated would have made things run more smoothly.
There were some difficulties when planning meetings to discuss how certain functions needed to interact with our parse path function. Implementing the parse path function was originally thought to be a tricky but non-time-consuming task, but was quickly proven to be a massive undertaking. The mismanagement of handling the parse path cost the group considerable time to debug and eventually led the entire group to work on refactoring it in order to continue working on most tasks regarding milestone 2.

Issues and Resolutions

- My first issue with implementing the free space system was understanding its requirements.
    - I knew I needed an array of pointers and functions to manipulate them. I decided on an array of int pointers because they're a convenient native size, and it would potentially save me time when reading the map.
- Another issue I had was mapping the different representations of the data, making sure to map from a word to a bit and reverse took some time working with it to wrap my head around.
    - I resolved my uncertainties with tests. I wrote debug prints and made functions that produced the expected outcome and could demonstrate it.
- For initializing the root directory, the success relied on the proper coupling between the Volume Control Block, extent table, and free space management. This meant that there was heavy uncertainty regarding the successful creation of the root directory initialization until all other parts were completed.
- Setting the init_dir() function to work beyond when the dirent * parent was NULL
    - Although the current requirement does not ask to make new directories aside from parent, it was extremely crucial that there was a way to create new directories if needed. However, as no directories were currently

made, this can be resolved after the space was properly formatted. This will be resolved later.

- A segmentation fault occurred after trying to run init_dir()
    -

- When creating the header file for the extents, I encountered issues with developing suitable function prototypes that would be helpful for my team members. Specifically, I was unsure about the return type and the value to be returned by the extent_append function in case of failure.
    - To address this problem, I organized a team meeting where we collectively discussed and agreed upon the desired return types for different functions. We also established clear naming conventions to ensure consistency and ease of understanding for all team members.

- While implementing the extent_append function, I faced difficulties when dealing with extents that needed to be added starting from the end of another block. Initially, my approach was to create a new extent at the end rather than modifying the last block. Even after making some code changes, I still encountered issues when the extent was not the last block in the table.
    - To resolve this issue, I modified the code to only edit an extent if it was the last extent in the table. This approach resolved the problem for the current implementation, but it might require further adjustments when we add the second and third extent tables to the system. Continuous monitoring and improvement of the implementation will be necessary to ensure smooth functionality in the future.
-

- My first issue with planning how to write a Volume Control Block was understanding the theory behind it, meaning what that meant in the first place. Although we weren’t technically starting from scratch on the file system, we needed to understand where to start in the first place.
    - This issue was resolved by moving into a planning phase, which did involve more research as to how a file control system works. Planning allowed me to implement the Volume Control Block designed in the File System Design assignment with the changes recommended during our group meetings.

- My last issue is understanding where the designed initialization of the volume will fit with the other portions being designed.
    - This will be resolved by making modifications to the initFileSystem function contained in the fsInit.c file as the pieces come in to make sure it gets initialized and written to memory properly. Testing will play a large part in understanding our code's components and ensuring it all works hand-in-hand.

- On the implementation of parsePath() helper function, there were several error-checking cases that needed to be accounted for in order to properly function. This function needed to tokenize a c string that was delimited by the ‘/’ character. The following error conditions needed to be accounted for in order to be considered a valid path: no consecutive ‘/’ characters, no null inputs.
    - Using the save pointer in strtok_r(), the consecutive ‘/’ character could be detected and return the error indicator (function returns null upon failure)
    - The function checks for an empty input before tokenizing
    - Illegal names also needed to be updated like, “...” so they cannot be passed as a valid directory entry
- There needed to be a way to validate the path in order for muhc of the file systems functions to work without redundant code
    - loadedDir* validatePath(parsedPath *ppath) function along with two additional functions, direntry* loadDir(direntry* dir, int index) and direntry *getRoot(), within the direntry.h were used to have directory entries accessible for various file system functions
        - These functions were very difficult to test initially as nothing but the root directory was initialized
- validatePath() needed a way to differentiate when it is called in the context of an absolute path or relative path.
    - Created an indexD that copies the pointer to one of the loaded directories depending on which flag was set by the parsePath function. It is also accompanied by iparentD that tracks the parent directory to the indexD.
        - This was done to ensure that the loaded parent and current directories will not be updated and therefore corrupted upon validatePath() failing
- There was a persistent double free or corruption (!prev) error that occcured
    - There was a redundant free() call within our tests for validatePath()
        - As validatePath() frees the parsedPath struct, the test also tried to free the path after the validatePath() function were to complete.
- There needed to be a way to encapsulate the parsePath functions to be usable across all required functions to ensure simple future modifications.

- Renamed parsePath() to pathstruct_create() and parsePath() takes the two functions pathstruct_create() and validatePath() to combine their functionalities and return a struct that is usable across the many functions that use it.
    - st
  - lastPath direntry pointer was also removed as it is not needed in most cases
- parsePath() was rewritten for clarity less redundant code
  - parsePath() originally had numerous globally accessed pointers that depended on the caller to free at the end of every instance they were used. They also needed to be reinitialized and allocated spaces at various different intervals. This caused a mess with memory management along with being difficult to read or modify based on what could be needed for the future.
  - The current revised version eliminated much of the redundant calling and limited the parsePath() function to only return a singular struct pointer that contained the information of the last valid subdirectory pointer alongside where it is present within the index of its parent directory.
- Within the directory entries, there was an error regarding the size member variable. While size should have reflected the number of bytes occupied by the directory or number of bytes the file occupies, the size only statically represented the number of bytes that the direntry struct could create. This created issues regarding how to determine a file's size for many of the b_io functions.
  - As directories themselves have no reason to change in size, any time a file were to change the size would reflect that change (i.e. writing to the file).
- For writing the fsshell command, mv, there were a few design difficulties that came with the implementation. To keep up with the idea of encapsulation and to minimize duplicate code, several factors needed to be considered. As with the other functions, mv must be accomplishable mostly with using the functions within mfs.h.
  -
- fs_getcwd: The decision was made to have getcwd depend on setcwd, as it did not make much sense to both malloc memory and to generate an I/O for this function by performing an LBAread on the block of memory that theoretically should already be loaded into memory. The challenge while designing fs_getcwd by writing out the problem was attempting to deal with not having an actual structure to test against.
  - The solution came once we were able to boil down how we wanted to create files, and interpret and traverse the files we had created within each

theoretical directory. It started with creating one file at root, and loading that into a global variable contained within mfs.c to read from, and from there it was able to be more robustly tested when we got comfortable creating more test files/directories. Creating a dummy file system to test against that didn't have any variables set than a filename and file flag was enough to start changing the current working directory and comparing the result of calling fs_getcwd. As simple as the function was, the logical considerations of the function still ate up some time.

