CSC415-03
File System Project
Team: Compiler Errors
Github: kylepet
https://github.com/CSC415-2023-Fall/csc415-filesystem-kylepet

Kyle Petkovic - ID: 923683186
Nicholas Pagcanlungan - ID: 922298361
Matthew Hernandez - ID: 920214682
Neal Olorvida - ID: 915905996

Final Report
November 26, 2023

What works

- pwd
- cd
- ls
- md
- rm
- cat
- mv
- touch

What does not work

- cp
- cp2fs
- cp2l

File System Description:

Milestone 1

Approach

For milestone 1, the main approach was based off of the “Steps for milestone 1.pdf” document we were given on canvas. We had a team meeting to discuss who would get each section of the steps listed in the PDF and assigned due dates so we would have time to integrate everything. At first we were focused on making the file system as simple as possible and therefore have less complex and more stable code; however, the initial design did not account for any ways for files to expand. If a file went past its last allocated block it would simply be unable to increase in size further. This would make for a pretty inflexible file system since the files created by the user could vary greatly in size, and it shouldn’t be up to the user to decide how much space they want for a file up front.

In class we went over different types of file systems, and a FAT Table seemed like the easiest method to allow for flexible block allocation. The initial designs for the volume control block and directory entry were made with file sizes staying the same, but we realized later in the building of the free space system that it would have to be more flexible. The FAT table is a pretty straight forward system with each index in its number of blocks length array corresponds to the block of the same number. This system is pretty nice, because for reading data only the initial starting block of the data needs to be provided and then the function can then simply read the FAT Table to gather the rest of the blocks. Before the addition of the FAT table, a free space bitmap was initially used to just keep track of which blocks were used. The bitmap is a holdover from before the FAT table was implemented, and functions were already written for it for getting free blocks and allocating blocks so it was kept in.

Initially the VCB was not a global variable and this made the code kind of messy as it would have to be LBAread into memory in every single function that used it. This was initially done because we kind of forgot about global variables being a thing since usually it's considered bad practice to use them. Changing the VCB to global significantly simplified the code.

The VCB has two pointers in it, to the FAT Table and to the free space bitmap. Our approach mirrored pdf for milestone one steps on Canvas, but we had to account for the pointers changing every time at runtime. If the program was ran for the first time this wouldn’t be an issue, but on initialization of the already created file system we had to calculate the locations of these two data structures, read them, and then update the VCB with this information so other functions could use it.

The approach to the functions in the free space system was mainly just reading from an array, whether it's an array of free blocks or the FAT table, and writing or reading the relevant data. The usage of incrementing a buffer to read block by block is an important aspect as the data is not guaranteed to be contiguous. This had to be accounted for because files can be deleted as well as expanded.

Volume Control Block, FAT Table, Free Space Bitmap, & Directory Entry Structures

The file system is made up of a couple of important data structures. The volume control block, directory entry, and FAT table structures. There is also a bitmap for tracking free blocks. The VCB will always be written to block 0 of the filesystem and never take up more than 1 block. After the VCB, the FAT table is stored followed by the free space bitmap and then the root directory.

The VCB contains information about the filesystem, most importantly the location of the root directory, the signature, and pointers to the freeSpace bitmap and FATtable. On load of the file system, if the VCB's signature matches `NealKyleMattNicholas` then that means our code has initialized the file system already. The pointers to the FATtable and the free space bitmap are also updated every time the file system is loaded since the pointers change at runtime. Otherwise, if the signature doesn't match it creates the VCB, FATtable, free space bitmap, as well as the root directory and they are written to the filesystem.

The FAT table is an array of FAT entries where each index in the array corresponds to the block of the same number. Each FAT entry contains an int that points to the next block of data to read for a given blob of data. This means it is simple to extend the amount of data a blob needs if the next block over is already used.

The free space table is a bitmap of integers where each of the bits in the number are manually changed to represent whether a block is used or not. For example the 10th bit in this array would be if the 10th block is used. This would be the second bit of the second integer in the array. This is used as well as the FAT table because it's faster to find and allocate blocks.

The DE, or directory entry, contains the file name, length in blocks, size in bytes, location of its stored data in the filesystem, if it's a directory, and some metadata. If it's a directory isDir is equal to 1, if it's a file isDir is 0, otherwise it's -1 by default. If it is a directory, the data in its loc field is the location of an array of MAX_DIR_ENTRIES (50 in our project) long of directory entries. This array of directory entries is the filesystem's concept of an inode. If isDir is 0, the DE's loc attribute will point to the block of data in the filesystem where a blob of data starts.

Milestone 1 Functions

(fsInit.c) initFilesystem

initFilesystem takes in the number of blocks in the file system followed by the size of the blocks. These are both parameters passed in at runtime. This function essentially checks to see if the filesystem is created. If it is, load in the VCB and update the pointers for FAT Table and freeSpace structures as well as the current working directory and the root directory. Otherwise it needs to create the filesystem. The VCB is always created at block 0, so that's where the data is read in from first. At this point the signature in the VCB can be checked against our signature, `NealKyleMattNicholas`, to see if the file system needs to be created. If this is the case, the FAT table and free space bitmaps are initialized. The VCB is populated with data and then the root directory is initialized with initDir. The VCB, FAT Table, and free space bitmap are written to storage.

(initDir.c) initDir

The initDir function handles creating folders, including the root folder, in the filesystem. This function takes in a parent directory entry as well as a blockSize integer. If this folder is the root directory the parent DE is null. InitDir allocated data for the directory and then iterated

through all the indices in the array to set default values. It then allocates data to prepare for writing. If the directory entry passed is null we know we are making the parent. This means that we need to save the location of this directory to the VCB, this also also where the global variables for keeping track of the root directory and current working directory are created in the case where the filesystem doesn't already exist. As the rootDirLoc field in the VCB is updated we then need to save the VCB back to the disk. In order to make this function work for both the root directory and any other directory, a dummy directory entry is created, in the case we are making the root we can then set this back to the "." entry in the root directory so that we can initialize the ".." entry as they are both the same in the root case. If a parent is passed the dummy directory entry is sent to that and the parent directory is then properly defined in the child's ".." entry. This directory entry array can then be written to the disk. This function returns the integer of the first block written to.

(freeSpace.c) initFAT

initFat takes in the number of blocks in the filesystem as well as the block size for calculating how to write the data. The FAT table has as many indices as there are blocks in the filesystem. Since we know the VCB is always at block 0, that means that whenever the next block that's being pointed to is 0 we have reached the end of a blob's data. After the FAT Table is created it preallocates data as we know that there will always be a vcb, free space bitmap, and the FAT table in the same spots. This data is then written to the file system. This function returns the size of the FAT Table.

(freeSpace.c) initFreeSpace

The initFreeSpace function essentially makes an array of ints where helper functions bit manipulate parts of the int in order to represent if blocks are used or not. This function takes in the number of blocks, block size for writing, and the size of the FAT Table so that the freeSpace bitmap can be written to the correct spot. This function returns the location of the free space bitmap.

(freeSpace.c) allocFreeBlocks

This function gets the amount of blocks required to write data. The free blocks are not necessarily contiguous as data can be unallocated, so this function reads through the freeSpace bitmap in order to find the first free blocks up to the passed value, numOfBlocks. This function will also handle updating the freeSpace bitmap, updating the FAT Table, and writing these modified tables to the disk. This function returns an array of integers that represent the blocks that can be written to. This function is meant to be called in tandem with writeData.

(freeSpace.c) writeData

writeData takes in an array of free blocks, the number of blocks to write, and a buffer. It can not be assumed that the data is contiguous. This means that only one block is written at a time and the buffer is then incremented by the block size. Each int in the blocksToWrite array is iterated over so the entire buffer can be written. Therefore, this function handles the writing of non-contiguous data that is already marked as used at this point.

(freeSpace.c) readData

readData takes in the first block of data and the buffer to write the read in data to. Only the first block is required as it then follows the fat table to get the rest of the blocks as the data is not guaranteed to be contiguous. The function will start with the startBlock passed to it, read in a block into the buffer, get the next block from the FAT table, increment the buffer by block size, and then repeat until the next block to read is 0. We know the VCB is at 0, and therefore have reached the end of file for this blob. This function returns the total amount of blocks read so it can be checked against the calculated expected.

Milestone 2

Approach

Milestone 2 consisted of many of the filesystem operations related to folders and deleting data. g_cwd and g_rootDir are two important global variables that needed to be added in order to keep track of the current working directory as well as the root dir. While g_rootDir would not be modified, the g_cwd global variable would change where the user changed the directory to.

Filesystems have a concept of an inode, in our filesystem the equivalent of this is the array of directory entries that each folder contains. This is what would need to be searched through and modified when creating or deleting folders and files. This was chosen because it already existed in the filesystem and wouldn't take up additional space. This directory array is also how the data is represented in the g_cwd and g_rootDir global variables. parsePath and cleanPath are two integral functions in this milestone and are heavily based off of the implementation that was based off of how it was described in lectures. cleanPath allows for a predictable path that was simplified from any directories that didn't need to be there like ".." and ".".

The main idea behind how mkdir would work is pretty simple, at a high level it would use the initDir function that's already used to make the root dir and then insert it into the corresponding index in the directory entry array. As this function was creating new data to write, that means the FAT table and the free space bitmap would need to be modified. In the design this is already handled when the mkDir allocates blocks for the data and writes it. The mkdir function is a good example of the design of the file system as it uses many of the functions already made and benefits from the abstraction of the free space system as it isn't specifically handled in the code of mkDir. Now that the folders could be created, the remove directory function was pretty similar in theory but had to have some modifications like making sure the folder was empty before it could delete the folder.

Functions in milestone two were able to use functions already written in milestone one to simplify the code and abstract concepts like the free space system. This made it simpler to build on top of the code that was written and helped make it easy to debug as if there was an error in writing the folder there was only one write function that was being used.

Milestone 2 Functions

(mfs.c) fs_setcwd

This function used to change the current working directory in a file system. As we have implemented it into this project it functions the same way as other file systems, taking a pathname as an argument and changing the current working directory to the directory specified in the pathname. It first cleans the pathname, removing any characters that do not provide any more necessary context to what pathname we are setting the current working directory to. For example, if we want to switch our current working directory to "testDir" but the path is "./testDir", cleaning the path will remove the "./" and only show testDir. The function will then parse the cleaned pathname using parsePath, check if the parent directory exists as we cannot continue if the parent directory is NULL, and check the result of the parsed path. Finally the function updates the current directory path variable with the cleaned pathname and changes the current working directory to either the root or the last element in the parent directory, exiting with -1 if it cannot find the last element.

(mfs.c) fs_getcwd

fs_getcwd is a simple, small function that retrieves the current working directory, useful for commands like "pwd". All this function does is check if the length of the current directory path is greater than the size of the buffer and copies the current directory path if it is not, returning the pathname of the current working directory. If it is, it will print an error message and return NULL.

(mfs.c) parsePath

parsePath is a helper function useful for parsing a given path into its components to locate the last element in the path. The function will check if the path and ppInfo structure variable are not NULL and will determine where to start parsing. If the path starts with a "/", it will recognize that as the root, otherwise it will start at the current working directory. Using strtok_r, the function will begin tokenizing the path, splitting each portion to where the "/" would go. Checking the tokens, the function will go through the first and second and see if they are either null or the root directory. If the first token is recognized as the root, then it will set the parent, index, and last element of the ppInfo structure to indicate that the path is the root, returning 0. If not, it will free the duplicate path and return -1. Finally getting to where it actually parses the path, for each token, the function will find the corresponding entry in the current directory. If the entry is not found it will set the parent, index, and last element of the ppInfo structure to indicate that the last element of the path is not found, returning -1. If the entry is found it will check if the next token is NULL. If it is, it sets the parent, index, and last element of the ppInfo structure to indicate that the last element of the path is found, and then returns 0. If the next token is not NULL, it checks if the found entry is a directory. If it is not, it returns -2. If it is, it loads the directory and sets it as the parent directory, and then continues to the next token. All these series of checks culminate in the tokens being parsed, finally freeing the duplicate path at the end of the function.

(mfs.c) fs_stat

fs_stat is a function used to retrieve information about a file or directory. It first checks if the path and fs_stat structure are not NULL, then parses the path using the parsePath function. It

will then check if the parsing was successful, retrieving the directory entry for the last element in the path. The entry contains information about the file or directory like the size, blocks, last accessed time, last modified time, and when it was created. It also differentiates directories from files. Finally, the function fills the fs_stat structure with the appropriate information from the directory entry.

(mfs.c) isUsed

isUsed takes in a directory entry and if the name is just a null terminator it will return a 0 meaning that it is not used, and other 1 meaning this directory is currently being used.

(mfs.c) getType

getType takes in a directory entry and returns its isDir attribute. This abstraction was originally done in case the way to check the type of file was changed.

(mfs.c) findUnusedEntry

This function takes in a directory entry array and returns the first unused entry inside it. It iterates through every entry in the array looking for a case where isUsed returns that a folder is not used. It then returns the index of the first unused entry and -1 if an entry is not found.

(mfs.c) fs_isFile

fs_isFile takes in a path with a filename at the end. First the path needs to be cleaned and then sent to parsePath. parsePath will then return a ppInfo struct where its index field is the file that is to be checked. The directory entry at the index field in the ppInfo struct is then passed to getType and fs_isFile will return 1 if the path leads to a file and 0 otherwise.

(mfs.c) fs_isDir

fs_isFile takes in a path with a folder at the end. First the path needs to be cleaned and then sent to parsePath. parsePath will then return a ppInfo struct where its index field is the file that is to be checked. The directory entry at the index field in the ppInfo struct is then passed to getType and fs_isFile will return 1 if the path leads to a directory and 0 otherwise.

(mfs.c) overwriteData

This function is used to overwrite data that has already been written to the file system. This is useful in cases of when data needs to be unallocated, or a parent has had a new directory added into it and it needs to be saved. overwriteData takes in the first block of the data to overwrite, the blocks needed which are used for error checking, and the buffer of data. Only the first block is needed as this function follows the fat table and rewrites all the data as the blocks are not guaranteed to be contiguous. The function essentially just iterates over each block, which it gets from the fat table using the previous written block, and then writing the data and incrementing the buffer by the size of a block as well as decrementing the blocksNeeded counter to see how much still needs to be written. In the case of the data exceeding the originally allocated space, the function will see if blocksNeeded is 0. If it is not that means more data needs to be written. It can then simply allocate more blocks with allocFreeBlocks(blocksNeeded), link the previous end of the blob to the beginning of this blob extension by updating the nextBlock attribute of the FAT Table for the last block in the initial

allocated data. Then the data can be written to the disk. Lastly, the amount of blocks written is returned so that this value can be checked against the expected amount of blocks and an error can be raised if it doesn't match up.

(mfs.c) fs_mkDir

fs_mkDir is one of the most important and comprehensive functions in mfs.c. At a high level this function will create a folder at the path provided and handle writing to the disk as well as updating the freeSpace system. It will start with cleaning the path and parsing it. The ppInfo struct modified by parsePath will have the parent directory as its parent attribute. This is the directory entry that gets passed to initDir along with the max amount of entries and the block size. The amount of blocks needed is calculated to check for errors. The new directory is then modified with metadata, that it's a dir, its location, and the amount of blocks it takes. The parent directory has to be modified with this data so other function like ls will display correct data if called right after creating a file. The entry to update in the parent is found with findUnusedEntry. The new parent then needs to be committed to the disk with the overwriteData function.

(mfs.c) fs_openDir

This function takes in a path and loads the array of directories located there. It will first parse the path. If the currentDirectoryPath is the root directory there is a special case for handling reading a file from there as there is no parent of the root directory. The fdDir is then updated with information about the directory, with the directory field simply being set to the global root directory variable defined earlier. The function then returns the fdDir struct. In the case where it is not the root the function is the same, but the fdDir's directory attribute needs to be set to the parent that is loaded in.

(mfs.c) fs_readDir

This function takes in an fdDir and returns the next unread entry that has data in the directory stored within the fdDir. It can just iterate over every value in the directory until it finds something that is used. The struct fs_diriteminfo is then populated with this directory and this struct is then returned.

(mfs.c) fs_closedir

This function takes in an fdDir and free the malloc'd data inside of it. If it is not null it is free and then set to null. This function will return 0 if successful or -1 if the passed fdDir was NULL.

(mfs.c) findEntryInDirectory

This function takes in a directory entry array and a name to search for. It iterates over every entry in the parent searching for one that has a name attribute that matches with the name passed. If found it will return that entries index. If a directory with the same name is not found this function returns -1.

(mfs.c) isDirectory

This function takes in a directory entry and returns the isDir attribute of it. This layer of abstraction was made in case the way checking if it was a file was changed.

(mfs.c) loadDir

This function takes in a single directory entry of which its content should be loaded from the filesystem. Space for the child is allocated and then readData is called with the location started from the loc attribute of the passed directory entry. The expected amount of blocks is checked to make sure it is correct and then the loaded in child is returned.

(mfs.c) cleanPath

This function receives two char arrays: inputPath for the string representing the directory path before processing, and outPath for the string representing the directory path after processing. This function's purpose is to simplify the directory path string for readability after the user inputs navigation commands like "." or ".." . The string processing begins by tokenizing the string from inputPath by the '/' character. Each token represents the name of a directory. Next, depending on the input, the tokens are reorganized to a more readable string minus the navigation commands, with respect to the user's current position in the directory. This function returns if the process was successful, and false if the process fails.

Milestone 3

Approach

Milestone 3's tasks were to complete functionality for the rest of the remaining commands, specifically touch, cat, rm, cp, mv, cp2fs, and cp2l. This milestone would be the last for completion of the project. Our approach for Milestone 3 was to split each of the necessary functions needing completion between each of us, while handling bug fixing for previous functions that were necessary for the specific milestone's tasks. Each of these commands take the results of different input/output functions in order to complete the task the user wants with the command they chose. Touch, for example, took the functions b_open() and b_close() for its functionality in fsshell.c to work. Kyle supervised bug fixes and memory issues, Neal took b_seek() and b_close(), Nicholas took b_write(), and Matthew took b_read() and b_open() with functions for reading flag bits.

Each function is essential in the commands' necessary functionality, some not working without the other. For a file system to take input from their users and provide output, the IO functions must be implemented correctly so that the commands users write can function in the same way as an operating system's file system. It became imperative that each function worked flawlessly in order for the commands to have correct outputs. For functions like cat, where b_open(), b_read(), and b_close() were called, it would be difficult to test when some functions were done and others weren't, so debugging while coding became a top priority.

Milestone 3 Functions

(b_io.c) b_open

b_open is a function used to create and open files, useful for commands like “touch” that are meant to create files that are able to be written to. It starts with identifying the parent directory of the new file, splitting the filename into directory path and file name. If the file name does not include a parent directory path, the program will use the current directory as the parent directory. From there, the function opens the parent and finds an unused entry using the findUnusedEntry helper function. After finding an unused entry, the function verifies if the file does not exist and the O_CREAT flag is set, allowing for files to be created. The function then allocates free blocks akin to the blocks needed for the file using another helper function, allocFreeBlocks. It will check if the allocation was successful and move on to updating the unused entry in the parent directory, setting the name of the new file as well as the size, number of blocks, and the type. It will also save the starting block for future use. Finally, the function will write the updated parent directory back to the disk using the overwriteData function, printing an error message if it fails. The function also handles truncate and append flags, using another helper function, truncateFile, to truncate files and b_seek to append files. Finally, b_open will free the memory allocated for the directory path and return the index.
b_read is a function for reading data from a file and storing data into buffers. Taking three parameters, a file descriptor, buffer, and count, the function first initializes the system and checks if the file descriptor is between 0 and the max possible file control blocks minus 1. Next are checks to see if the file control blocks, count, buffer, or directories are NULL, stopping crashing when files that are empty attempt to be read. While the count is greater than 0, we read more data into the buffer until we have no more data to read. Using the LBAread function provided in the assignment, we read the data, increment the block index, set the buffer length to the length of the file control blocks buffer, and copy the necessary bytes of data from the buffer to the user’s buffer. It will keep looping until all the requested number of bytes have been read.

(b_io.c) b_read

b_read is a function for reading data from a file and storing data into buffers. Taking three parameters, a file descriptor, buffer, and count, the function first initializes the system and checks if the file descriptor is between 0 and the max possible file control blocks minus 1. Next are checks to see if the file control blocks, count, buffer, or directories are NULL, stopping crashing when files that are empty attempt to be read. While the count is greater than 0, we read more data into the buffer until we have no more data to read. Using the LBAread function provided in the assignment, we read the data, increment the block index, set the buffer length to the length of the file control blocks buffer, and copy the necessary bytes of data from the buffer to the user’s buffer. It will keep looping until all the requested number of bytes have been read.

(b_io.c) b_close

The purpose of b_close is to serve as the interface for closing a file, according to the received file descriptor fd. When called, this function uses the fd to access its associated file control block, frees the included buffer, resets the file control block, and releases the fd. For cases where the file control block’s buffer was open for write but not written to disk, it

overwrites the buffer's data, and updates the directory entry to the latest access time, modification time, and size.

(b_io.c) b_seek

The purpose of b_seek is to reposition the offset of a file. When called, it accepts a file descriptor fd and an unmodified offset value, and returns an updated offset value, according to the directive whence. The value of whence is expected to match one of the preset values provided by <unistd.h>. If whence is equal to SEEK_SET, then the function returns the unmodified offset value. If whence is equal to SEEK_CUR, then the function returns the sum of the file's current location and the received offset value. If whence is equal to SEEK_END, then the function returns the file's size plus the value of the received offset.

(b_io.c) b_write

For b_write, the main logic revolves around manipulating the bytes in the buffers, and preparing them to be written to disk. In our case, we simply need to take the buffer given to us by the user, the bytes of the file we need to write to a new destination, and take those bytes, and write them into our files buffer to be written to disk. For this assignment, we do not have to worry about writing to the LBA because LBAWrite is a function already defined for us, so here we have to worry about just the buffers to prepare the data to be written. Simply we would just take what is in the buffer given to us by the user, and copy it into the files buffer; however things are never this simple with file systems. We have to account for many edge cases. One such edge case, is when the amount of blocks needed to be written, count in the case of our parameters, is more than the remaining bytes in the block. Here we would make sure that we write to the remaining bytes in the block before proceeding to continue to write to the rest of the blocks. In addition, we have to make sure that count is more than 0, and that count is within the buffer length. All these checks have to be made to make sure that function correctly functions inside the system. After all of these checks, we return the amount of bytes that were written. In addition; we handle each flag accordingly.

Issues & Resolutions

Matthew

Milestone 1

Issues with initDir.c at first were due to confusion in the start of the file system project, I had started making an LBAWrite() function not realizing that one is already built into the project, like with Assignment 5. Once I realized that, I started on a memory allocation check but we decided to put that into different files like mfs.c. We initially began the project using extents, but switched to FAT which would provide us with a more streamlined approach which stuck through to the end of the project.

Milestone 2

Problems with parsePath occurred right from the start of the milestone, as I had trouble understanding the pseudocode that Professor Bierman was explaining during his lecture. I was following the pseudocode too literally, and was overthinking how different the pseudocode would be for our specific file system implementation. As a result, parsePath would not actually parse the path that we chose. I worked with Nicholas on debugging the function where we found that the temporary directory entry was not initialized to the dereferenced parent's index, causing some issues when attempting to parse through. Other issues that occurred was the use of global variables, as I was confused on where to initialize them so they could be used across the entire application. The global variables I had to initialize were g_cwd and g_rootDir, both of which had to be initialized in initDir.c. By initializing them there and allocating memory for them, parsePath finally began working.

