File System- Milestone 3

Group Name: File Explorers

Team Github Repository: [CSC415-2022-Fall/csc415-filesystem-HadiRam]()

Team Members List:

Name	Student ID	Github ID
Manish Sharma	922220894	mscchapagain
Hadi Ramlaoui	922155413	HadiRam
Kimheng Peng	921847378	TotifyKH
Rajan Shrestha	922223741	rajshrestha015

A description of our file system:

Our file system consists of a series of different functions that facilitate the initialization, creation and utilization of a file system. Our team created the file system using the C programming language, and wrote code that pertains to the essential aspects of a file system such as the volume control block, bitmap free space structure, directory entries, as well as any functions that facilitate the use of commands. Our file system shall support all of the important functions that exist within the file system of the Linux operating system.

The supported commands include:

Ls - used to list all contents within a directory.

Md - used to make a directory.

Rm - used to remove a file or directory.

Pwd - used to print the working directory.

Touch - used to create a file.

Cp2l - copy files from test file system to linux file system

Cp2fs - copy files from linux file system to test file system

Cd - used to change the current working directory.

Pwd - used to print the working directory.

mv - Moves a file - source dest

cp - Copies a file - source [dest]

NOTE: ***cp*** *command works only on the same directory. It can't copy a file to another directory.*

Cat - displays the contents of a file

History - prints out history of commands

Help - prints out information about commands

In order to ensure a functional file system, our team also had to complete functions relating to opening, reading, writing, seeking and closing a file. If the open function flag is set to O_CREATE, this means a new file will be created, which will require the initialization of an extent table which maps out where the file is located in the logical block array (LBA). Each row within the extent table has a specified location relating to the starting block of the LBA, and a count which represents the amount of occupied blocks. The extent table itself takes up one block of space and each file has an extent table.

Within the following section of the write up, all of the essential functions and code our team wrote will be described, relating the logic and purpose behind the code, as well as displaying code snippets. The code will be described and displayed in chronological order of completion.

Our Approach to building the file system -

The initial steps our team took was to ensure that the volume control block, the free space management system and the root directory were written.

Below is the description of our volume control block struct, which is used during the initialization of our global VCB variable.

The size of our VCB struct is 24 bytes, the struct contains several variables, first a long signature, which is our unique “magic number” which is used to check if a VCB has already been initialized or not. Next we have several integers to represent the amount of blocks, the size of a block and the amount of free space. The VCB also has a pointer to our freeSpaceBitMap, this pointer is valid only during runtime but has no meaning on the disk. There is also an int to store the bitmap byte size and lastly an int to store the location of the root directory.

```
//Size of VCB is 24 bytes
//Volume Control Block struct
typedef struct VCB{
                //unique magic number
                long signature;
                int numBlocks;
                int blockSize;
                int freeSpace;
                //Memory Pointer allocated at runtime
                unsigned char* freeSpaceBitMap;
                int bitMapByteSize;
                int RootDir;
} VCB;
```

We have used bitmap to represent the free space structure in our file system. Our free space map starts at index 1 of the logical block array of our disk and ends at index 5. Since our disk is of size 10 MB we have 19531 bits to represent each block of size 512 bytes in the disk. We decided if the bit is 1 it is in used/allocated state and if the bit is 0 then it is free/available.

To initialize the free space map we created a helper function initBitMap:

void initBitMap(char* bitMapPointer, u_int64_t blockSize)

This function takes a pointer to the bitmap and size of a block. It then sets the first byte or 8 blocks to 0xFC which means that the first 6 blocks are already in use for VCB and bitmap. It then loops through the second byte to the size of 5 blocks and sets it to 0x00 to represent that they are in free state. And after that we write back to the disk using LBAWrite.

We have created other helper functions to manage our free space structure. They are listed and explained below:

int bitCounter(unsigned char myByte);

bitCoutner takes in a byte and counts the amount of bits within that byte that are set to 1. This will return an integer from 0 to 8.

unsigned char mask(int offset);

Mask takes in an “offset” integer from 0 to 7 and gives the byte where all bits within that byte are set to 0 except for the bit at the position of the “offset”.

int freeSpaceCounter(unsigned char myByte);

This function will count the bits that are 0 and return the value. In another word, it gives you the remaining number of free bits in a byte.

int checkABit(unsigned char myByte, int offset);

This function takes in a byte as well as an integer offset and checks whether or not the bit at the offset position of within the given byte is set to 1, if that bit is 0 the function returns 0, if that bit is 1 the function returns 1.

int checkForConsecFreeSpace(unsigned char myByte, int count);

This function takes a byte and desired number of bits that are free consecutively as count as a parameter to check if there is consecutive free space in the bitmap. If the byte has consecutive free space it returns 1 which means true.

void setABit(unsigned char* bitMap, int offset);

This function takes in a bitMap and an integer offset, and changes the specific bit within the bitMap at the offset from 0 to 1.

void clearABit(unsigned char* bitMap, int offset);

This function takes in a bitMap and an integer offset, and changes the specific bit within the bitMap at the offset from 1 to 0.

int getConsecFreeSpace(unsigned char* bitMap, int bitMapSize, int numOfBlocks);

This function takes a pointer to bitMap, size of the bitmap, and desired number of blocks as contiguous blocks. Inside this function, first we calculate the minimum amount of free bytes we would need, based on the desired number of blocks, then we have a loop that finds the first free byte within the bitmap, and once that byte is found, it adds to the count of free consecutive bytes, it then continues to the next byte within the bitMap to check if that is also free, and continues to add to that count if that is true. Once the count of free consecutive bytes is equal to the minimum amount of free bytes needed, we have another loop to check for and utilize any free bits at the previous byte, before the first free byte. If there are free bits within that previous byte, then we will then set the last bit that is 0 to the first free block, otherwise the first free block will simply be the first bit of the first free byte. This function then sets the bits to 1 for the bits that are allocated. At the end, this function returns the index to the first free block.

int releaseFreeSpace(unsigned char* bitMap, int location, int size)

This function takes in a bitmap, a location and a size. This function sets all the bits to 0, starting at specified location in the bitmap, stopping once the specified size is reached.

void updateBitMap(unsigned char* bitMap)

This function takes in a bitmap and writes the bitmap to disk using the given LBAwrite() function with the bitmap, BITMAP_SIZE and BITMAP_LOCATION as the parameters. This will ensure that the latest bitmap is written to disk.

Our directory entry struct is of size 64 bytes. It contains a variable pointer to char to store the name of the directory entry. An int variable to store the location of the entry. An int variable to determine the size of the entry. An int variable dirType to know which state is the entry at, if it's free it will be set at -1, if it is a file then it is set to 0 and if it is a directory itself then it's set to 1. An int to represent extent location, We have used two time_t variables for time created and time last modified.

```
//Directory Entry struct with the size of 64 bytes

typedef struct dirEntry{
        char name[32];
        time_t created;
        time_t lastModified;
        int location;
        int size;
        //Free state is -1 || is a file directory entry 0 || is a directory 1
        int dirType;
        int extentLocation;

} dirEntry;
```

Now our team implemented the main functions that will be used for initializing our file system, initBitMap(), initFileSystem() and exitFileSystem(). These functions were written in the fsInit.c file, as well as defining the “MAGIC_NUMBER”.

void initBitMap(char* bitMapPointer, u_int64_t blockSize)

The initBitMap function takes in a bitMapPointer and a blockSize. The function sets the first byte (unsigned char) within the bitmap at index 0 to the hex value 0xFC. This byte is the value 1111 1100, which shows that the first 6 blocks of space within the bitmap are used by the VCB and the bitmap itself. NExt the function simply initializes all other bytes within the bitmap to free space. Finally, the function writes the initialized bitmap onto the disk.

int initFileSystem (uint64_t numberOfBlocks, uint64_t blockSize)

This function takes in the number of blocks and the block size and ensures that the file system is initialized with that amount of blocks. The function first checks if the VCB has been initialized by comparing it to the magic number. If the VCB has not been initialized, all the variables relating to the VCB will be initialized. Including the signature, number of blocks, block size, bitmap and related bitmap variables, setting all the directory entries to the free state and then initializing the root directory entry and the parent directory entry, and lastly updating the bitmap and VCB onto the disk. If the VCB has already been initialized, the above steps will be skipped and the bitmap will be loaded into the VCB bitmap pointer. On a successful run, this function shall return 0.

void exitFileSystem ()

This function frees the freeSpaceBitmap in the VCB.

The next steps our team took was to ensure that the following functions were fully implemented within the file mfs.c:

- fs_setcwd
- fs_getcwd
- fs_isFile
- fs_isDir
- fs_mkdir
- fs_opendir
- fs_readdir
- fs_closedir
- fs_stat
- fs_delete
- Fs_rmdir

Before the above functions could be implemented, some helper functions had to be implemented, the most important one being Parse Path as it needed to be used in almost all of the other functions within mfs.c.

Helper functions:

pathInfo* parsePath(const char *pathname)

parsePath is the bread and butter function of our file system. This function takes in the pathname, and returns the pathInfo struct which contains the directory entry pointer to the file/directory if it exists, an integer which tells us whether the path exists or not, and the simplified absolute path. The first thing we needed to do was check whether or not the path is relative or absolute. If it's relative, we append the current working directory path to it in order to make an absolute path.

Next, we tokenize the path. For this we use strtok with the delimiter “/” and we created an array to store the tokens. This array acts like a stack so that during our strtok while loop, we can check if the token is equal to “..”, we would pop the top of the stack. We also have to take care if the path is just “/” which would give us a null token.

After having an array of tokenized strings, we would enter a while loop that processes those strings by checking from the root directory if those strings exist. If a string doesn’t exist and it’s not the last element, the path is invalid returning the value -2. If a string does exist and is a directory, we load that directory and process the next string. If a string doesn’t exist, and it is the last element, that means the path is valid and we will return the value -1. If the string exists and it's the last element, we will return the value as their index in the parent directory. We also return the simplified absolute path without the “.” and “..”.

char * getLastPathElement(const char *pathname)

This function takes in a path name and returns the last element of the path, for example “/hadi/desktop/folder” passed in would return “folder”.

char * getParentDirectory(const char *pathname)

Function that takes in a path name and returns the path excluding the last element, for example “/home/foo” passed in would return “/home”. Another example would be if “/home” was passed in, the function would simply return “/”.

void initGlobalVar()

Updates the global variables cwdPath and cwdEntries.

void freeGlobalVar()

It frees the memory of the pointer to global variables: cwdPath and cwdEntries.