- fs_setcwd: The current problem with designing the logic for this function is with the testing. Testing would require a file directory that we can traverse to set the current working directory, and with something to test against, I can at least work toward a solution. That, and my personal understanding of designing a file system at the moment is a major contributing fact to the trouble I am having.
    - Steps taken so far:
        - calculateOffset(): Method to calculate the current offset in the buffer to use with memcpy() to save nearly declared direntries in the correct position of the respective block. (bytes_in_use - (blocks_in_use * BLOCK_SIZE)). Currently does not take into consideration the leftover 8 bytes in a block (One block can accept 3 direntry) - This ended up not being necessary as a result of being able to calculate the offset (if necessary) from the extents.
        - Adding variables to the VCB:
            - Bytes_in_use - Meant to track the number of bytes currently in use. To be used in calculating the offset of the buffer to save information into. (Not necessary)
            - Blocks_in_use - Seems to be a redundant variable. Meant to be used in the calculateOffset function. (Not necessary)
            - Direntry_size - Current size of direntry (sizeof(direnty) = 168) Only meant to be there for reference
        - Testing:
            - Wrote into the SampleVolume in order to identify the steps required to understand the implementation of a basic file system. After writing, I started testing reading bytes with the appropriate offset into a new direntry structure, and successfully retrieved the data written loaded back into the structure.
    - Conclusion and Reflection for fs_setcwd: The difficulty around implementing the logic described in the manpage for cd, ironically enough, was not from traversing the file structure to validate if it was a valid pathname. The parsePath function was implemented to do so for all the

functions that would depend on validity of said pathname in order to continue their operation. My difficulty with this function, for whatever reason it may be, came from splicing the pathname with all the conditionals set by the manpage. All it is really defining is how a shell the absolute path of what the shell just interpreted would look like, and in retrospect, there is absolutely no good reason as to why this function provided such a headache for me to implement. Having the correct PWD is important in traversing a file system where you would not know where you really were otherwise unless it was planned and mapped out, which was eventually required while testing and debugging this function.

    - Debugging and testing led to four attempted redesigns of the algorithm, with two of these attempts being successful implementations of the requirements. However, the current version of the algorithm is, in my personal opinion, clunky and hard to read or maintain. Although there should be no reason to maintain the algorithm, the fact that one of the failed implementations in fact made the logic easier to follow will haunt me for the time being. That implementation was based on using a c-string array, taking into consideration "." and ".." while tokenizing the path, and taking the appropriate steps to modify the directory prior to setting it as the PWD.
    - At the time of writing, I will make one more attempt at implementing an algorithm which is easier to read and maintain, but Milestone #3 is the main priority. Until then, this current implementation depends on using two tokens. A redesign should not impact the operation of the rest of the project, as it should essentially perform the same actions. It will be immediately reverted back to the current algorithm if there is any negative impact detected after extensive testing.

- fs_delete : Encountered a problem while trying to wipe the name that was written.
  - Being that the name was saved in an array of char's, it was identified that each individual bit needed to be cleared in order to actually wipe the bits within the hexdump. Original plan was to use a strcpy to wipe the name from the variable, but the hexdump proved that this only inserted the null terminator in the first bit, tricking calls to the 'name' variable of the file being deleted into thinking it was empty. As stated before, the hexdump proved otherwise. Memcpy also didn't help in the situation, as copying the null terminator into the size of the name (100 bits) caused some uncontrolled behavior, such as print statements called shortly afterward saving into the block of memory.

        - This solution to this problem was dumbing down the problem instead of thinking it was a function call. Implementing a for loop, which would iterate through the length of the name for the file, setting each individual bit to the null terminator. Doing so would prevent volatility that was occuring while using the functions that I had originally believed were the best approach. We are able to remain safely within the bounds of that specific entries memory do the for loop.
    - The second problem encountered with this function was the mistake of both loading into the wrong cluster of blocks, as well as deleting the locations of the file stored within the extent prior to deallocating the blocks appropriately.
        - This was remedied by changing the location of the reassignments of both count and start for the extent after the deallocation, as well as a long game of printf's to see where the function was actually loading into.
- b_seek : The problem that arose with this function came from two items. First, the manpage only indicates that fail conditions are available for this function if there are #define variables used: SEEK_DATA and SEEK_HOLE. As they are not defined within our program, we moved forward assuming we do not need to consider these options. That leaves SEEK_SET(0), SEEK_CUR(1), and SEEK_END(2), creating a problem that Jacob actually mentioned troubleshooting with b_read.
    - The solution to this problem was to leave it alone. We have to assume that the user will assign the 'whence' variable properly, not combining values to modify the result of the whence variable. Although the manpage specifies that the seek function does not technically have any return values that indicate a failure, there are some fail conditions to catch if anything passed in is outside the scope of what is specified for the assignment. In particular:
        - If the fd variable passed in is outside the range of our set limits
        - If the file we're expecting doesn't actually exist in the file descriptor array
        - If the 'whence' parameter was used improperly - If multiple flags equal a number greater than 2, or if a number less than 0 was passed in. There are no ways to detect if someone passes in SEEK_SET and SEEK_CUR together, as it would simply read a 1. This would cause b_seek() to operate within the SEEK_CUR directive. For the sake of this assignment, it must be assumed that the user will not do so.