Milestone 3

Problems with b_open() were that touched files were not able to be created in subdirectories. To fix this, the inclusion of libgen.h in the library includes was necessary as there is a function called "basename" that returns a pointer to a string that is the base name of the file path. The function omits the slashes or dots that come with a path to a file or directory, allowing the b_open function to create files regardless of whether we are in the root or a subdirectory. Other problems with b_open include the touched files showing up in the host file system as opposed to the one we are building. This was due to me creating the files with the open() function, as since that is a built-in Linux function, it would think that we are creating a touch command for the Ubuntu files, not files for the application we were building. To circumvent this issue, I used the helper function allocFreeBlocks() to allocate the blocks of the file created into the app file system. After that, we still had to write the updated parent directory to the disk, so I used another helper function, overwriteData(), to provide the functionality of writing the data to the disk, with a check to see if the task was successful or not.

Problems with b_read() were that when any files that were empty got read by the function in order to cat or any other command that used b_read, it would crash and exit. This was due to a segmentation fault, where the function was attempting to read memory that was not there. To fix this, several checks were added to make sure that certain parts of the file system were not NULL or empty. This included checking if the file control block, the buffer, or the directory were NULL and checking if the count was less than 0. By including these conditional statements, the function could recognize empty files as being readable and did not crash when reading these empty files.

Kyle

Milestone 1

When using structs in other header files the make run would fail because there would be an infinite loop of importing files. I had to fix this by using #ifndef and #define to ensure that data structures would only be declared one time. I also had to forward declare structs.

I was overcomplicating the code by reading in the VCB from the filesystem every time. This added a lot of extra complexity. I resolved this by creating a global variable for the VCB so it can just stay in memory and easily be accessed.

I was receiving malloc assertion errors when running the code. This was because I wasn't malloc'ing enough memory and the read was going past the allocated memory. The reason was because I was getting memory based on the exact size of the struct and not taking into account that it would be a bit larger since I was reading it based on blocks.

Data wasn't being saved across runs. This was because I forgot to re-write data back to the volume after changing it. I also had issues with my FAT entries not being written correctly since my variable counts were slightly off. I was able to confirm this was working properly by directly reading the volume.

Initially the filesystem design didn't have any way to increase the size of a file past its last block. In order to resolve this I implemented a FAT table that keeps track of what blocks data was written at and a pointer to the next block or EOF if it was at the end.
The free space system uses a bitmap to quickly see what blocks have been allocated and which blocks haven't; however, the data wasn't changing in the volume. I realized I actually forgot to update it with the allocated blocks after writing the VCB, FAT, and the free space bitmap and committing it to the volume.

When running the program after the volume was already initialized the pointers for the FAT table and the free space bitmap weren't working. This was because every time you run the program the pointers change, so the written portions in the VCB are essentially garbage data after the program finishes running. I solved this by reading the two data structures from the volume and then saving them to the VCB in memory if the volume was already initialized.

Because a FAT table is being used, that means there is a chance that the data is not contiguous. This isn't really an issue now since nothing can be unallocated, but when I was re-writing the function that writes data to the disk, I kept on getting duplicate data for every block. The issue was that I needed to increment the buffer pointer by the block size in order for the rest of the data to properly be written.

Milestone 2

One of the hardest to debug issues in the code was that there were malloc assertion errors all over the code after I finished writing the mkDir function. These errors were happening because the allocated memory wasn't enough and the read function was reading past the allocated memory. This was because when mallocing data for functions I calculated the exact amount of space in bytes instead of the space to the nearest block. Because the read functions were reading in blocks this meant that it would read empty data in to fill the rest of the allocated memory that should have been there.

In the VCB there are two structs that are malloc'd, the free space bitmap and the FAT Table. The bitmap would be erased shortly after runtime and data would then be written to the wrong spots in the filesystem. This bug was hard to realize without print statements for the location of the write because the VCB was getting overwritten and on the next run the signatures would not match up and then the file system would be recreated. The issue also ended up being that I was allocating too much space for the FAT table and I guess it was overwriting the free space bitmap in memory because it would be all zeros when read back after allocating the memory for the FAT Table. Calculating the correct amount of space for the malloc ended up solving this issue.

When creating a folder with mkDir the folder would be written to the storage, but when calling LS it would not show up. This ended up being because when a folder is created in the cwd the g_cwd global variable was not being updated and so when reading it to list the files it wouldn't show up. This did not occur when creating a file in a different directory than the cwd or rootDir because those directories would have to be read in again and thus update from the newly written data in the file system. The same issue also occurred with rmDir and delete.

rmDir would remove folders that still contain files in it. This would unlink the data and essentially make all the existing files in the folder wasted space. This was due to the check for data only seeing if there were directories. Modifying this check solved the issue.

When using the overwrite function in freeSpace.c it would take into account if new data needed to be allocated; however, it would not link the newly allocated block to the existing allocated blob and there would just be creating a new block of invalid data. After writing the data I modified the FAT Table to link these two blobs together so the data could properly be written.

A commonly occurring issue in functions that I was using to read and write to the filesystem was that I forgot to advance the buffer by the blocksize after something was read in. This would cause the same block to be written over and over to the filesystem for example. Incrementing the buffer properly solved this issue.

There would be occasional segfaults when changing directories, running LS or pwd, and making directories. This was because some directories weren't properly being loaded in and had invalid loc attributes that would then be invalid and cause the program to read in nonexistent data as well as mess up calculations. After fixing the amount of data I was allocating in malloc these issues were resolved. I also had an old SampleVolume so I had to remove it since the data in it got corrupted and that was causing further issues.

Neal

Milestone 1

There was some confusion during the making of the VCB struct, mainly on the member variables and their roles and purposes. This was clarified after reviewing what was discussed in class lectures, and applying it to our project.

Milestone 2

The function cleanPath() suffered from multiple issues. One issue was that the function did not properly handle empty input paths, and outputted empty strings. This was fixed by reworking the handling for empty input paths to preserve the string representing the current directory path, rather than setting the output path to a string without characters. Another issue was that cleanPath() frequently failed to detect and properly handle input path strings. This issue was rectified by increasing the strictness of the if statement conditions for determining which string was an absolute path or a relative path, as well as making support for absolute paths more robust. Another issue for this function arose from the refactoring of mfs.c, the .c file where cleanPath() was located. This function became a source of segmentation faults. We eventually

found out that this function was attempting to access unavailable memory because the string manipulation features were counting to the wrong value. Setting these features to count to the proper value reduced the amount of segmentation faults, as well as tightened up the memory management.

There was a bit of confusion during the making of the function fs_closedir(). This was mostly because I over thought about the requirements of this function. This was clarified once I realized that all this function needed to do was free a pointer and set it to null.

For my part in the parsePath() function, I was lost on managing the char pointer that represented the path. After reviewing the video lecture recordings on how parsePath() is supposed to work, I got a stronger grasp on what is expected of this function, and that I was meant to tokenize the string and handle each token.

Milestone 3

The making of the function b_seek suffered from a minor issue. There was much overthinking about how the required arithmetic was meant to be implemented. However, after studying the Linux manpages about the function that b_seek was based on, b_seek was completed. Turns out we are expected to determine which preset value whence was equal to. The preset values are provided by <unistd.h> . After recognizing the value for whence, the required math could be conducted for each case.

The problems surrounding b_close() mostly involved keeping track of the correct struct members. Properly closing a file required this function to pass the proper values to certain helper functions, as well as utilizing their return values. The solutions stemmed from remembering the path of pointers: fcb->fdDir->DE. This knowledge allowed this function to pass the required values to overwriteData() in order to write dirty buffers to disk, as well as update the DE to the proper values. One more issue was that the fd was not being released. This was because the implementation was forgotten. Once remembered, this was fixed.

Nicholas

Milestone 1

There was some confusion in the beginning of the project with some of the variables of the VCB and initializing them, but after closely looking at the structure we designed, and reviewing the material in class, things became more clear.

Milestone 2

One of the main issues with milestone 2 functions for me specifically, was working on fs_mkdir in conjunction with parsePath(). One of the early issues with parsePath(), was that the code was freeing up things more than it needed. Which led to a compilation error when the function was called. This was ultimately solved, by tracing back to what was actually being freed and removing anything that was unnecessary. Secondly, The other biggest issue with the initial version of mkdir, was there were many segmentation faults while trying to run the program. The approach I took to fixing these issues was by documenting out most of the code and introducing each chunk one at a time. This helped isolate where the segmentation faults were happening, and made troubleshooting the problem easier. Through this I found out that a majority of the problems resided in mallocs and how I was accessing pointers. By breaking down the code with comments, the solution became easy, and cleaning up the code was straightforward. In addition, to mkdir, there were some other issues with mallocs, but those were easy to isolate and fix. The next biggest issue was with mkdir. The function itself seemed to be compiling, but the actual function did not write the directory into our file system. I was not able to reasonably find a solution to this issue.

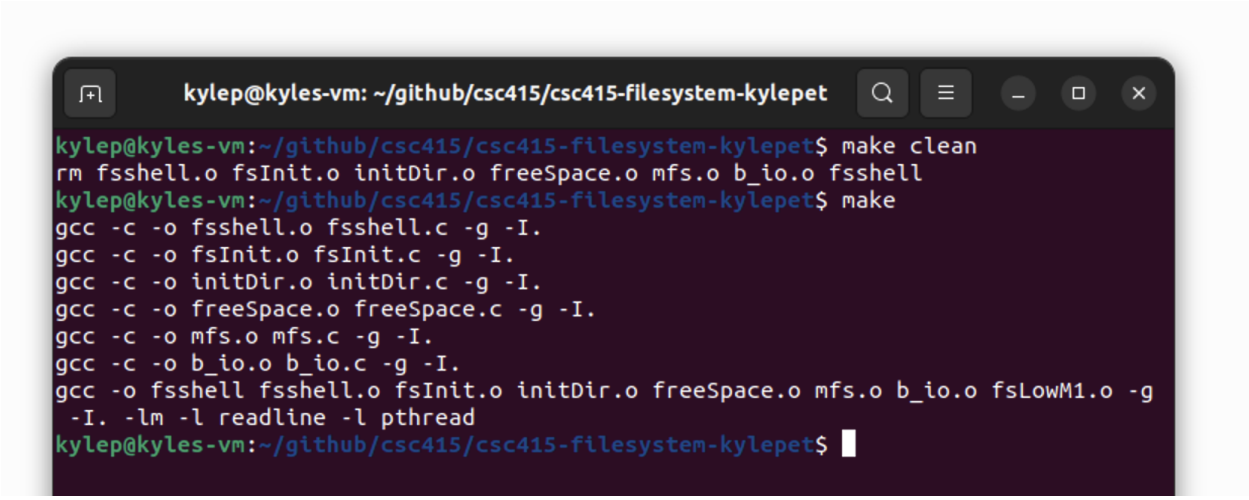
Milestone 3

The main issues that occurred in milestone 3 was with understanding how logically the write function works within our file system, and understanding the variables of all the moving parts of the system. Having to deal with different structs was disorienting, but by carefully tracking what each variable represents things became more clear. Lastly, there were issues with cp and cp2fs commands where they would show not having an invalid file descriptor, even though the file used was in the file system. I was not able to figure out this issue and it made it hard to fully test the write functions.

