File System Project

Group Submission

Team Name	Alibaba	
Team Github	https://github.com/CSC415-2023-Fall/csc415-filesystem-siid14	
Team Member	Sidney Thomas	918656419
	Hoang-Anh Tran	922617784
	Ruxue Jin	923092817
	Yee-Tsing Yang	922359864

TABLE OF CONTENTS

1. Description of our file system

Our file system serves as a framework for handling various file and directory operations. It's centered around a command-line interface, offering users a versatile array of commands like ls, cp, mv, md, rm, touch, and cat. These commands file management tasks, allowing for efficient handling of data.
A key highlight of our system is its capability to interact between the test file system and the Linux file system. This feature facilitates smooth file transfers between these environments, enhancing the overall usability and flexibility of the system.

At its core, the system employs a structured approach to interpret and execute commands using a parser. This parser leverages functions designed to manipulate directory entries and manage file attributes. Consequently, users can navigate through the file system, examine file characteristics, and execute operations crucial for effective data organization.
In essence, our file system equips users with a foundational toolkit, empowering them to perform essential file operations efficiently and manage their data with ease.

Here is a specific description for each file.It will detail the content of all .c files if the corresponding .h file solely comprises declarations. If the .h file stands alone with more content, it will be described separately.

1.1 File Name: b_io.c

<u>Purpose/Functionality:</u>
The b_io.c file serves as a crucial file in our Basic File System, managing key file I/O operations, including file opening, reading, writing, seeking within files, and closing files. It's designed to handle file control blocks (FCBs), manage buffers, and interact with the underlying file system.

<u>Key Functionalities:</u>

a. Initialization and File System Setup:
    - b_init: Initializes the file system, setting up the FCB array and other internal variables.

b. File Operations:
    - b_open: Opens a file, allocates buffers, and initializes FCBs based on file access mode and path.
    - b_seek: Moves the file pointer within an open file based on 'whence' and 'offset'.
    - b_write: Writes data to an open file, managing buffer sizes, and interacting with disk blocks.
    - b_read: Reads data from an open file, handling reads from buffers and disk blocks.
    - b_close: Closes an open file, updating directory entry information and freeing allocated memory.

1.2 File Name: fsDir.c

Purpose/Functionality:
This file implements functionalities related to our basic file system, handling directory operations, file manipulation, and managing the structure of our system.

Key Functionalities:

a. Initialization Functions:
    - initDir: Initializes a directory.
    - fs_setcwd: Sets the current working directory.
    - fs_mkdir: Creates a new directory.
    - findFreeDE: Finds an empty entry in the parent directory.
b. File and Directory Status Functions:
    - fs_isDir: Checks if a given path represents a directory.
    - fs_isFile: Checks if a given path represents a file.
    - fs_stat: Retrieves file or directory information.
c. Directory Management Functions:
    - fs_closedir: Closes a directory after reading it.
    - fs_opendir and fs_readdir: Open and read directories.
    - checkIfDirEmpty: Checks if a directory is empty.
d. File and Directory Removal Functions:
    - fs_rmdir: Removes a directory.
    - fs_delete: Deletes a file.
e. File System Movement and Operations:
    - fs_move: Moves a file from one location to another.
f. Auxiliary Functions:
    - cleanPath: Cleans the path string.
    - freeBlocksDE and markUnusedDE: Manage block allocation and mark directory entries as unused.

1.3 File Name: fsFree.c

Purpose/Functionality:
The fsFree.c file manages free space within our file system by handling bitmaps that track the allocation status of blocks on the disk.

Key Functionalities:

a. Initialization Functions:
    - initFreeSpace(): Initializes the free space management system by setting up the bitmap to track used and free blocks.
    - loadFreeSpace(): Loads the free space system if the volume is already initialized.
b. Bitmap Management:
    - setBitUsed(): Marks a block as used in the bitmap.

  - setBitFree(): Marks a block as free in the bitmap.
  - isBitUsed(): Checks if a block is used based on its bit in the bitmap.
c. Block Allocation:
  - getFreeBlockNum(): Finds the first free block available.
  - allocBlocksCont(): Allocates contiguous blocks if available and marks them as used in the bitmap.

1.4 File Name: fsLow.h

<u>Purpose/Functionality:</u>
fsLow.h acts as the primary interface for reading and writing Logical Blocks, serving as a crucial interface for our file system project. It provides functionalities to interact with the physical hard drive file and presents it to the logical layer as a series of logical blocks.

<u>Key Functionalities:</u>
a. Partition System Management:
  - startPartitionSystem(): Initializes the partition system, setting up the file that represents the physical drive with specified volume and block sizes. Handles file creation and opening, checking for existing files, and allocating space for the volume.
  - closePartitionSystem(): Properly closes the file representing the physical drive.
b. File System Initialization:
  - initFileSystem(): Initializes the file system, preparing it with the specified number of blocks and block size. Manages the initialization of the file system metadata.
c. Read and Write Operations:
  - LBAwrite(): Writes logical block data to the specified location on the physical drive file.
  - LBAread(): Reads logical block data from the specified location on the physical drive file.
d. Testing Functionality:
  - runFSLowTest(): Reserved for testing purposes, not to be used in production. Likely used for internal testing of the low-level file system operations.
e. Constants and Definitions:
  - Defines constants related to block sizes, partition signatures, and error codes.
  - Defines data types like uint64_t and uint32_t.

1.5 File Name: fsInit.c

<u>Purpose/Functionality:</u>
The fsInit.c file serves as the main driver for our file system, handling the initialization and exit routines of the file system.

Key Functionalities:

a. Initialization:
   - initFileSystem(): Initializes the file system with a given number of blocks and block size. It checks for the existence of a signature in the Volume Control Block (VCB) to determine whether to initialize or reload the file system metadata.
   - loadFreeSpace(): Loads the free space system if the signature matches, reloading the free space information.
b. Volume Control Block (VCB) Handling:
   - Manages the VCB structure, storing essential information about the file system, including the signature, block count, block size, and locations of the root directory and free space bitmap.
   - Initializes or reloads the VCB based on the signature verification.
c. Directory and Path Handling:
   - Manages the root directory, current working directory, and directory entries.
   - Handles parsing paths, managing paths within the file system, and initializing necessary data structures.
d. Cleanup and Exit:
   - exitFileSystem(): Releases memory and performs cleanup tasks when exiting the file system.

1.6 File Name: fsParse.c

Purpose/Functionality:

fsParse.c contains functions related to parsing paths within the file system, enabling the extraction of information from paths provided as strings.

Key Functionalities:

a. Path Parsing Function:

   → parsePath(const char *path, ppInfo *ppi): Parses the given path to extract information.
   - Receives a path string and a structure (ppInfo) to store parsed information.
   - Breaks down the path and extracts relevant details like parent directory, last element, and index.
   - Handles scenarios of both absolute and relative paths.
   - Returns codes (-1, -2, 0) indicating success, invalid paths, or failure to find directories.
b. Helper Functions:
   - findEntryInDir(DE *parent, char *token): Looks for a specific entry (directory or file) within a given directory.
   - loadDir(DE *parent): Loads a directory's content from disk into memory.
   - loadRootDir(int initialDirEntries): Loads the root directory content into memory.
c. Internal Functionality:
   - Uses directory entry structures (DE) and interacts with the low-level file system via functions from fsLow.h.

- Manages loading directories into memory and performing directory-related operations.
- Handles memory allocation and disk I/O operations for parsing paths and directory manipulation.

1.7 File Name: mfs.h

Structures:

a. DE (Directory Entry):
    - fileName: Name of the file or directory.
    - size: Size of the directory or file in bytes.
    - location: Starting block number of the directory or file.
    - isDir: Flag indicating whether this entry is a directory or a file.
    - timeCreated, timeLastModified, timeLastAccessed: Time stamps for file operations.

b. VCB (Volume Control Block):
    - numberOfBlocks: Number of blocks in the volume.
    - blockSize: Size of each block in bytes.
    - signature: Signature for the VCB struct.
    - bitMapLocation: Starting block number of the bitmap.
    - rootDirLocation: Starting block number of the root directory.

c. fs_diriteminfo:
    - Structure for providing information about each file during directory iteration.
    - Contains record length, file type, and the filename.

d. fdDir:
    - Structure for directory iteration. Keeps track of the directory entry position and directory content being iterated.

e. fs_stat:
    - Structure filled in from a call to fs_stat.
    - Contains file attributes such as size, block size, block count, and various timestamps.

Functions:

a. Directory Manipulation:
    - fs_mkdir: Creates a directory.
    - fs_rmdir: Removes a directory.

b. Directory Iteration:
    - fs_opendir: Opens a directory for iteration.
    - fs_readdir: Reads the next entry in a directory.
    - fs_closedir: Closes a directory.

c. Miscellaneous Directory Functions:
    - fs_getcwd: Gets the current working directory.
    - fs_setcwd: Changes the current working directory.
    - fs_isFile: Checks if a given entry is a file.
    - fs_isDir: Checks if a given entry is a directory.

  - fs_delete: Removes a file.
  - fs_move: Moves a file from source to destination.

d. File Status:
  - fs_stat: Retrieves status information about a file.

Definitions:

a. MAX_FILENAME_LEN: Maximum filename length.
b. Various data type definitions (uint64_t, uint32_t) if not previously defined.