- Volume Control Block: There is a need to determine what additional variables and methods should be added in order fulfill the needs of the file system.
    - LBAread and LBAwrite makes it easier to read blocks of data into our declared buffers, but there still needs to be a way to track individual block positions so we can properly read the written data back into the structures we've created.
- Testing methods - One of the biggest hurdles with this assignment personally has been grasping the abstracts of what is required when attempting to implement a file system. While the structure of the file system calls for the same things generally, selecting how to implement it when is a journey on its own. Trying to comb through reference material on the internet proved more helpful than the textbook, but they didn't necessarily help with the abstract concepts…and how can it? Embracing the fact that a lot of the "how's" with creating this file system would come from collaboration with each other did it start to click for me. Admittingly, I was the last to grasp the concepts of this assignment. The next thing that pushed me over would be actually having something to test against.

    - Taking inspiration from how Matthew and Jacob implemented tests to have something to develop against while the file system was in its infancy, it gave me the structure to write the following tests.
        - **test_fsgetcwd** located in mfs_tests.c was created after the test directory was created to test against. This function allowed me to refine what sort of variables I would need to define and access while preparing to write fs_setcwd. All it was testing was that I was able to access the global variable created declared in mfs.c, and compare the value returned. The test was a pass if it returns root, or "/".
        - **test_fssetcwd** located in mfs_tests.c was created after the test directory was completed to test against. This function must be either tested again or rebuilt due to the revision of fs_setcwd as of 7/28. In theory, the test should pass as well, but it may expose some other problems I might have missed. Similar to getcwd, it passes in a valid absolute path based on the test directory, but it contains delimiters intended to make parsing and navigating the path more difficult. Test is successful if the function is able to successfully retrieve the valid absolute path minus the delimiters from getcwd. Prior to activating the shell, this gave me the environment to make most of the method functional.
        - **make_testdir** located in file_system_unit_test.c provided an actual file structure to test against. This was personally very handy for

testing against with getting the ball rolling for Milestone #2, although I’m not sure just how useful it will be for Milestone #3. It does provide a safe environment to test and play around with commands in, as the directories and files are already populated minus the metadata that will come with using the functions in the shell.

  - Originally it only went up two levels, it eventually got built up to five levels to get an idea of how to traverse multiple directories, helping to grasp where the information would need to be accessed from to move either forward or backward in the hypothetical file system structure.
  - The following data within each entry was modified:
    - isfile flag was changed if the file was intended to be a directory
    - Name was assigned to differentiate between entries. It was assumed that all entries in the directory would be populated.
    - With the first version of init_dir (Directory initialization function), block locations needed to be noted and stored as it was originally the return value of the function. Upon the revision of the function, the test needed to be modified to work off this redesign. This also provided the opportunity to understand the function on a deeper level, which was incredibly handy later on.

- **print_test_directory** located in file_system_unit_test.c. Purpose of this test was to print the file names located in the set of blocks loaded if something needed to be visually verified with a print statement. Only intended to load one level at a time, and not the entire test structure.
- **setdir_test** located in file_system_unit_test.c was one of the first attempts at implementing a test trying to get an idea of how to implement what the fs_setdir function would require. All it would really do was enter a mock fs_setdir function named fs_testsetcwd which would set a global variable located in the same c file, and return the result of a comparison between the path passed in, and the contents of the global variable. Although it didn’t provide much insight, it gave me enough to have some sort of clue of what variables I would need to implement in the completed version of fs_setdir.

- fs_is_file:
    - Returns the state of the appropriate field
    - Had to make sure the parse_path contract lined up
- fs_is_dir:
    - Returns the state of the appropriate field
    - Had to make sure the parse_path contract lined up

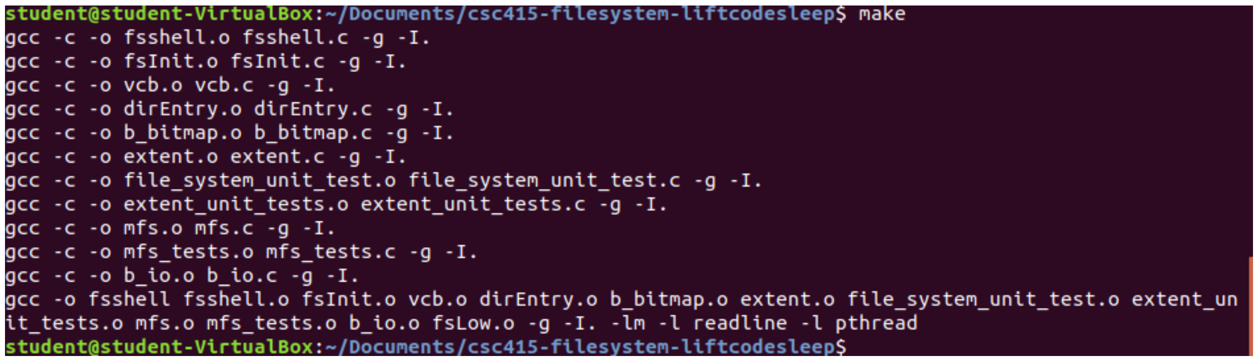
- I started working on the b_io functions for m3. I started with b_read thinking I could copy it in from A5 and translate the code to refer to the locations on disk as appropriate instead of in the input file. Logically it seemed to work but I realized I needed a new way to refer to the beginning of the blob of data I'd be opening, where A5 had a field "location" referring to the starting block of the file, I needed to map an adjusted way to contextualize the file pointer.
    - Digging into that I realized the location information I was looking for was made available by b_open, and without a b_open implementation, or at least the shell of one, I wouldn't have locations to reference as easily as I could. I shifted to working on b_open and again copied in my work from A5, this again seemed to fit well enough, but also created questions about the function of b_open. It wasn't clear to me what b_open would be responsible for, so I checked fs_shell.c and saw it was called by touch to create files, aha. Suddenly flags. I forgot about flags. This function will have many modes to account for and also they will need to be able to work in tandem as requested. At this point I need to read more to develop a better plan for our design.
    - For the flags we needed to have some way of tracking the permissions set for the file, so I added a field to the FCB for this purpose. We also needed to check for illegal flag usage like creating an existing file or truncating a nonexistent one or demanding write only as well as read write permissions on the same file. ReadOnly proved to be different because it has an undetectable value, 0. It is not so much a flag as the lack of one, and so I handled it as a default case and ignored what would otherwise be illegal flag usage with it. If you ask for read-only access to a file along with some other method of access, you get the other method of access.
- For fs_rmdir(), there was an issue that it could remove subdirectories that contained additional subdirectories or file contents. This would create an issue where the space that those subdirectories occupied would have no way to be accessed and freed.