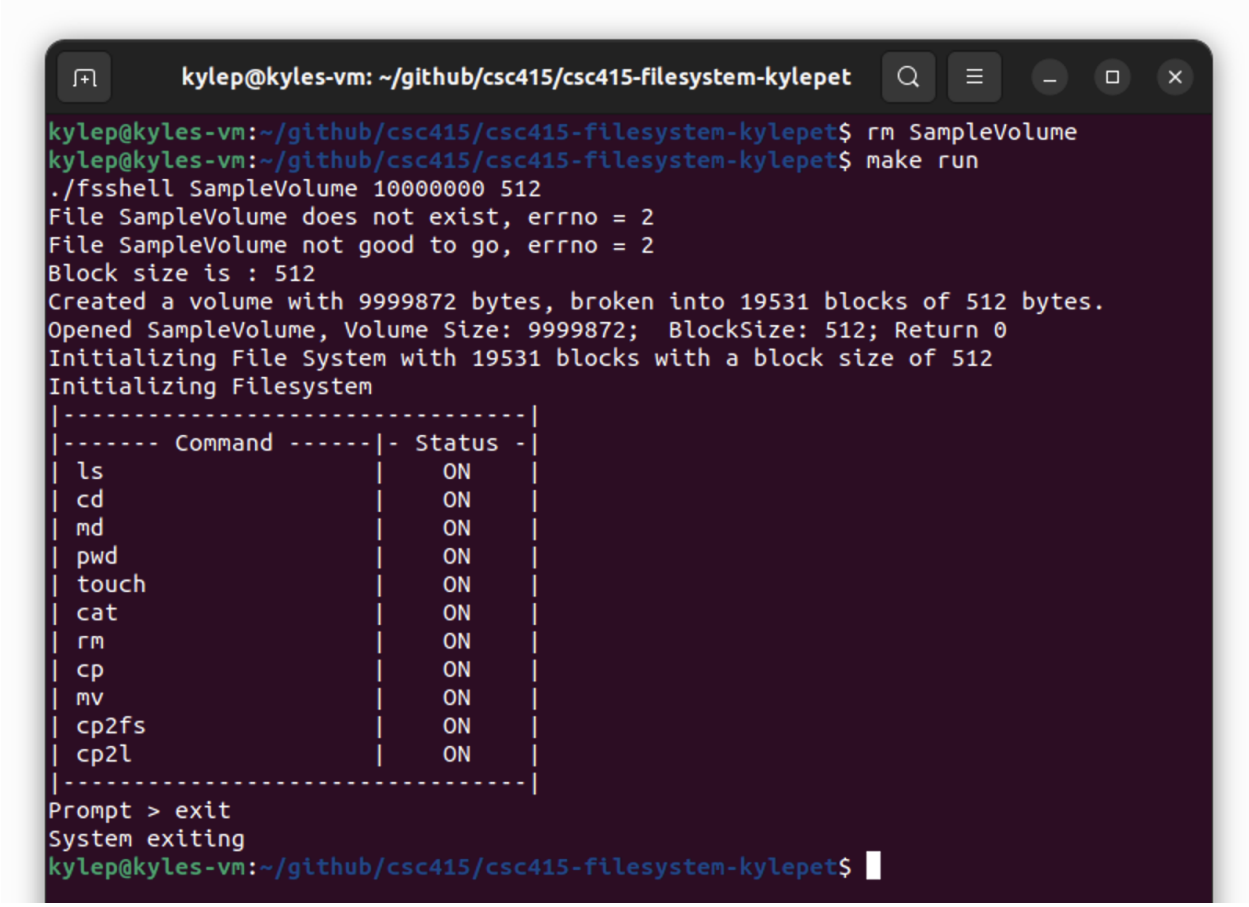
Command Execution Screenshots

Make

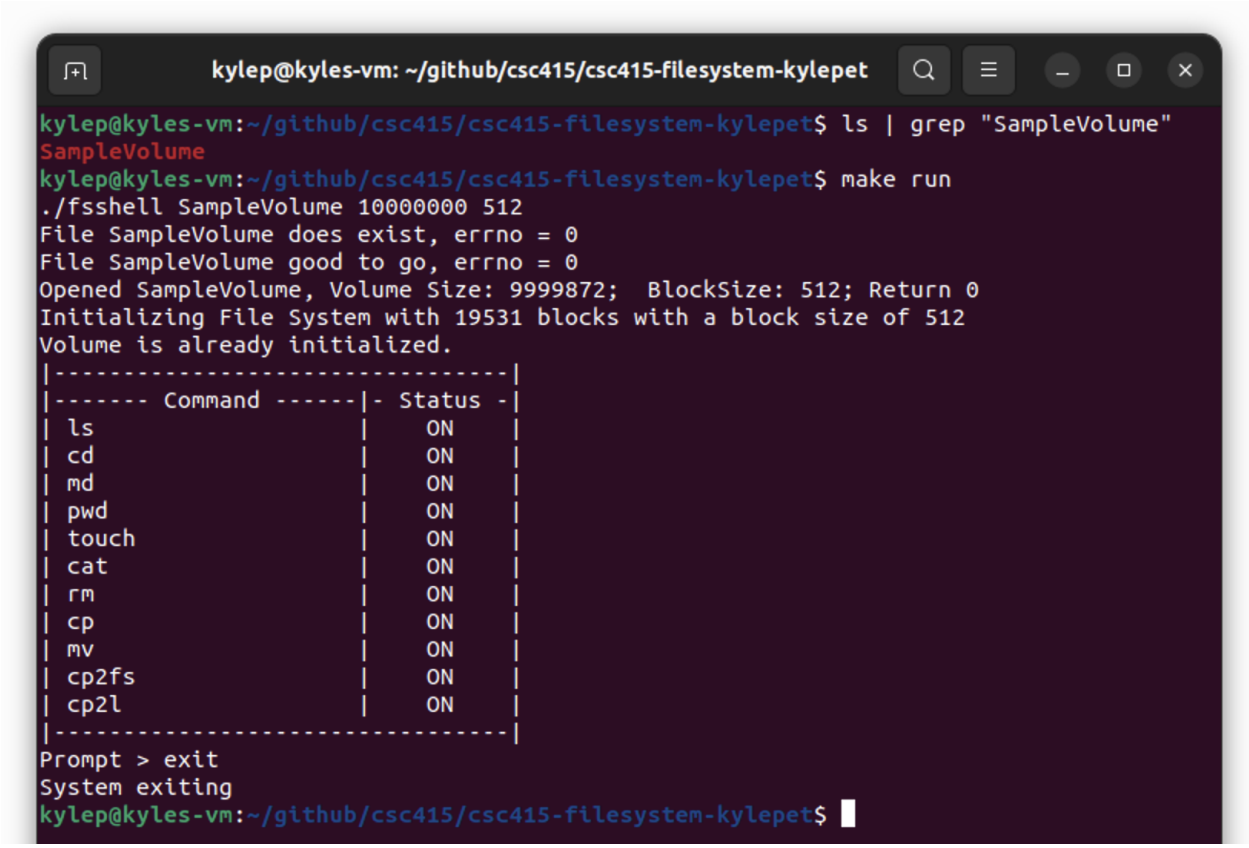
Make



Make run with no Filesystem

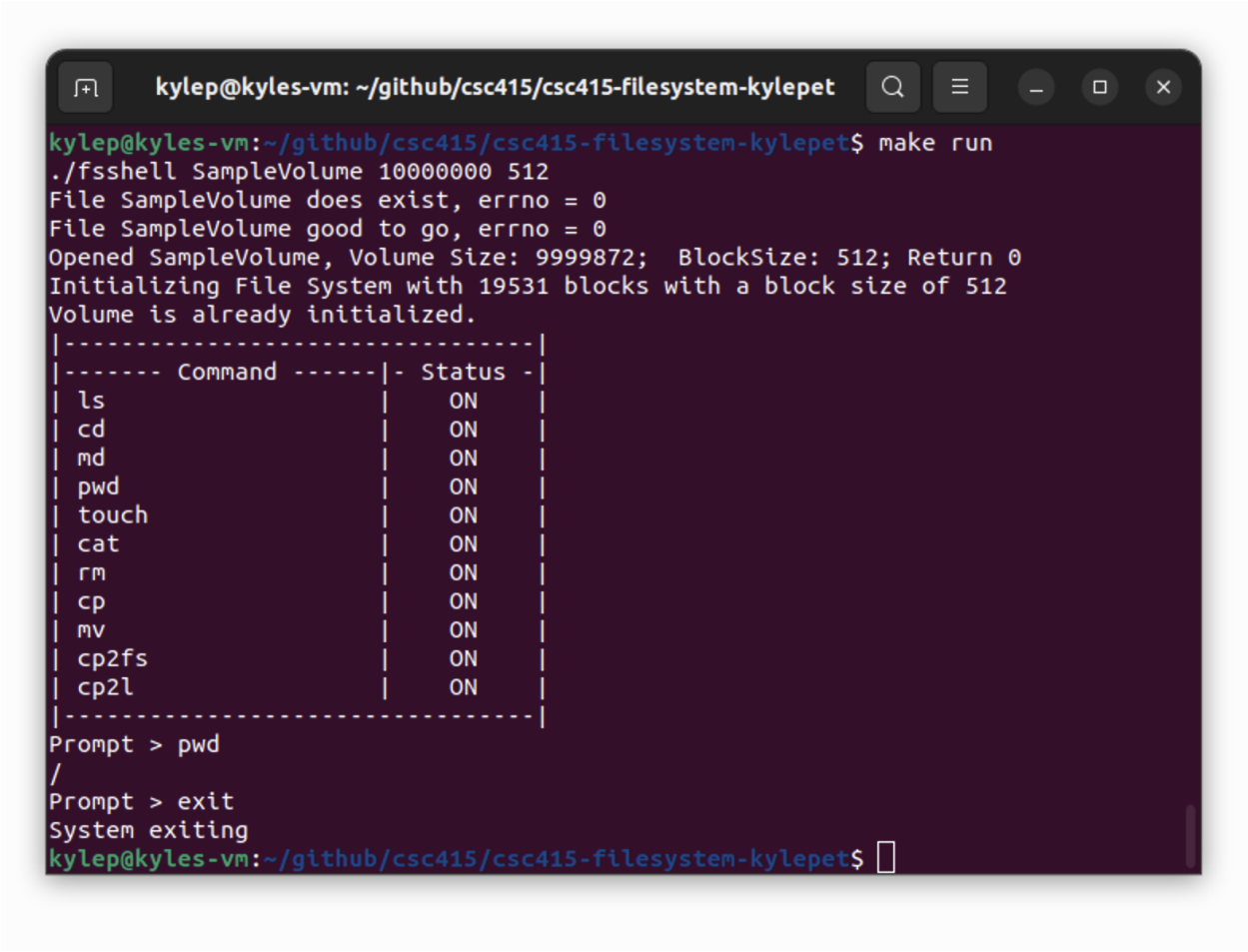


Make run with filesystem

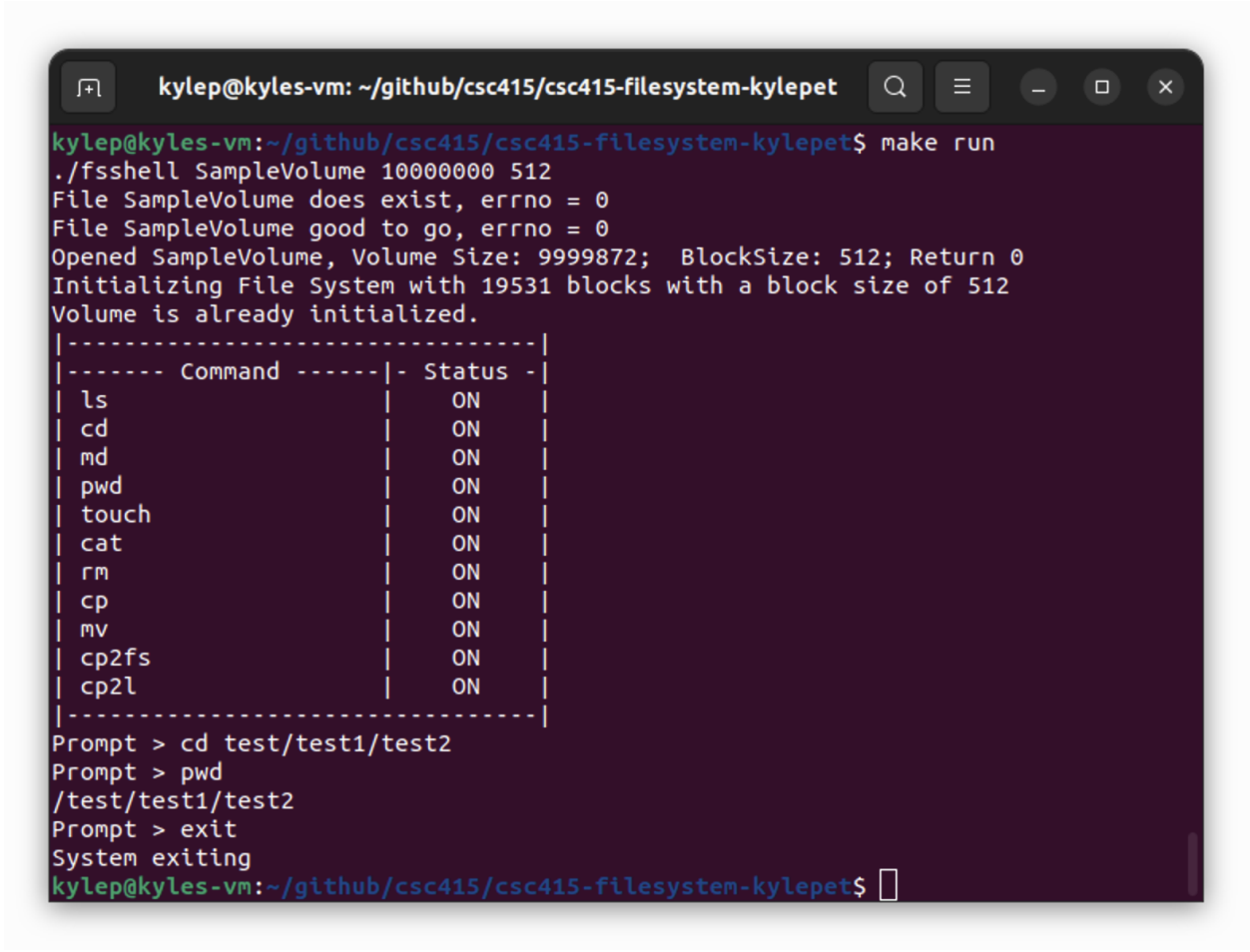


pwd

pwd in root

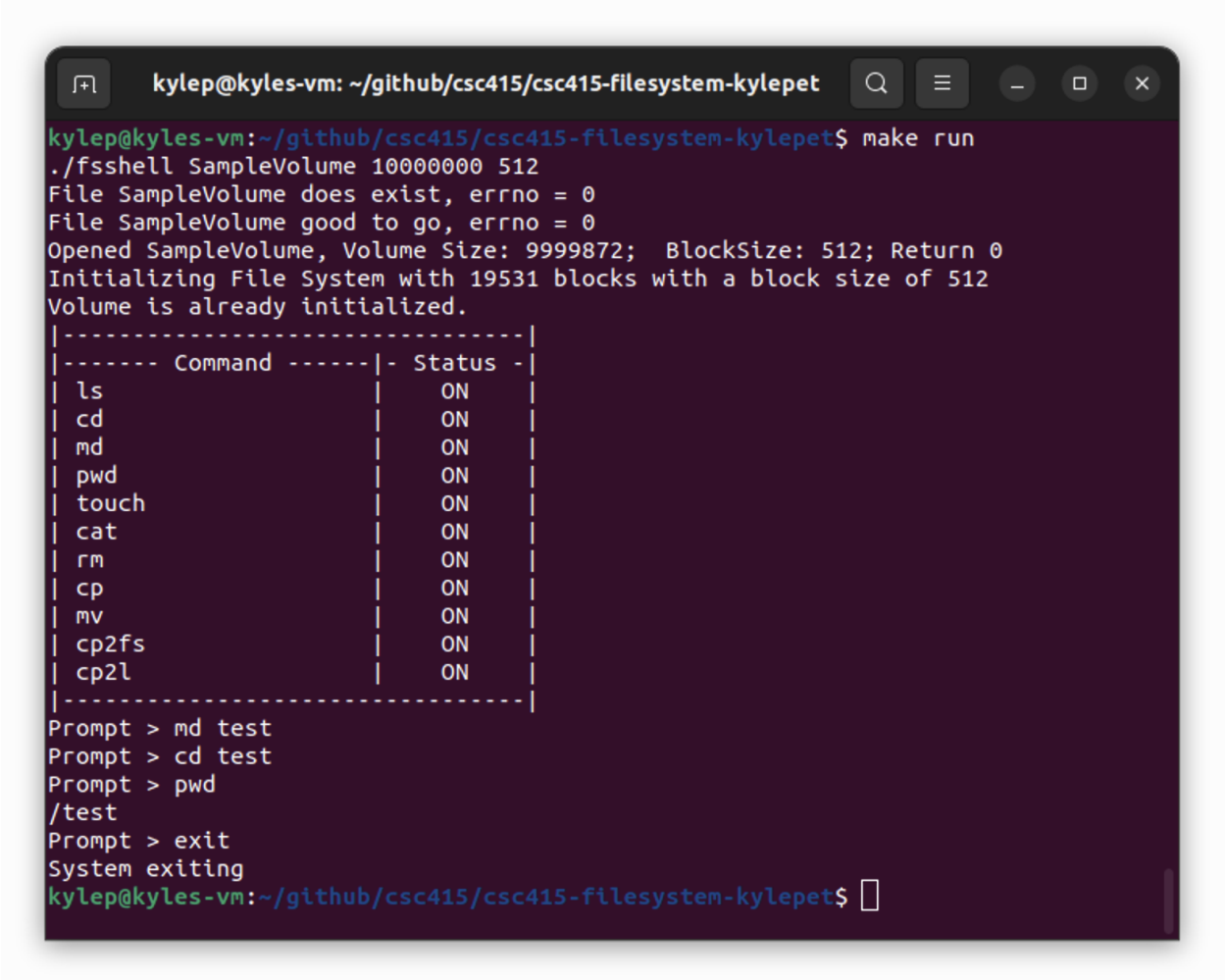


pwd in subfolder

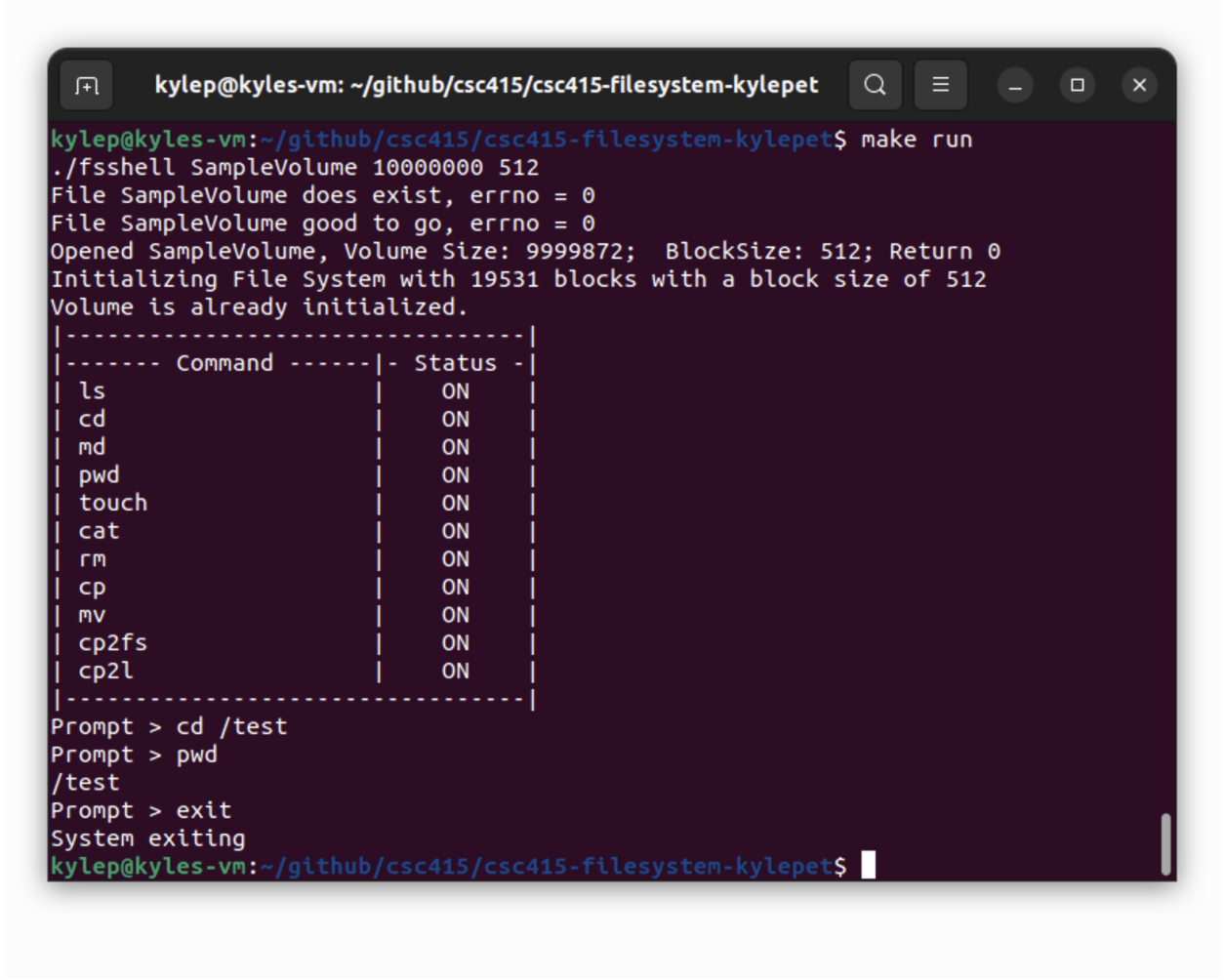


cd

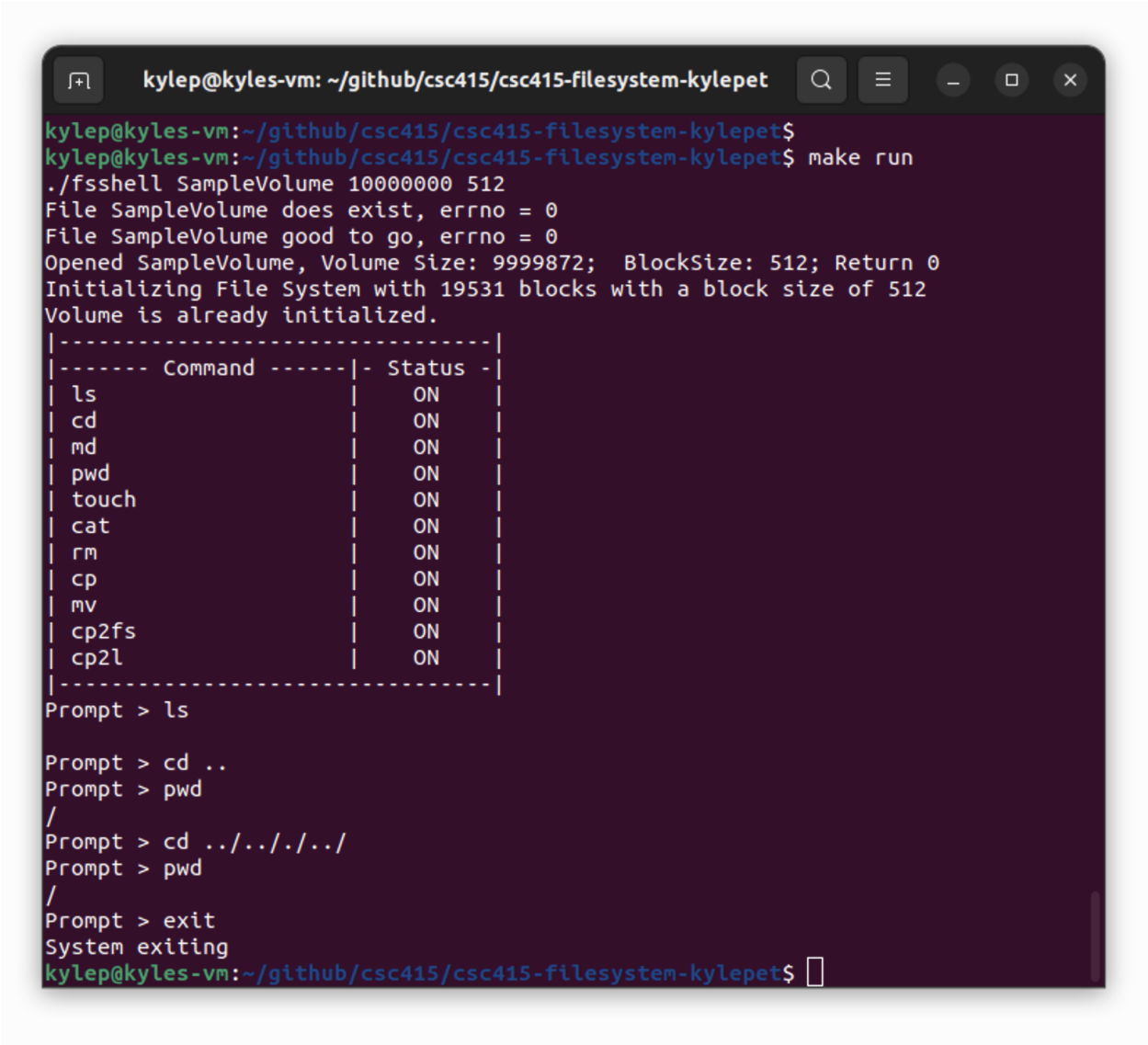
cd into existing folder with relative path



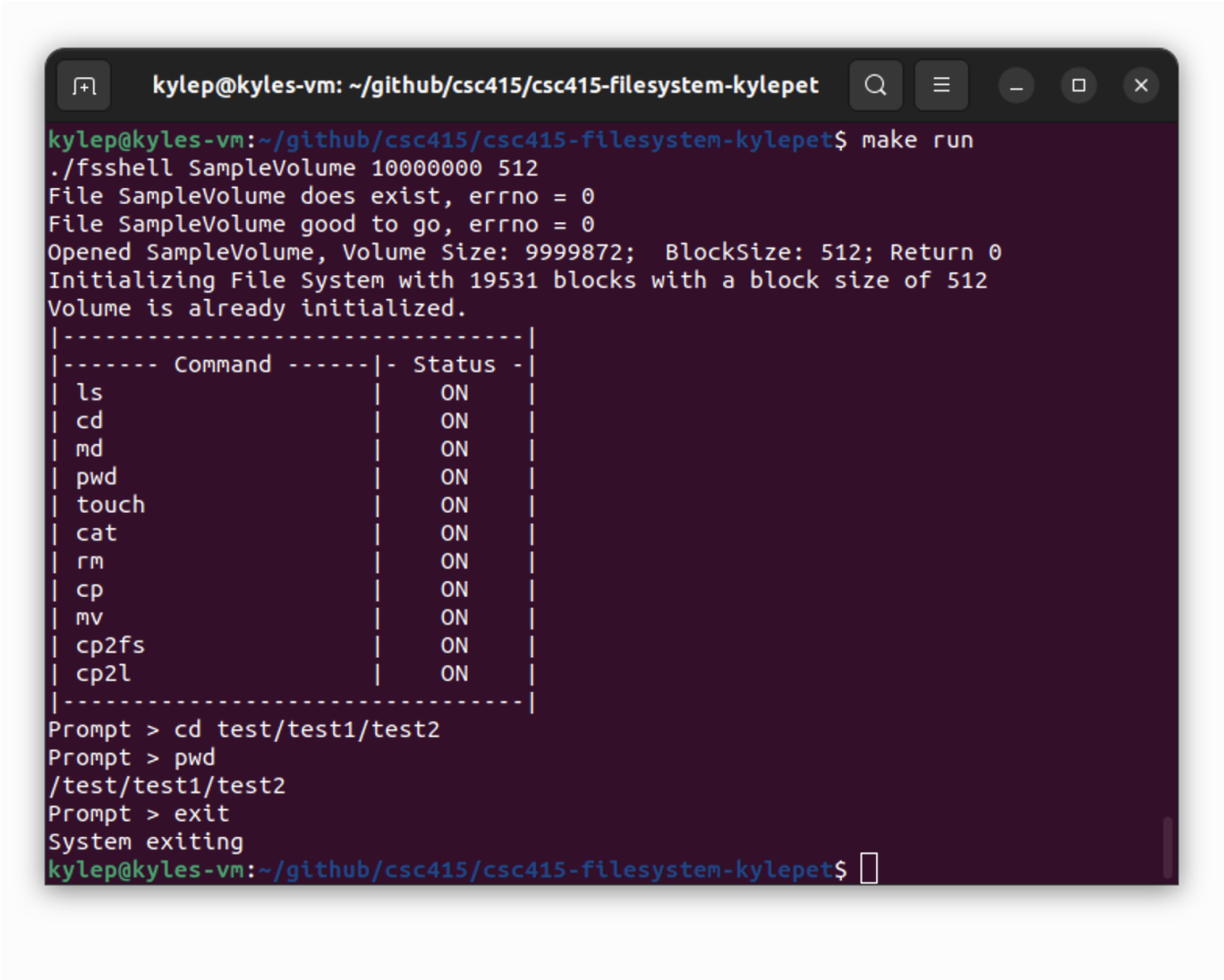
cd into existing folder with absolute path