1.8 File Name: fsshell.c

a. Purpose:
  - The file serves as a command-line interface to interact with the file system. It allows users to execute various file system operations through defined commands.

b. Command Handling:
  - Defines a structure dispatch_t holding command-related information: command name, function pointer, and description.
  - Functions like cmd_ls, cmd_cp, cmd_mv, etc., implement specific commands using functions from mfs.h.

c. File System Commands:
  - Each command function interacts with the file system using functions defined in mfs.h to perform operations like listing files, copying, moving, creating directories, removing files/directories, etc.
  - Examples include cmd_ls, cmd_cp, cmd_mv which perform corresponding file system operations based on input arguments.

d. Command Line Parsing:
  - The processcommand function tokenizes the input command and checks if it matches any defined command. It then calls the corresponding function.

e. File System Interaction:
  - Functions like fs_opendir, fs_readdir, fs_closedir, fs_isDir, fs_isFile, etc., are part of the file system interface (mfs.h) that command functions use to interact with the file system.

f. Initialization and Termination:
  - The main function initializes the file system and implements a basic command-line interface. It reads user commands, processes them, and executes corresponding file system operations.

2. Approach/ What we did:

2.1 Milestone 1

To format a disk for the file system(prepare the disk for the specific file system), we need to set up three things: volume control block, freespace, and root directory.

a. <u>VCB</u>

"A volume control block is the master file table in UNIX or the superblock in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks." (from google)
We defined our Volume Control Block structure and store it in the first block. We set our signature as 1234.

```
#define SIGNATURE 1234
typedef struct VCB
    {
    unsigned int volumeSize;  // Total number of blocks in the volume
    unsigned int blockSize;   // Size of each block in bytes
    unsigned long signature;  // Signature for VCB struct

    unsigned int bitMapLocation;    // starting block num of bitMap
    unsigned int freeSpaceLocation; // starting block num of free space
    unsigned int rootDirLocation;   // starting block num of root directory
    } VCB;
```

b. <u>Freespace</u>

In the file system, we need to keep track of which block is used or not. We choose to use bitmap as prof showed us in the lecture. We have 10M disk storage, that is 10,000,000/512 = 19531 blocks. If 1 bit indicates a block, that is 19531/8 = 2441 bytes. If 1 block has 512 bytes, we need 2441/512 = 5 blocks to keep track of all blocks in the file system. The bitmap data stores between block 2 and block 6.
We have three helper functions to manage the freespace.

setBitUsed: This function takes the unsigned integer type of block number, and then sets the corresponding bit to 1.

```
// set the bit corresponding to blockNum to 1 (mark the block as used)
void setBitUsed(unsigned char *bitMap, unsigned int blockNum)
   {
   unsigned int byteIndex = blockNum/8; //calculate the byte index in the bitmap
   unsigned int bitIndex = blockNum%8; //calculate the bit index within the byte
   unsigned char mask = 1<<(7-bitIndex);  //create a 1-byte mask with 1 at the
                                          //bit position 0 for other bits
```

```
    bitMap[byteIndex] = bitMap[byteIndex] | mask; //set the bit at the
                                              //specified position to 1
    }
```

setBitFree: this function takes the unsigned integer type of block number, and sets the corresponding bit to 0.

```
// set the bit corresponding to blockNum to 0 (mark the block as free)
void setBitFree(unsigned char *bitMap, unsigned int blockNum)
   {
   unsigned int byteIndex = blockNum/8;
   unsigned int bitIndex = blockNum%8;
   unsigned char mask = ~(1<<(7-bitIndex)); //create a 1-byte mask with 0 at the
                                            //bit position 1 for other bits
   bitMap[byteIndex] = bitMap[byteIndex] & mask; //set the bit at the specified
                                                 //position to 0
   }
```

getFreeBlockNum: this function finds the first free block after the given block number.

```
// Find the first free block after blockNum
int getFreeBlockNum(unsigned char *bitMap, unsigned int blockNum)
   {
   unsigned int byteIndex = blockNum / 8;
   unsigned int bitIndex = blockNum % 8;
   unsigned char mask = ~(1 << (7 - bitIndex));
   while (bitMap[byteIndex] & mask) //keep searching for the first 0 bit
      {
      bitIndex++;
      if (bitIndex == 8)
         {
         bitIndex = 0;
         byteIndex++;
         }
      if (byteIndex * 8 >= blockNum) //check if have searched through all blocks
         {
         return -1;   //no free block found
         }
      mask = 0xFF;    //reset the mask to 0xFF (all 1s) to search for any
                      //free bit in the next byte
      }
   return byteIndex*8 + bitIndex; //calculate the block number of free block
   }
```

c. Root Directory

directory entry: directory entry is a structure that contains metadata about the file or the directory.

```
#include <time.h>
#define MAX_FILENAME_LEN 255   // the maximum filename length

// Directory Entry structure
typedef struct DE
  {
  char fileName[MAX_FILENAME_LEN + 1]; // file name cstring, +1 for the NULL
  unsigned long size;    // size of the directory/file in bytes
  unsigned int location; // starting block number of the directory/file
  unsigned int isDir;    // flag indicating if this entry is
                         // a directory (1) or a file (0)
  time_t timeCreated;      // time when the file created
  time_t timeLastModified; // time when the file last modified
  time_t timeLastAccessed; // time when the file last accessed
  } DE;
```

The root directory is an array of 50 directory entries. So, it needs 50 X 296 = 14800 bytes. We malloc 29 blocks, which result 14848 bytes. 48 bytes would be wasted.

2.2 Milestone 2

For milestone 2, we finished directory functions in fsParse.c and fsDir.c
Following the professor's instruction, we implemented the most fundamental functions first like initDir() and parsePath().
After making sure the initDir and parsePath function works well, we started to work on directory functions.

a. initDir()

This function takes three parameters: initialDirEntries which is default 50, blockSize, which is the default 512, and the parent directory entry. It sets up a directory in the parent directory entry, and returns the first block number of this new directory.

b. parsePath()

This function is the key function in milestone2, because many other functions depend on it, such as fs_setcwd, fs_mkdir, fs_isDir, fs_isFile, fs_rmdir, fs_delete.
In the file system, path is just an array of characters, we need the parsePath function to navigate this string to get useful information: the parent directory entry, the string of the last element, and the index of this last element in the parent directory. So we design a structure ppInfo to store the information:

```
typedef struct ppInfo
    {
    DE *parent;         // point to parent directory
    int index;          // index of lastElement, -1 if it does not exist
    char *lastElement; // point to name of last element
    } ppInfo;
```

This function allows us to set up any directory in a specific directory entry, including root directory.

c. fs_mkdir

We first worked on fs_mkdir, since other functions depend on this function to test.
To successfully make a directory, we first need to parse the path.
We check:

- parent directory must existed
- parent entry must be a directory, instead of a file
- the last element must not exist in the parent directory

Then we init a directory in the disk, find the free directory entry in the parent directory, and store the directory in the disk.
We have two helper functions:

- findFreeDE: to find the free directory entry in the directory by iterating the array and finding the first directory entry with name "\0".
- copyDE: to set up the first entry in the directory (current directory).

d. fs_setcwd

To successfully set the current working directory(cd), we first need to parse the path.
We check:

- the path is valid
- the last element must exist in the parent directory
- the last element must be a directory
- then set the global variable currentPath to the path.

We also need a cleanPath function to get the clean current path. For example, "./././../../"
Following the professor's instruction, we tokenize the string with "/", get each token and store them in an array. Then we need an index to indicate the current directory entry. If we got a token ".", we stay the current index in the array. if we got a token "..", we decrease the index to get the former directory entry. If we get a token other than "." and ".." we increase the index and store the token in the array. Then we can get a clean current working directory path.

e. fs_isDir

To successfully know the entry is a directory or a file, we still need to parse the path.
We check:

- the path is valid
- the last element exist in the parent directory
- the last element is a directory instead of a file

(Hint: the DE structure has the variable isDir, 0 is file, 1 is dir)
When checking the path, we need to notice one situation: “/”, which is a directory and should return 1 in this function.

f. **fs_isFile**

To successfully know the entry is a directory or a file, we still need to parse the path.
We check:

- the path is valid
- the last element exist in the parent directory
- the last element is a file instead of a directory
  (Hint: the DE structure has the variable isDir, 0 is file, 1 is dir)

When checking the path, we need to notice one situation: “/”, which is a file and should return 0 in this function.

g. **fs_stat**

To successfully get the metadata of a file, we still need to parse the path.
We check:

- the path is valid
- the last element exist in the parent directory

Then we get the parse path info about this directory entry, fill the data to buffer.

h. **fs_getcwd**

To copy the global variable currentPath to pathname.

i. **fs_rmdir**

To successfully remove the directory, we still need to parse the path.
We check:

- the path is valid
- if the last element is “.” or “..”. we cannot remove in this case
- the last element exist in the parent directory
- the last element is a directory instead of a file if the directory is empty

Then we free blocks of this directory, and write into disk.

j. **fs_delete**

To successfully delete the file, we still need to parse the path.
We check:

- the path is valid
- if the last element is “.” or “..” then we cannot delete in this case
- the last element exist in the parent directory
- the last element is a file instead of a directory

Then we free blocks of this file, and write them into disk.

k. fs_opendir

To successfully open the directory, we still need to parse the path.
We check:

- the path is valid
- if the path is "/" then we load root directory in this case
- the last element exist in the parent directory
- the last element is a directory instead of a file

Then load the directory and assign it to the struct fdDir * fdd.

l. fs_readdir

The function fs_readdir takes a pointer to a directory structure (fdDir) and reads the directory entries from the specified position (dirEntryPosition) onward.
Calculates the total number of directory entries by dividing the size of the opened directory by the size of each directory entry.
Allocates memory for a fs_diriteminfo structure using malloc.
Iterates over the directory entries starting from the current entry position (dirp->dirEntryPosition).
For each directory entry, we check if the file name is not equal to "\0" (indicating that the entry is in use).
If the entry is in use, populates the fs_diriteminfo structure with following information:

- d_reclen is set to the size of the fs_diriteminfo structure.
- d_name is set using strcpy to the file name from the directory entry.
- fileType is set to FT_REGFILE if it's a regular file and FT_DIRECTORY if it's a directory.