    - To remedy this, after checking if the path specified is a valid directory, fs_rmdir() would then check if that directory contained any content by looping through each index and checking if the name of the content was not NULL.
- In b_write(), after writing anything to a file, a critical error occurred that seemed to corrupt the values of the file's directory entry although the contents appear to be written to the appropriate areas within the disk.
    - What had happened was that the extent that held the value of the file's directory entry (extent[0]) had been overwritten and thus when reading back the information, LBAread() was unable to find it.
    - To fix this, the extent[0].start and extent[0].count would never be allowed to change as these values contain the vital information to find the file data.

- When making write there was a problem with getting the lba for at the point in the extent because we save the first value of the extent to where it is in the direntery, this threw everything off by 1 extent. Although this bug was hard to find it was an easy fix by just offsetting the value by 1 in the extent.

- When writing move I hit some problems when switching the target and the destination, this is because I was going to child of the target and not the target itself.

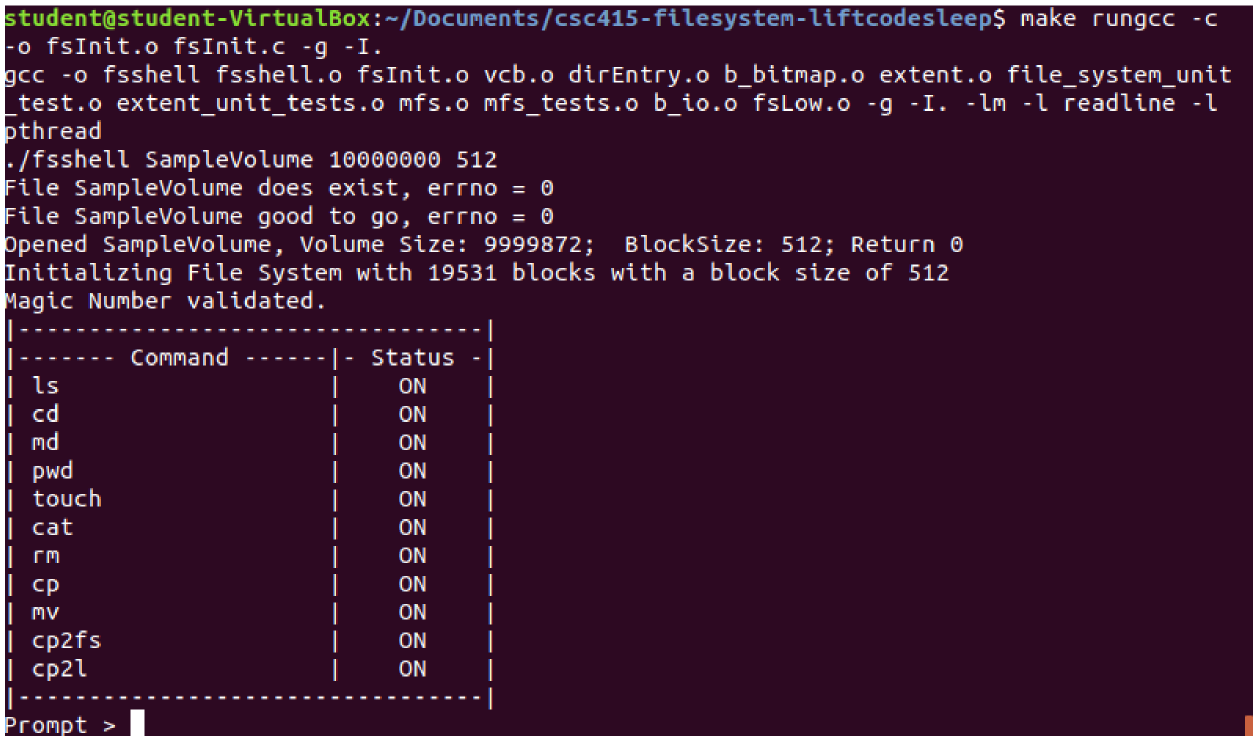
Screenshot of Compilation



```
student@student-VirtualBox:~/Documents/csc415-filesystem-liftcodesleep$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o vcb.o vcb.c -g -I.
gcc -c -o dirEntry.o dirEntry.c -g -I.
gcc -c -o b_bitmap.o b_bitmap.c -g -I.
gcc -c -o extent.o extent.c -g -I.
gcc -c -o file_system_unit_test.o file_system_unit_test.c -g -I.
gcc -c -o extent_unit_tests.o extent_unit_tests.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o mfs_tests.o mfs_tests.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o vcb.o dirEntry.o b_bitmap.o extent.o file_system_unit_test.o extent_un
it_tests.o mfs.o mfs_tests.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Documents/csc415-filesystem-liftcodesleep$
```