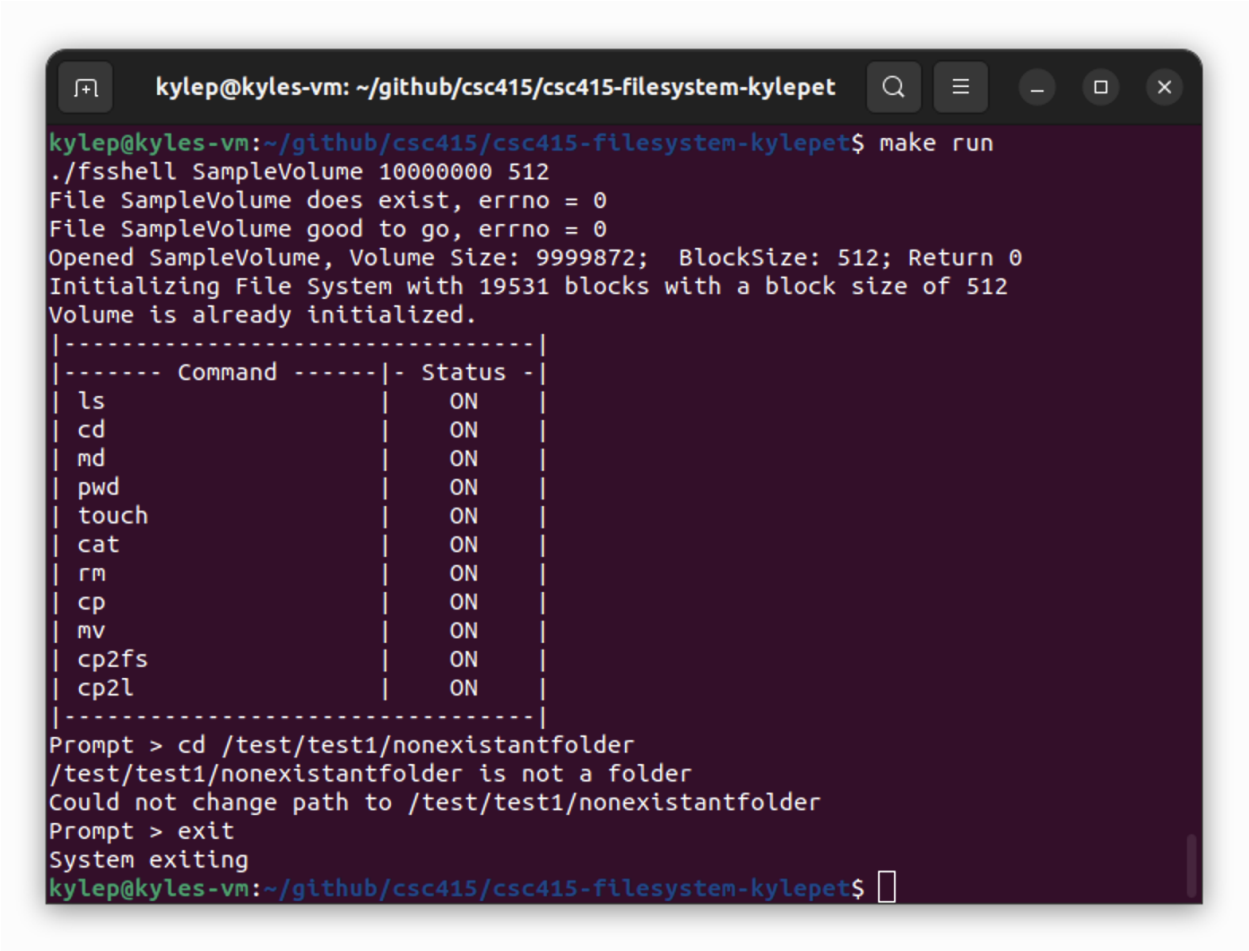
Parse “..” and “.” in cd at root



cd directly into existing subfolder

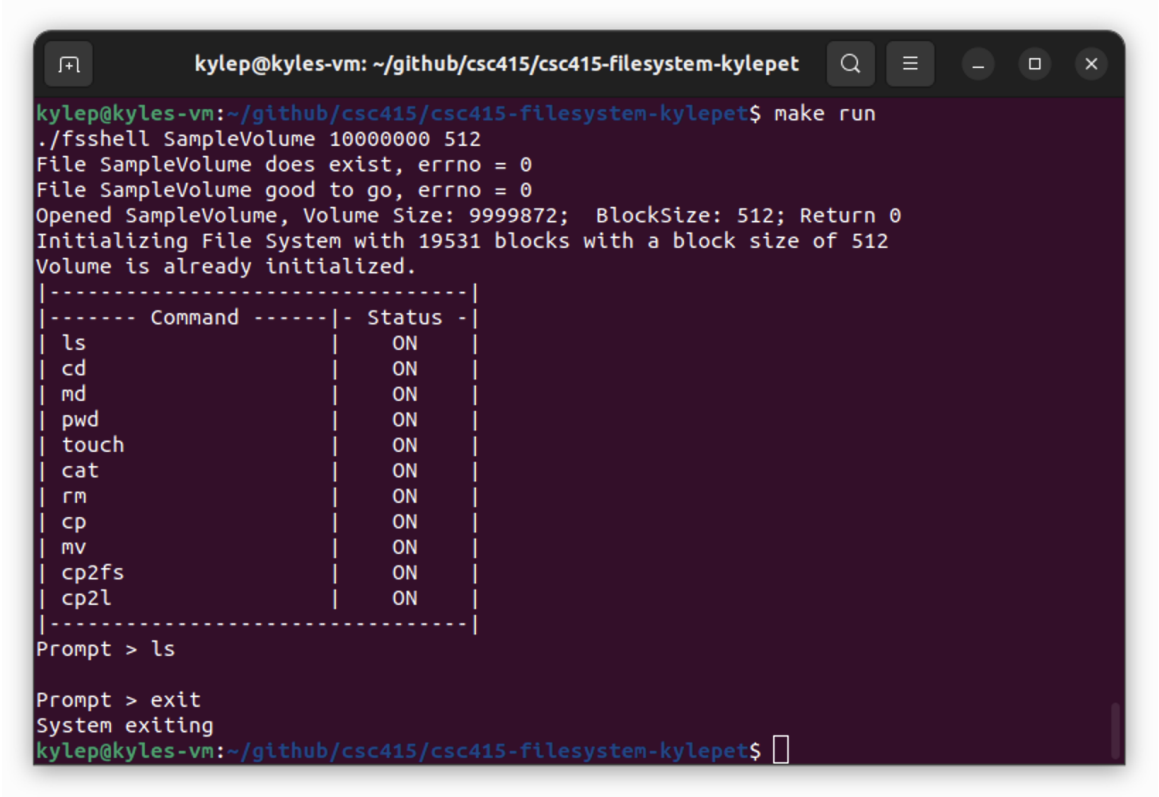


cd directly into non-existing sub-folder

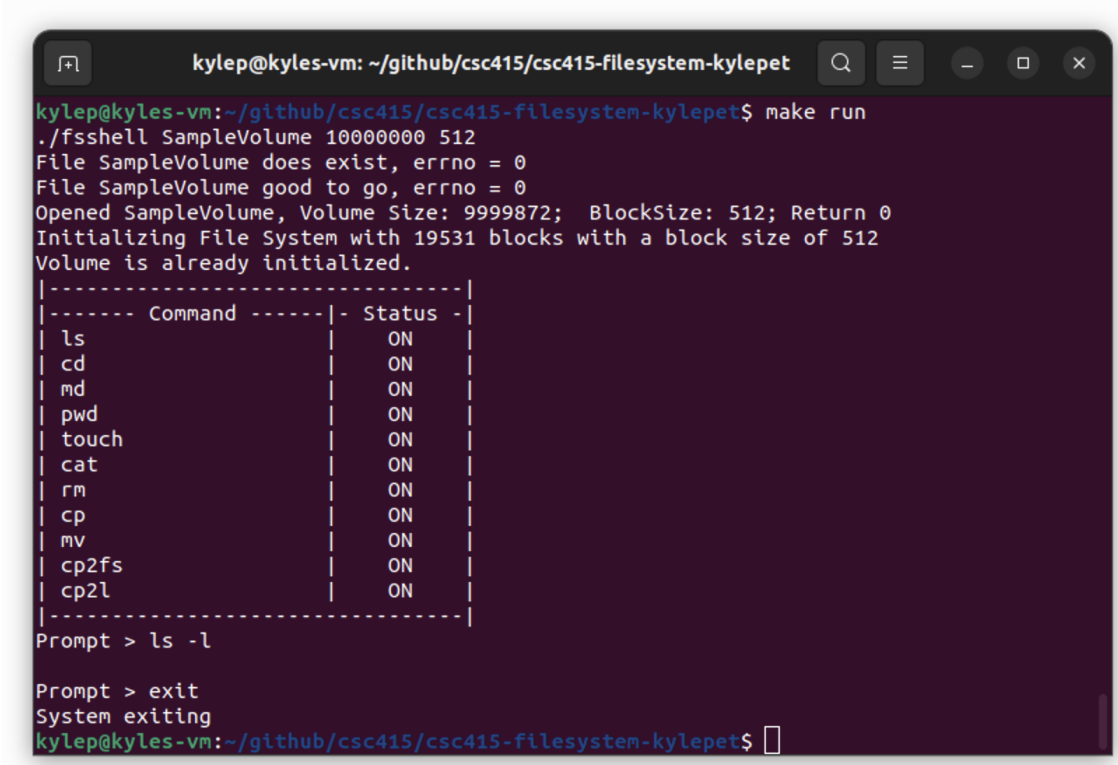


ls

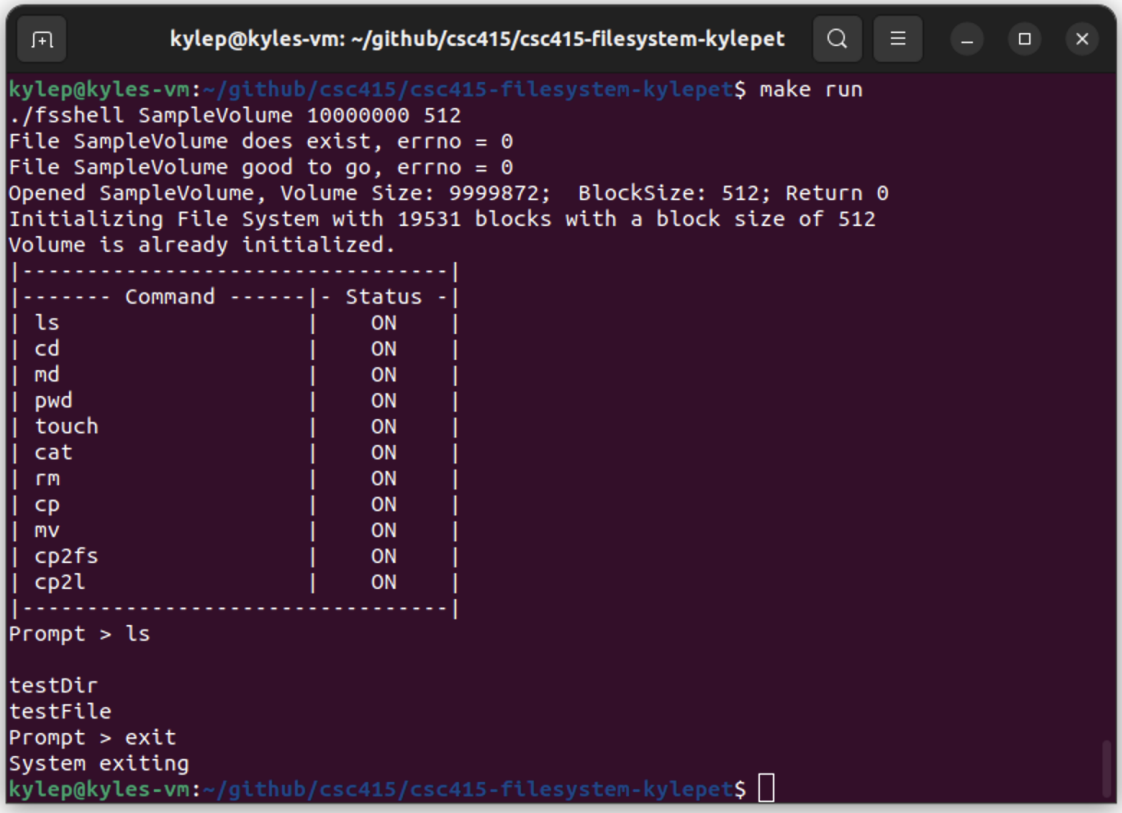
ls with no files



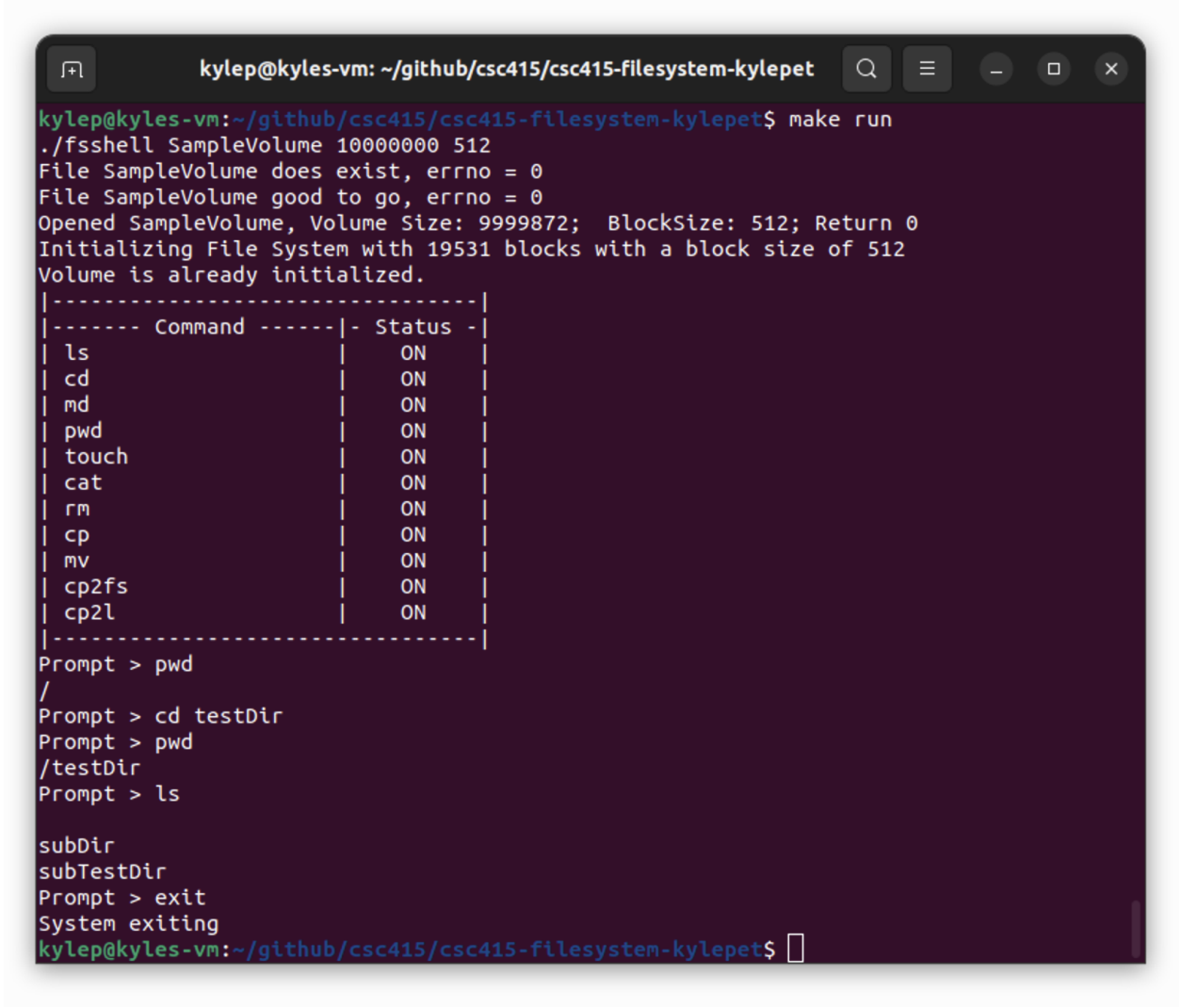
ls -l with no files



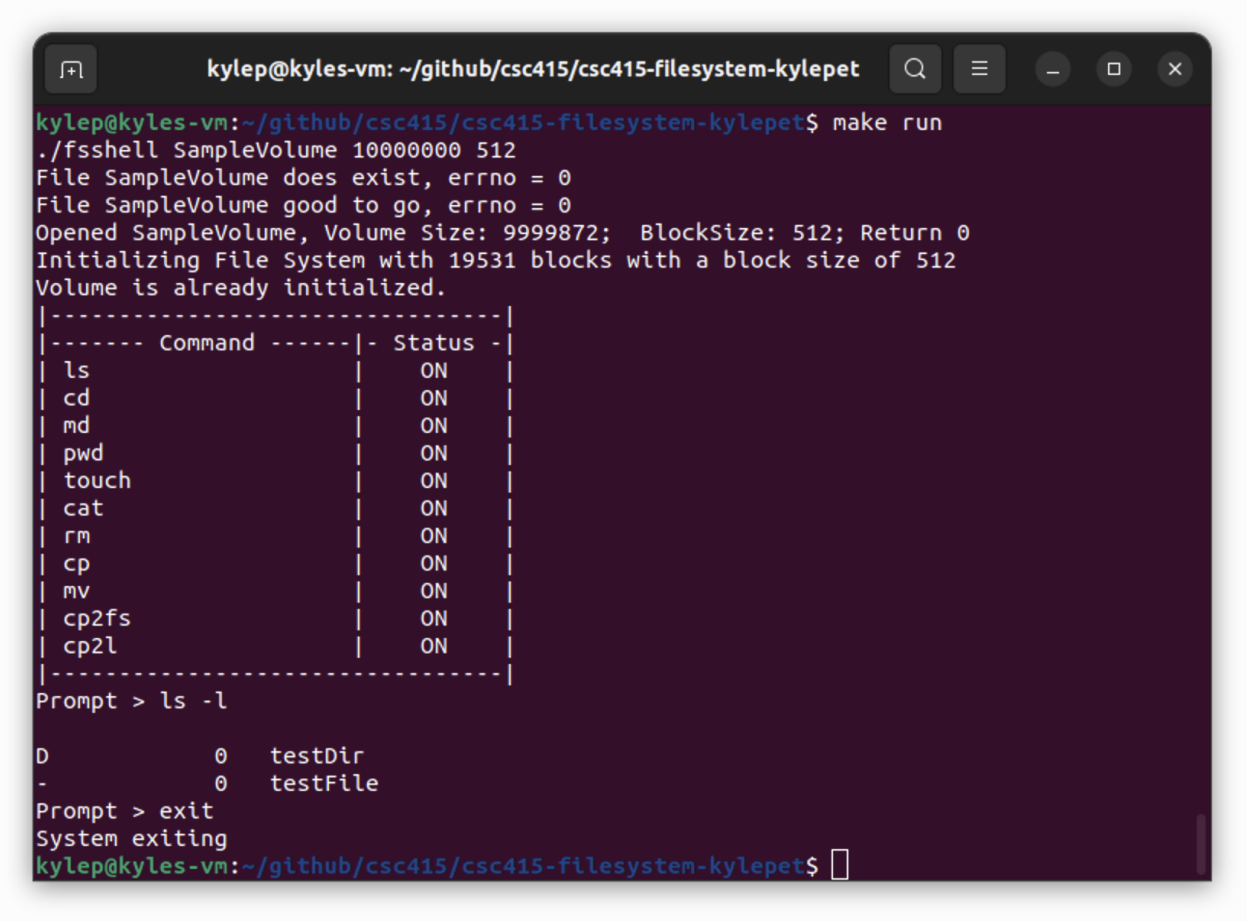
ls with files



ls with files in sub directory

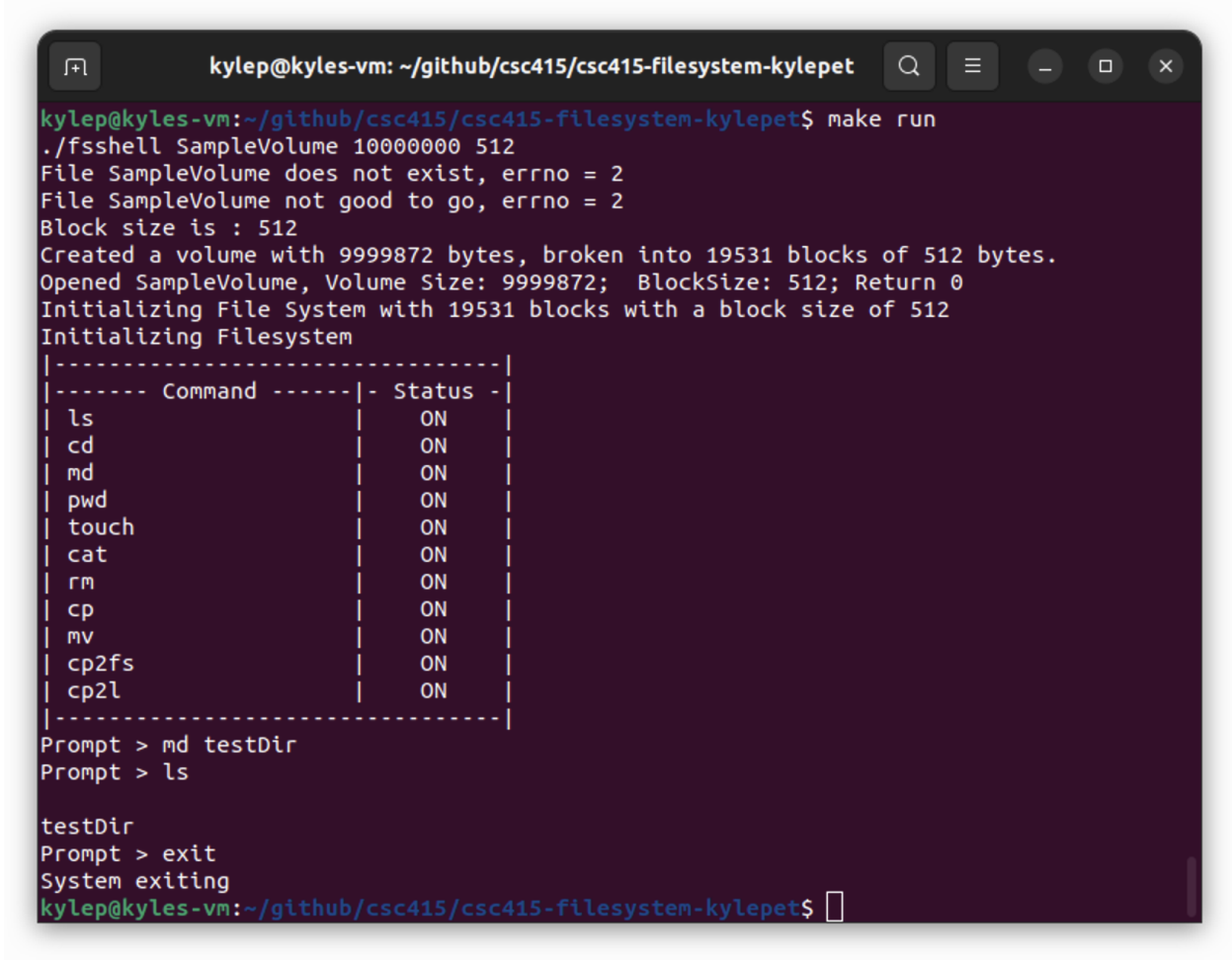


ls -l with files