Updates the dirEntryPosition to the next entry position and returns the populated fs_diriteminfo structure.
If no valid entry is found in the remaining directory entries, returns NULL.

m. fs_closedir

The fs_closedir function closes a directory that has been opened by the fs_opendir function. It frees the memory associated with the fdDir pointer and sets it to NULL to avoid any potential pointer issues. The function returns 0 on success and -1 if the input pointer dirp is already NULL.

2.3 Milestone 3

After making sure that the directory functions works well, we started to implement file functions:

a. b_open

Initialization Check:
It checks if the system has been initialized (startup == 0).
If not, it initializes the system using b_init().
File Descriptor Allocation:
It retrieves a file descriptor (returnFd) using b_getFCB() and checks if all file control blocks (FCBs) are in use. If no free FCB is available, it returns an error indicating that all FCBs are in use.

Path Validation:
It validates the provided file path using the parsePath function. If the path is invalid or if the last element in the path is not found (ppi->lastElement == NULL), it returns an error.
Directory Check:
It checks if the path points to a directory (ppi->parent[ppi->index].isDir == 1).
If it does, it returns an error since the function is intended for opening files, not directories.
Memory Allocation:
Allocates memory for the file buffer (buf) and file information (fileInfo).
If the memory allocation fails, it returns an error.
Setting File Control Block Fields:
It initializes various fields within the file control block (fcbArray[returnFd]) such as buffer length, current block, index, parent location, file type (isDir), and access mode based on the provided flags.
File Existence Check:
It checks if the file exists. If it doesn't exist and the O_CREAT flag is provided, it creates a directory entry (DE) for the file, sets its attributes, and initializes its information like file name, location, size, and number of blocks.
File Existence Handling:
If the file already exists, it retrieves the file information and sets the appropriate fields in the file control block. If the O_TRUNC flag is set, it truncates the file by freeing its blocks and resetting size-related attributes.
Returning the File Descriptor:
Finally, it returns the allocated file descriptor (returnFd) indicating a successful file opening or appropriate error codes (-1) in case of failures or invalid scenarios.

b. b_read

The b_read function is designed to read data from a file, taking into account the current state of the file buffer and the need to fetch additional blocks from storage for managing the reading process.
Check the validity of the file descriptor and permissions:

- Ensure that the file is open for reading (O_RDONLY permission check).
- If the file is not open for reading, print an error message and return -1.

Determine the remaining buffer space:
Calculate the number of bytes remaining in the current buffer
remain = fcbArray[fd].buflen - fcbArray[fd].index
Calculate Bytes Delivered:
Calculate the total number of bytes already delivered by the file system (bytesDelivered), which is the product of the current block and block size subtracted by the remaining bytes.
Adjust Count Based on File Size:
If the requested count exceeds the file size, limit the count to the remaining bytes until the end of the file.

<u>Determine the parts (part1, part2, part3)</u>
Part 1. Reading from Buffer:
- Check if enough data is available in the buffer to fulfill the requested count of bytes.
- If yes, set part1 to count and set part2 and part3 to 0.

Part 2. Reading Additional Blocks:
- If not, calculate part2 as the number of complete blocks needed to fulfill the remaining data requirement.
- Use LBAread to read data from storage and update the current block pointer.

Part 3. Refill Buffer for Remaining Data:
- If there is still data to be read (part3 > 0), use LBAread to fetch a new block from storage.
- Reset the buffer offset and length, and copy the remaining data to the user's buffer.

<u>Return Total Bytes Read:</u>
- Calculate bytesReturned as the sum of part1, part2, and part3.
- Return bytesReturned as the total number of bytes read.

c. b_write

The b_write function helps to write the content of the user buffer to the file. It takes three parameters: file descriptor, user buffer, and the number of bytes needed to write. If the file does not contain enough blocks, 50 extra blocks are allocated to process the write operation.
The rest of it is similar to the b_read function. We need several variables to keep track of the buffer. Integer part1 indicates the left content of the current buffer. Integer part2 indicates the number of the whole block. Integer part3 indicates the remaining bytes left to write to the file.
At the end, we return the total number of bytes we write into the file.

d. b_close

The b_close is responsible for closing a file opened by b_open. It takes the file descriptor returned by b_open as a parameter.
<u>Check valid fd:</u>
Check if the provided file descriptor (fd) is within valid bounds (between 0 and MAXFCBS-1).
If not, it returns -1 to indicate an invalid file descriptor.
<u>Write Unused Buffe & free unused blocks:</u>
If the file size is greater than 0 and the file's current block index is less than the block size, it allocates a temporary buffer, copies the remaining data from the file's buffer, and writes it to the specified location on the disk. Then, calculate how many blocks are unused using the actual file size, then mark those unused blocks as free.
<u>Update Parent Directory Entry:</u>
Loads the directory corresponding to the parent location of the file being closed. It updates the information of the file (name, size, type, location) in the parent directory entry. And updates the timestamps (creation time, last access time, last modified time) in the parent directory entry.
<u>Write Updated Directory Entry:</u>
Write the updated directory entry back to the disk.
Finally, free the memory allocated for the file buffer, file information structure, and the loaded directory entry. Then b_close returns 0 to indicate successful file closure.

e. b_seek

<u>Initialization Check:</u>
It first checks if the system has been initialized (startup == 0). If not, it initializes the system using b_init().
<u>File Descriptor Validation:</u>
Verifies if the file descriptor (fd) is within the valid range (0 to MAXFCBS-1). If it's not within this range, it prints an error message and returns -1 (indicating an invalid file descriptor).
<u>Switch Case for Whence Parameter:</u>
It utilizes a switch-case construct to determine the action based on the whence parameter.

- Case SEEK_SET: Sets the file position pointer (filePositionPtr) to the provided offset. However, if the resulting position exceeds the file's size, it limits the pointer to the end of the file.
- Case SEEK_END: Positions the pointer at the end of the file (fileInfo->size).
- Case SEEK_CUR: Moves the file position pointer by the given offset relative to the current position. Similar to SEEK_SET, it limits the pointer to the end of the file if the resulting position exceeds the file's size.
- Default Case: If none of the above cases match, it prints an error message indicating invalid parameters and returns -1.

<u>Returning the File Position Pointer:</u>
Finally, the function returns the updated file position pointer (filePositionPtr), indicating the new position within the file.

f. fs_move

The fs_move() function moves a file to another directory. It takes the source path and destination path as parameters. Before the function is called, the source file path is checked by fs_isFile(), and the destination directory path is checked by fs_isDir().
<u>Loading source directory & Get source file DE info:</u>
Call parsePath() on the source path to get updated parse path info. Then use that information to load the parent directory of the source file (src) using the loadDir() function. It also retrieves the index of the source file within its parent directory. The information about the source file (such as filename, size, location, and timestamps) is copied for later use.
<u>Loading destination directory & updating destination DE info:</u>
Loads the destination directory using similar logic as for the source directory.
If the source and destination directories are the same, the function does nothing and returns 0.
Otherwise, the function checks for a free directory entry in the destination directory using the findFreeDE() function. If no free entry is found, an error is printed, and the function returns -1.
If a free entry is found, the source file's directory entry is marked as unused using the helper function markUnusedDE(), and the modified directory is written back to disk.
The destination directory entry is then updated with information from the source file, and the modified destination directory is also written back to disk.
Finally, the memory allocated for the temporary source and destination directories is freed.
Then, the function returns 0 on success and -1 on failure.

3. Issues we met and Resolutions

3.1 File Name: b_io.c

a. b_open() :

Error Handling:

- File Open Errors: Been using fs_open() to open the file but didn't sufficiently handle errors that might occur during this process.
- Memory Allocation Errors: The code checked for memory allocation issues after using malloc(), but didn't properly handle these errors. If malloc() failed, it returned -1 without freeing any allocated memory, leading to potential memory leaks.

File Descriptor Validity Check:
The code checked if returnFd was not equal to -1 after the fs_open() call, assuming it would always succeed. This assumption might not be true all the time, leading to potential issues if the file descriptor wasn't valid.

Access Mode Handling:
While the code attempted to determine the access mode specified by the flags (O_RDONLY, O_WRONLY, O_RDWR), it didn't enforce these modes when interacting with the file. So the file operations didn't comply with the specified access mode.

b. b_seek() :

Missing Calculation of New Position:
The code didn't calculate the new position based on the whence parameter and the offset. This means it didn't perform the task of moving the file pointer to the desired location within the file.