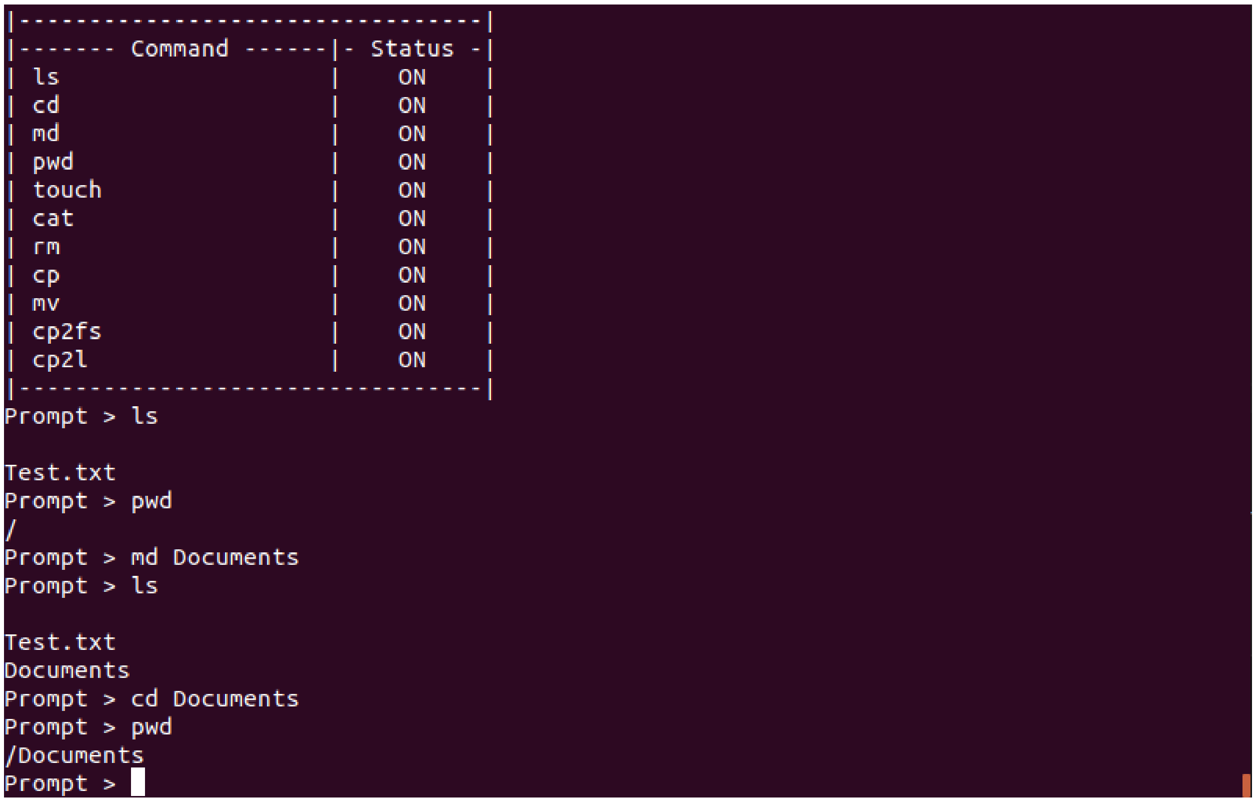
Screenshot of Execution

Make run



```
student@student-VirtualBox:~/Documents/csc415-filesystem-liftcodesleep$ make rungcc -c
-o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o vcb.o dirEntry.o b_bitmap.o extent.o file_system_unit
_test.o extent_unit_tests.o mfs.o mfs_tests.o b_io.o fsLow.o -g -I. -lm -l readline -l
pthread
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Magic Number validated.
|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|-------------------------------|
Prompt >
```

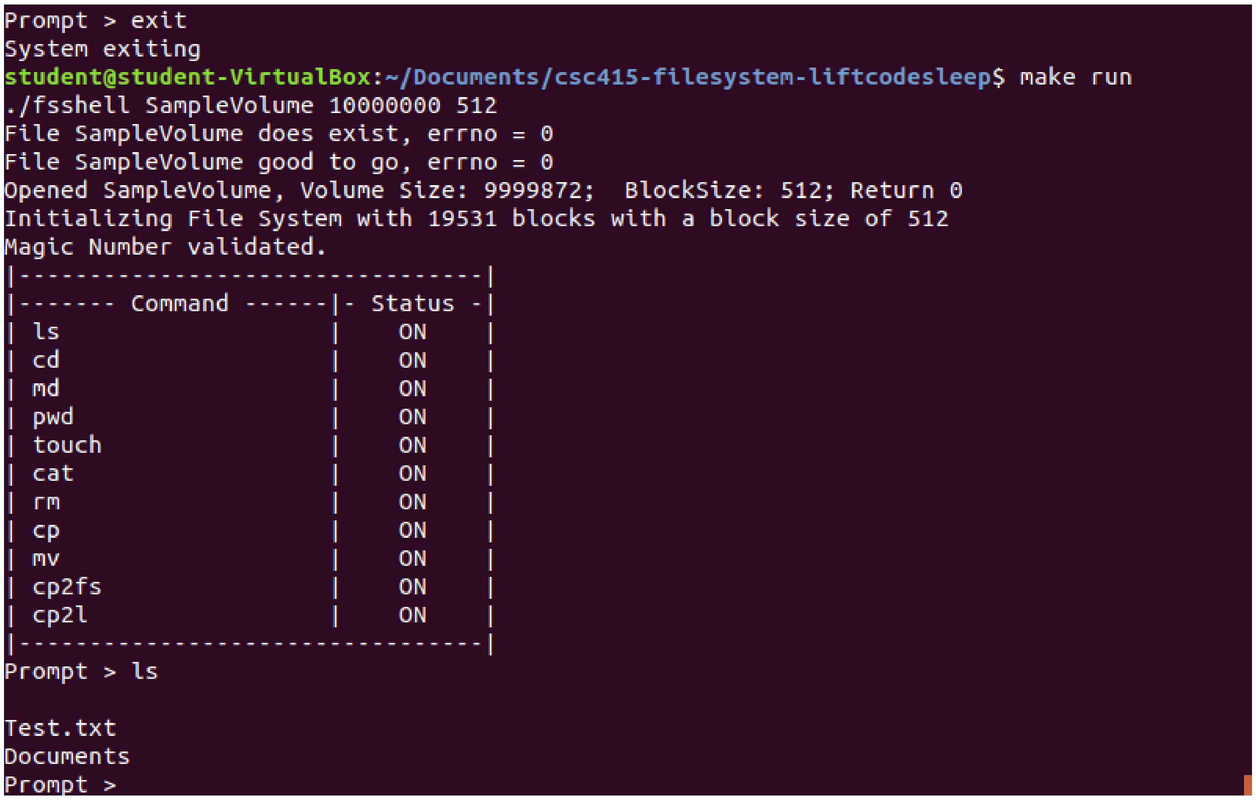
Demonstration of commands: ls, pwd, md, cd