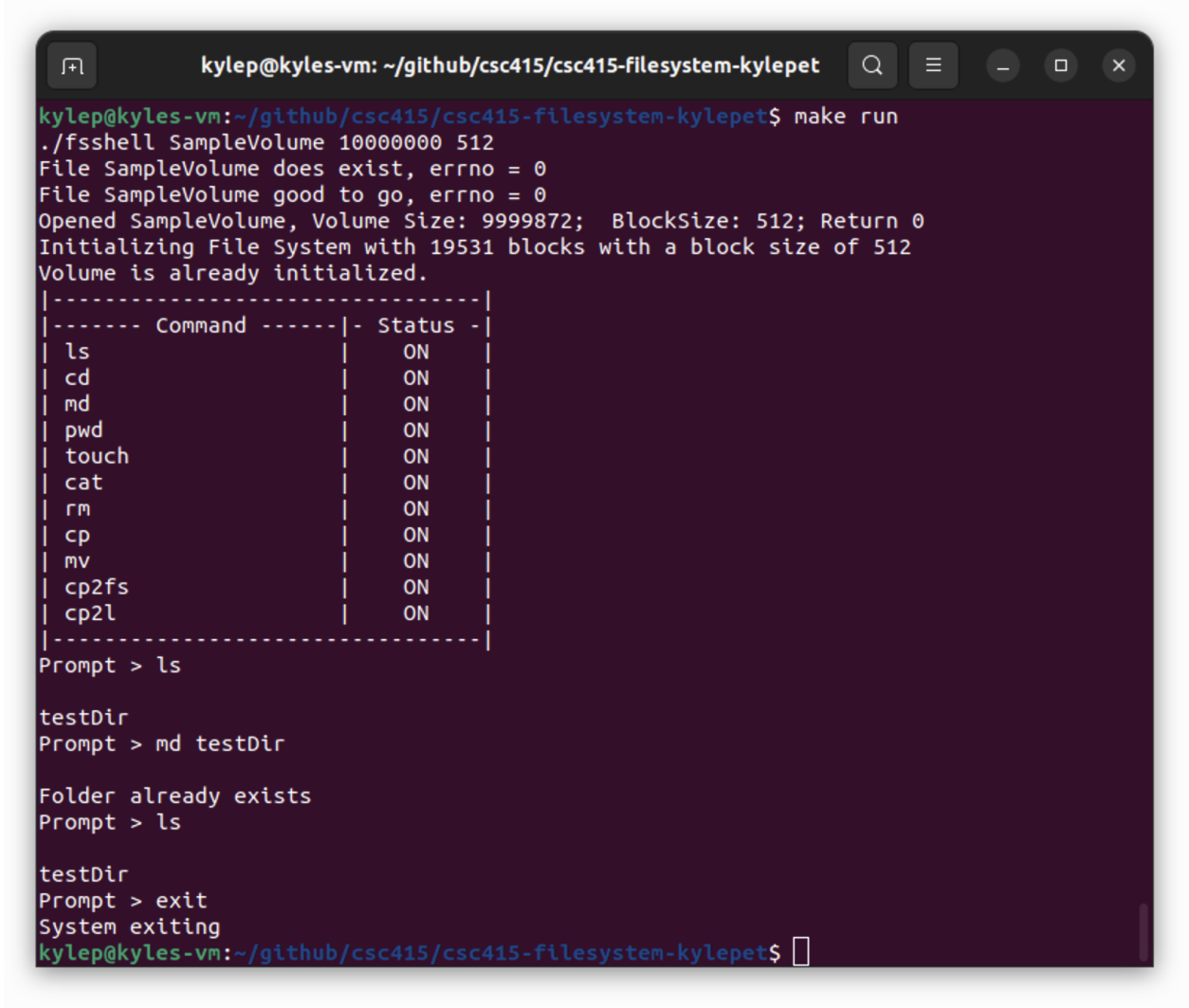


md

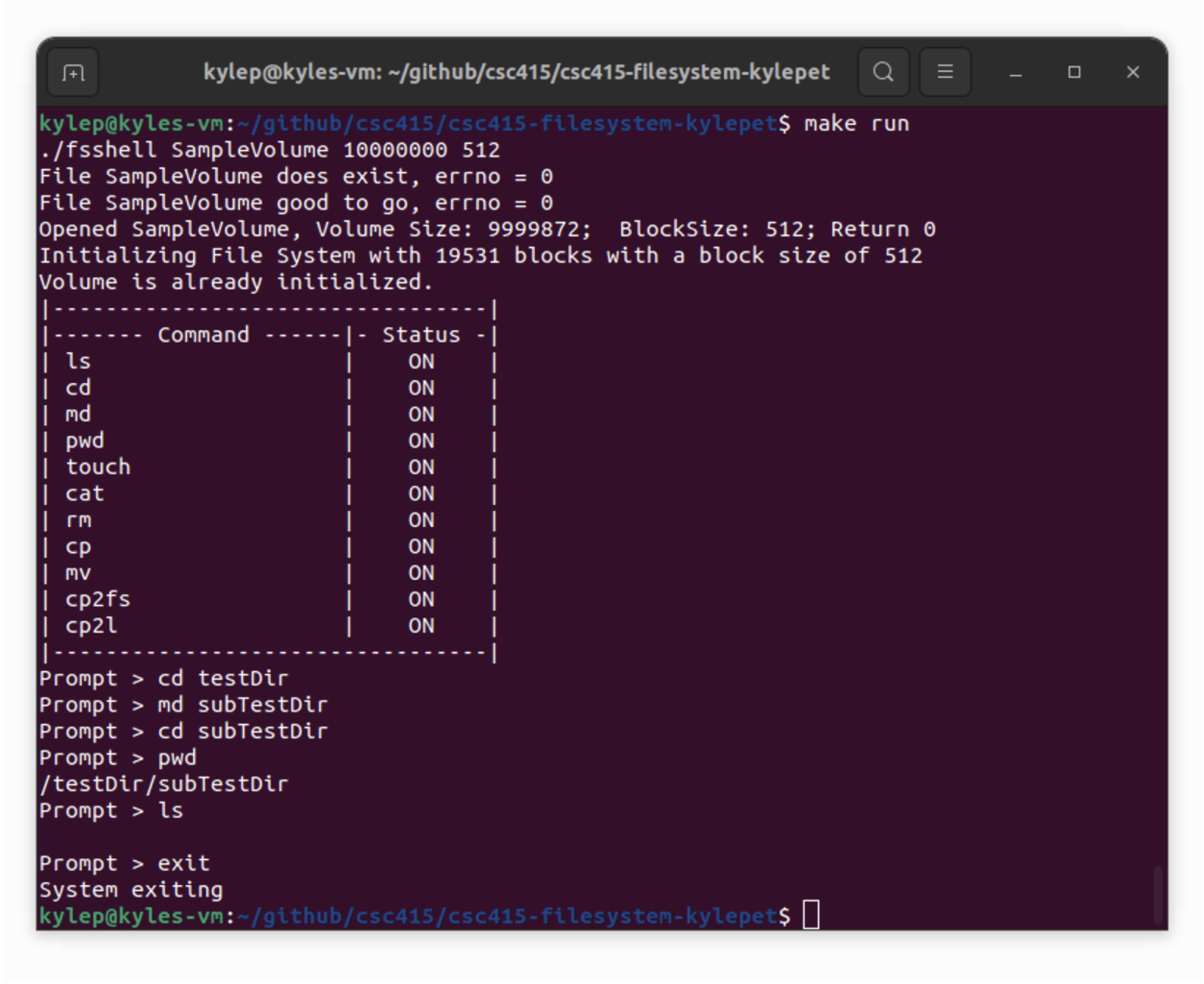
md for new folder



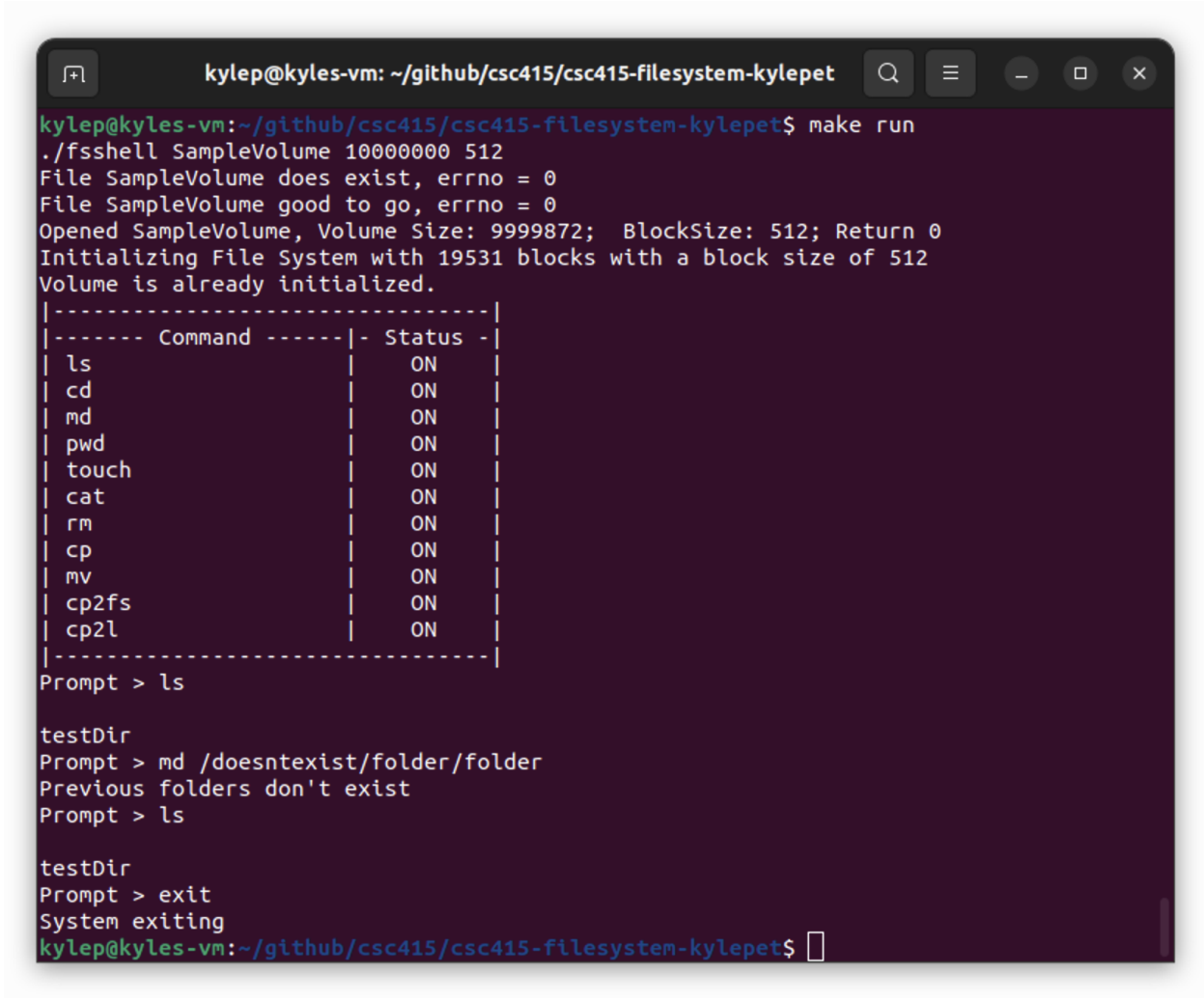
md for existing folder



md for folder in subdirectory

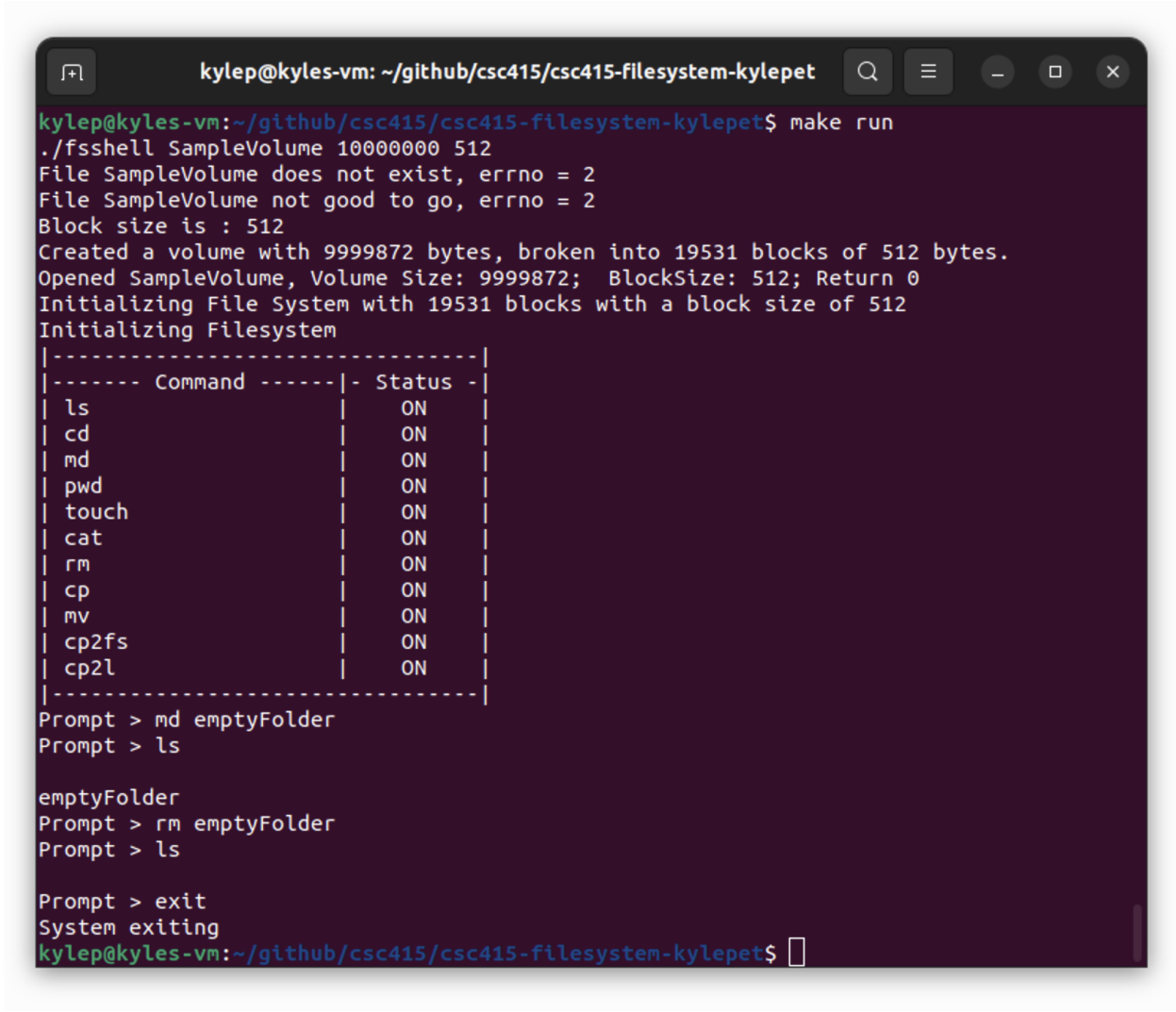


md for folder whose parents don’t exist

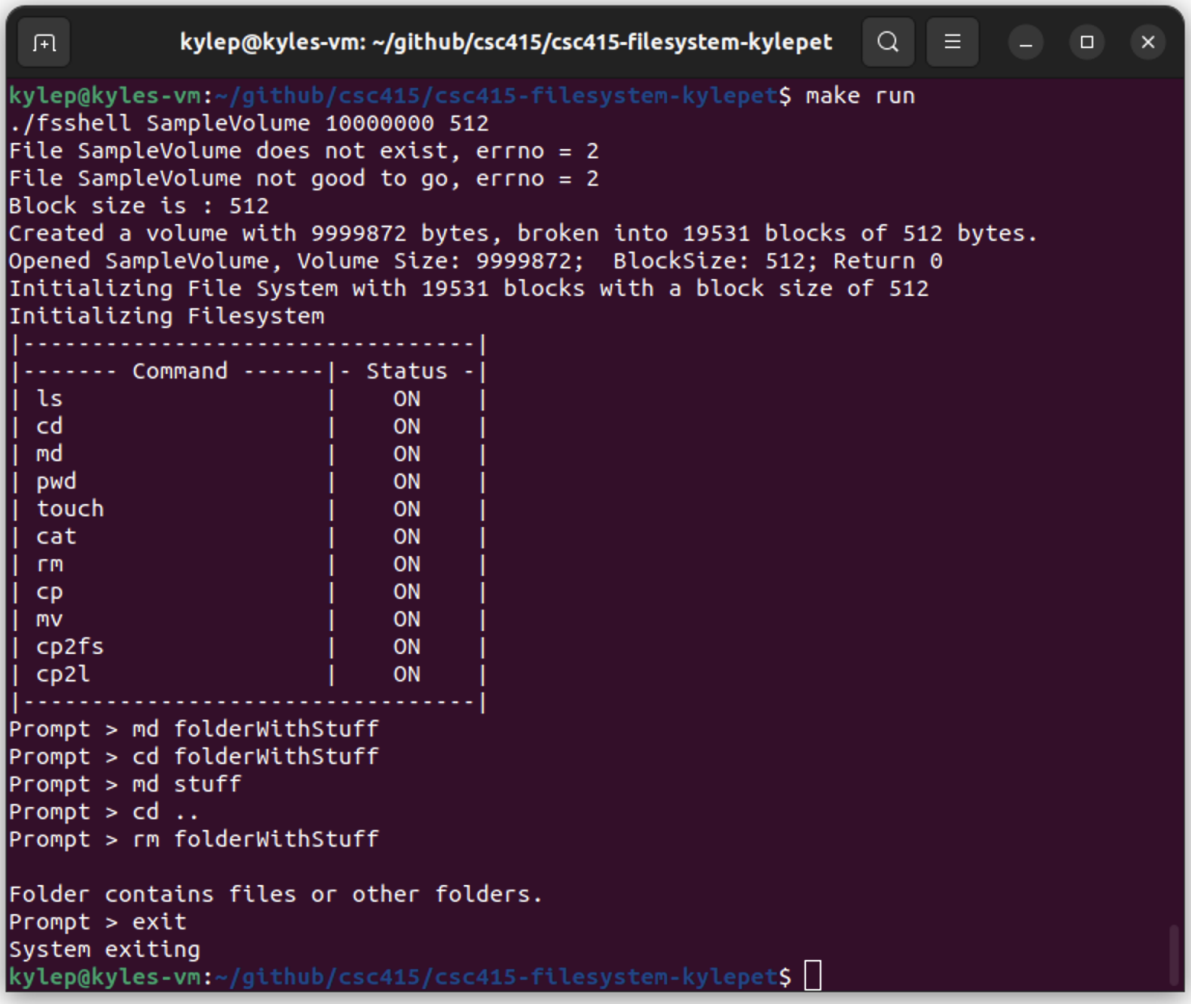


rm

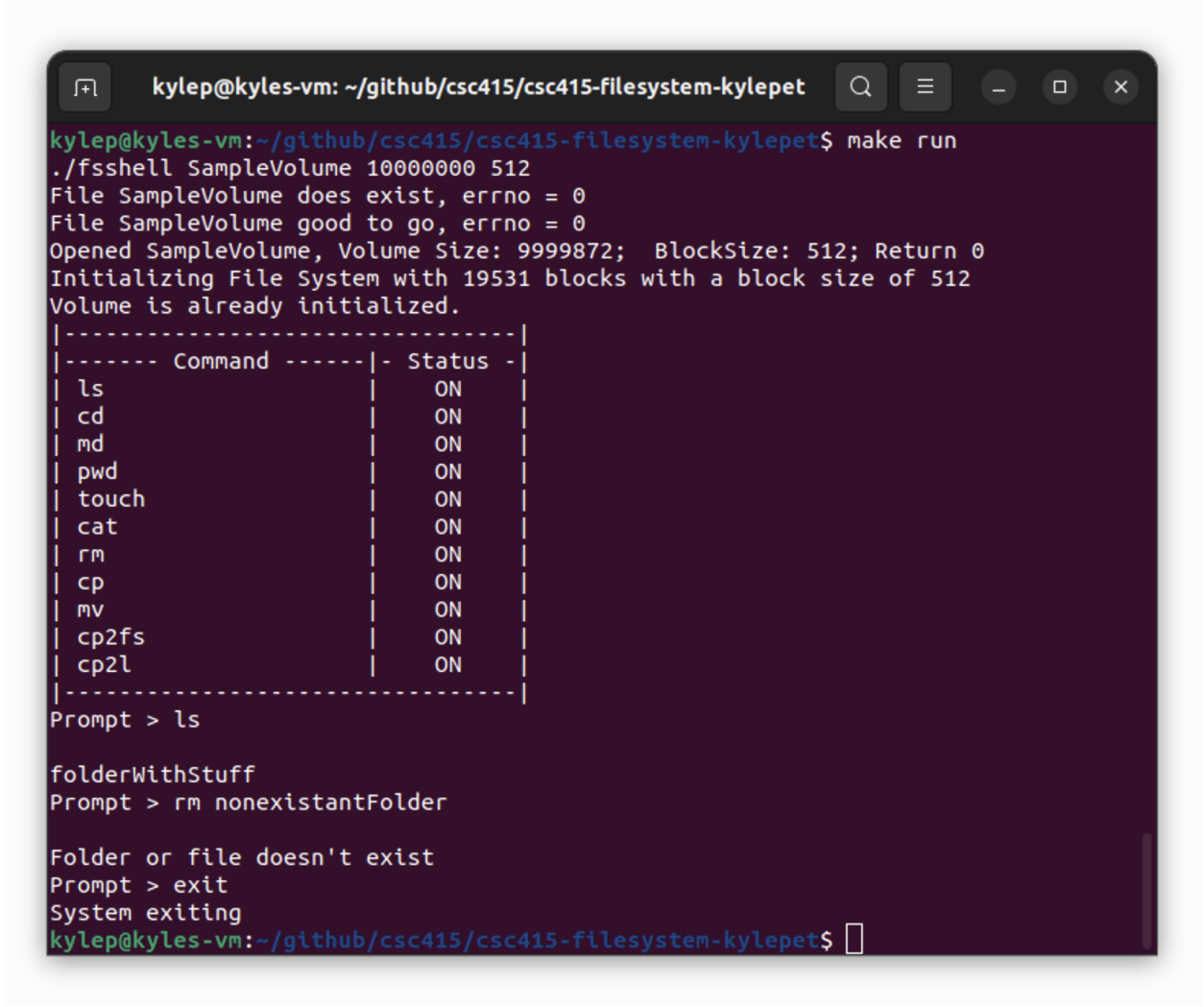
rm folder with no contents



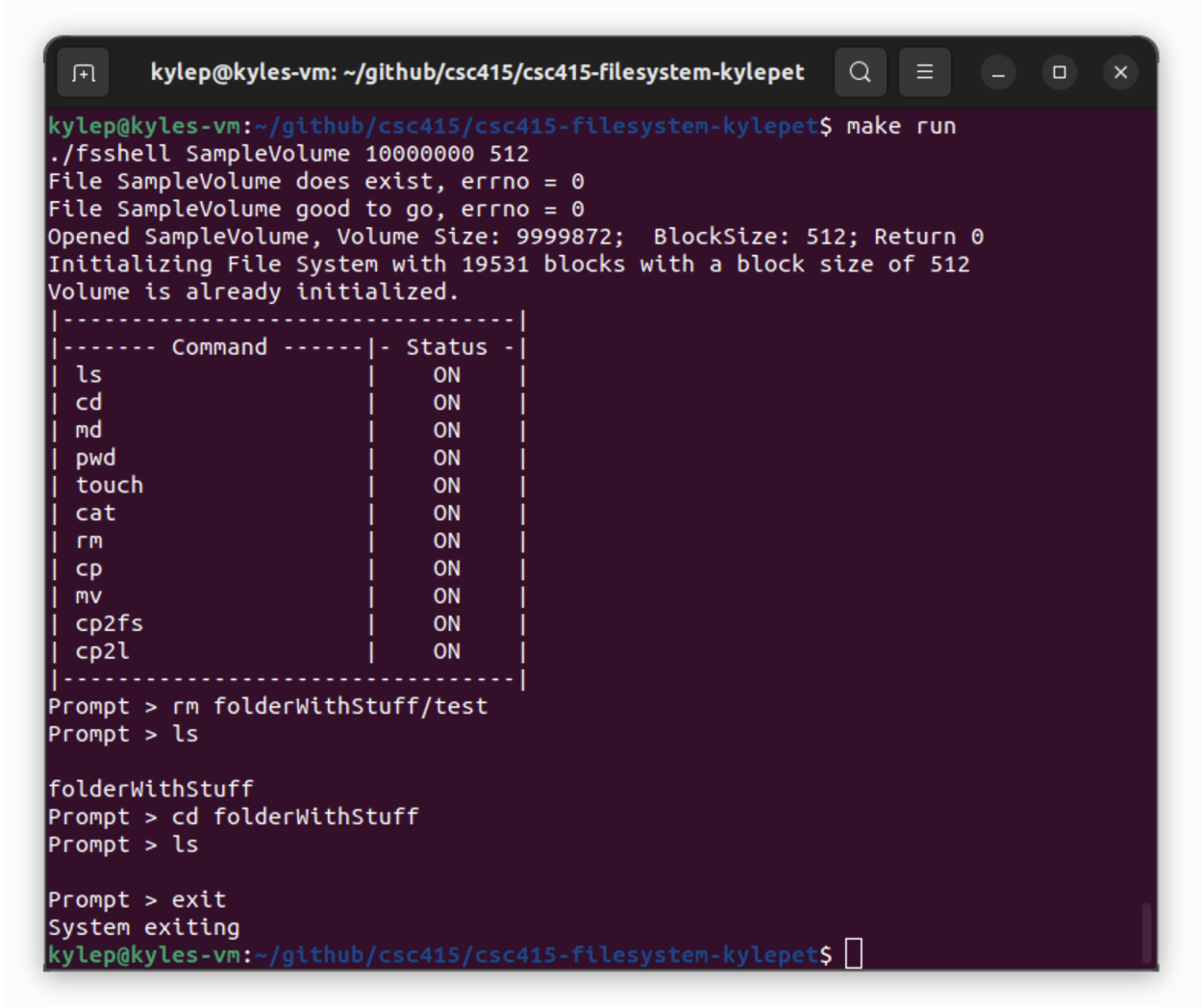
rm folder with files and directories



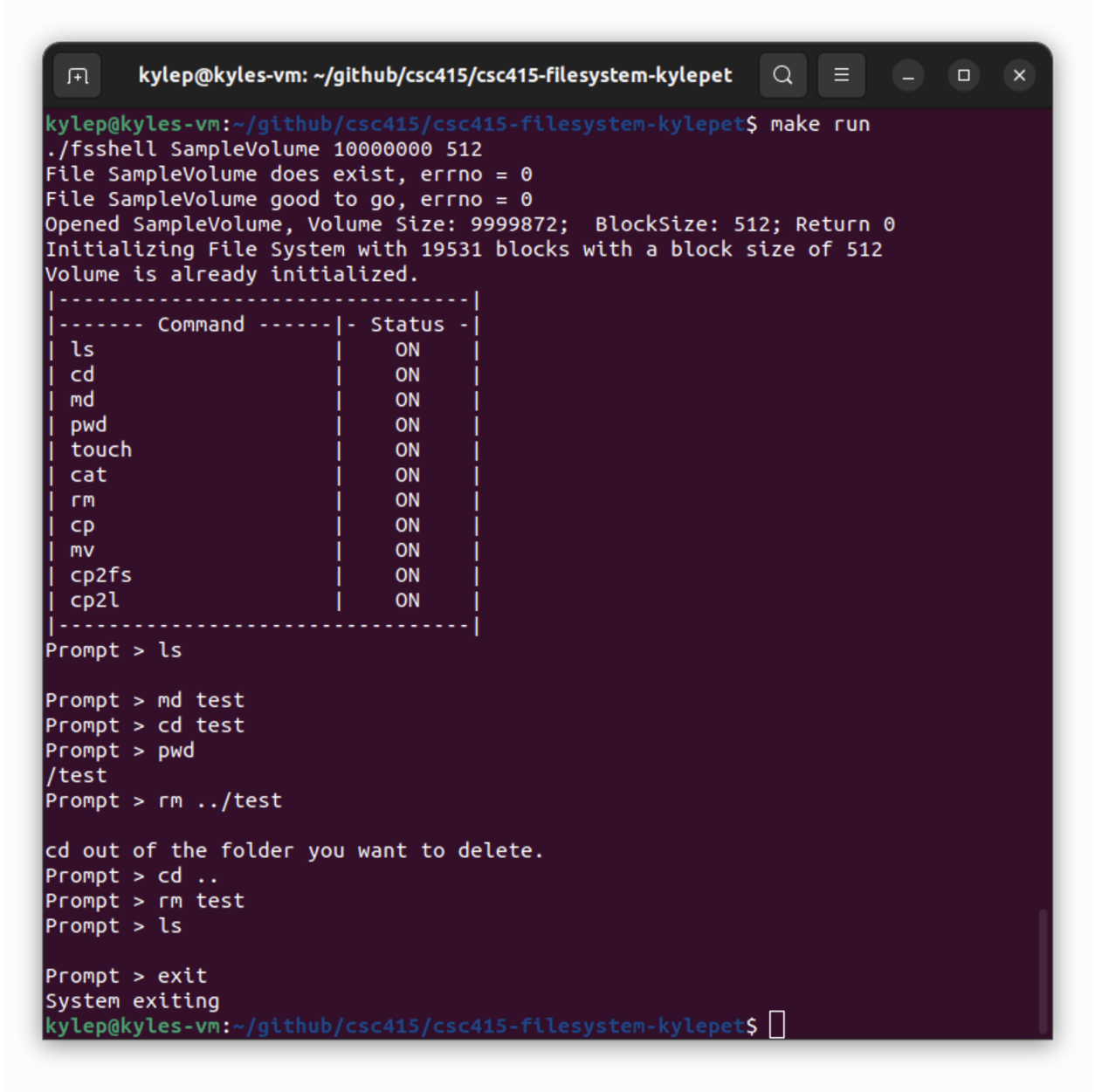