Primary functions:

fdDir * fs_opendir(const char *pathname)

This function takes in a path a file path, and if the path is valid, returns a file descriptor with the correct record length, directory start location, directory size and directory entry position set to 0 and file index set to 0. Otherwise returns NULL.

struct fs_diriteminfo *fs_readdir(fdDir *dirp)

This function takes in a dir pointer, and mallocs a diritemInfo struct and initializes the variables within that struct such as the name, file type and record length based off of the passed in dir pointer. The function then returns the initialized struct.

int fs_closedir(fdDir *dirp)

This function frees the passed in dir pointer and sets it to NULL.

char *fs_getcwd(char *pathname, size_t size)

Returns the name of the path of the current working directory.

int fs_setcwd(char *pathname)

This function takes in a path name and first writes the current working directory back to the disk, then if the pathname is valid, load all the directory entries into the global variable cwdEntries and set the current working directory path to the past path name. On success return 0 and on failure return -1.

int fs_mkdir(const char *pathname, mode_t mode)

Function used to create a directory at a specified path. The function first gets the parent of the passed in path and sets that to the current working directory. The function then loops through the entries within the parent path in order to find an entry in a free state. Once found, set the name of that directory entry in the parent path to the last element of the passed in path, set the dir type to 1 to represent a directory, and set the location to the area within the freeSpaceBitmap where there is enough consecutive free space and update the bitmap. Also initialize time created, last modified and size variables. Next, initialize the directory entries of the newly created directory, firstly by setting all entries to a free state and then initializing the root and parent directory entries of the new directory.

int fs_isFile(char * filename)

Function used to check if the passed filename is a file and not a directory or null. Returns 1 if it is a file and 0 otherwise.

int fs_stat(const char *path, struct fs_stat *buf)

Function checks if path is valid, if so updates the variables within the fs_stat buffer with the following information about the path - access time, size, create time, modified time and then returns 1 on success and -1 on failure.

int fs_rmdir(const char *pathname)

Function used to remove a directory. Function will fail and return -1 in the following cases: the passed pathname is the root, as this directory can not be removed, the passed pathname leads to a file and not a directory or the passed pathname leads to a directory that is not empty. If these cases are not hit, the directory will be set to a free state, meaning the following variables will be changed - name set to the null character, dirType set to -1, location set to -1, size set to 0, extentLocation set to -1. The function then writes the changes onto the disk and returns 0.

int fs_delete(char* filename)

This function is similar to the above function “rmdir” but for a file instead. Once the function ensures that the passed filename is a valid file, it changes the directory Entry. Name of the directory Entry at that position is changed to the null character, the dirType to -1(free state), the location to -1(free state). Then, it calls the helper routine releaseFreeSpace to set the bits in the bitmap as free state. It then updates the bitmap and updates the size of the directory.

It then writes the changes to the disk and returns 0 on success.

int fs_move(char* src, char* dest)

Function used to move a file from one directory to another. The passed variable “src” is the file that is to be moved, and the passed variable “dest” is either the directory where you want the file to be moved to or a new file name that you would like to use. If the source is a valid file and the destination is a valid directory, the function will create a new file in the directory, with all the information of the source file, and delete the original source file. If the destination is another file, it will have the effect of essentially renaming the file.

The next functions were the key file input and output operations within b_io.c - Open, seek, write, and close.

Before we were able to implement these functions, we needed to create functions to initialize and manage an extent table, which maps out where a file is located in the logical block array (LBA). We decided to add these functions to a separate file called extent.c.

void initExtentTable(int extentLocation)

Initialize an extent table with the specified number of extent rows, with each row have a location and count initialized to -1. It takes the location of the extent in bitMap.

int getExtentTableSize(extent* extentTable)

Function used to loop through extent table rows, incrementing the size for every row that has a location value that is not -1. Returns the size.

void mergeNewRow(extent* extentTable)

Function used to add the count of the final row to the count of the row at i, if the row at i’s location + count is equal to the location of the final row and then set the final row location and count to -1.

extent* getExtentTable(int extentLocation)

Function takes the location of extent in free space bitMap and returns the extent table at the passed in extent location.

int addToExtentTable(extent* extentTable, int location, int count)

Function used to loop through the extent table, finding the first row location equal to -1, setting that row to the passed location and count, and then calling the function mergeNewRow. Returns 0 on success and -1 if the extent table is out of rows.

int getLBAFromFile(extent* extentTable, int location)

Function that returns the location within the extent table that is equal to LBA position. This function takes in a relative block location and iterates through the extent table to figure out which LBA position we are at.

void releaseFile(int extentLocation)

Function used to release files by calling the releaseFreeSpace function on each row of the extent table, and setting the location of each row to -1. Function will then update the vcb bitmap as well free all the blocks that are allocated for this file.

void releaseFreeBlocksExtent(extent* extentTable, int location)

Function used to release free blocks within the passed extent table. This function takes in the extentTable and a block position in order to free them in the extent table.

void updateExtentTable(extent* extentTable, int extentLocation)

Function used to update the extent table by writing it to the disk.

void printExtentTable(extent* extentTable)

Function used to print all the rows of the extent table.

Key file input and output operations within b_io.c - Open, seek, read, write, and close

b_io_fd b_open (char * filename, int flags)

The open function takes in a filename and a flag.

First path was checked using the parse path function.

If the file did not exist and the flag passed was O_CREAT, the open function must create a file with size 0 bytes. First, a free directory entry is searched and initializes the directory entry with name, dirType set to 0(because it's a file), adds the extent's location, location of the directory entry, and updates the time created and last modified. After that, it updates the extent table associated with the file. Then, it updates the current working directory size since a new dir entry is added. The function also checks if it's a root directory or not. If its root directory, the ".." directory size is also updated. If it's not a root directory it updates the parent directory size. Then, it writes back to the disk using LBAWrite. And updates the cwdEntries (global variable). It also needs to initialize the fcb struct elements and return the file descriptor.

If the file exists, the open function initializes elements inside the struct fcbArray. After that, if the flag is O_TRUNC and O_RDWR or O_WRONLY, the file will be truncated to the length of 0. It then updates the current working directory and returns the file descriptor.

int b_seek (b_io_fd fd, off_t offset, int whence)

b_seek repositions the file offset of the open file description associated with the file descriptor fd to the argument offset according to the directive whence as follows:

SEEK_SET: it sets the file offset to the given offset

SEEK_CUR: The file offset is updated to current location plus the given offset.

SEEK_END: The file offset is updated to the size of file plus given offset bytes.

It returns the file offset which is inside the fcbArray[index at file descriptor] struct.

int b_write (b_io_fd fd, char * buffer, int count)

b_write function writes into the caller’s buffer with the given count bytes amount of data associated to the file descriptor of the file. It returns the amount of bytes written to the caller’s buffer.

Implementation of the b_write:

First the permission is checked; whether the file has write permissions or not.

It tracks the remaining bytes in the file so that if it's not enough to write, it can call another helper function getConsecFreeSpace to get free space from the bitMap. And the extent table associated with the file is updated. After it makes sure the file has space to write in it is set to loop until the given count (bytes to write) is greater than zero.

In the loop, it calculates the number of bytes left in the buffer associated with the file’s fd. If this buffer has some data, it is written to the file in the disk. Else, the empty buffer of the file is filled with the caller’s buffer and it is written back to the disk in the location of the file where it was last left off. It then updates the tracking variables such as buffer position tracker, block location and others that are related to it.

Once it writes every byte from the caller’s buffer, it updates the size of the file, updates the extent table and writes it back to the disk with this updated info. Finally, it returns the bytes written.

int b_read (b_io_fd fd, char * buffer, int count)

The purpose of the b_read function is to read in the requested amount of bytes, represented by the passed integer “count”, into the caller's buffer. The bytes of data that is read in is relating to the associate file descriptor of a file, which is also a passed in variable.

The implementation of this method is as follows:

First, initialize all variables needed to keep track of:

Caller buffer’s offset (callerBufferOffset), amount of bytes to memcpy (copyAmount), remaining size of file(remainingFileSize), remaining bytes that the caller requested(neededBytes), current block (fileBlockIndex).

After these variables have been initialized, the function first checks if neededBytes is greater than remainingFileSize, if this is the case, set neededBytes to the remainingFileSize as the maximum possible amount of bytes the function can read in will be whats remaining in the file, regardless of the requested count being greater.

Within a while loop, the first line of code will recalculate the remainingFileSize, by subtracting fileSize from the fileOffset, which is updated within the loop.

The first if statement will check if the index is equal to 0 and the needed bytes is greater than or equal to one block, if that's the case the function will LBAread directly into the callers buffer, the number of blocks needed as long as the blocks needed is greater than 0. The number of blocks needed can be calculated by dividing the neededBytes by the chunk size.

If the first if statement is false, the function will then check if the index is equal to zero, if that's the case, the function will LBAread into the fcbBuffer, then check if the neededBytes is larger than the remaining space in the fcbBuffer, if that's the case, set the CopyAmount to the chunkSize minus the index, and increment the fileBlockIndex, otherwise set the copyAmount to the neededBytes. Now the function can memcpy into the caller buffer at the callerBufferOffset, from the fcbBuffer at the index, for the copy amount. Then update the fileOffSet, CallerBufferOffset, and index. Lastly subtract the copyAmount from the needed bytes. This process will repeat in the while loop, until the neededBytes hits 0, then the callerBufferOffset will be returned which is equivalent to the memcpy amount.

int b_close (b_io_fd fd)

b_close takes the file descriptor and cleans up the memory taken by the other file functions such as open, read and write.

It checks if it ever allocated more blocks for the file than needed. For example, if the file was allocated 50 blocks and it only used 2 blocks, then the rest 48 blocks are freed by the b_close.

After freeing unused blocks, it updates the directory entry associated with the file.

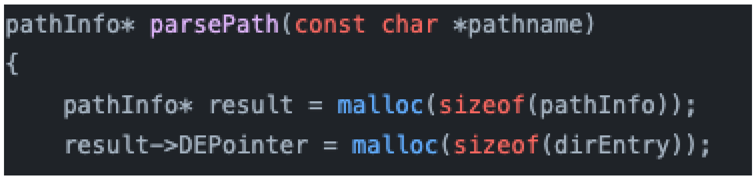
And it cleans up the memory because safety first.

On success it returns 0.

Issues Our Team Solved:

When our team implemented functions that create and return structs, our code did not malloc the required amount of space needed for the struct itself. This would cause the data to get corrupted whenever a function that creates a struct was called multiple times, as the memory was being shared.

In order to solve this issue, we ensured that before any code that creates a struct, we malloced the required space. The following screenshot is an example of how we fixed this issue within the parsePath function.

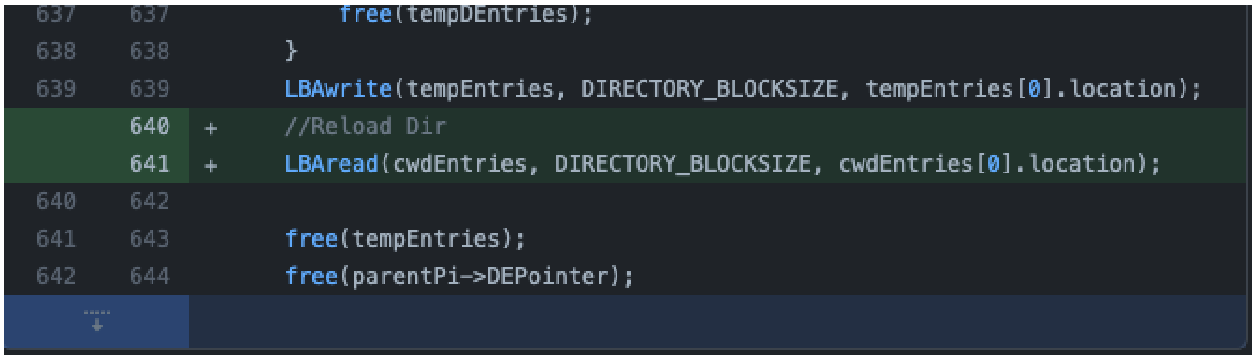


```
pathInfo* parsePath(const char *pathname)
{
    pathInfo* result = malloc(sizeof(pathInfo));
    result->DEPointer = malloc(sizeof(dirEntry));
```

As can be seen, for the pathInfo and dirEntry struct, a malloc was first called with the appropriate amount of memory.

Another issue our team faced was a bug with the **rm** command on an empty directory. So, whenever we removed a directory, it wasn't fully removed because we could still **cd** into that directory even though it was not showing up after using the **ls** command.