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

Test.txt
Prompt > pwd
/
Prompt > md Documents
Prompt > ls

Test.txt
Documents
Prompt > cd Documents
Prompt > pwd
/Documents
Prompt > 
```

- Test1.txt was a file written for testing persistence. As it appears in the first “ls” call, it shows that writing a file into root is possible and accessible.
- As this is the start of the file system running, calling “pwd” should show the current directory is the root directory.
- Upon calling “md Documents” the Documents subdirectory is created. The “ls” call afterward reflects this change, as “Test.txt” and “Documents” are displayed.
- When calling “cd Documents” when the current directory is root (“/”) the parse_path() function is able to interpret the path as relative to the root and is able to move the current directory to Documents. This is further proven with the subsequent “pwd” call that displays the path from root.

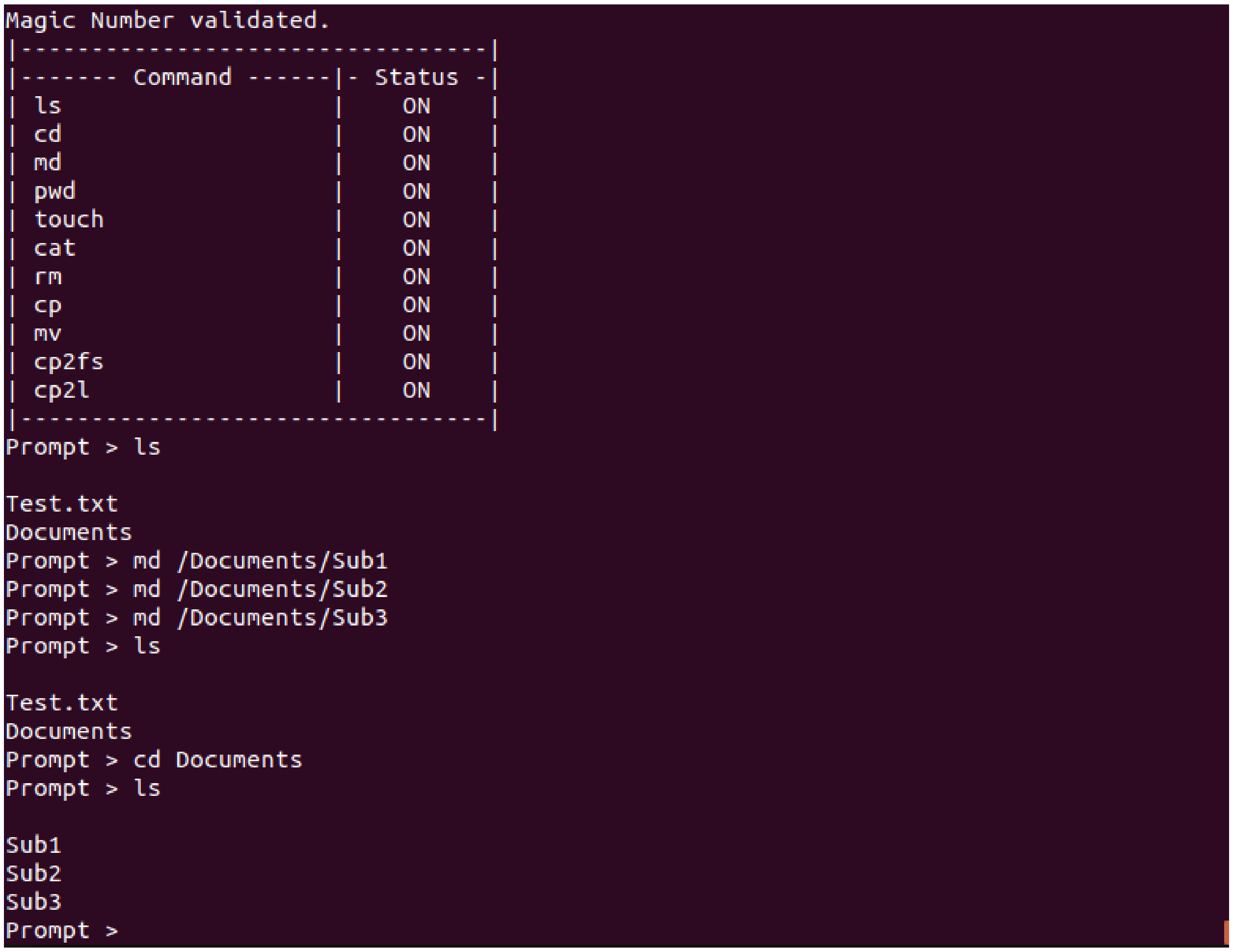
Demo for persistence



```
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-liftcodesleep$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Magic Number validated.
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