rm folder that doesn’t exist



rm folder in subfolder

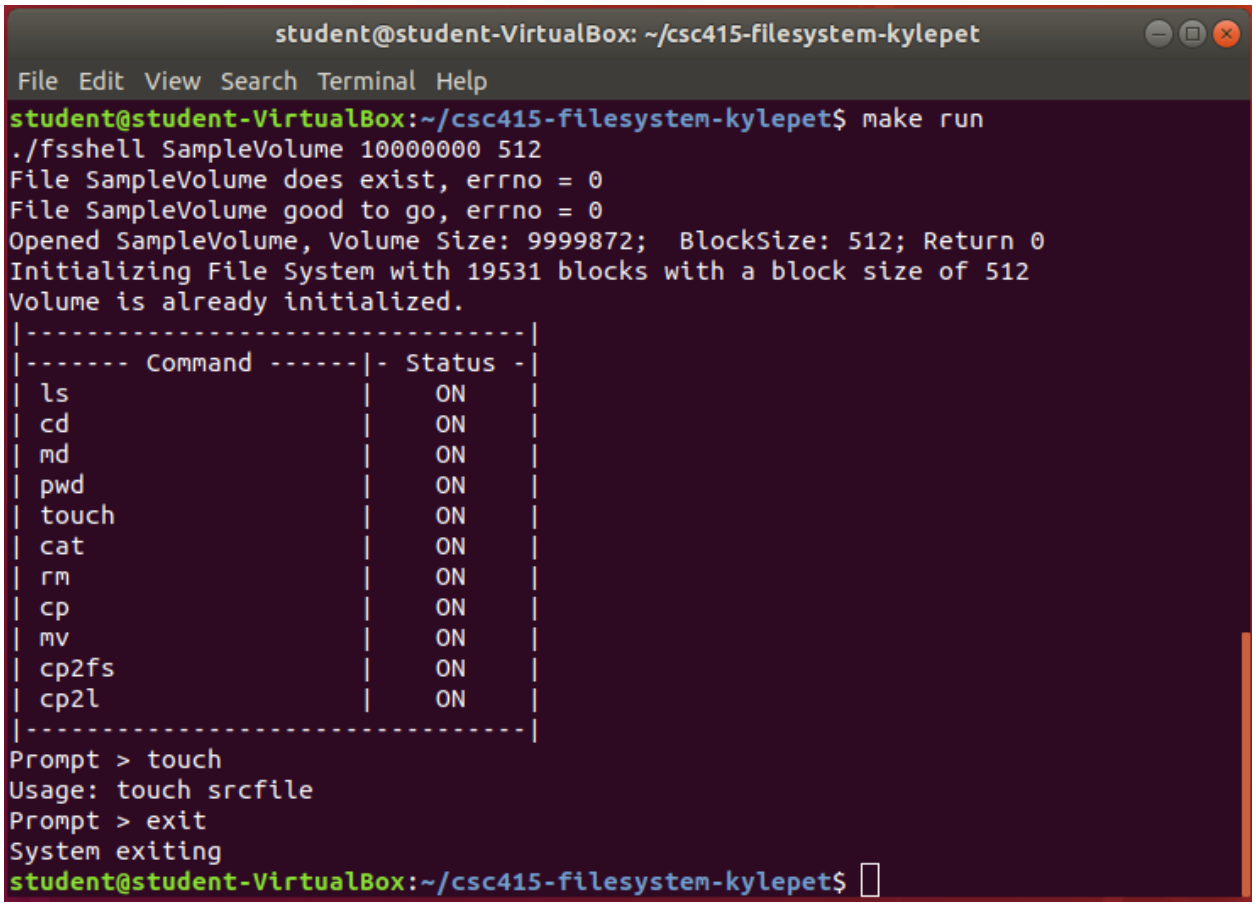


rm folder that's also the current working directory

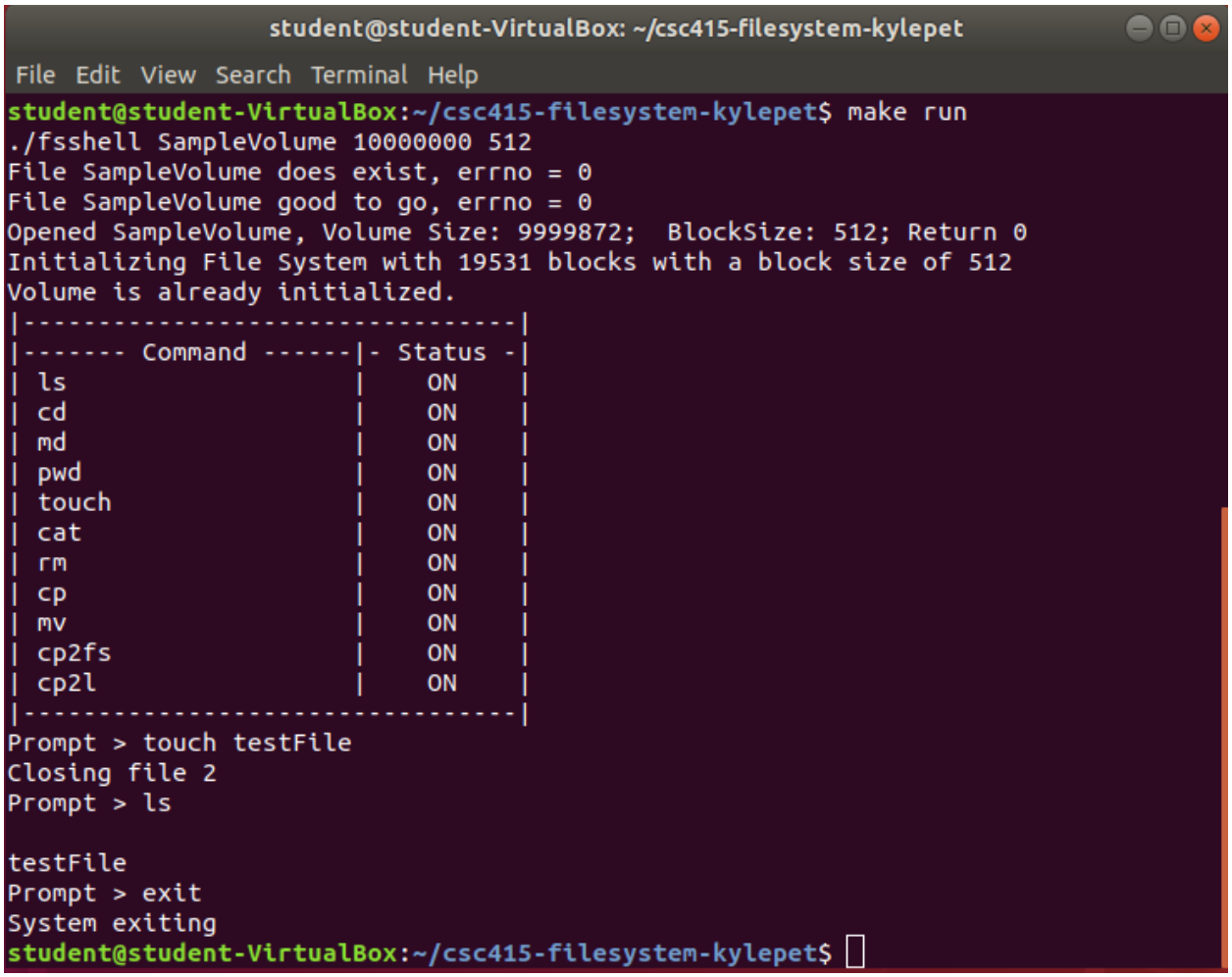


Touch

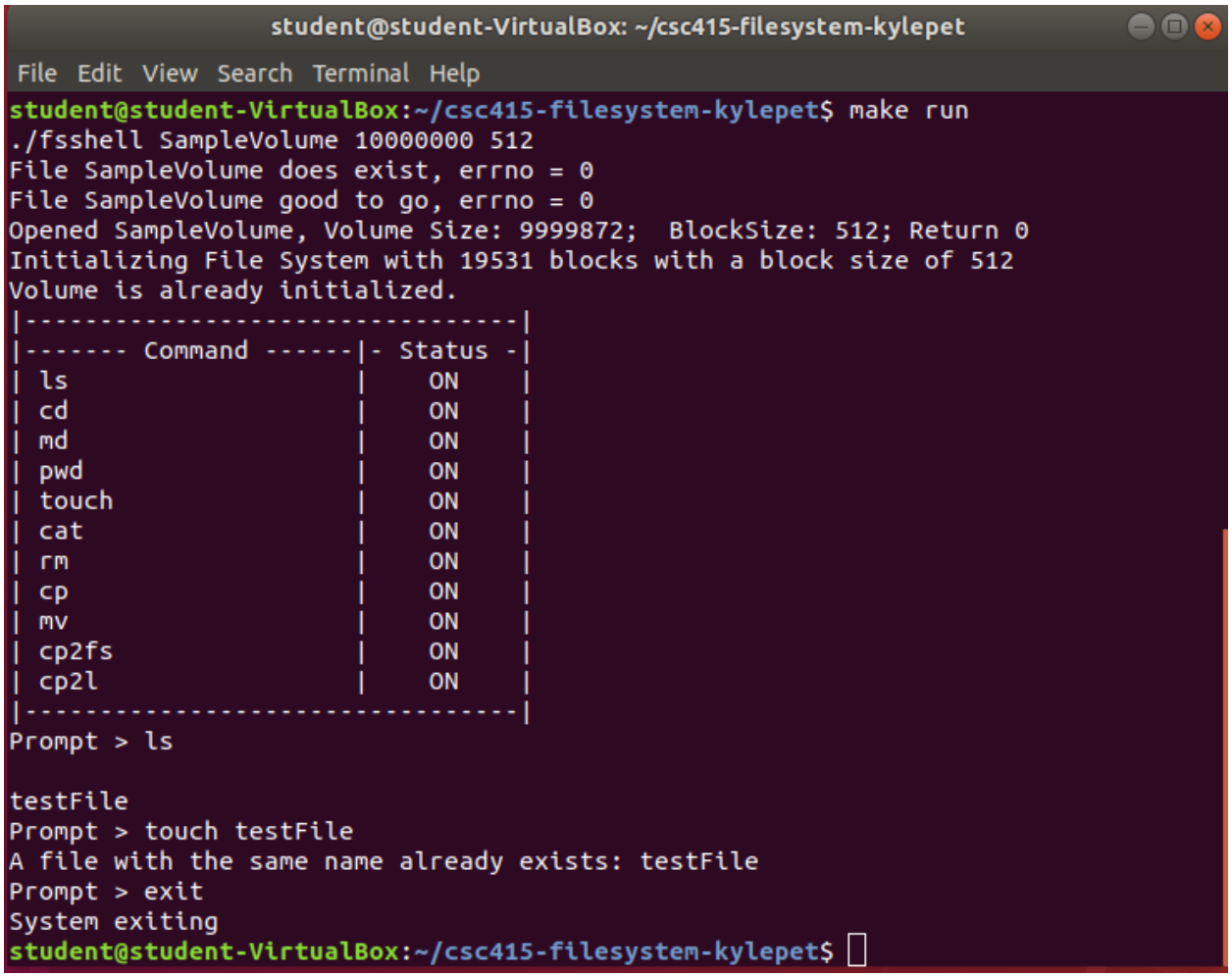
touch file but with no srcFile



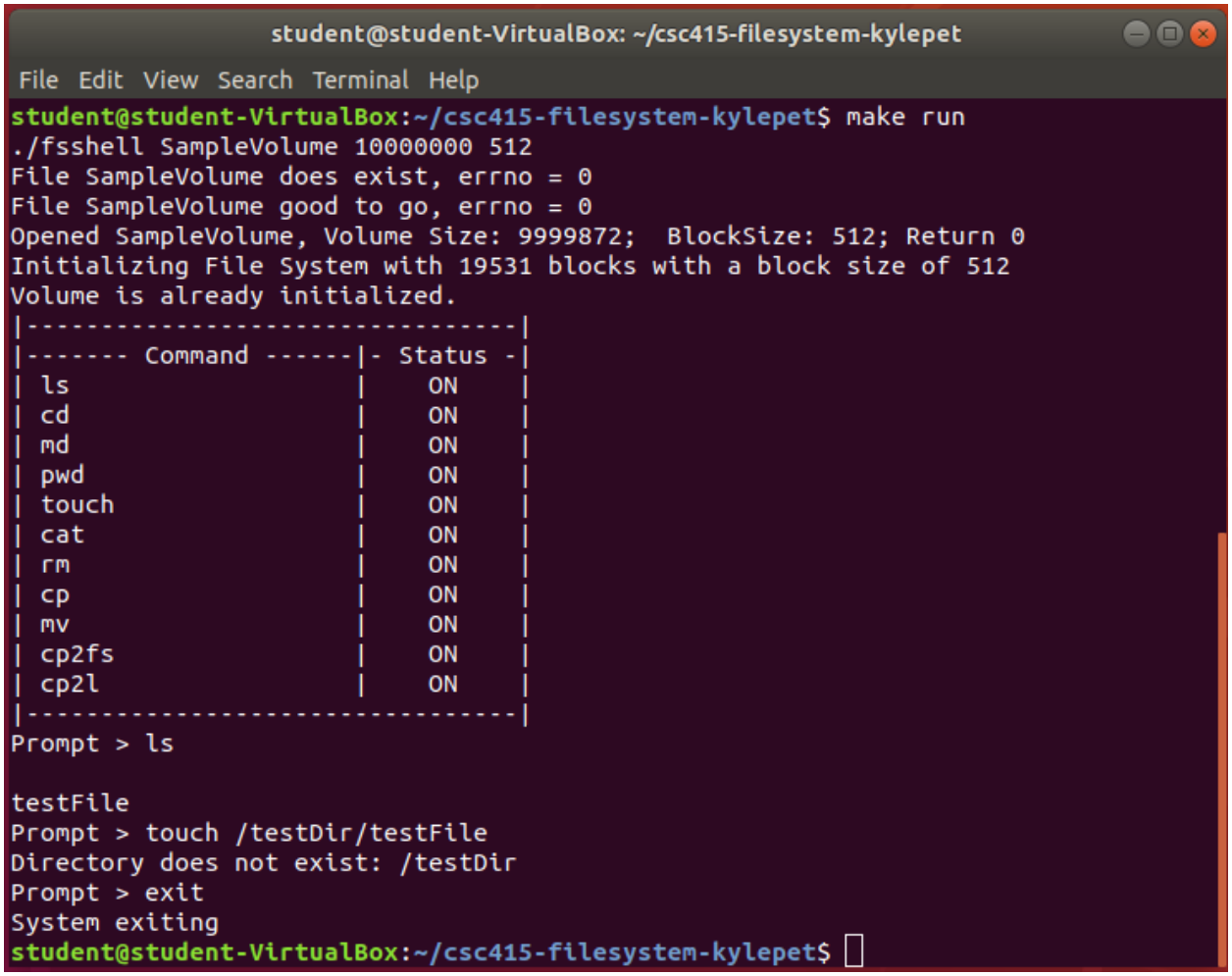
touch (creating) a file



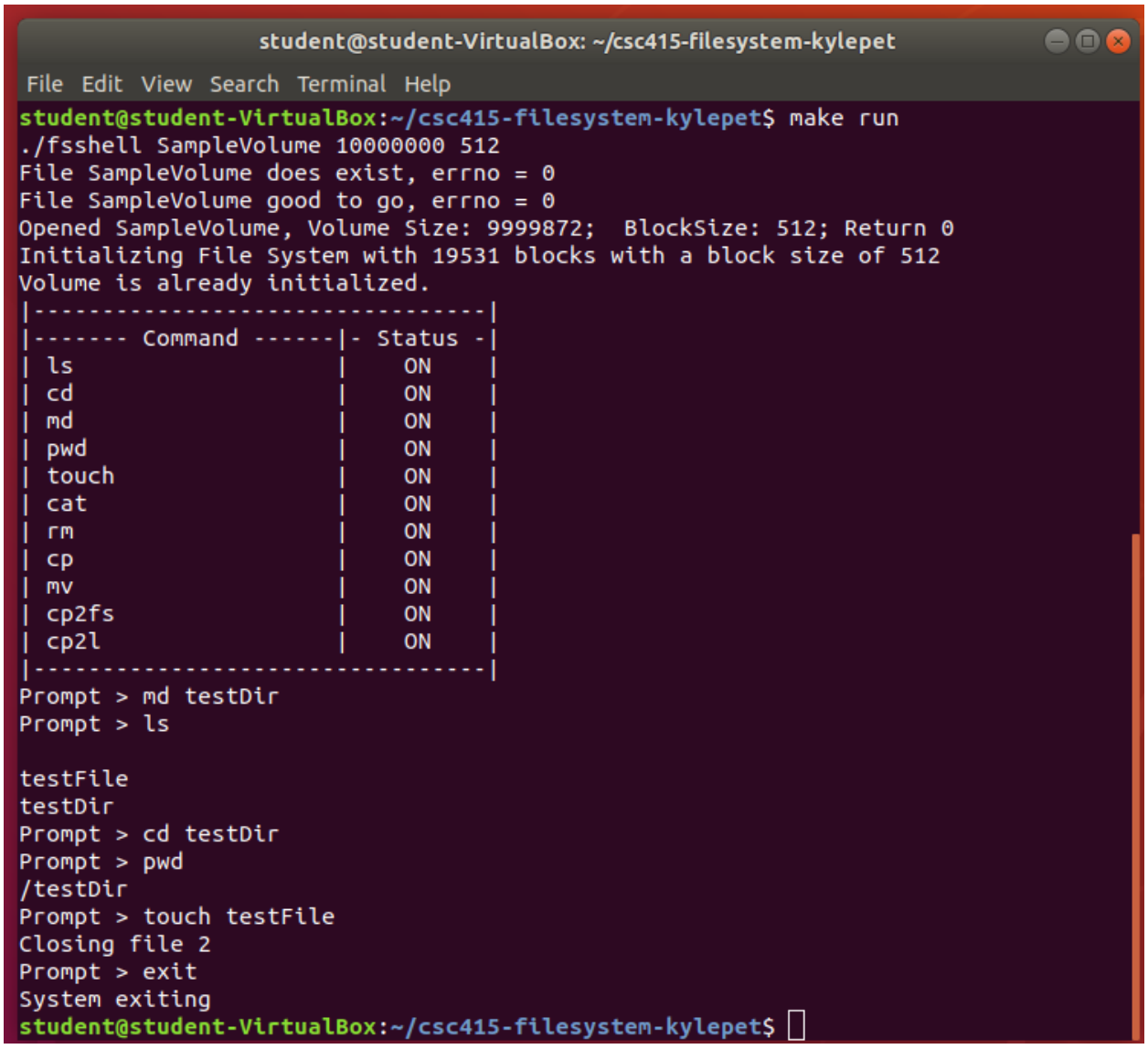
touch a file that already exists



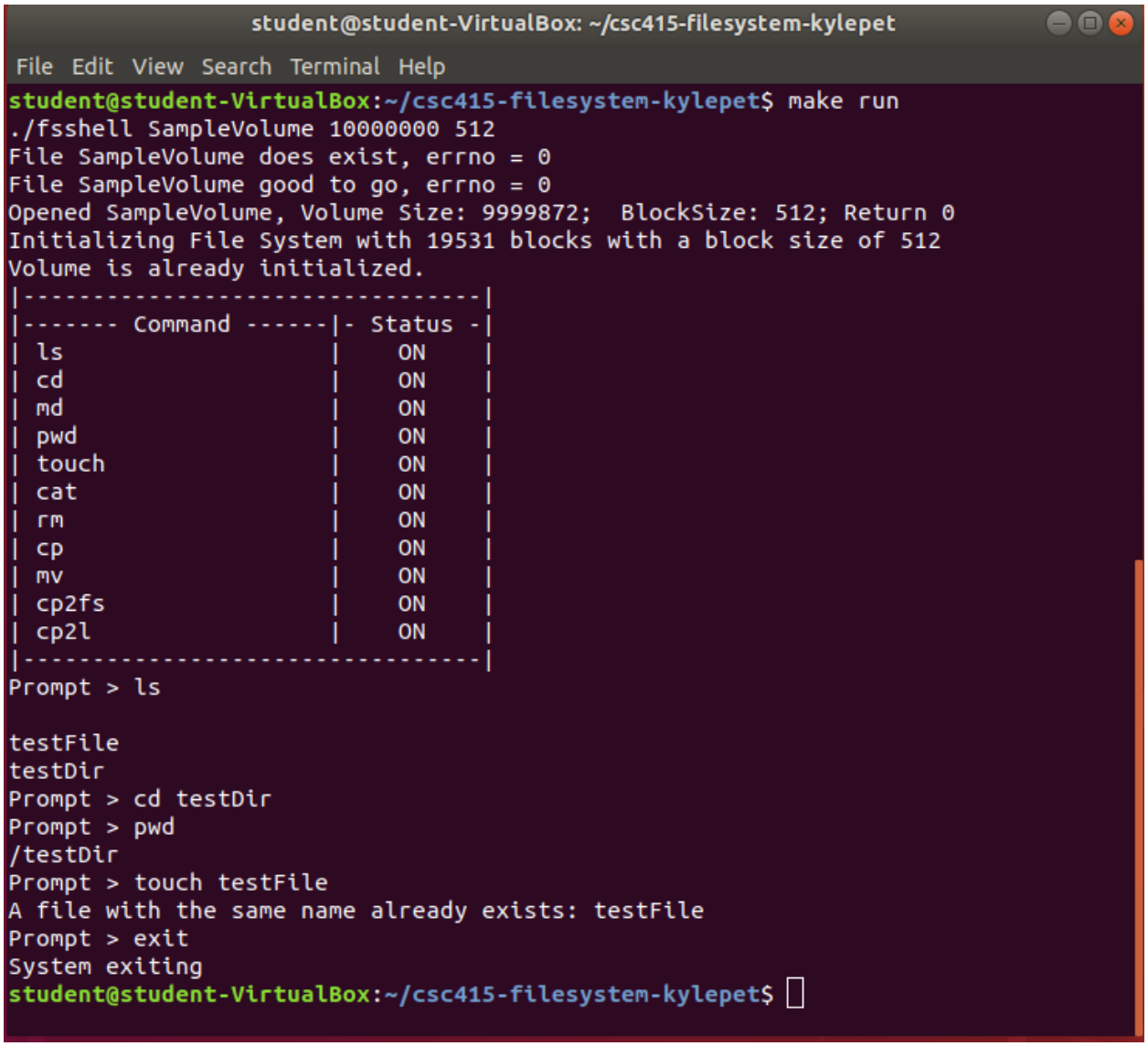
touch into subdirectory that does not exist