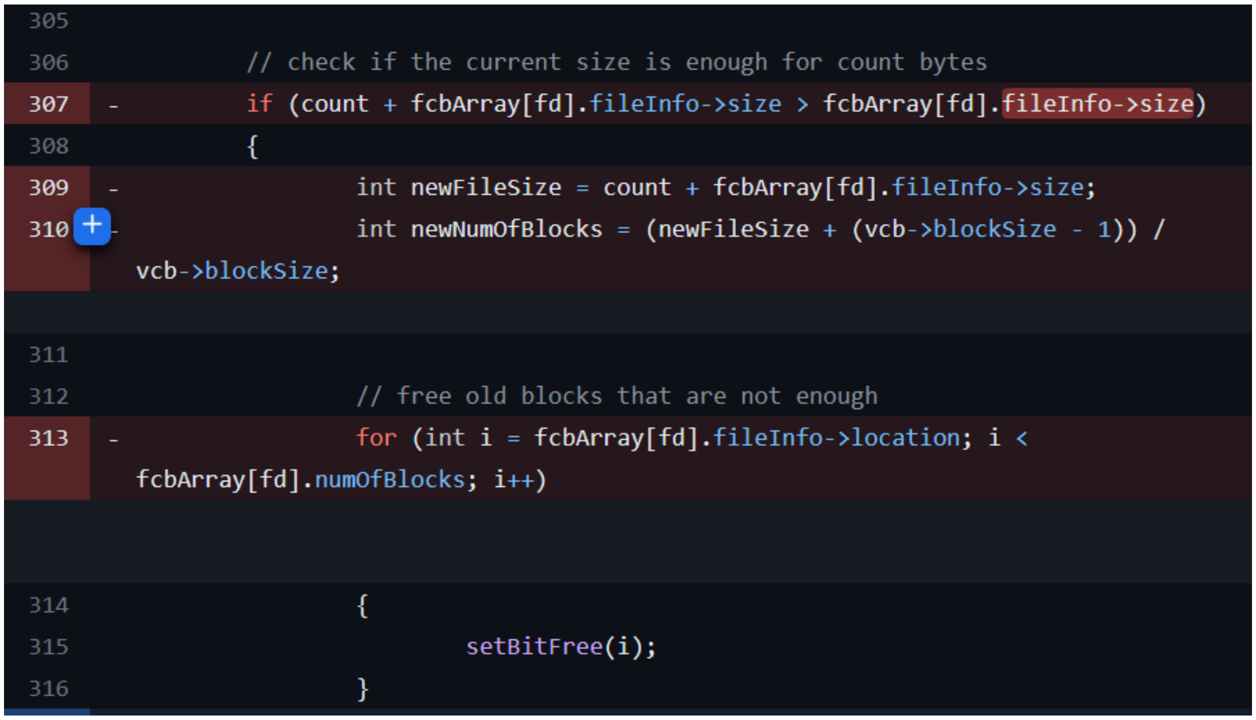
No Bounds Checking:
The code didn't check if the new position is within the bounds of the file, it was missing this validation because it's essential to ensure that the seek operation remains within the file's size to prevent seeking beyond the file's boundaries.

Lack of Error Reporting:
The code was lacking detailed error reporting, making it challenging to diagnose issues when the file descriptor is invalid or when the seeking operation fails due to other reasons.

c. b_write():

<u>Issue:</u>

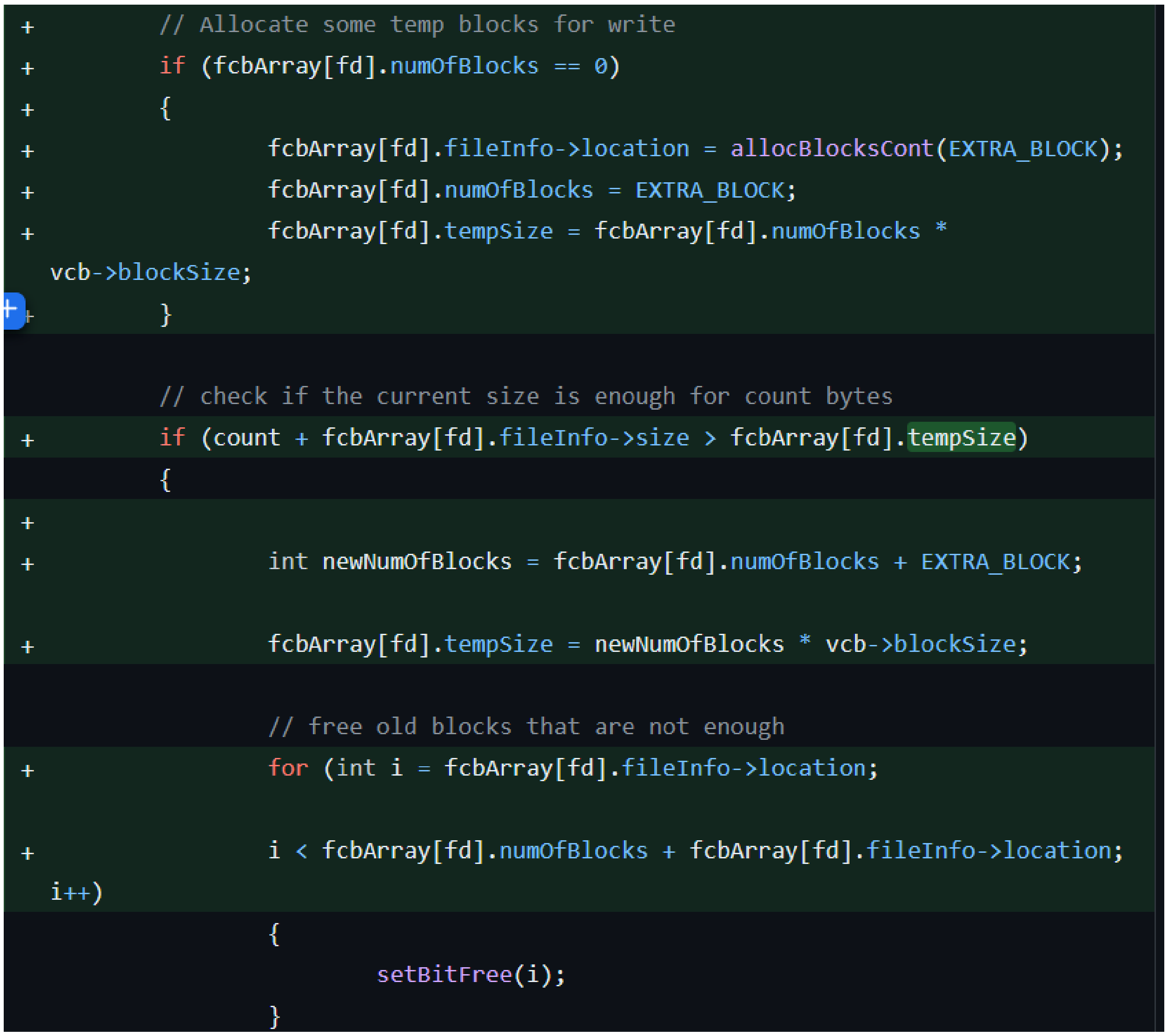
The condition of checking enough file size is always true. The loop that frees old blocks has an incorrect loop condition.



```

        // check if the current size is enough for count bytes
-       if (count + fcbArray[fd].fileInfo->size > fcbArray[fd].fileInfo->size)
        {
-               int newFileSize = count + fcbArray[fd].fileInfo->size;
-               int newNumOfBlocks = (newFileSize + (vcb->blockSize - 1)) / vcb->blockSize;

                // free old blocks that are not enough
-               for (int i = fcbArray[fd].fileInfo->location; i < fcbArray[fd].numOfBlocks; i++)
                {
                        setBitFree(i);
                }
```

<u>Solution</u>

Allocates temporary blocks (EXTRA_BLOCK is defined as 50) if the file has no blocks allocated yet. Updates the temporary size (tempSize) based on the new number of blocks. Comparing the current file size plus count bytes with the temporary file size for write operation. If more bytes are needed, another EXTRA _BLOCK is allocated. Finally, frees old blocks that are not enough, using the correct loop condition.

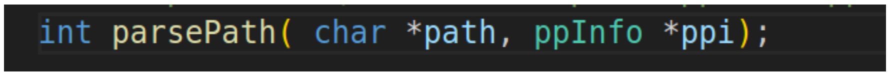


```
+           // Allocate some temp blocks for write
+           if (fcbArray[fd].numOfBlocks == 0)
+           {
+                   fcbArray[fd].fileInfo->location = allocBlocksCont(EXTRA_BLOCK);
+                   fcbArray[fd].numOfBlocks = EXTRA_BLOCK;
+                   fcbArray[fd].tempSize = fcbArray[fd].numOfBlocks *
   vcb->blockSize;
+           }

            // check if the current size is enough for count bytes
+           if (count + fcbArray[fd].fileInfo->size > fcbArray[fd].tempSize)
            {
+
+                   int newNumOfBlocks = fcbArray[fd].numOfBlocks + EXTRA_BLOCK;

+                   fcbArray[fd].tempSize = newNumOfBlocks * vcb->blockSize;

                    // free old blocks that are not enough
+                   for (int i = fcbArray[fd].fileInfo->location;

+                   i < fcbArray[fd].numOfBlocks + fcbArray[fd].fileInfo->location;
   i++)
                    {
                            setBitFree(i);
                    }
```