To fix this bug, what we realized was we had to update or reload the current working directory that is a global struct variable. Once that was done, the **rm** command started to work perfectly.



```
            free(tempDEntries);
        }
        LBAwrite(tempEntries, DIRECTORY_BLOCKSIZE, tempEntries[0].location);
+       //Reload Dir
+       LBAread(cwdEntries, DIRECTORY_BLOCKSIZE, cwdEntries[0].location);

        free(tempEntries);
        free(parentPi->DEPointer);
```

Testing some of the functions was another issue our team faced. Sometimes, we would create a dummy file with dummy data and call our functions to check if it's working properly.

Time constraint was another issue. As this project is really big, our team wouldn't have completed it without the given extension.

Coordination with teammates. Since this project was big and we had to plan a lot for the implementation of our idea, there would be issues among the team members on which one's to follow. When we initially began milestone 1 our process was to meet all together and code within the meeting, but by milestones 2 and 3 we realized that we would not be enough to code only during the meeting but also individually.

Debugging. Debugging was really tough for us. And we were also using virtual machines for the project, we were just using Visual studio code for writing code which doesn't have that much feature as on the IDE's. Visual Studio Code is sometimes laggy and buggy when used on the virtual machine which makes the process of coding and debugging slightly more difficult. Majority of our time was consumed during debugging as well. The only tools we used for debugging were printf and Hexdump.

Segmentation fault was an evergreen issue we faced while working with this project. We would never know the exact issue what caused it and had to debug using printf() all over the place to find out the root cause of the segmentation fault.

Github Issues: As we were using Github to do the project, we encountered a number of issues like when two persons were working on the same file pushing and pulling problems would occur. For that we had to do a number of google searches about the error. We learned a lot about github during this project. In the future, we decided to use different branches to work so that it wouldn't conflict with other's work.

Detail of how your driver program works

The fsshell.c is our modified program driver. Most of the codes are implemented for us already since we only have to do the move command. fsshell.c utilizes all the other functions we implemented in order to create an interface for our file system. There are a handful of commands for our file system that are implemented here including ls, cp, mv, md, rm, cp2l, cp2fs, cd, pwd, touch, and cat.

The ls command makes use of the fs_diriteminfo and fdDir struct as well as a displayFile helper function that uses the fs_opendir, fs_readdir, and fs_closedir functions. displayFile uses these functions in order to print out all the files and directory that is in the requested directory.

The cp command takes in 2 arguments which are the source and destination. In this function, it uses b_open to open both files to read from the source file, and write to the destination file. At the end, b_close is called on both files.

The mv command takes in 2 arguments which are the source and destination as well; however, the cmd_mv function will call the fs_move function to do most of its work. The fs_move function basically copies the file to the destination directory/file, and deletes the original directory entry in its source directory.

The md command takes in 1 argument which is the path and the name of the new directory. This cmd_md function calls the fs_mkdir function which will create a new directory in the directory specified in the path.

The rm command takes in 1 argument which is the name of the directory or path. This command function will call fs_delete or rmdir whether or not if the argument is a path or directory.

The cp2l command takes in 2 arguments. The first argument is the path in our file system while the second argument is the path in our linux. This command function will call b_open and b_read on the first argument to copy the file content using open, and read to create a file in the linux system.

The cp2fs command takes in 2 arguments. The first argument is the path in our linux while the second argument is the path in our file system. This command function will call b_open and b_write on the second argument to copy the file content using open, and write to create a file in the file system.

The cd command takes in 1 argument which is the path. This command function will call the fs_setcwd function to set the current working directory.

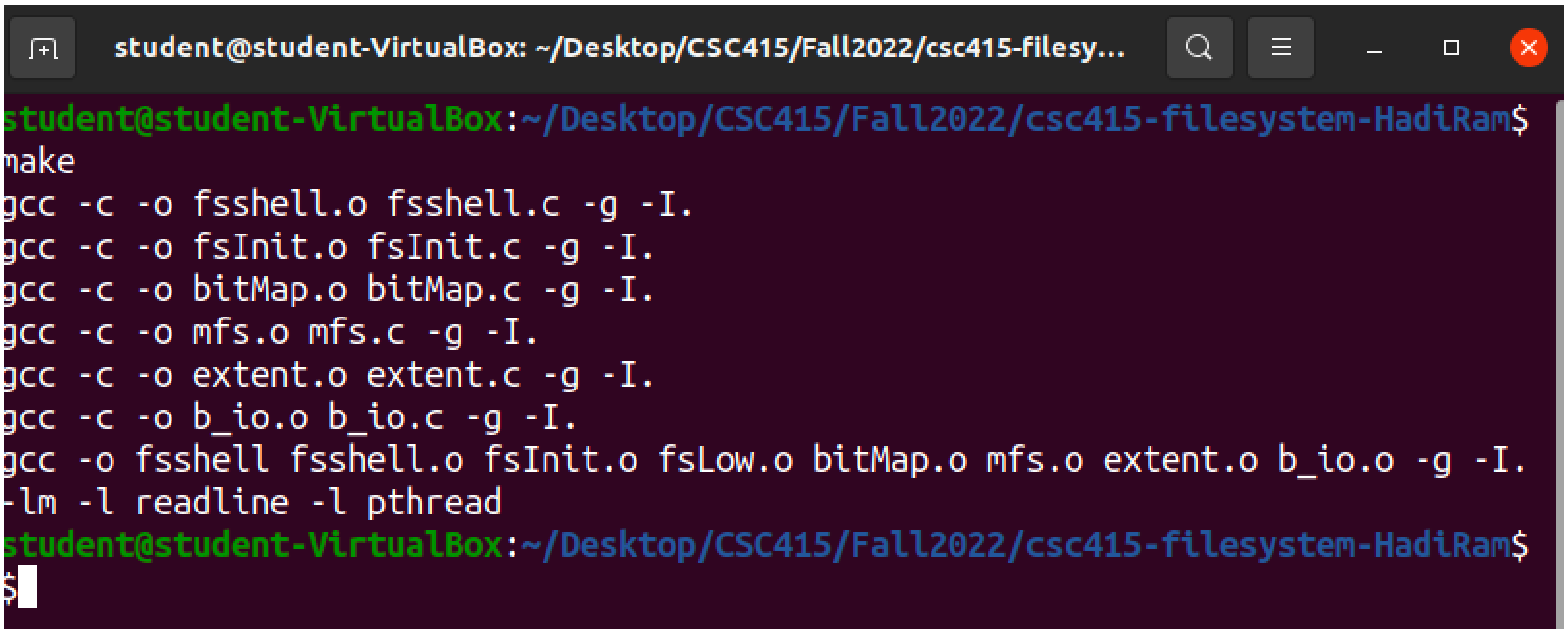
The pwd command uses the fs_setcwd command to get the current working directory path.

The touch command takes in 1 argument which is the filename. This command function will call the b_open command with the flag O_CREATE which will create a new file.

The cat command takes in 1 argument which is the filename. This command function will call b_open and b_read in order to read the file and print it out onto the console.

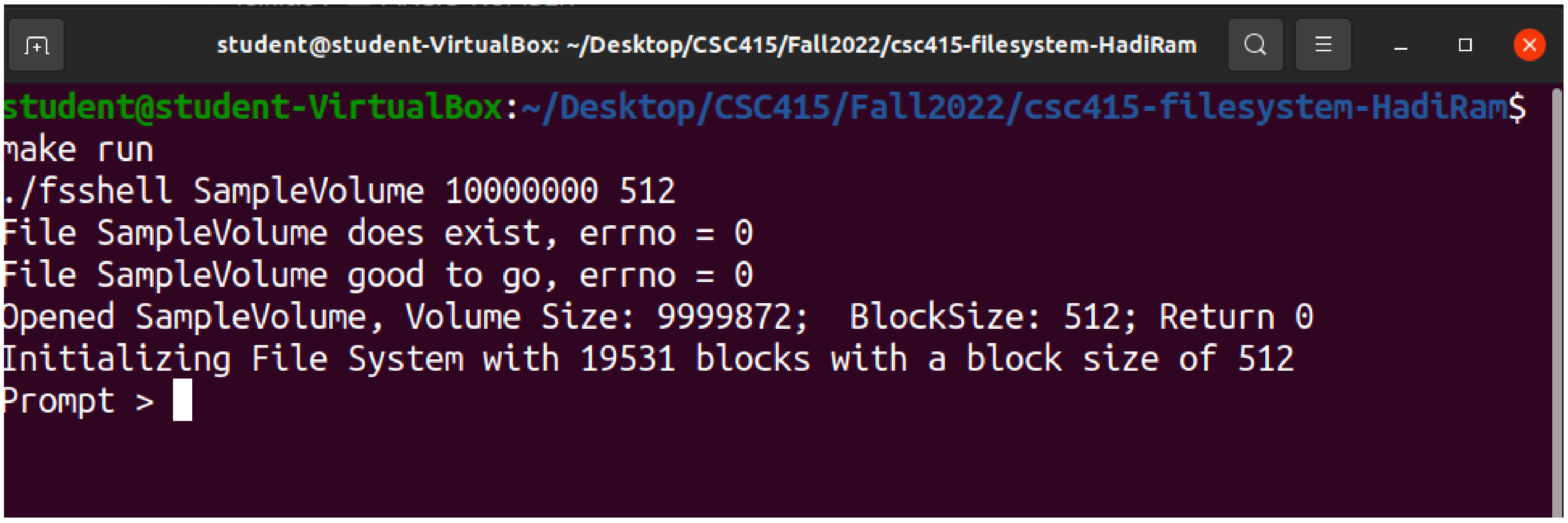
Screenshots showing each of the commands

Showing **make** and **make run** working:



student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesy...

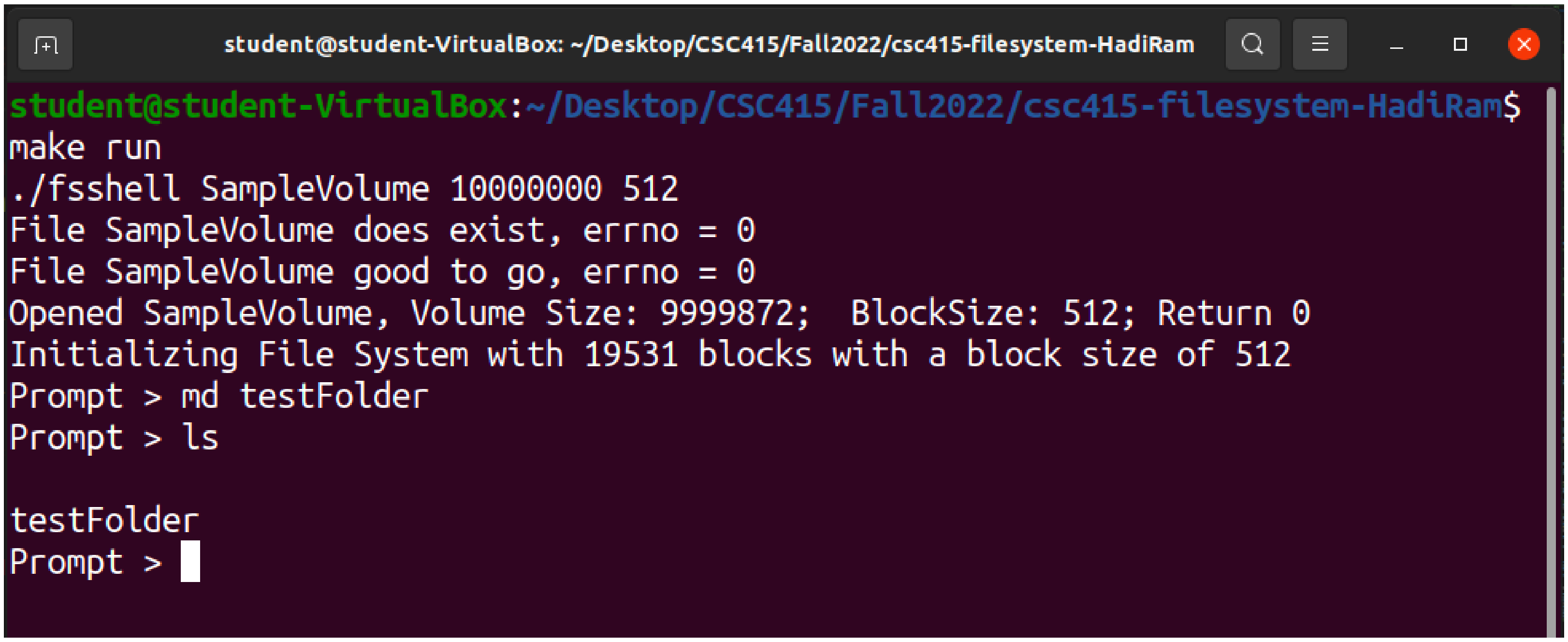
```
student@student-VirtualBox:~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam$
make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o bitMap.o bitMap.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o extent.o extent.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLow.o bitMap.o mfs.o extent.o b_io.o -g -I.
-lm -l readline -l pthread
student@student-VirtualBox:~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam$
$
```