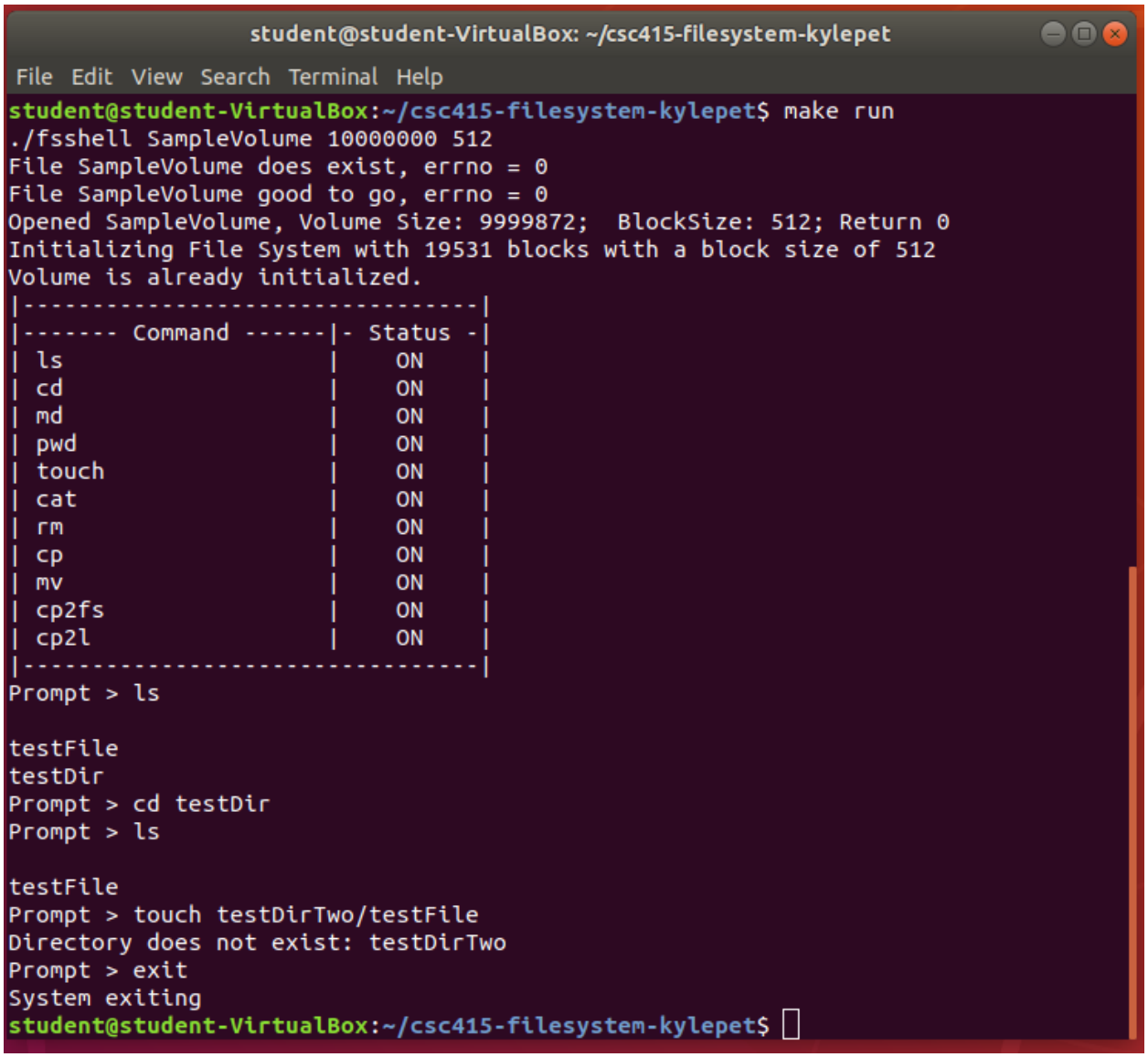
touch in a sub directory



touch a file that already exists in a subdirectory

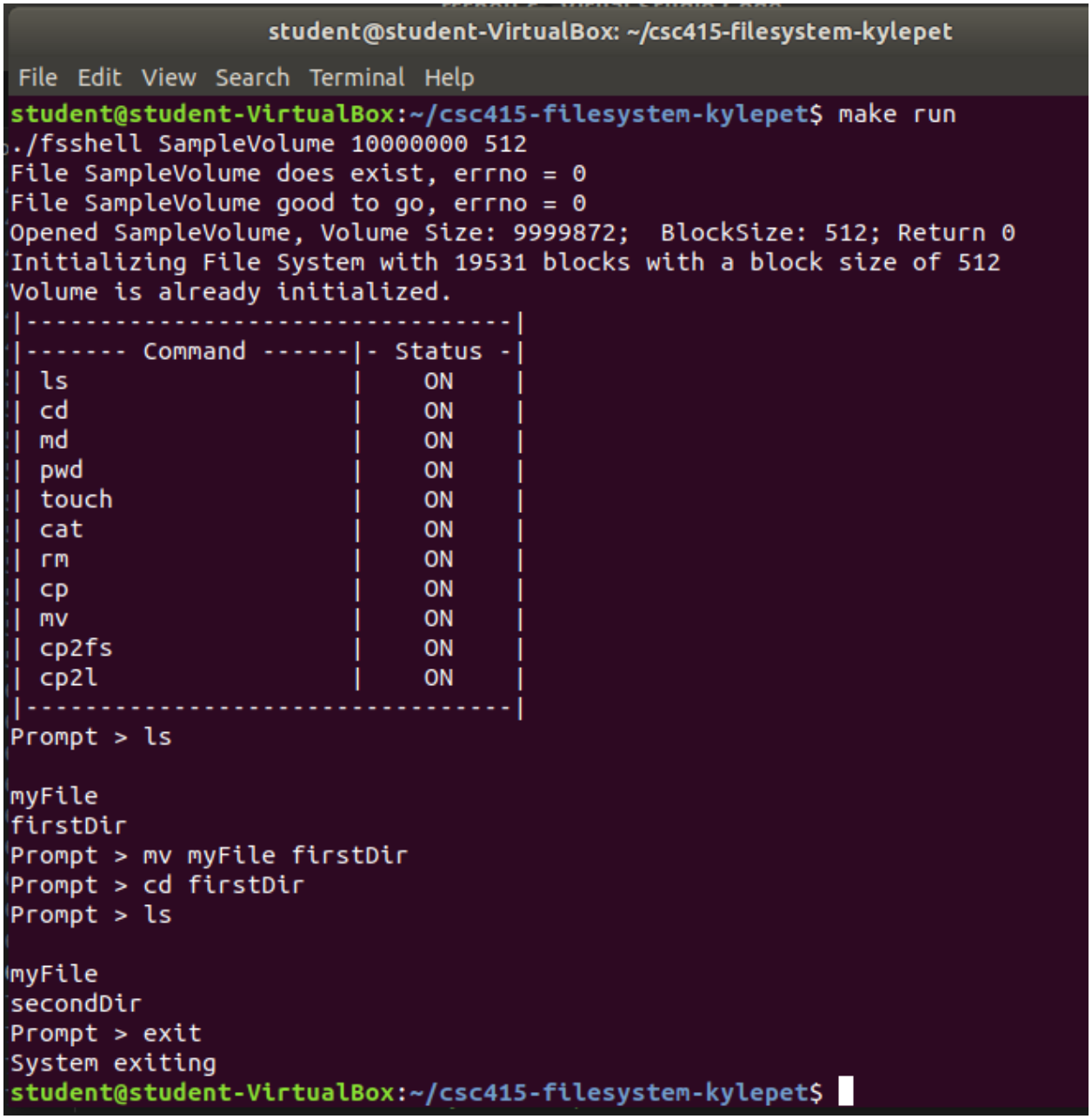


touch a file in a subdirectory to a subdirectory that does not exist

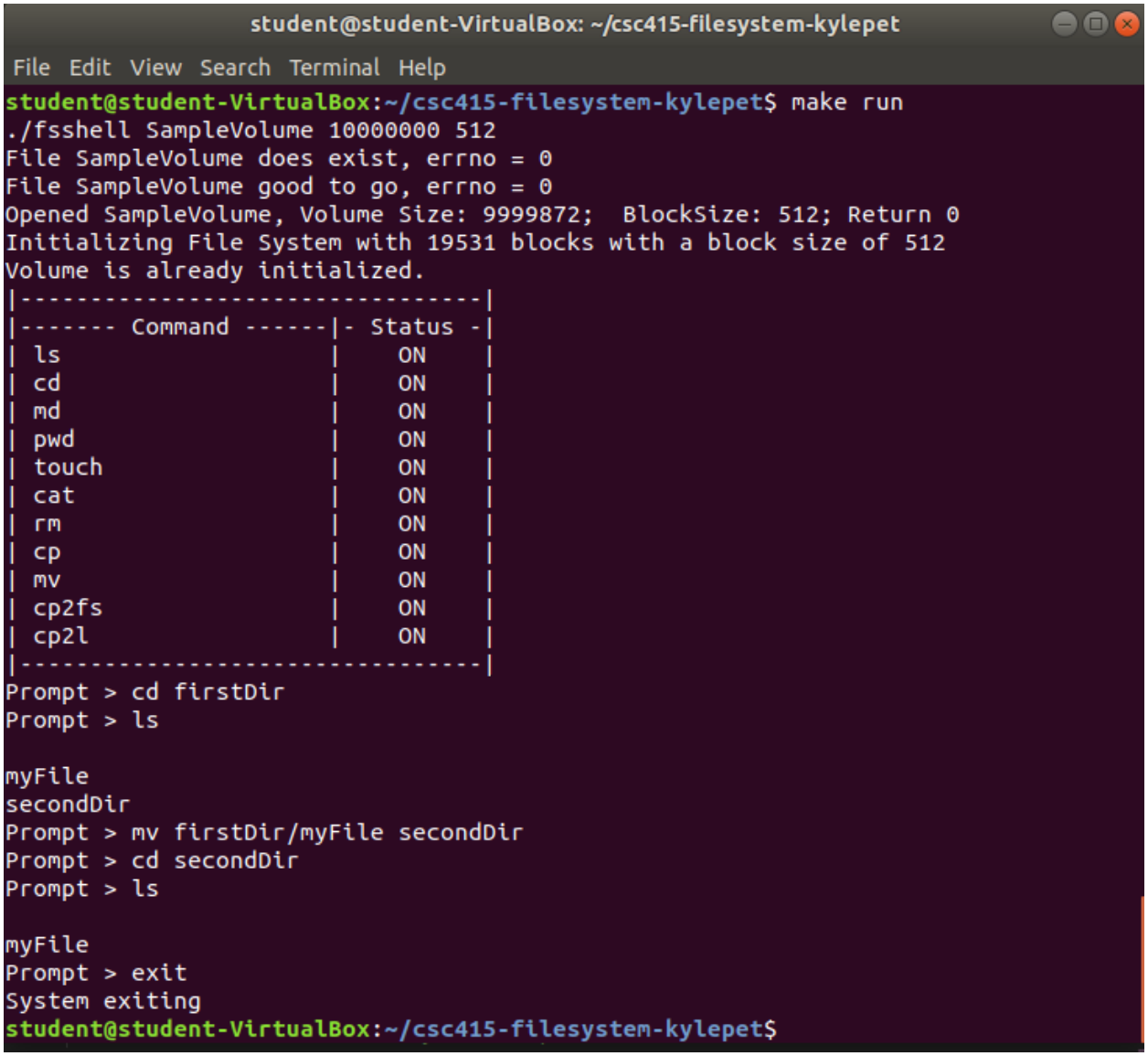


mv

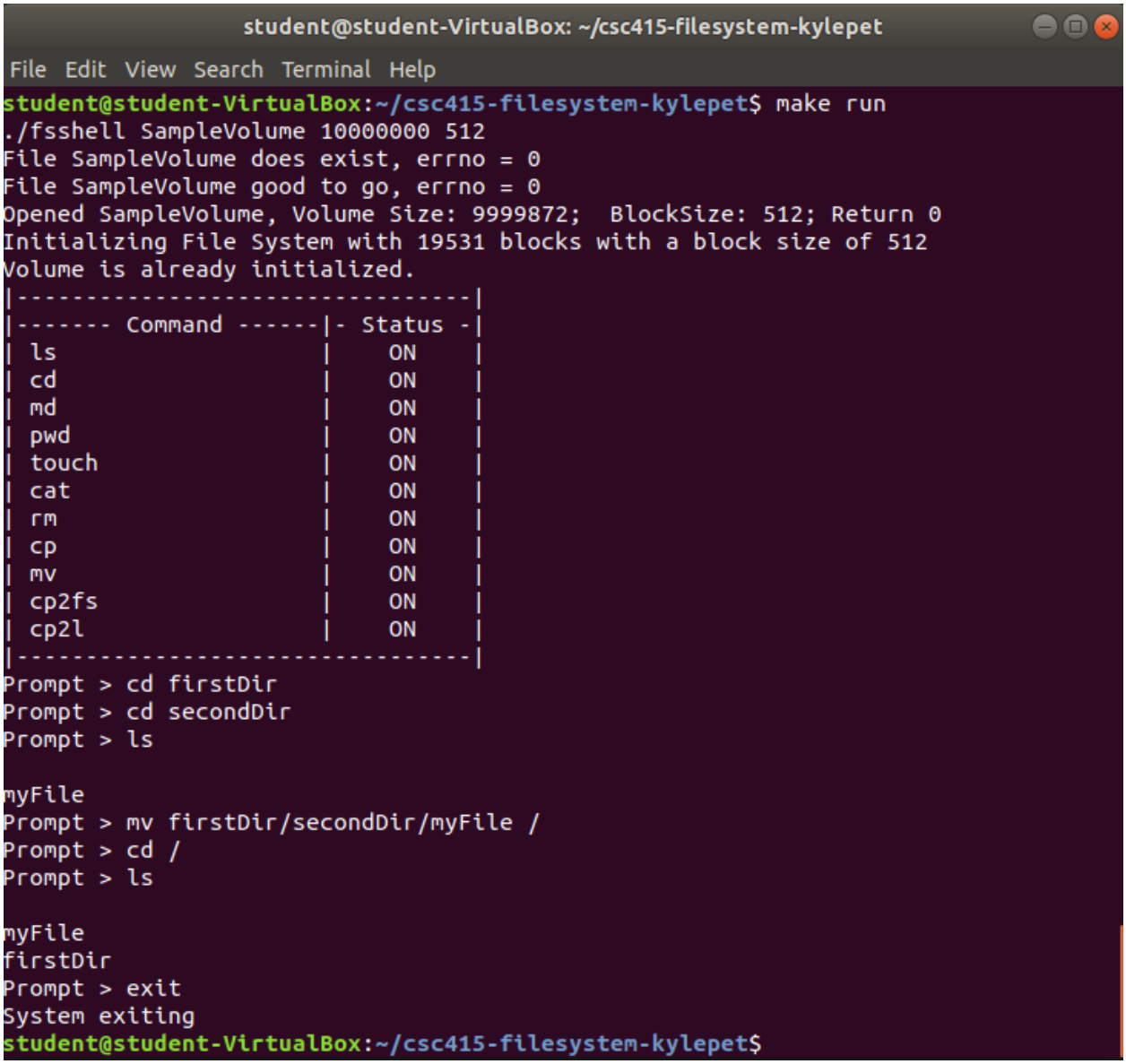
mv file from root to a directory



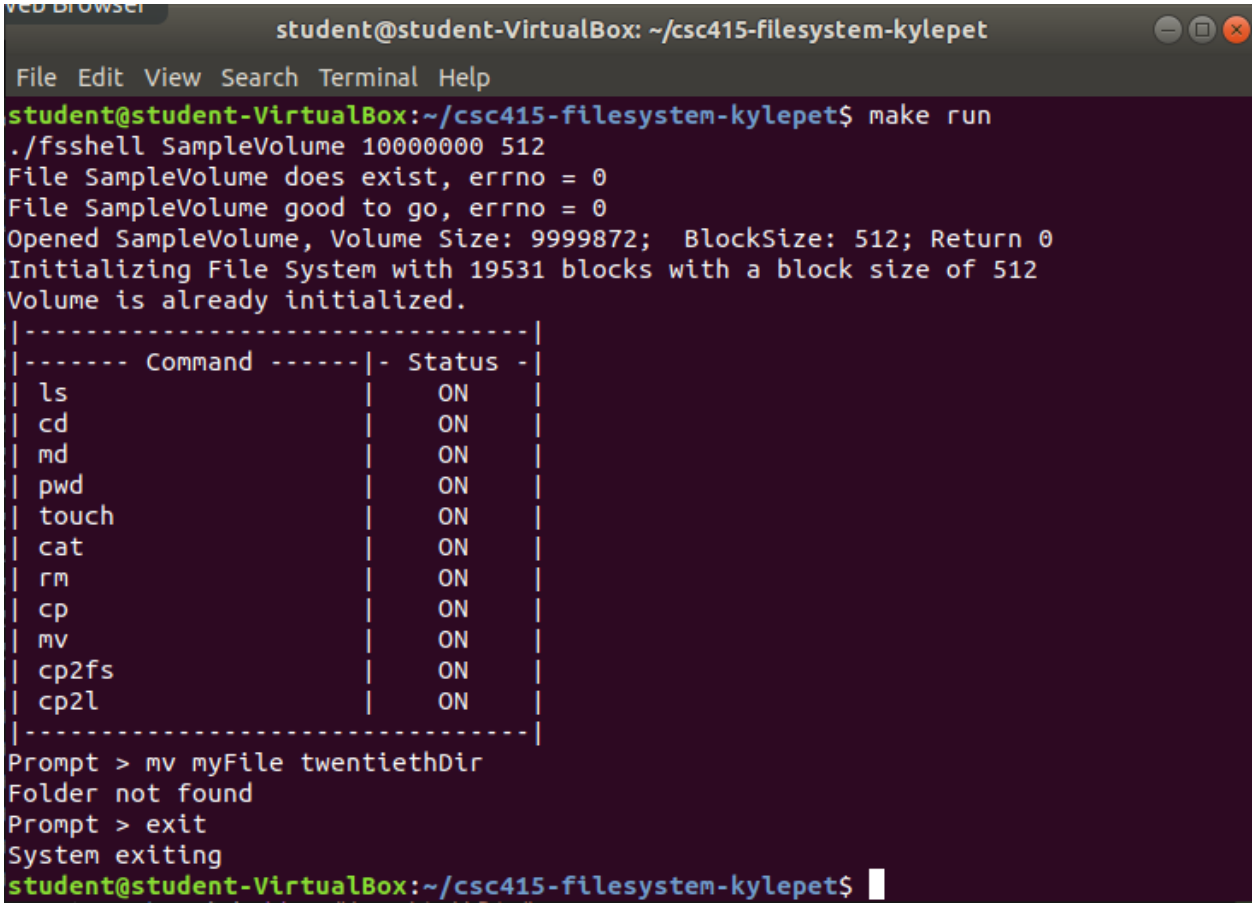
mv file from directory to subdirectory



mv from subdirectory to root

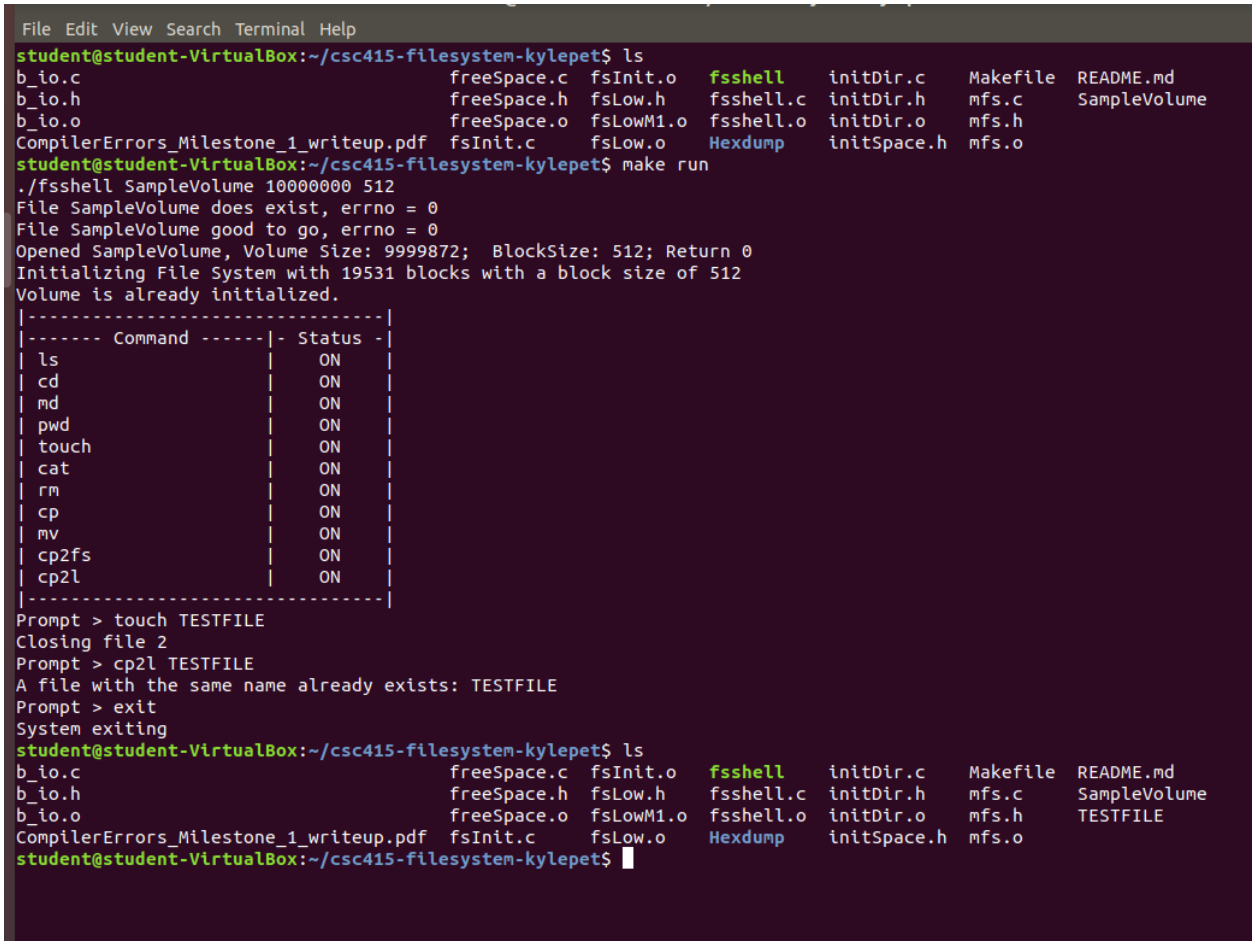


mv file to nonexistent directory

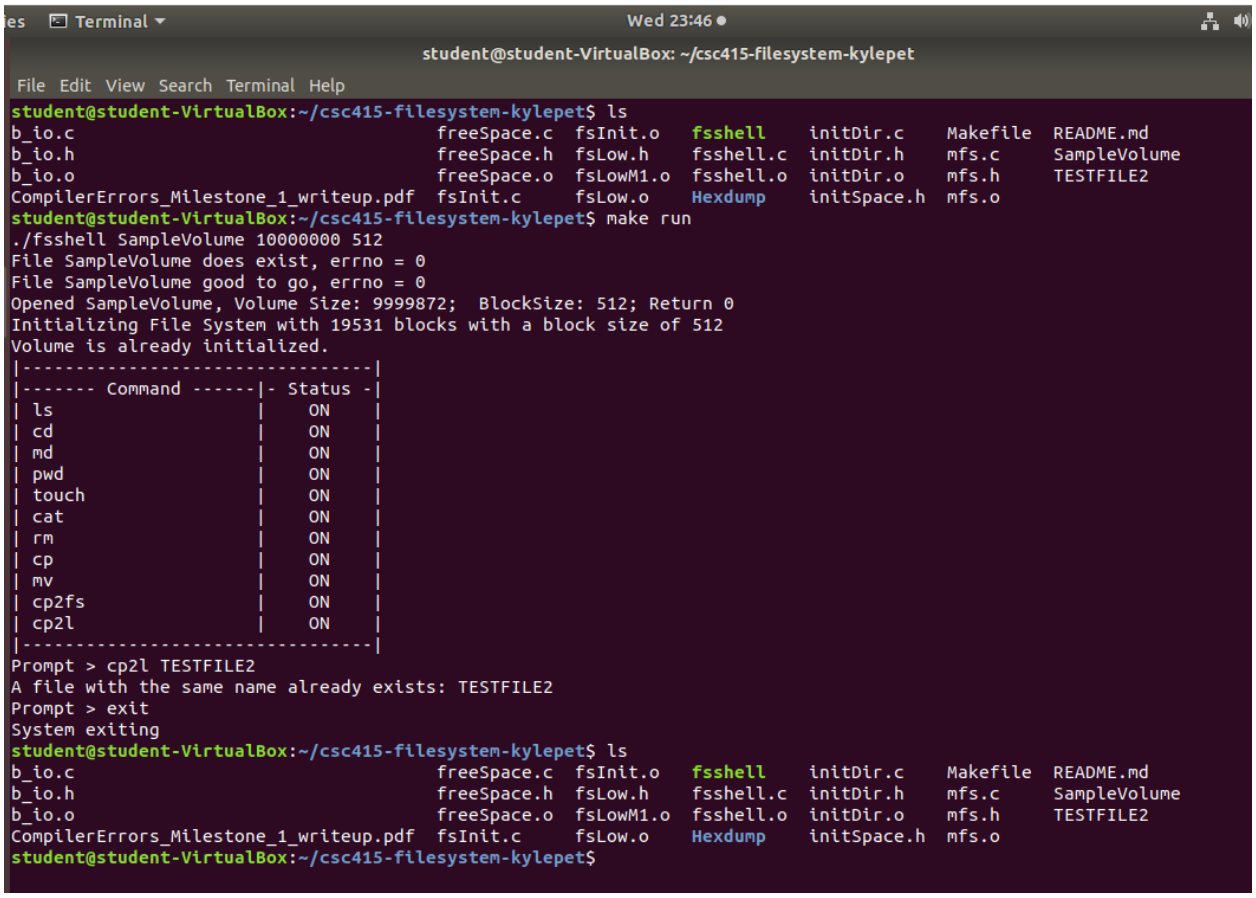


cp2l

cp2l file from file system to Linux