3.2 File Name: fsParse.c

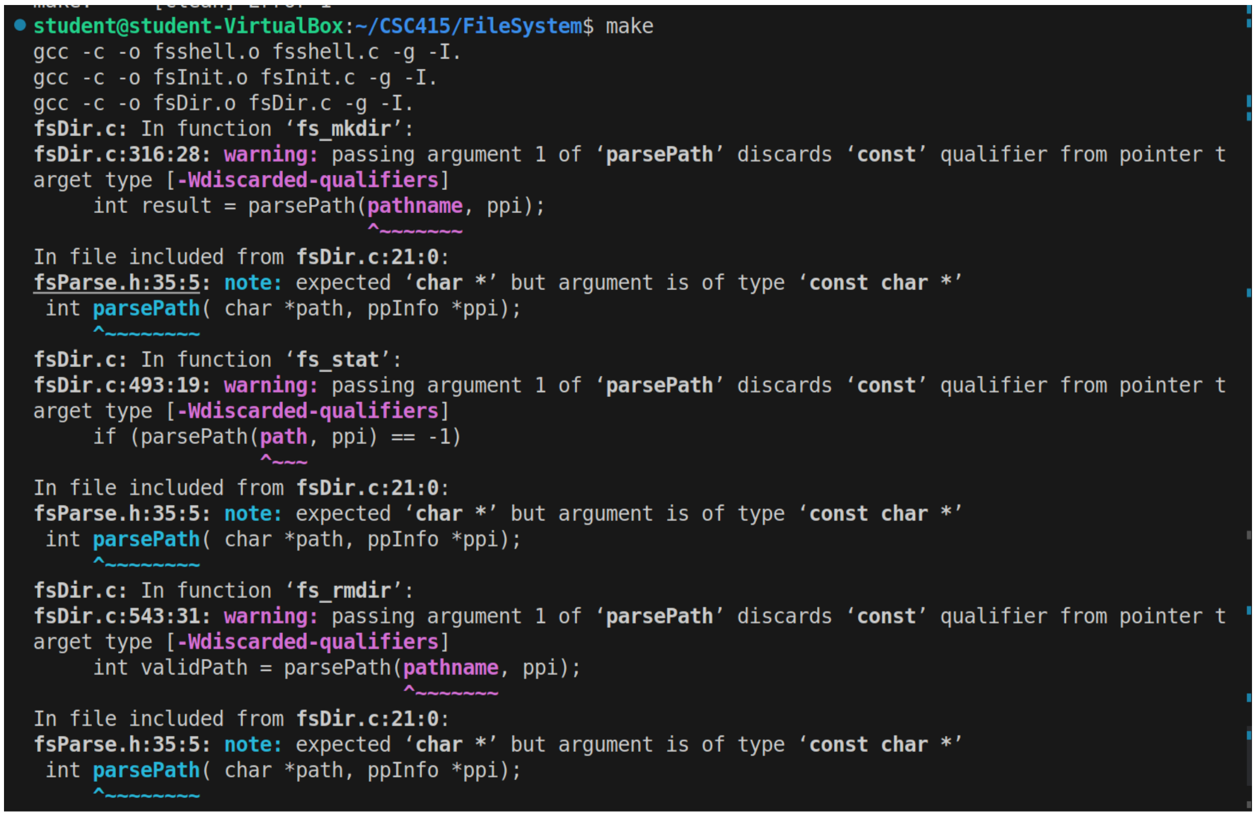
<u>Parameter type with parsePath():</u>
parsePath is a key function that many other functions will reference. It takes a string of characters as path, and updates the PPI structure, containing the parent directory, the index in the directory, and the string of the last Element.

When we declare parsePath,



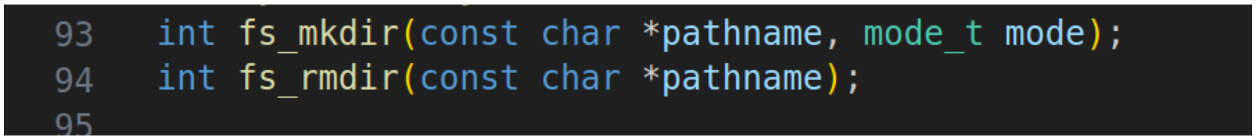
```
int parsePath( char *path, ppInfo *ppi);
```

we got this error:



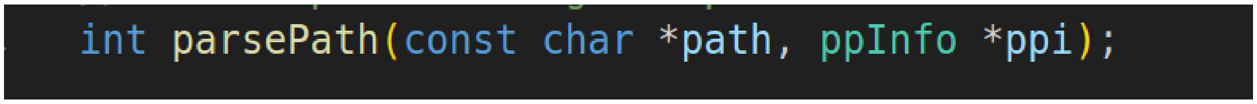
```
student@student-VirtualBox:~/CSC415/FileSystem$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsDir.o fsDir.c -g -I.
fsDir.c: In function ‘fs_mkdir’:
fsDir.c:316:28: warning: passing argument 1 of ‘parsePath’ discards ‘const’ qualifier from pointer t
arget type [-Wdiscarded-qualifiers]
     int result = parsePath(pathname, ppi);
                            ^~~~~~~~
In file included from fsDir.c:21:0:
fsParse.h:35:5: note: expected ‘char *’ but argument is of type ‘const char *’
 int parsePath( char *path, ppInfo *ppi);
     ^~~~~~~~~
fsDir.c: In function ‘fs_stat’:
fsDir.c:493:19: warning: passing argument 1 of ‘parsePath’ discards ‘const’ qualifier from pointer t
arget type [-Wdiscarded-qualifiers]
     if (parsePath(path, ppi) == -1)
                   ^~~~
In file included from fsDir.c:21:0:
fsParse.h:35:5: note: expected ‘char *’ but argument is of type ‘const char *’
 int parsePath( char *path, ppInfo *ppi);
     ^~~~~~~~~
fsDir.c: In function ‘fs_rmdir’:
fsDir.c:543:31: warning: passing argument 1 of ‘parsePath’ discards ‘const’ qualifier from pointer t
arget type [-Wdiscarded-qualifiers]
     int validPath = parsePath(pathname, ppi);
                               ^~~~~~~~
In file included from fsDir.c:21:0:
fsParse.h:35:5: note: expected ‘char *’ but argument is of type ‘const char *’
 int parsePath( char *path, ppInfo *ppi);
     ^~~~~~~~~
```

This is because the function fs_mkdir() and fs_rmdir() take const char * pathname.



```c
int fs_mkdir(const char *pathname, mode_t mode);
int fs_rmdir(const char *pathname);
```

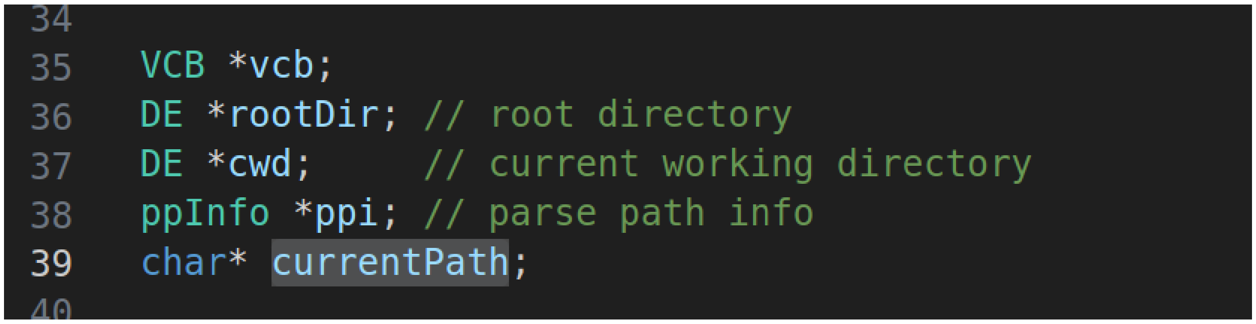
So, we modify the prototype of the function and make a copy the string to modify it:



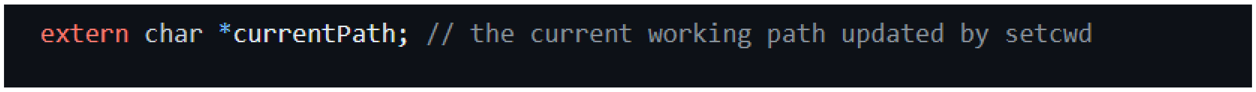
```c
int parsePath(const char *path, ppInfo *ppi);
```

3.3 File Name: fsInit.c

<u>Global variables:</u>
When we implement directory functions, we need global variable DE * cwd and DE * rootDir to keep track of the current working directory and the root directory.
When we implement and reference parsePath, we need a global variable ppInfo *ppi to store the current parsing path information.
When we implement fs_getcwd(), we need a global variable char * currentPath.
We declare them in the fsInit.c file, and use extern to share it in the header file:



```c
VCB *vcb;
DE *rootDir; // root directory
DE *cwd;     // current working directory
ppInfo *ppi; // parse path info
char* currentPath;
```

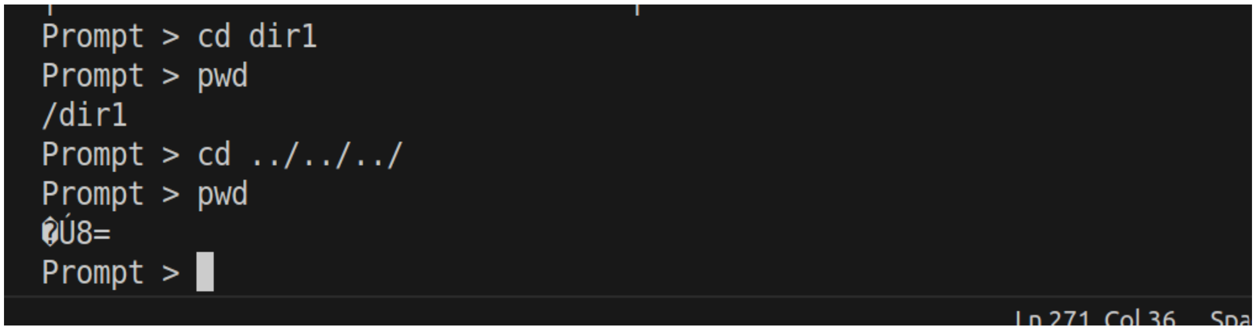


```c
extern char *currentPath; // the current working path updated by setcwd
```