Test.txt
Documents
Prompt >
```

- With the previous addition of the Documents subdirectory, exiting and running the file system again shows that the Documents subdirectory persists within the file system.

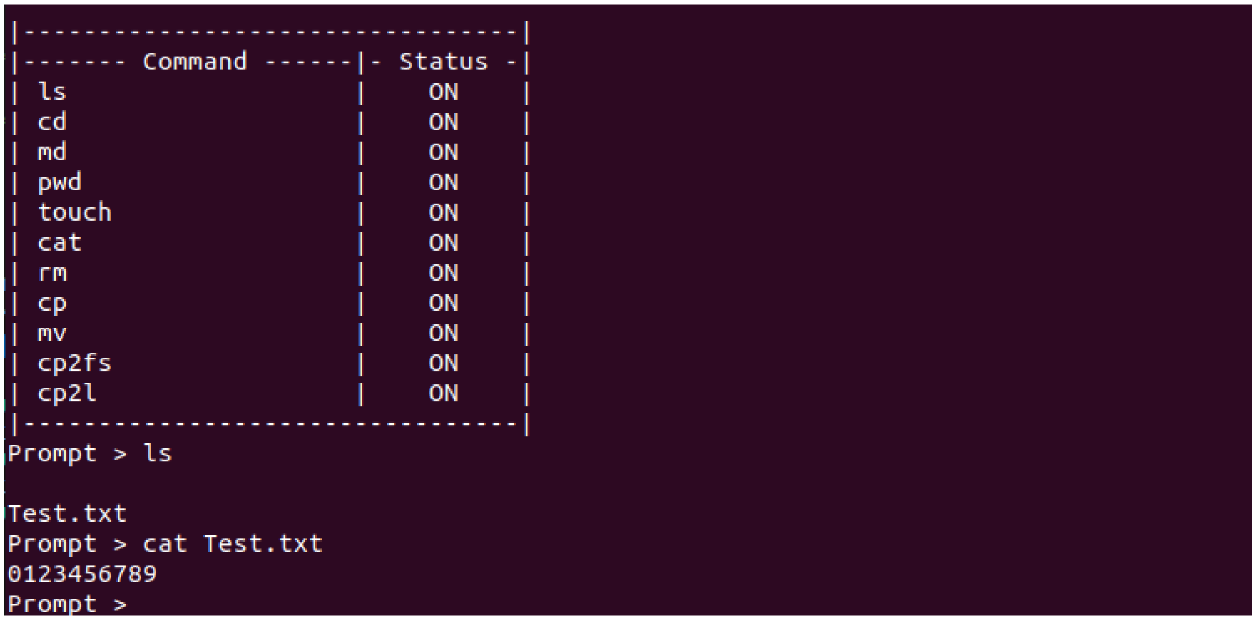
Demo for creation of subdirectories



```
Magic Number validated.
|--------------------------------|
|------- Command ------|- Status -|
| ls                  |    ON    |
| cd                  |    ON    |
| md                  |    ON    |
| pwd                 |    ON    |
| touch               |    ON    |
| cat                 |    ON    |
| rm                  |    ON    |
| cp                  |    ON    |
| mv                  |    ON    |
| cp2fs               |    ON    |
| cp2l                |    ON    |
|--------------------------------|
Prompt > ls

Test.txt
Documents
Prompt > md /Documents/Sub1
Prompt > md /Documents/Sub2
Prompt > md /Documents/Sub3
Prompt > ls

Test.txt
Documents
Prompt > cd Documents
Prompt > ls

Sub1
Sub2
Sub3
Prompt >
```

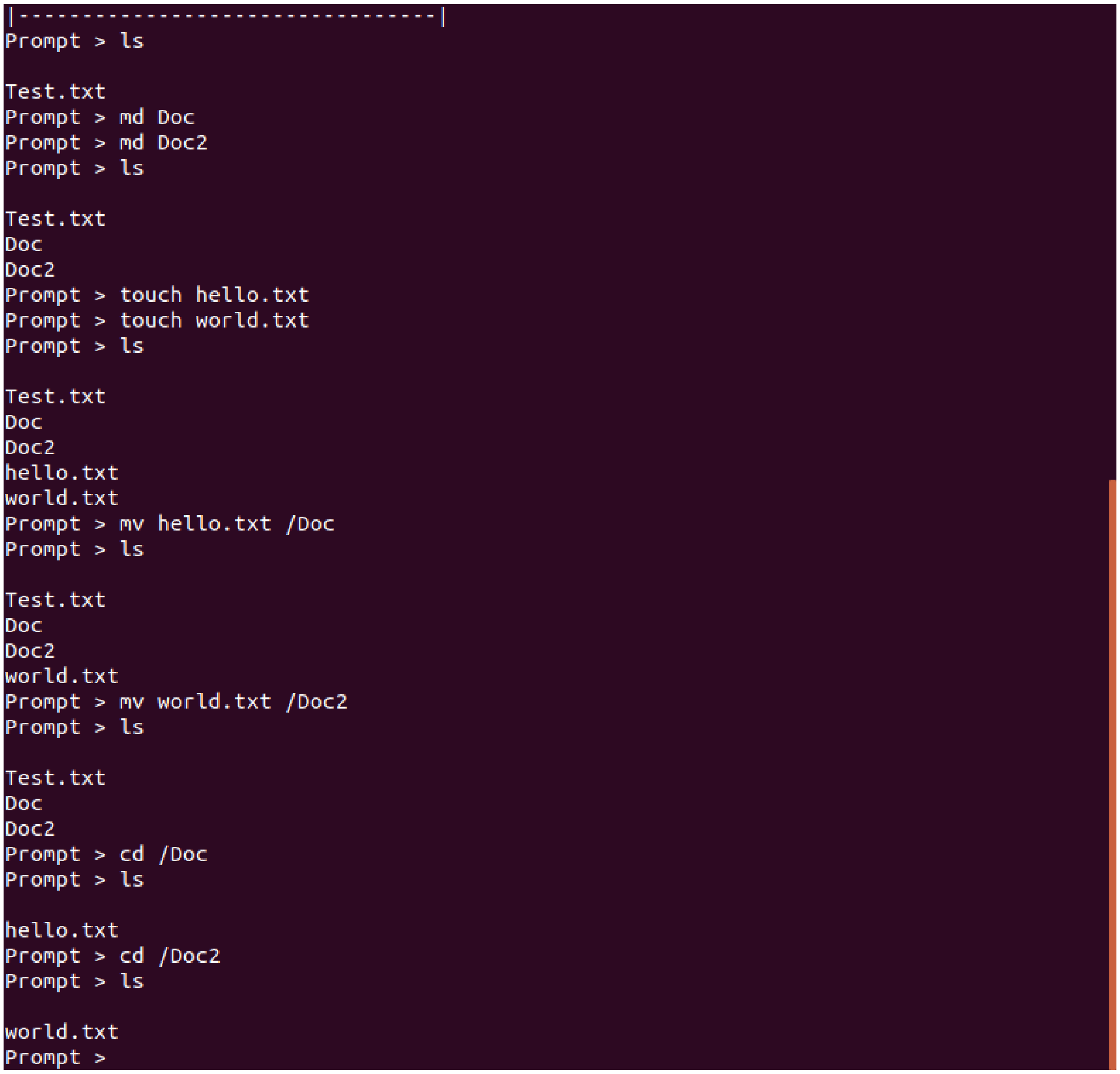
- The sequential “md” commands show how subdirectories can be created using the absolute path.
- When calling “ls” from the root directory, the subdirectories are not visible as they are part of the Document’s subdirectories.
- When changing the directory to Documents, calling “ls” displays the subdirectories created earlier

Demo for “cat” command


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

Test.txt
Prompt > cat Test.txt
0123456789
Prompt >
```