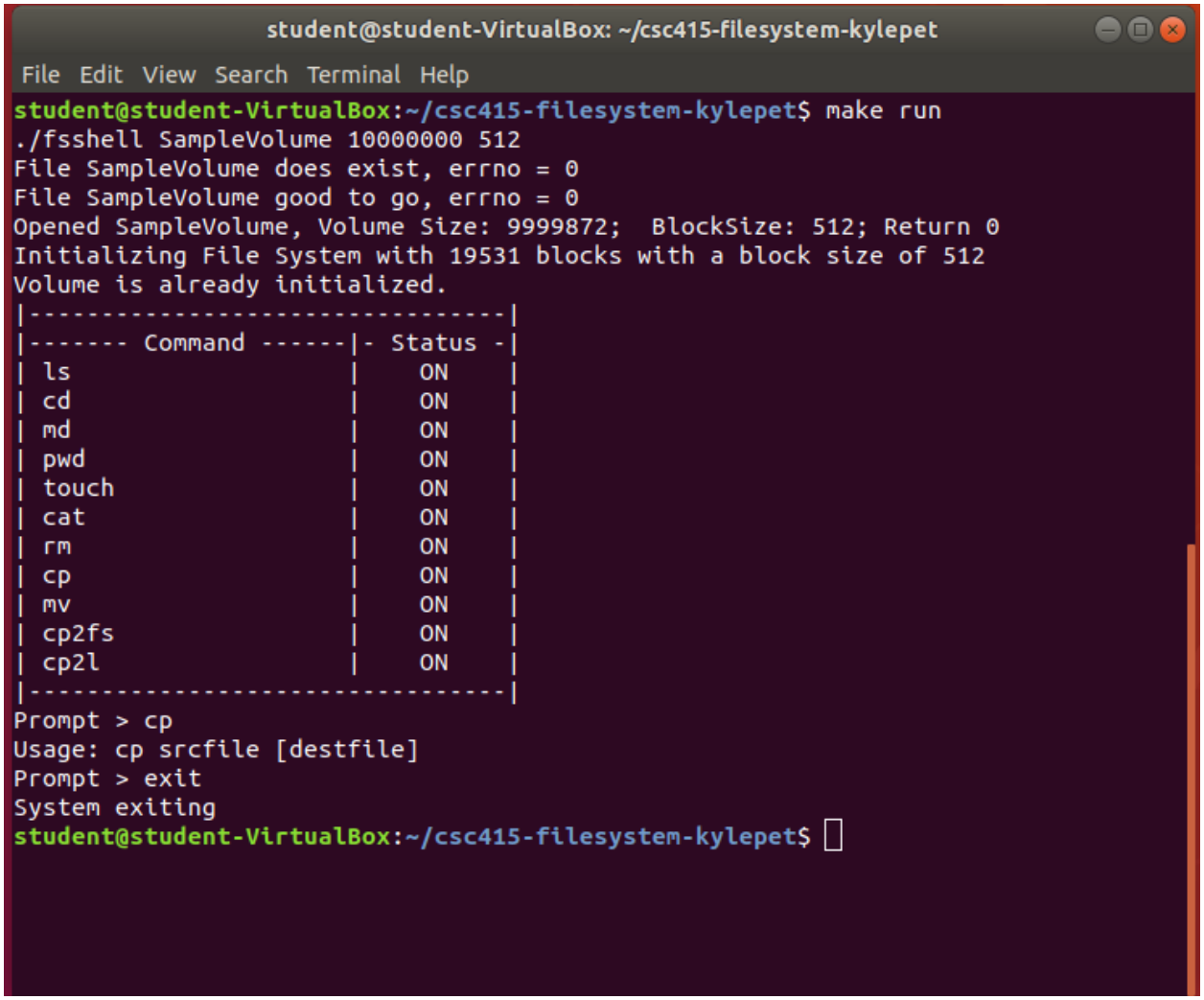


cp2l test to make sure duplicate isn't made

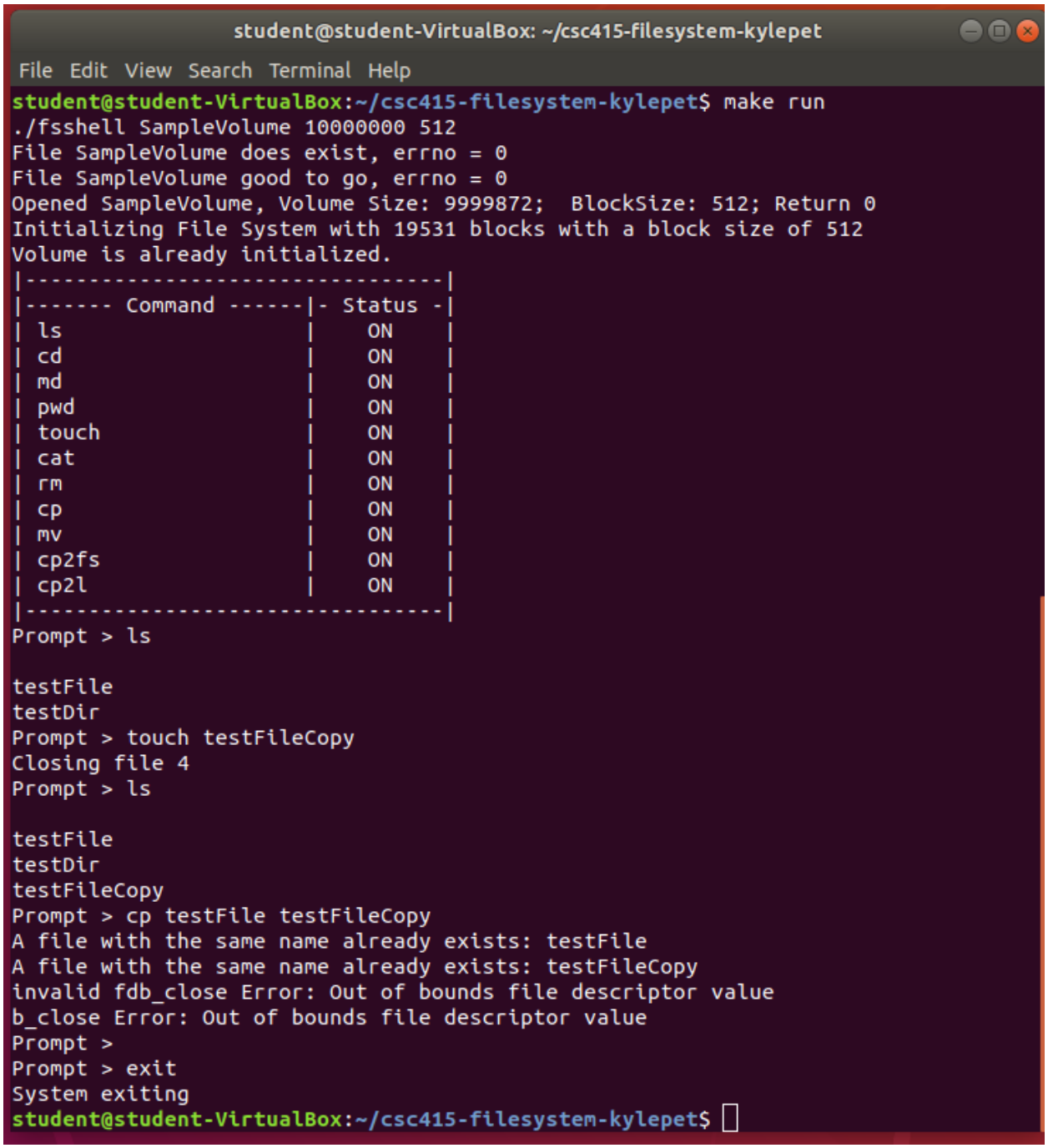


cp

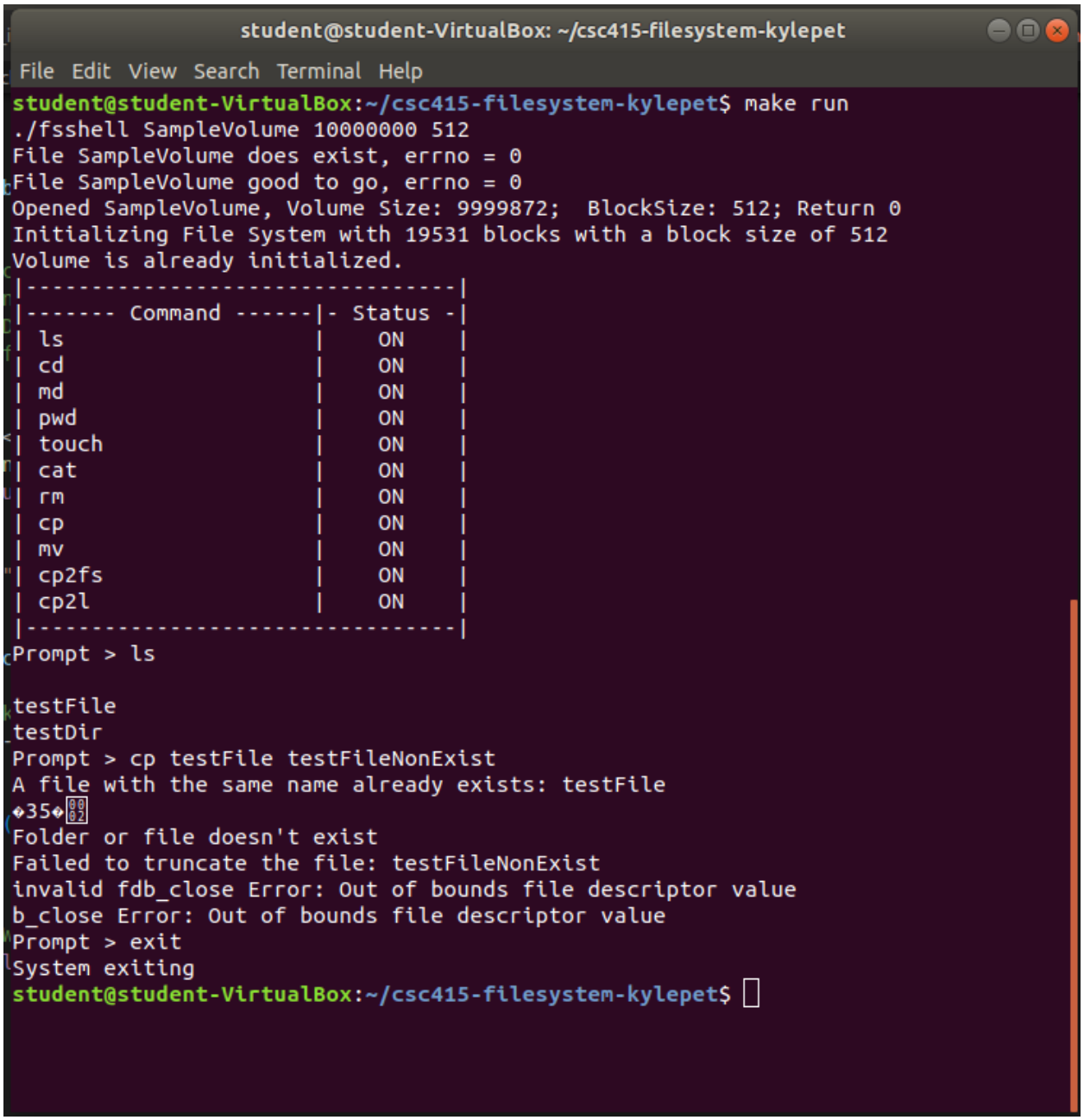
Copy nothing



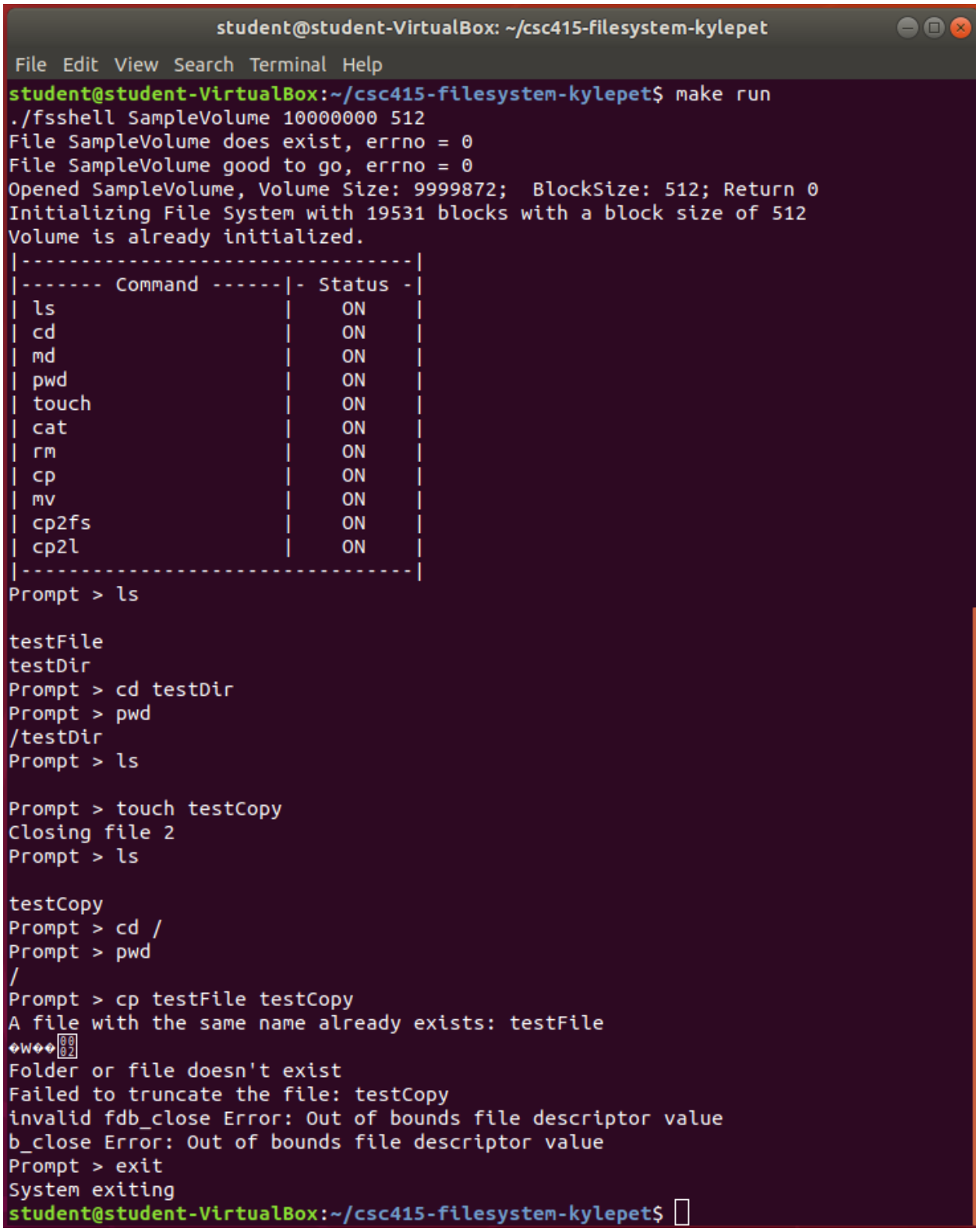
Copy file to file



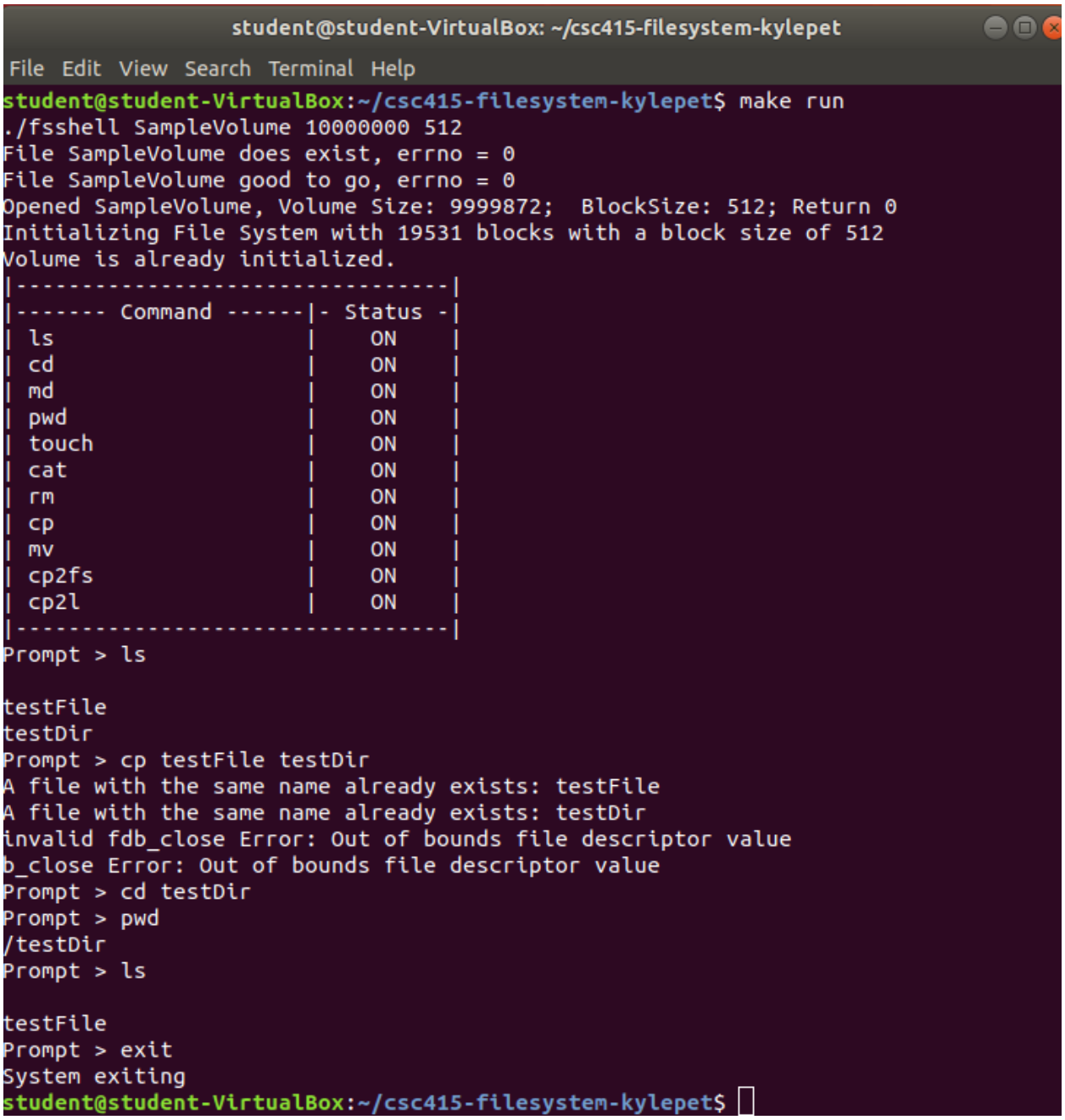
Copy file to nonexistent file



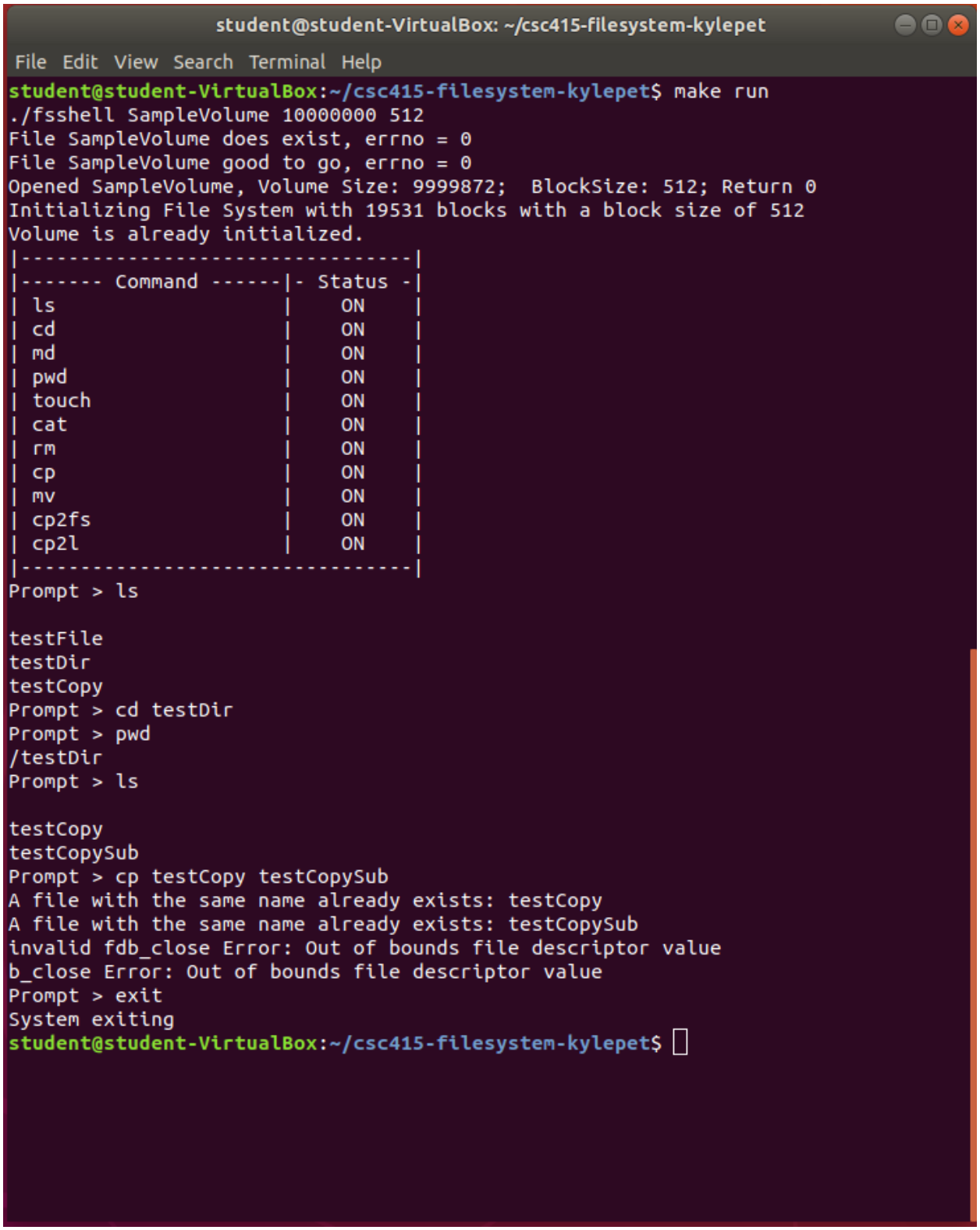
Copy file to sub directory file



Copy file to subdirectory that already has a file of the same name



Copy file in sub to file in sub



Copy file in subdirectory that does not exist to file in subdirectory that also does not exist