student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

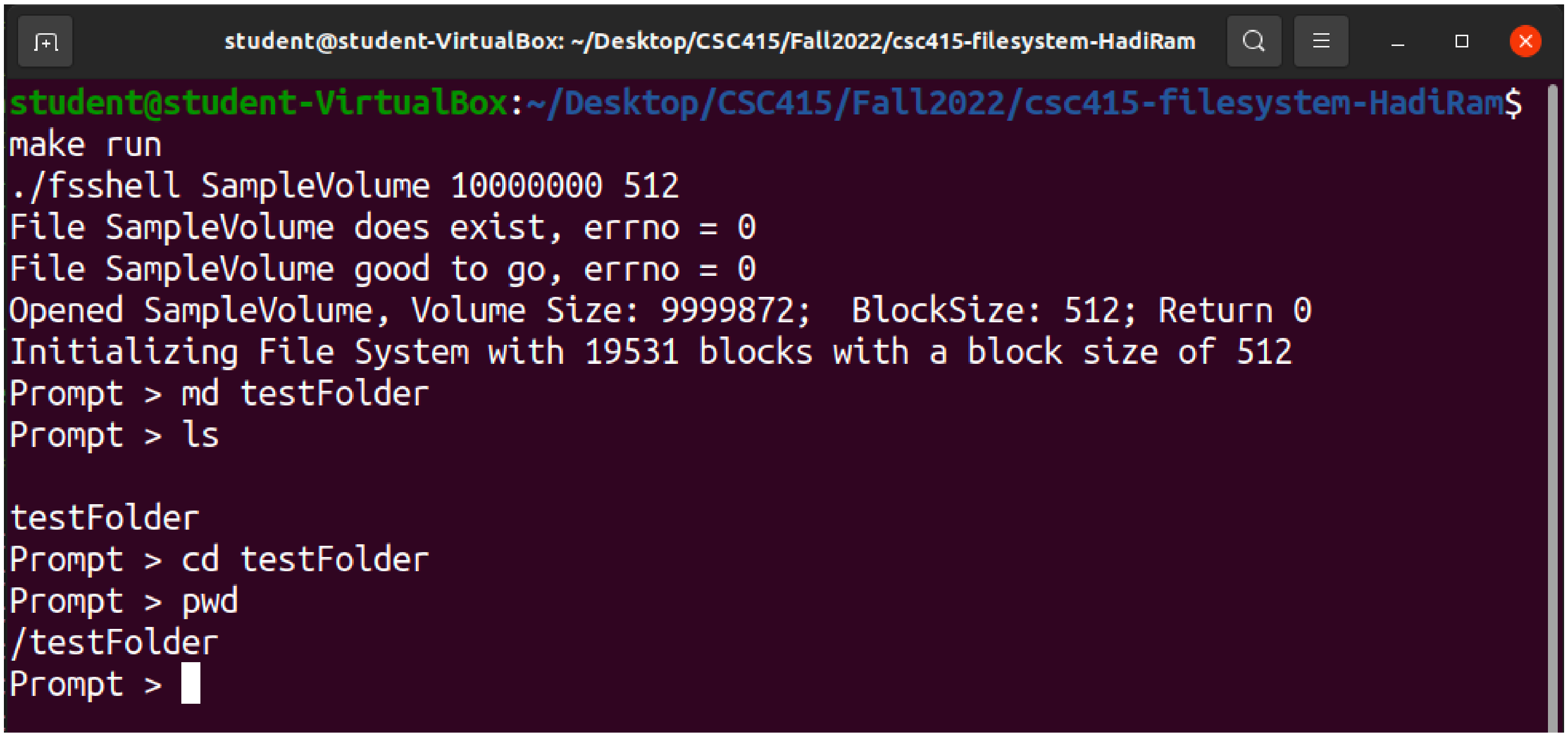
```
student@student-VirtualBox:~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam$
make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt >
```

Using the command **md** to make a directory “testFolder”, and using **ls** to display the directory.


student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

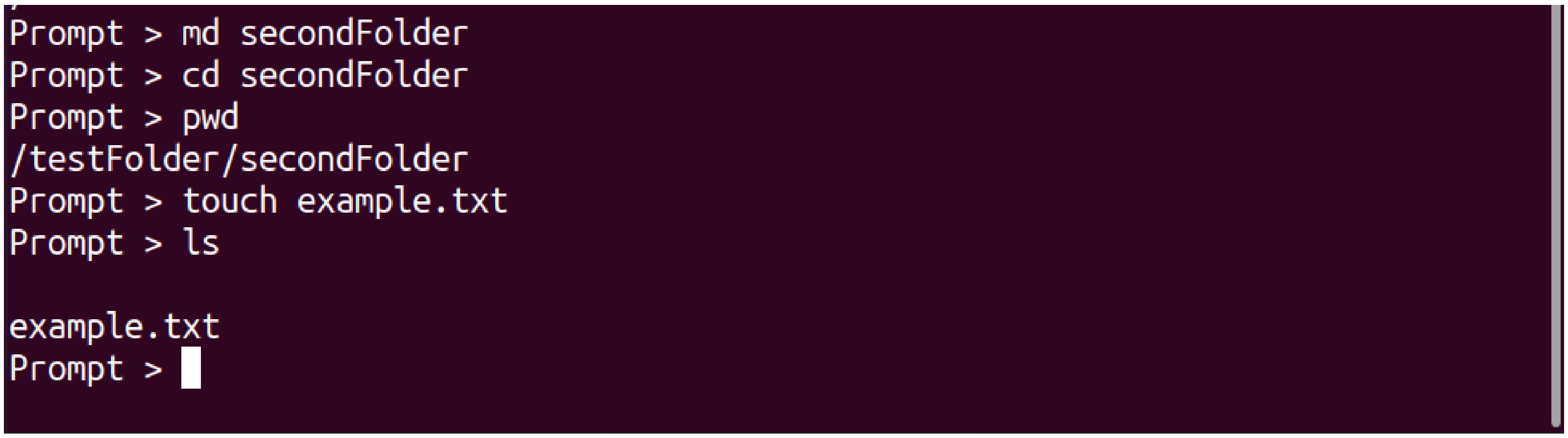
```
student@student-VirtualBox:~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam$
make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md testFolder
Prompt > ls

testFolder
Prompt > 
```

Using **cd** to change the current working directory to “testFolder” and printing the path with **pwd**.


student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

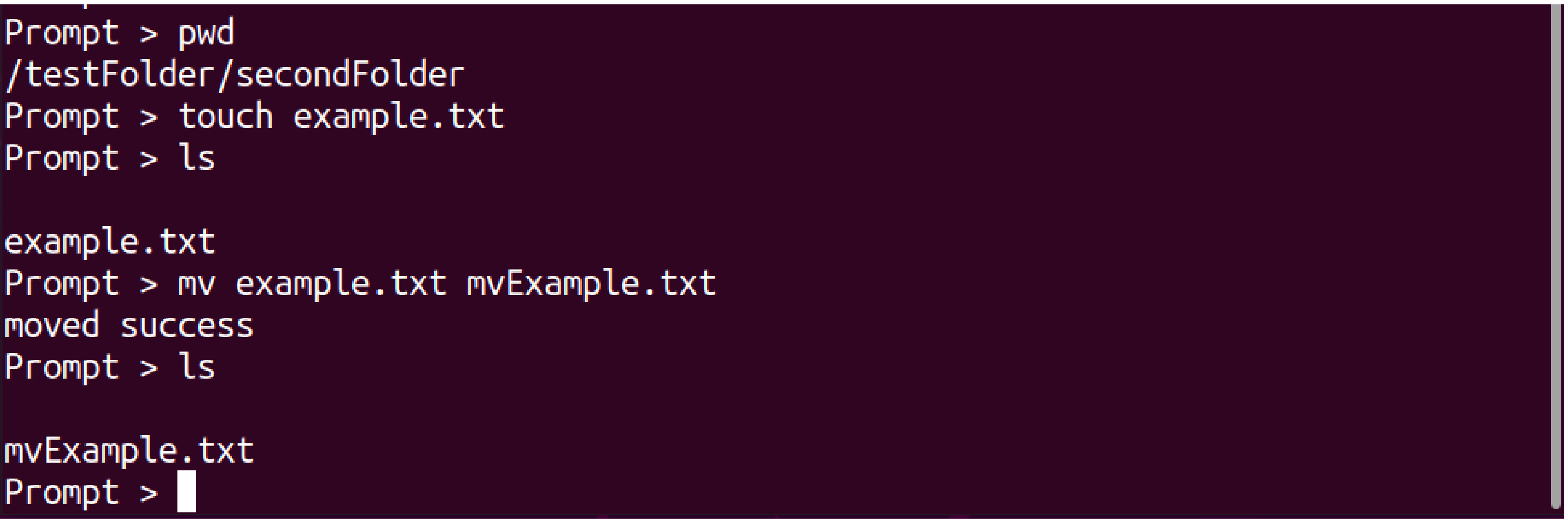
```
student@student-VirtualBox:~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam$
make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Prompt > md testFolder
Prompt > ls

testFolder
Prompt > cd testFolder
Prompt > pwd
/testFolder
Prompt > 
```

Creating another directory within “testFolder” called “secondFolder”, **cd** into “secondFolder” and then use **touch** to create a text file called “example.txt”