- Upon calling “cat Test.txt,” it reads the contents of the file, Test.txt, and displays its contents.

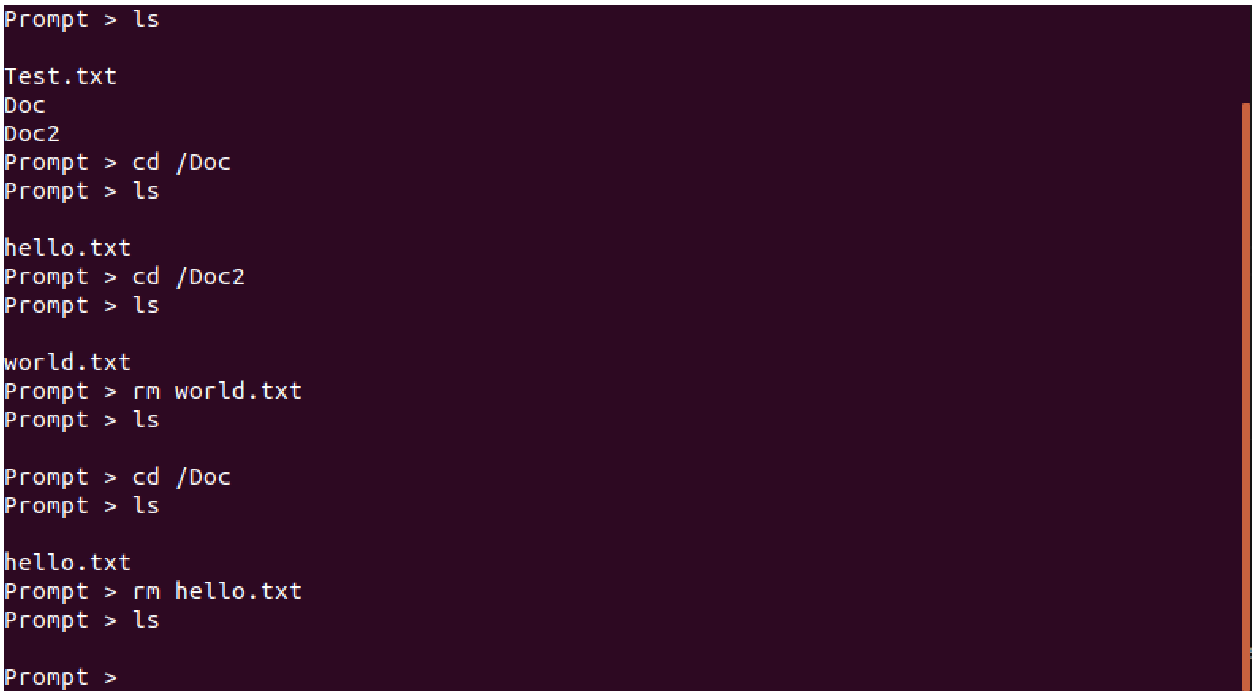
Demo for “mv” and “touch” command



```
|--------------------------------|
Prompt > ls

Test.txt
Prompt > md Doc
Prompt > md Doc2
Prompt > ls

Test.txt
Doc
Doc2
Prompt > touch hello.txt
Prompt > touch world.txt
Prompt > ls

Test.txt
Doc
Doc2
hello.txt
world.txt
Prompt > mv hello.txt /Doc
Prompt > ls

Test.txt
Doc
Doc2
world.txt
Prompt > mv world.txt /Doc2
Prompt > ls

Test.txt
Doc
Doc2
Prompt > cd /Doc
Prompt > ls

hello.txt
Prompt > cd /Doc2
Prompt > ls

world.txt
Prompt >
```

- Here, subdirectories Doc and Doc2 are created
- “hello.txt” and “world.txt” are files created within the root directory
- Using the “mv hello.txt /Doc” and “mv world.txt /Doc2”
- Going into the subdirectories Doc and Doc2, we can see hello.txt and world.txt

Demo for “rm” command



```
Prompt > ls

Test.txt
Doc
Doc2
Prompt > cd /Doc
Prompt > ls

hello.txt
Prompt > cd /Doc2
Prompt > ls

world.txt
Prompt > rm world.txt
Prompt > ls

Prompt > cd /Doc
Prompt > ls

hello.txt
Prompt > rm hello.txt
Prompt > ls

Prompt >
```

- Using the hello.txt and world.txt from before, “rm” is demonstrated
- Going into the directories Doc and Doc2 respectively and calling to remove each file, the “ls” command demonstrates the removal of both files.