3.4 File Name: fsDir.c

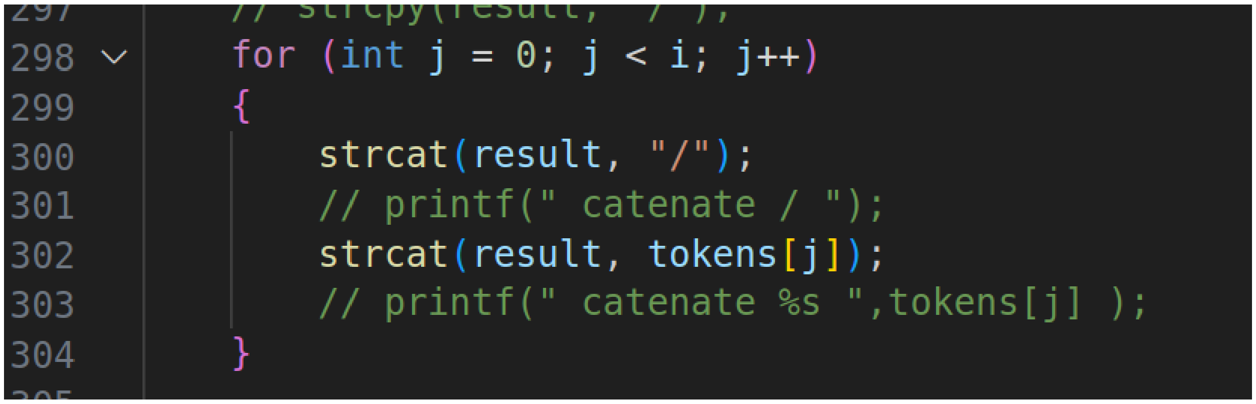
<u>cleanPath():</u>
This function takes a string parameter path, and gets the most clean path of the current working directory. We tokenize the string, and store each token in an array. We have a variable i to keep track of which index we need to read to get the clean path.

One issue we met is that when the index is 0(should be the root directory), we have some invalid value:



```
Prompt > cd dir1
Prompt > pwd
/dir1
Prompt > cd ../../../
Prompt > pwd
�Ú8=
Prompt > 
```

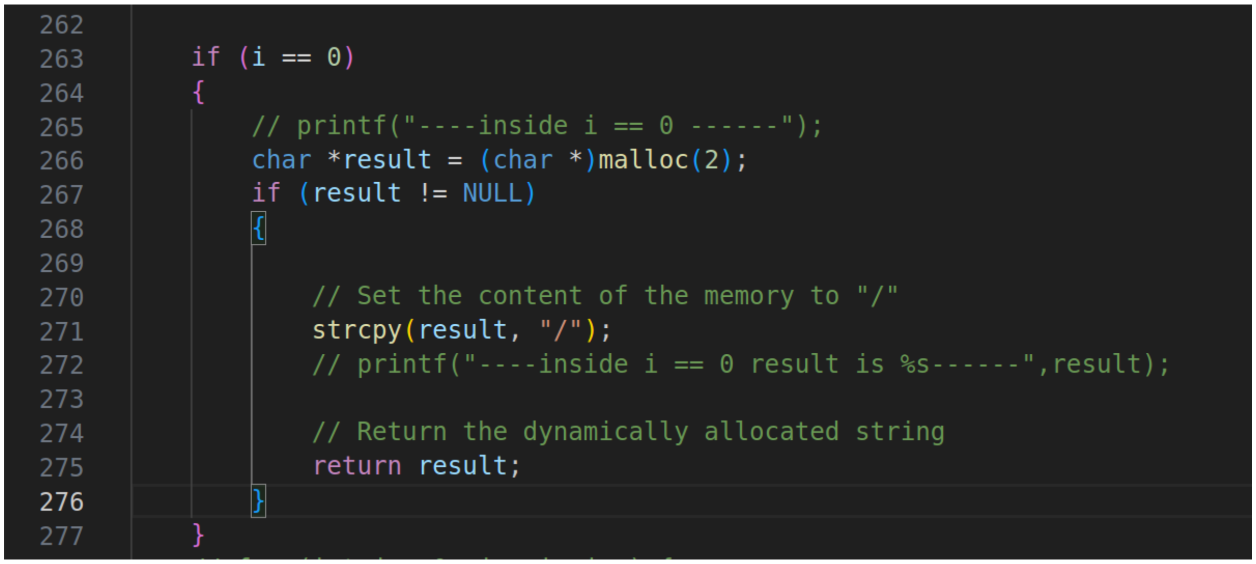
When we analyzed our program, we found the issue. We get the clean path by accessing the value in the array. If there is valid value, we strcat “/” and tokens[i] to the result:



```c
    // strcpy(result, "/");
    for (int j = 0; j < i; j++)
    {
        strcat(result, "/");
        // printf(" catenate / ");
        strcat(result, tokens[j]);
        // printf(" catenate %s ",tokens[j] );
    }
```

But when i == 0, we did not store any value to the result, so the result takes whatever is in the memory(invalid value).

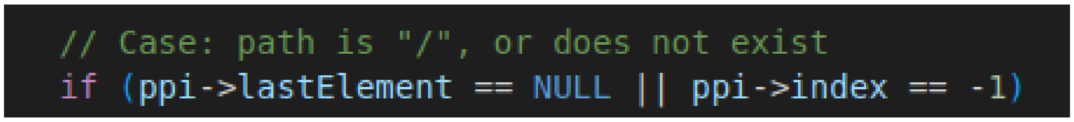
We fixed it by malloc 2 bytes to store the “/” character to the result.



```
    if (i == 0)
    {
        // printf("----inside i == 0 ------");
        char *result = (char *)malloc(2);
        if (result != NULL)
        {

            // Set the content of the memory to "/"
            strcpy(result, "/");
            // printf("----inside i == 0 result is %s------",result);

            // Return the dynamically allocated string
            return result;
        }
    }
```

fs_isFile() & fs_isDir()

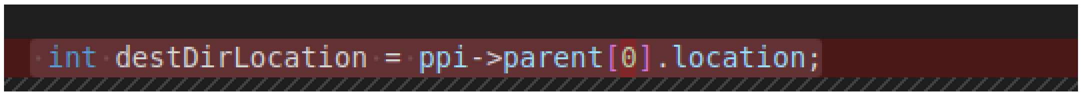
- Issue: Not checking for the path is "/" or the last element does not exist.
- Solution: Add condition check, last element is NULL for “/” path or index of last element is -1 if it does not exist.



```
// Case: path is "/", or does not exist
if (ppi->lastElement == NULL || ppi->index == -1)
```

fs_move()

Issue: Not getting the correct location of the destination directory.



```
int destDirLocation = ppi->parent[0].location;
```

Solution: We use condition check to handle root directory since the parent of root directory is itself, the location can be retrieved from the directory entry at index 0 or 1. For other directories, use the index in its parent directory to get location.

4. A table of who worked on which components

NO	Topic	Task	Member
Milestone 1 (due Thu 10/26 at 11:59PM)			
1	HexDump	HexDump analysis	Ruxue, Yee-Tsing
2	VCB	DE and VCB struct	Ruxue
3		mfs.h	Hoang-Anh
4	Free Space	Implement some functions (loadFreeSpace()) etc…)	Sidney
5		Bitmap, block allocation	Yee-Tsing
6	Root Directory	Help debug the initDir function	Hoang-Anh
7		Implement initDir function	Ruxue
8	Document	The writeup for FS Milestone 1	*all members*
9	GitHub	GitHub management & support	Sidney
Milestone 2 (due Thu 11/16 at 11:59PM)			
10	parsePath	fsParse.c and fsParse.h	Ruxue, Sidney, Yee-Tsing
11	fsDir.c	fs_setcwd, fs_isFile, fs_isDir, fs_mkdir	Ruxue
12		fs_opendir, fs_readdir	Hoang-Anh
13		fs_getcwd, fs_delete, fs_rmdir, debug	Yee-Tsing
14		fs_closedir, fs_stat	Sidney
15	fsshell.c	cmd_mv() function	Yee-Tsing

Milestone 3 (due Tue 11/28 at 11:59PM)			
16	b_io.c	b_write	Ruxue
17		b_read	Hoang-Anh
18		b_close, debug b_io.c	Yee-Tsing
19		b_open, b_seek	Sidney
20	Writeup	ls, mv, rm, cp, touch, cat/ cp2l, cp2fs	Yee-Tsing
21		cd, pwd,md	Ruxue
22		Part 1 and 3	Sidney
23		Part 2	Ruxue
24		Part 4 and 5	Hoang-Anh
25		Part 6	Yee-Tsing

5. Detail of how our driver program works

Summary:

Commands	Purpose
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	Creates a file
cat	(limited functionality) displays the contents of a file
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

Details:

a. ls

To list files in a directory and supports options like -l for long format and -a for showing hidden files. The ls command is implemented using the fs_opendir and fs_readdir functions. It opens the current directory using fs_opendir and then iterates through the directory entries using fs_readdir to display information about each file or directory.