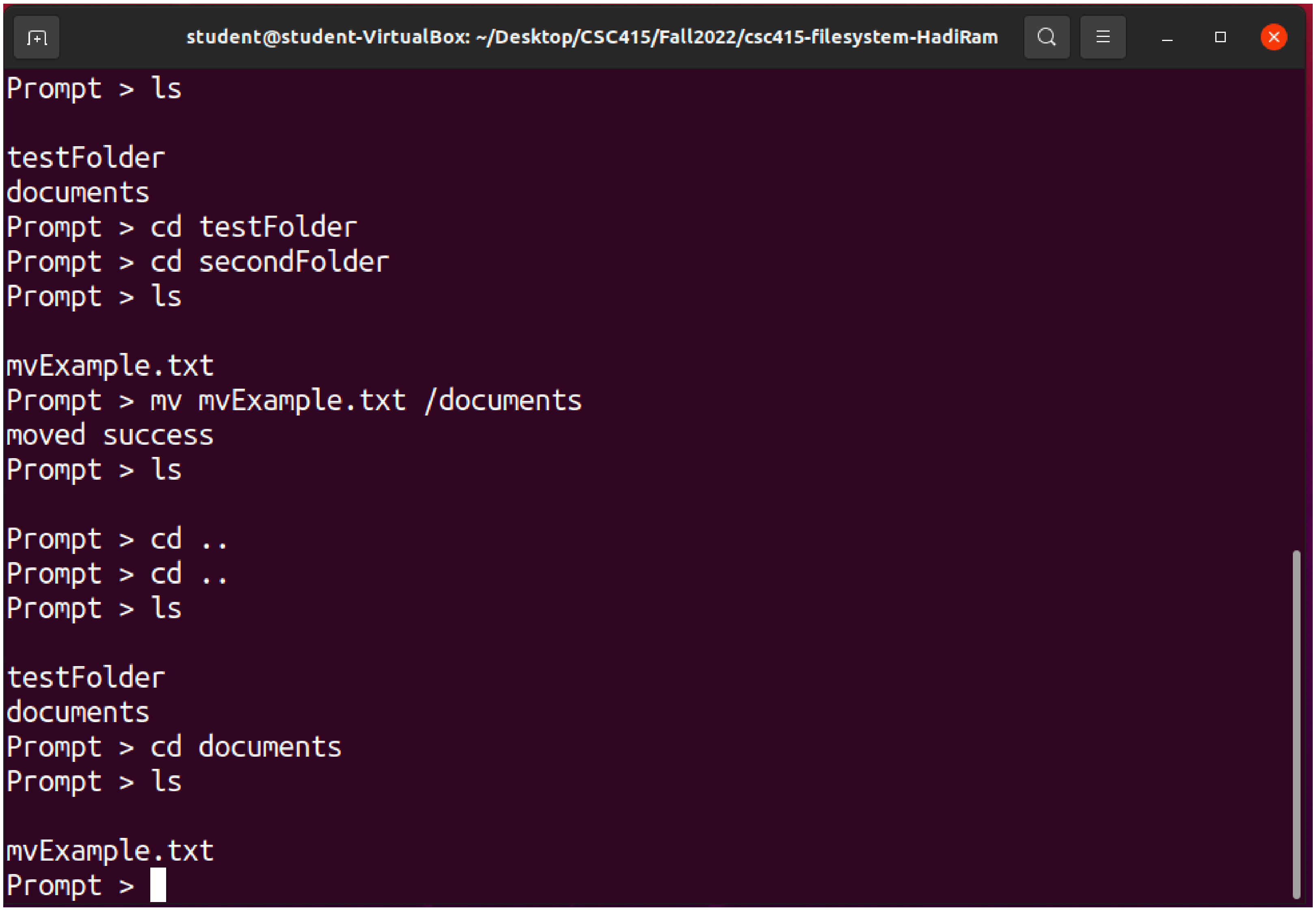
```
Prompt > md secondFolder
Prompt > cd secondFolder
Prompt > pwd
/testFolder/secondFolder
Prompt > touch example.txt
Prompt > ls

example.txt
Prompt > 
```

Using **mv** command as a way to rename the file “example.txt” to “mvExample.txt”



```
Prompt > pwd
/testFolder/secondFolder
Prompt > touch example.txt
Prompt > ls

example.txt
Prompt > mv example.txt mvExample.txt
moved success
Prompt > ls

mvExample.txt
Prompt > 
```

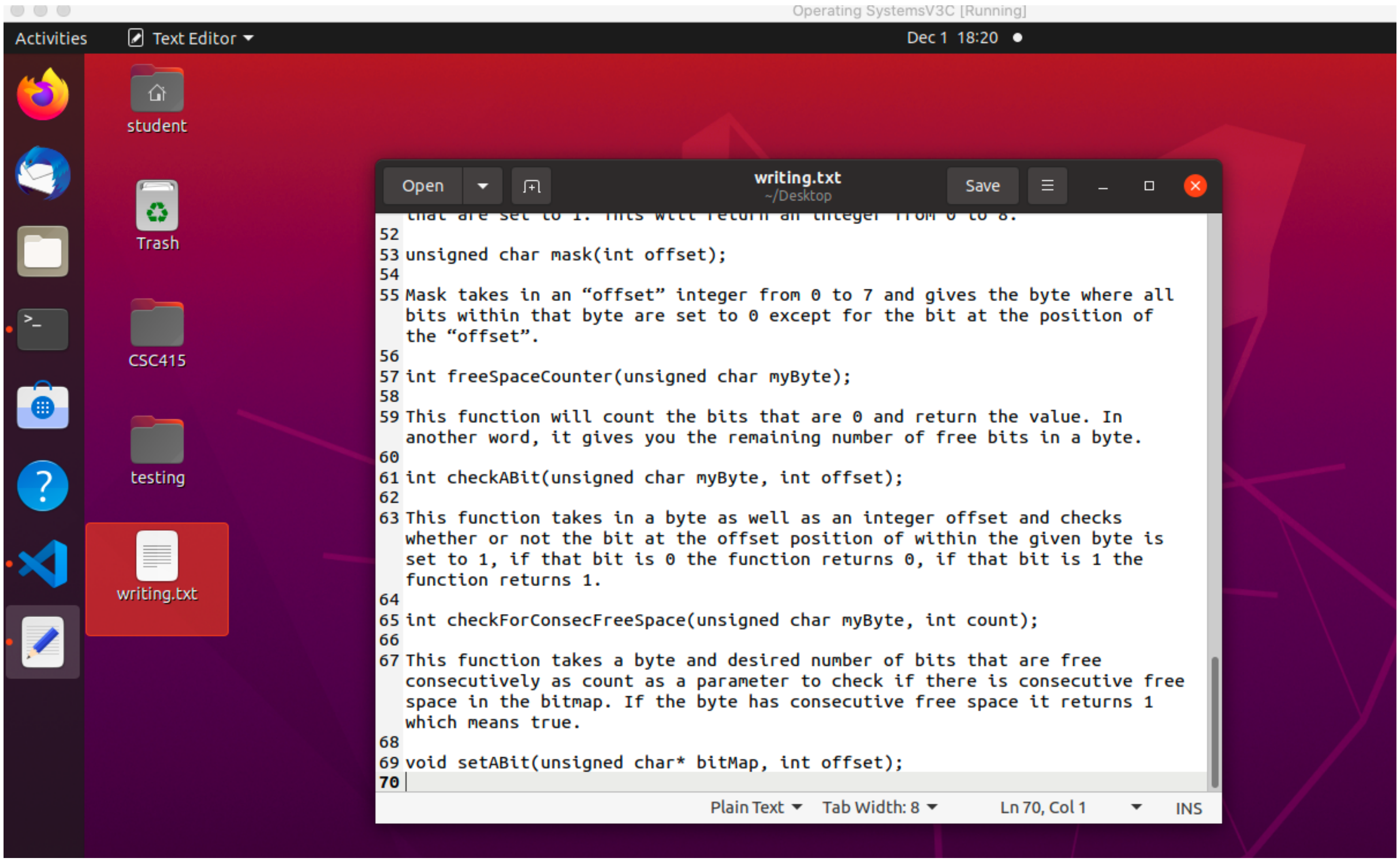
I have now created another directory inside the root directory called “documents” in order to test the **mv** command by moving “mvExample.txt” from “/testFolder/secondFolder” to “/documents”.



student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

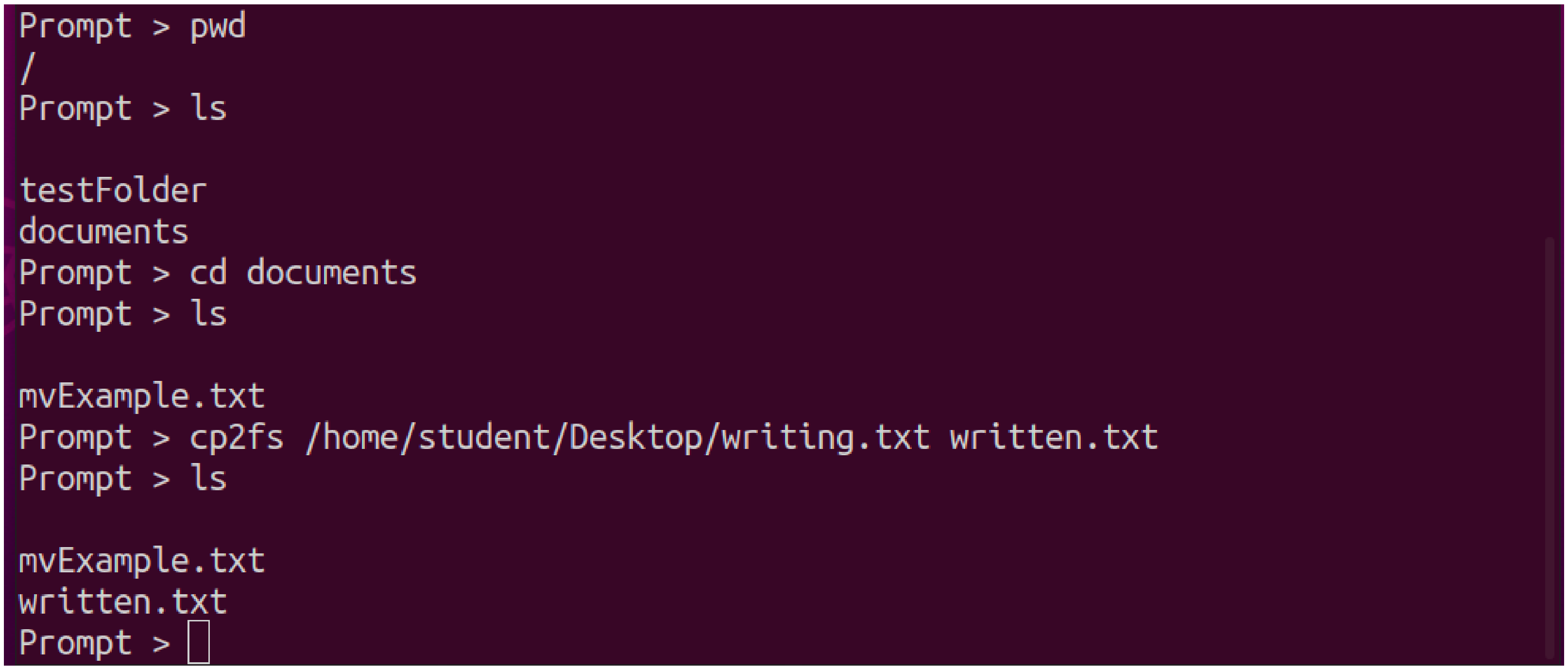
```
Prompt > ls

testFolder
documents
Prompt > cd testFolder
Prompt > cd secondFolder
Prompt > ls

mvExample.txt
Prompt > mv mvExample.txt /documents
moved success
Prompt > ls

Prompt > cd ..
Prompt > cd ..
Prompt > ls

testFolder
documents
Prompt > cd documents
Prompt > ls

mvExample.txt
Prompt > 
```

Now the **cp2fs** function will be shown, first I have created a file called “writing.txt” with a lot of random text, and saved it to my linux desktop as seen here.



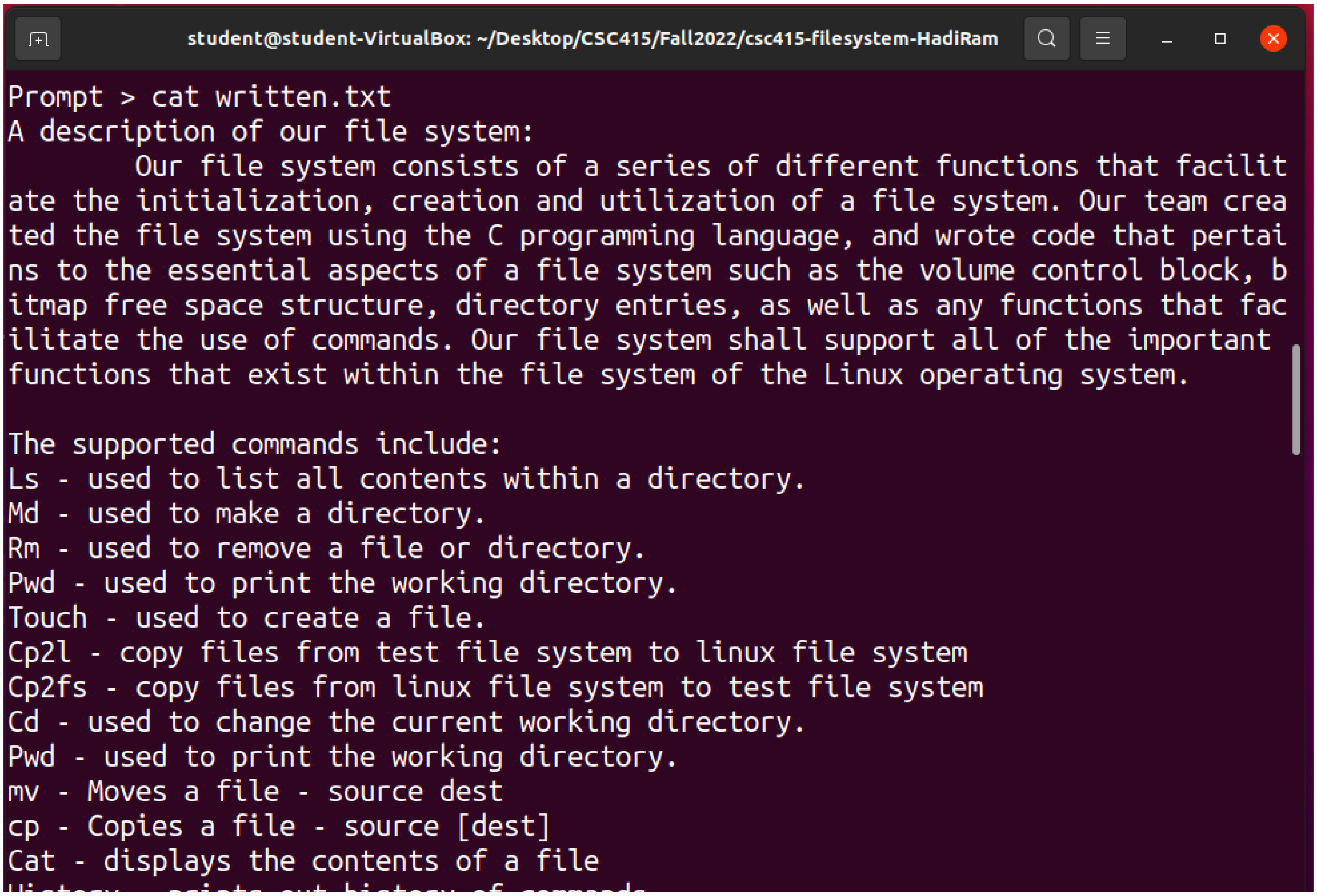
```
that are set to 1. This will return an integer from 0 to 8.

unsigned char mask(int offset);

Mask takes in an “offset” integer from 0 to 7 and gives the byte where all
bits within that byte are set to 0 except for the bit at the position of
the “offset”.

int freeSpaceCounter(unsigned char myByte);

This function will count the bits that are 0 and return the value. In
another word, it gives you the remaining number of free bits in a byte.

int checkABit(unsigned char myByte, int offset);

This function takes in a byte as well as an integer offset and checks
whether or not the bit at the offset position of within the given byte is
set to 1, if that bit is 0 the function returns 0, if that bit is 1 the
function returns 1.

int checkForConsecFreeSpace(unsigned char myByte, int count);

This function takes a byte and desired number of bits that are free
consecutively as count as a parameter to check if there is consecutive free
space in the bitmap. If the byte has consecutive free space it returns 1
which means true.

void setABit(unsigned char* bitMap, int offset);
```

The goal of **cp2fs** is to copy a file from the linux file system to our file system, so in this case I will use the command to copy this file “writing.txt” from my linux desktop to a new file called written.txt in the current working directory of my file system - /documents.



```
Prompt > pwd
/
Prompt > ls

testFolder
documents
Prompt > cd documents
Prompt > ls

mvExample.txt
Prompt > cp2fs /home/student/Desktop/writing.txt written.txt
Prompt > ls

mvExample.txt
written.txt
Prompt > 
```

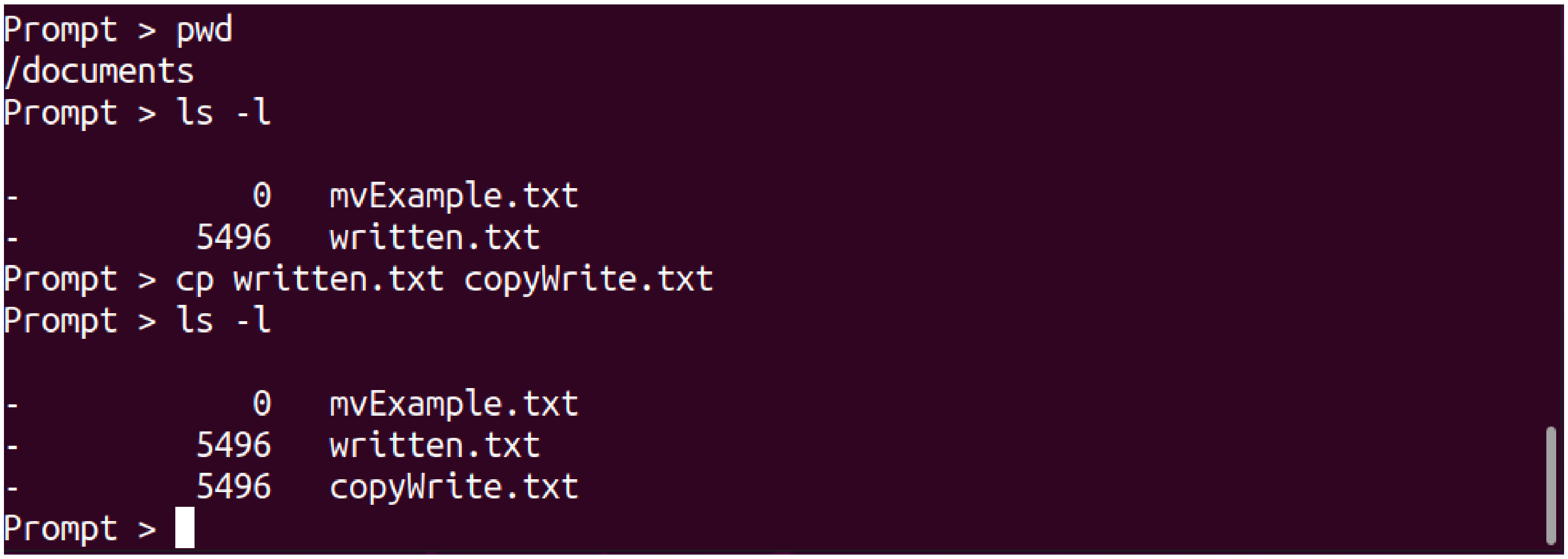
Now I will use the **cat** command to display the contents of “written.txt” to show that it is a copy of the file from the linux desktop.



student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

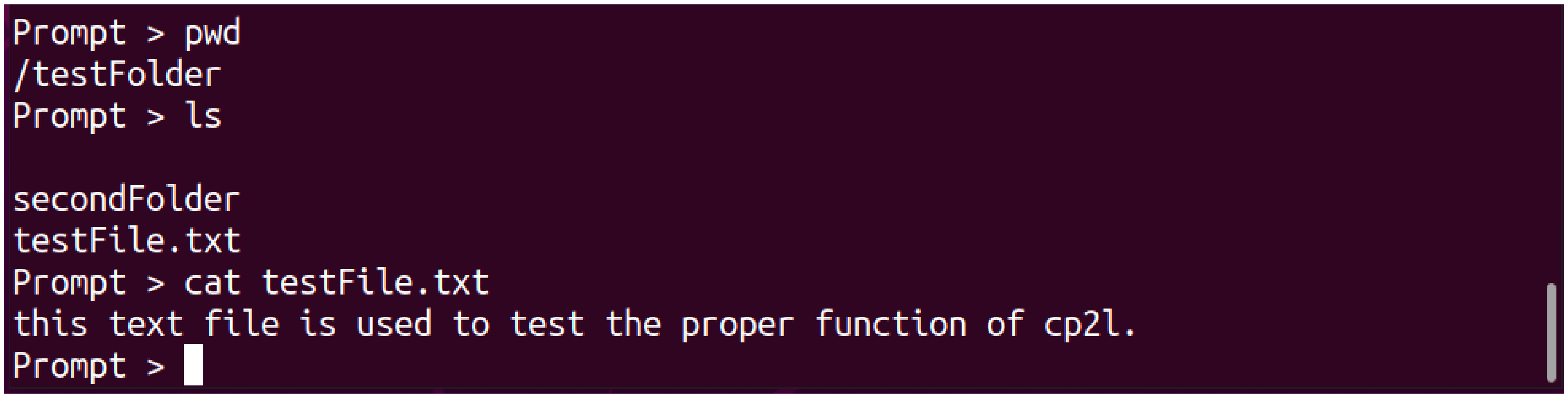
```
Prompt > cat written.txt
A description of our file system:
        Our file system consists of a series of different functions that facilit
ate the initialization, creation and utilization of a file system. Our team crea
ted the file system using the C programming language, and wrote code that pertai
ns to the essential aspects of a file system such as the volume control block, b
itmap free space structure, directory entries, as well as any functions that fac
ilitate the use of commands. Our file system shall support all of the important
functions that exist within the file system of the Linux operating system.

The supported commands include:
Ls - used to list all contents within a directory.
Md - used to make a directory.
Rm - used to remove a file or directory.
Pwd - used to print the working directory.
Touch - used to create a file.
Cp2l - copy files from test file system to linux file system
Cp2fs - copy files from linux file system to test file system
Cd - used to change the current working directory.
Pwd - used to print the working directory.
mv - Moves a file - source dest
cp - Copies a file - source [dest]
Cat - displays the contents of a file
```

In order to test the **cp** command, I will use **cp** on the file “written.txt” to create a copy called “copyWrite.txt” in the same directory “/documents” of our file system. Important to note is that in our file system we can only **cp** a file into another file in a single directory.



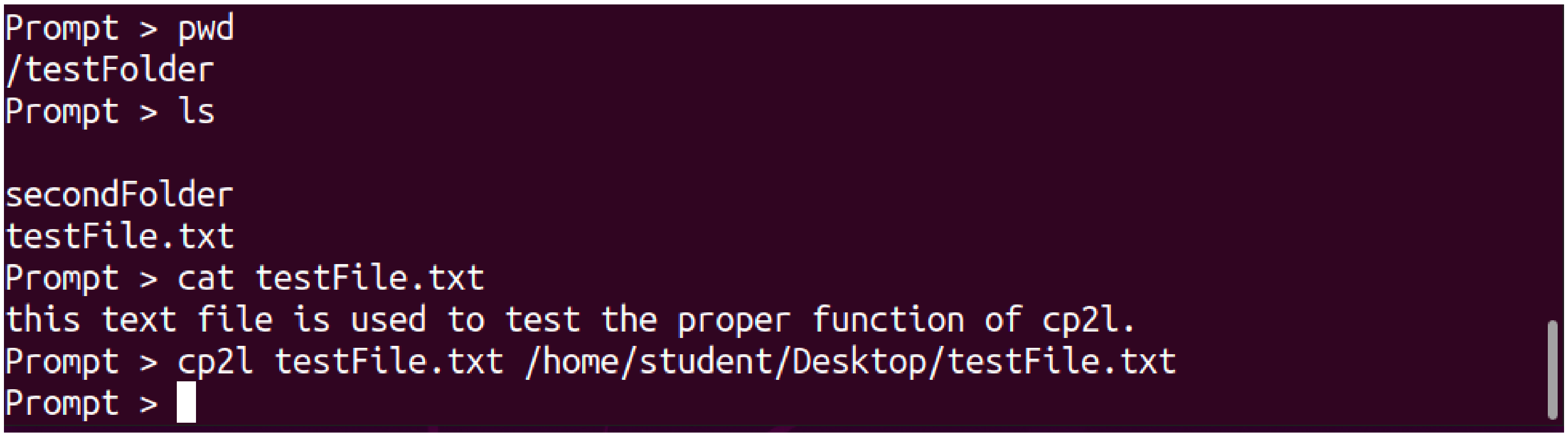
```
Prompt > pwd
/documents
Prompt > ls -l

-            0   mvExample.txt
-         5496   written.txt
Prompt > cp written.txt copyWrite.txt
Prompt > ls -l

-            0   mvExample.txt
-         5496   written.txt
-         5496   copyWrite.txt
Prompt >
```

For the command **cp2l** I have created a text file called testFile.txt within the “/testFolder” directory of our file system, as can be seen in the following screenshot.



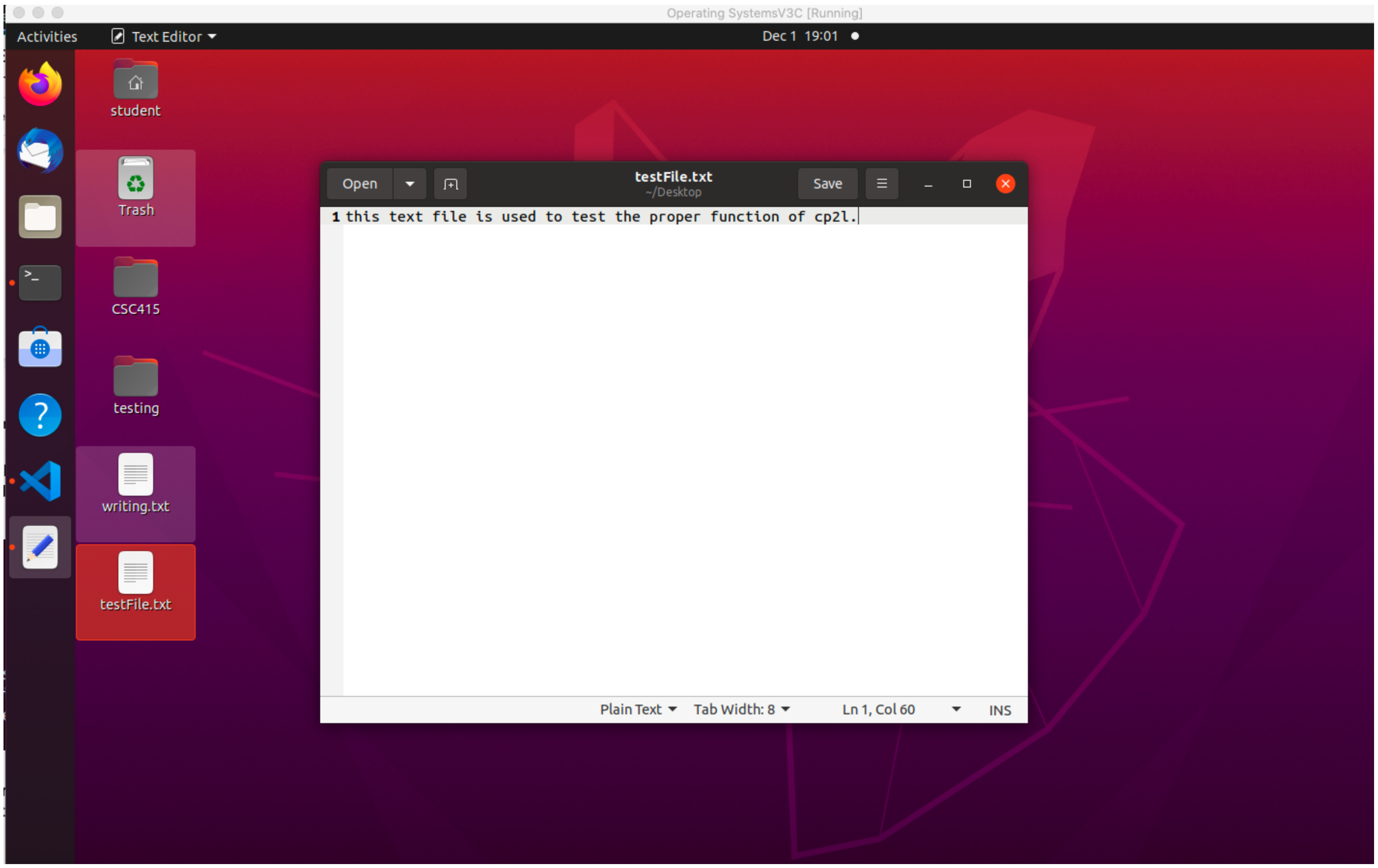
```
Prompt > pwd
/testFolder
Prompt > ls

secondFolder
testFile.txt
Prompt > cat testFile.txt
this text file is used to test the proper function of cp2l.
Prompt >
```

Now I will use the command **cp2l** , to copy testFile.txt from our file system directory "/testFolder" to my linux desktop /home/student/Desktop/



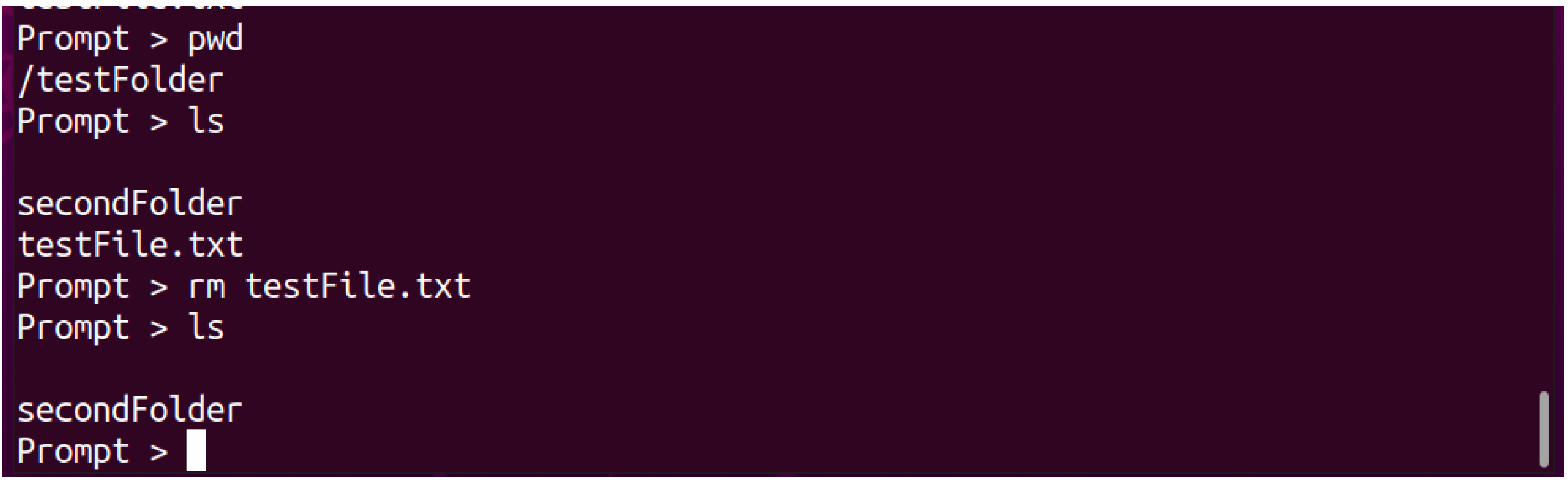
```
Prompt > pwd
/testFolder
Prompt > ls

secondFolder
testFile.txt
Prompt > cat testFile.txt
this text file is used to test the proper function of cp2l.
Prompt > cp2l testFile.txt /home/student/Desktop/testFile.txt
Prompt > 
```

The command worked as can be seen in the following screenshot, the text file "testFile.txt" was copied correctly to the linux desktop.



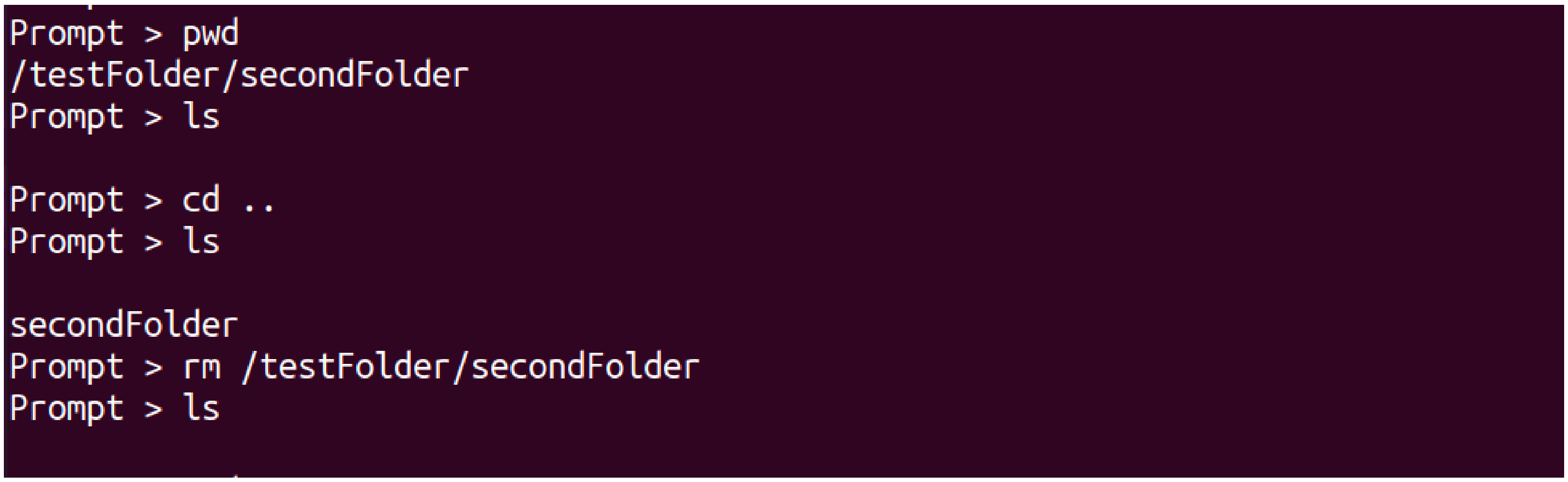
```
this text file is used to test the proper function of cp2l.
```

Now the command **rm** will be tested, firstly to remove a file. I will be removing the file "testFile.txt" from the directory "/testFolder".



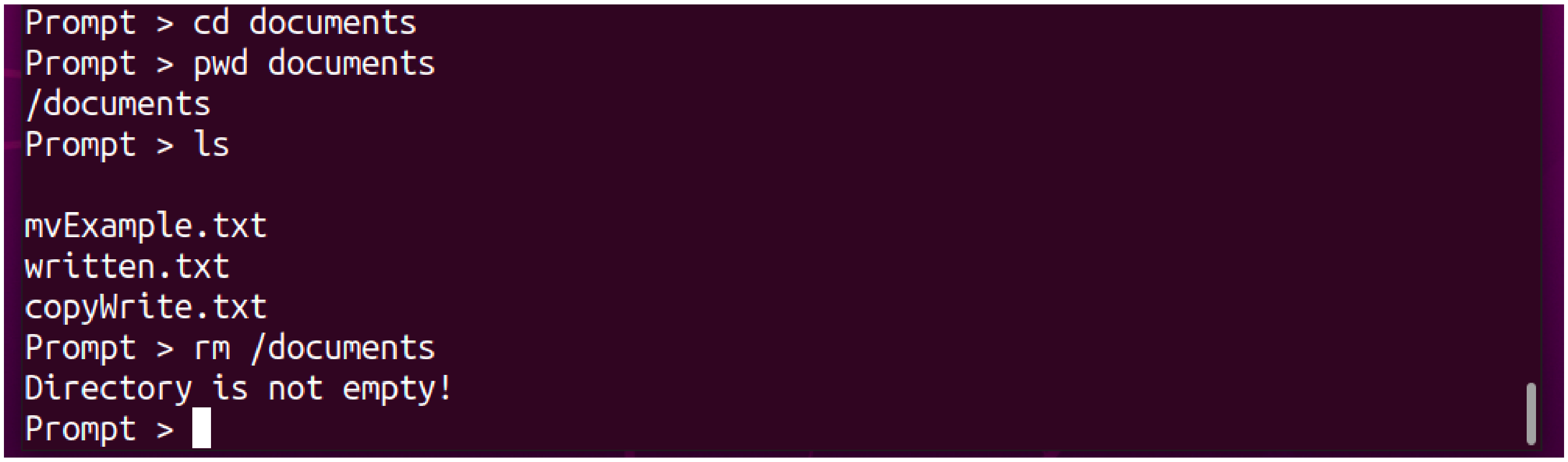
```
Prompt > pwd
/testFolder
Prompt > ls

secondFolder
testFile.txt
Prompt > rm testFile.txt
Prompt > ls

secondFolder
Prompt >
```

Now I will use the **rm** command on an empty directory called "secondFolder" in order to remove that directory.



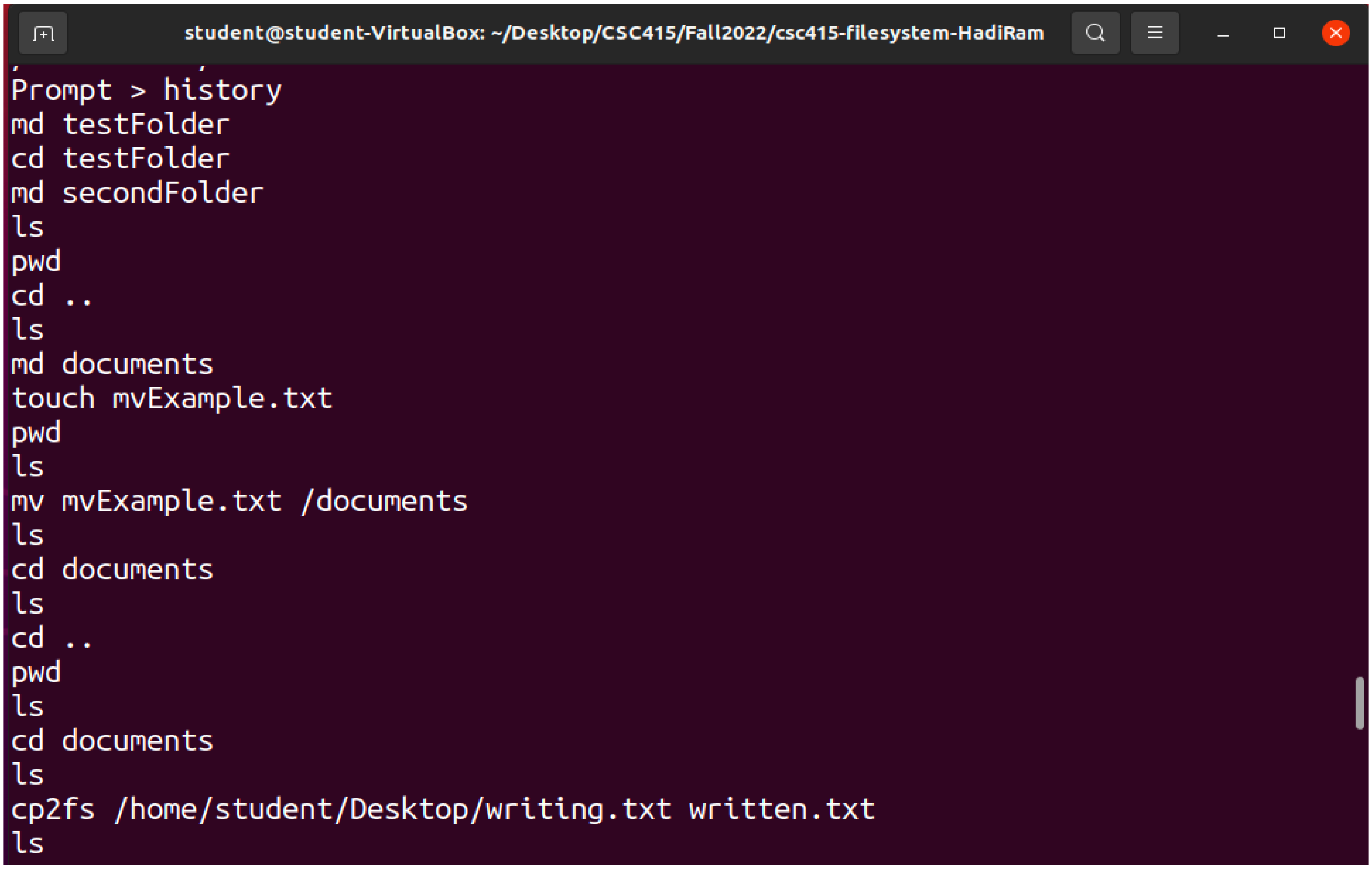
```
Prompt > pwd
/testFolder/secondFolder
Prompt > ls

Prompt > cd ..
Prompt > ls

secondFolder
Prompt > rm /testFolder/secondFolder
Prompt > ls
```

Now I will attempt to use rm on a non-empty directory, this will not work and let the user know that deleting a non-empty directory is not supported. As can be seen the directory "/documents" was not empty, and the command did not delete the directory, and warned the user.



```
Prompt > cd documents
Prompt > pwd documents
/documents
Prompt > ls

mvExample.txt
written.txt
copyWrite.txt
Prompt > rm /documents
Directory is not empty!
Prompt > 
```

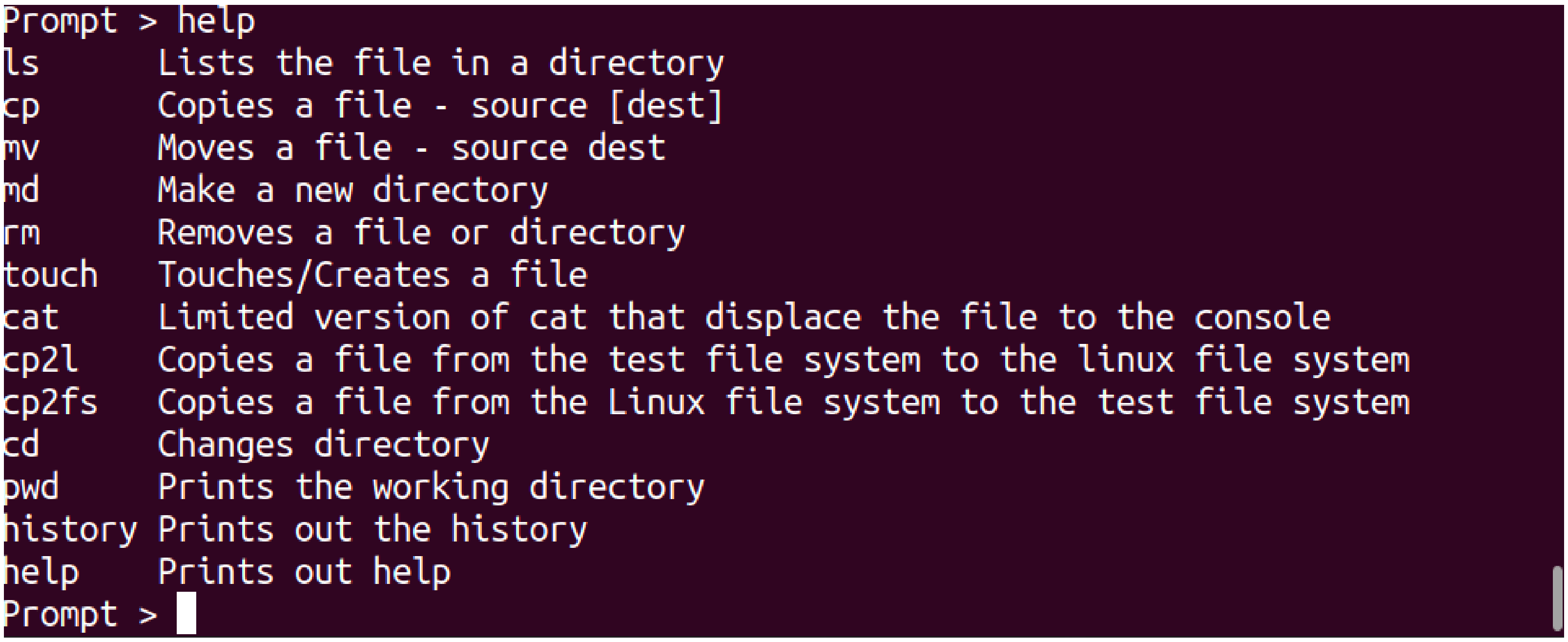
The **history** command shows all the previous commands used



student@student-VirtualBox: ~/Desktop/CSC415/Fall2022/csc415-filesystem-HadiRam

```
Prompt > history
md testFolder
cd testFolder
md secondFolder
ls
pwd
cd ..
ls
md documents
touch mvExample.txt
pwd
ls
mv mvExample.txt /documents
ls
cd documents
ls
cd ..
pwd
ls
cd documents
ls
cp2fs /home/student/Desktop/writing.txt written.txt
ls
```

The **help** command displays and describes all the supported commands.



```
Prompt > help
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
Prompt >
```