b. cp

To copy a file that supports both source and destination arguments. The cp command is implemented using the parsePath function to identify the source and destination paths. It uses functions like fs_isFile and fs_isDir to check the types of the source and destination. If the source is a file, it uses fs_read and fs_write to copy the file content to the destination.

c. mv

To move a file that requires both valid source and destination arguments. The fs_move function moves a file (src) to a new location (dest). The mv command involves moving a file from one location to another. It uses parsePath to get information about the source and destination paths. If the source is a file, it uses fs_move to move the file, which involves updating directory entries.

d. md

To create a new directory. The fs_mkdir function is used to create a new directory. It uses parsePath to identify the path where the new directory needs to be created. The initDir function is used to initialize the directory and allocate necessary blocks.

e. rm

To remove files or directories. For directories, it uses fs_rmdir after checking if the directory is empty. For files, it uses fs_delete after checking if the file exists.

f. touch

To create or update a file. The fs_mkdir function is used to create a new directory. A new directory is initialized using the initDir function. The touch command is implemented using the parsePath function to get information about the file path. It uses fs_open with appropriate flags to create an empty file.

g. cat

To display the content of a file. The cat command uses fs_open to open a file and fs_read to read its content. It then displays the content to the console.

h. cp2l

To copy a file from the test file system to the Linux file system. The fs_opendir and fs_readdir functions are involved in listing files in a directory. The LBAread function involves reading a file from the file system using functions like fs_open and fs_read. It then writes the content to a local file.

i. cp2fs

To copy a file from the Linux file system to the test file system. The LBAwrite function is used to write data to the file system. cp2fs involves reading a local file and using functions like fs_open and fs_write to copy the content to the file system.

j. cd

To change the current directory. The fs_setcwd function changes the current working directory (cwd). It involves parsing the given path and loading the corresponding directory by updating the currentPath variable.

k. pwd

To print the current working directory. The fs_getcwd function retrieves the current working directory (cwd) and then prints it.

l. history

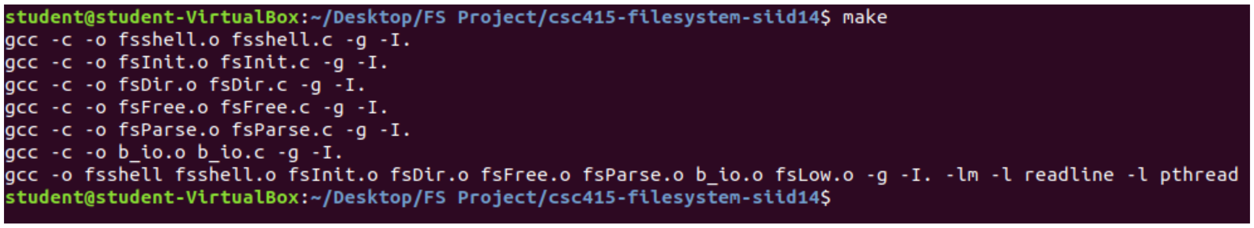
To print a history of previously executed commands.

m. help

To print helpful information about available commands and their usage.

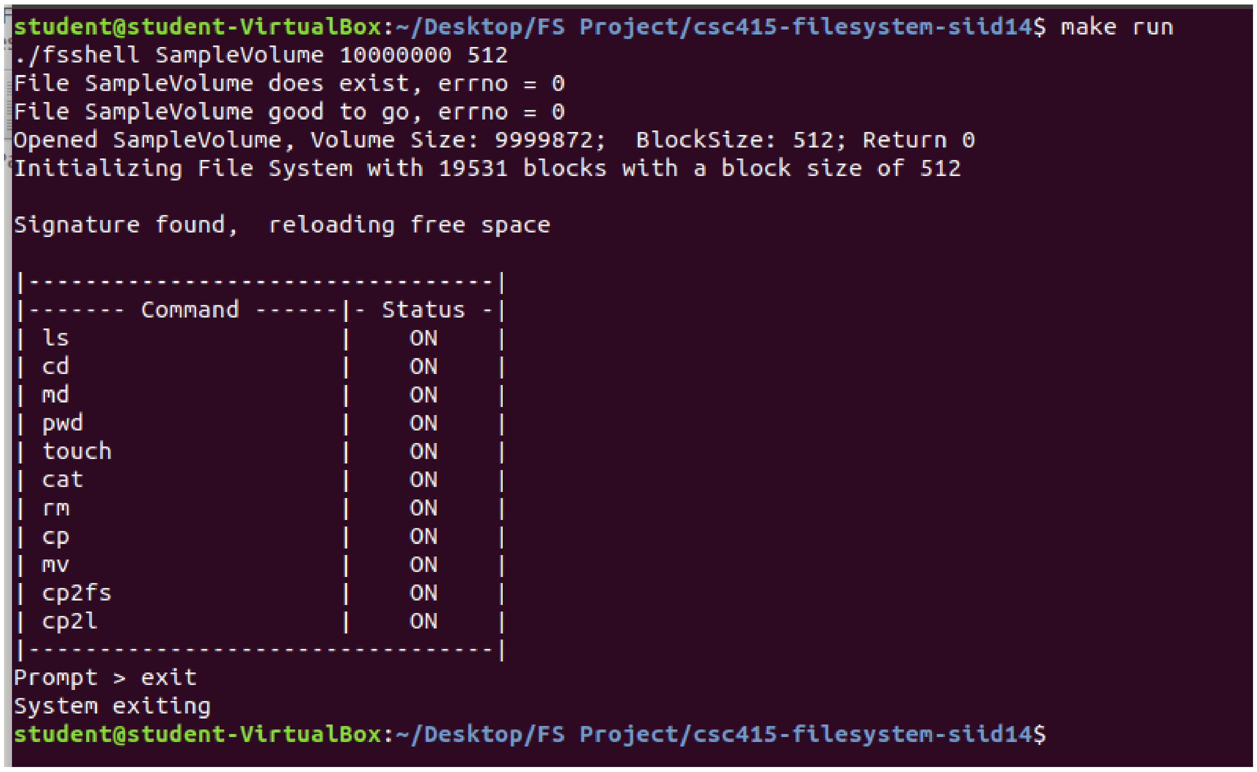
6. Screenshots showing each of the commands listed in the README.md

Screenshot of compilation:



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsDir.o fsDir.c -g -I.
gcc -c -o fsFree.o fsFree.c -g -I.
gcc -c -o fsParse.o fsParse.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsDir.o fsFree.o fsParse.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

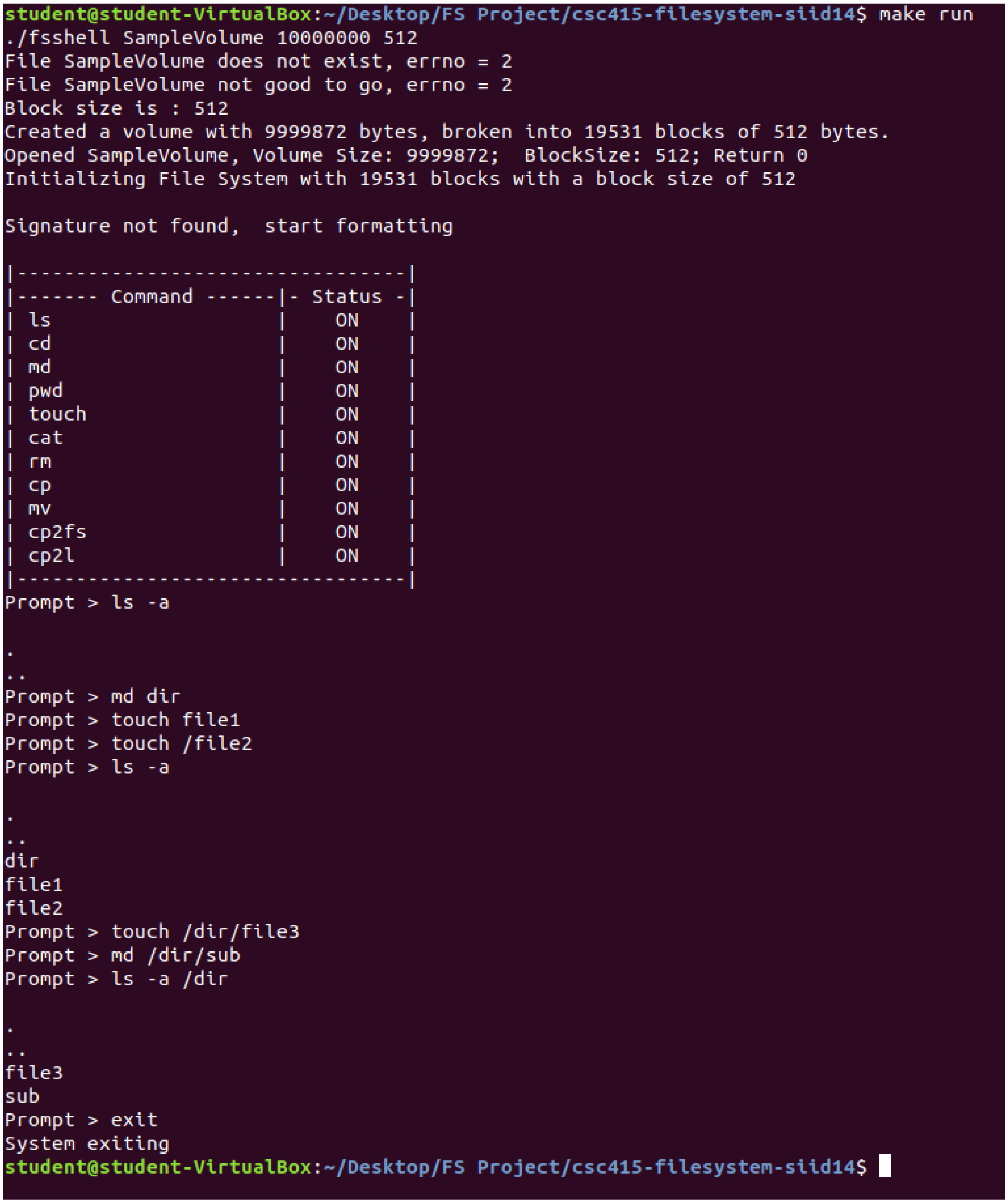
Screen shot(s) of the execution of the program:



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ 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

Signature found,  reloading free space

|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.1 ls, touch, and md

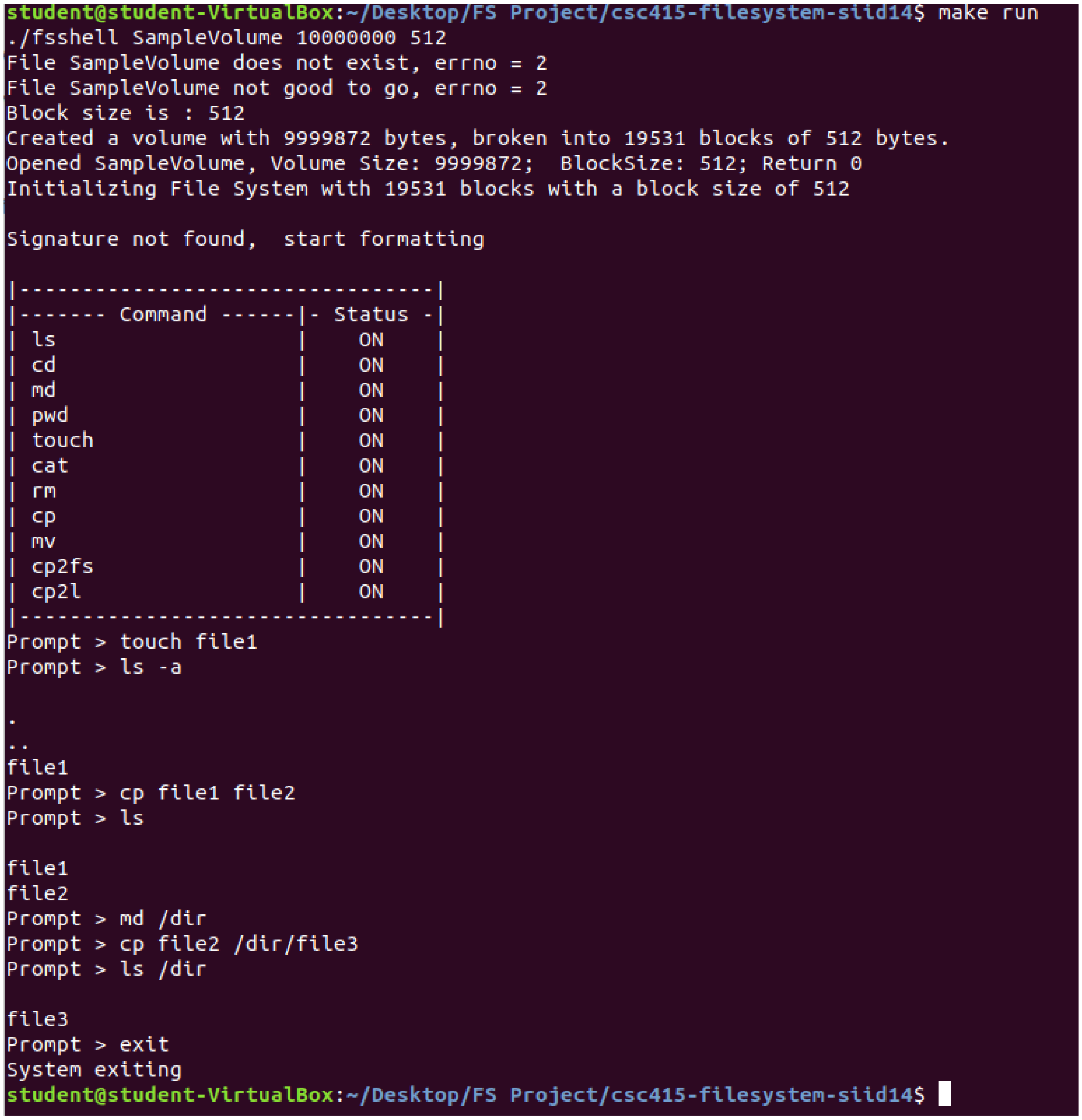
Start with an new sample volume, make directory “dir”, create files “file1” and “file2” in root directory, and then make directory “sub” and create “file3” in /dir



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

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

.
..
Prompt > md dir
Prompt > touch file1
Prompt > touch /file2
Prompt > ls -a

.
..
dir
file1
file2
Prompt > touch /dir/file3
Prompt > md /dir/sub
Prompt > ls -a /dir

.
..
file3
sub
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.2 cp

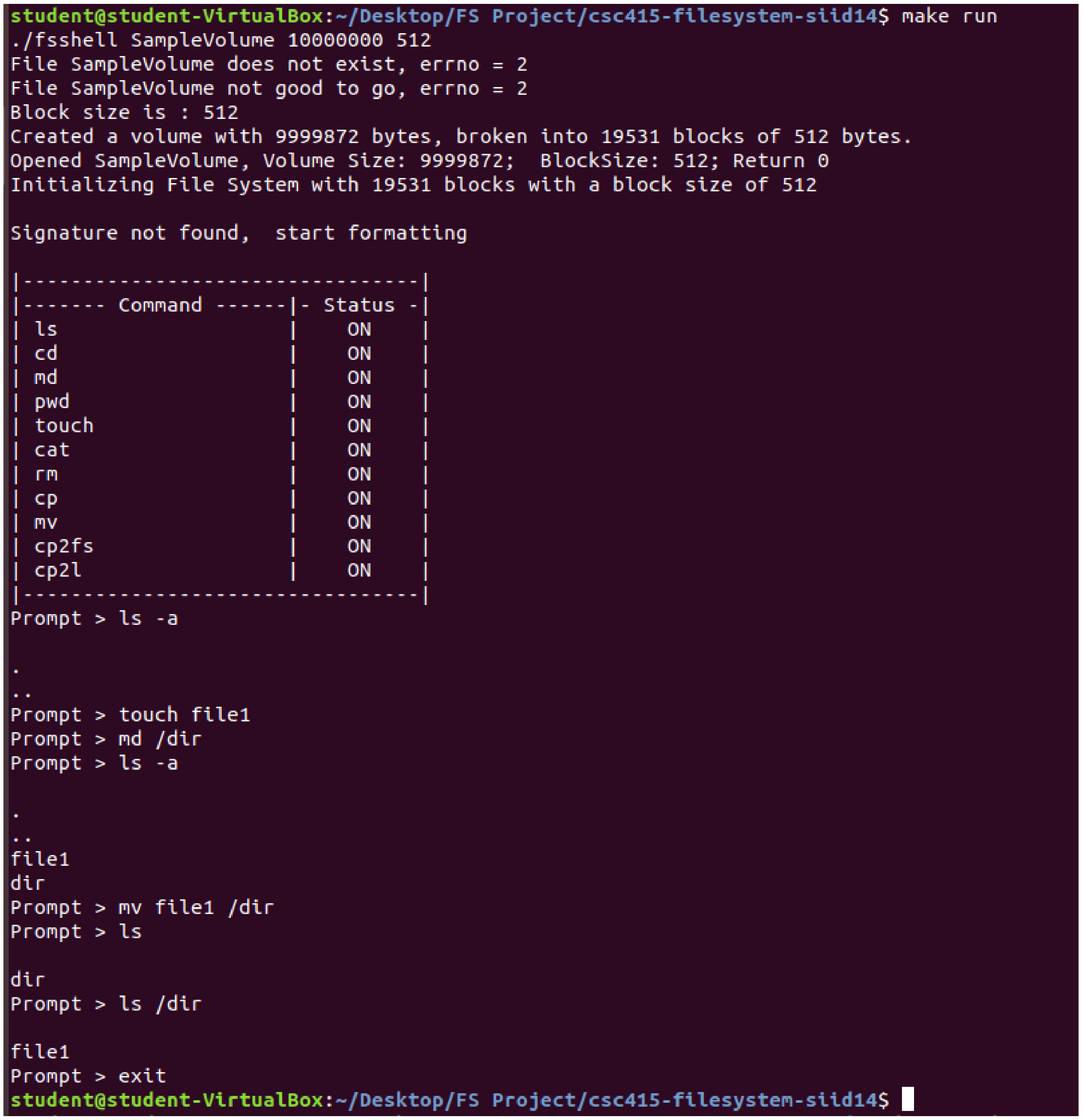
Start with an new sample volume, create “file1” and copy “file1” to “file2” in root directory make “dir” directory in root, copy “file2” to “file3” in /dir



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|-------------------------------|
Prompt > touch file1
Prompt > ls -a

.
..
file1
Prompt > cp file1 file2
Prompt > ls

file1
file2
Prompt > md /dir
Prompt > cp file2 /dir/file3
Prompt > ls /dir

file3
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.3 mv

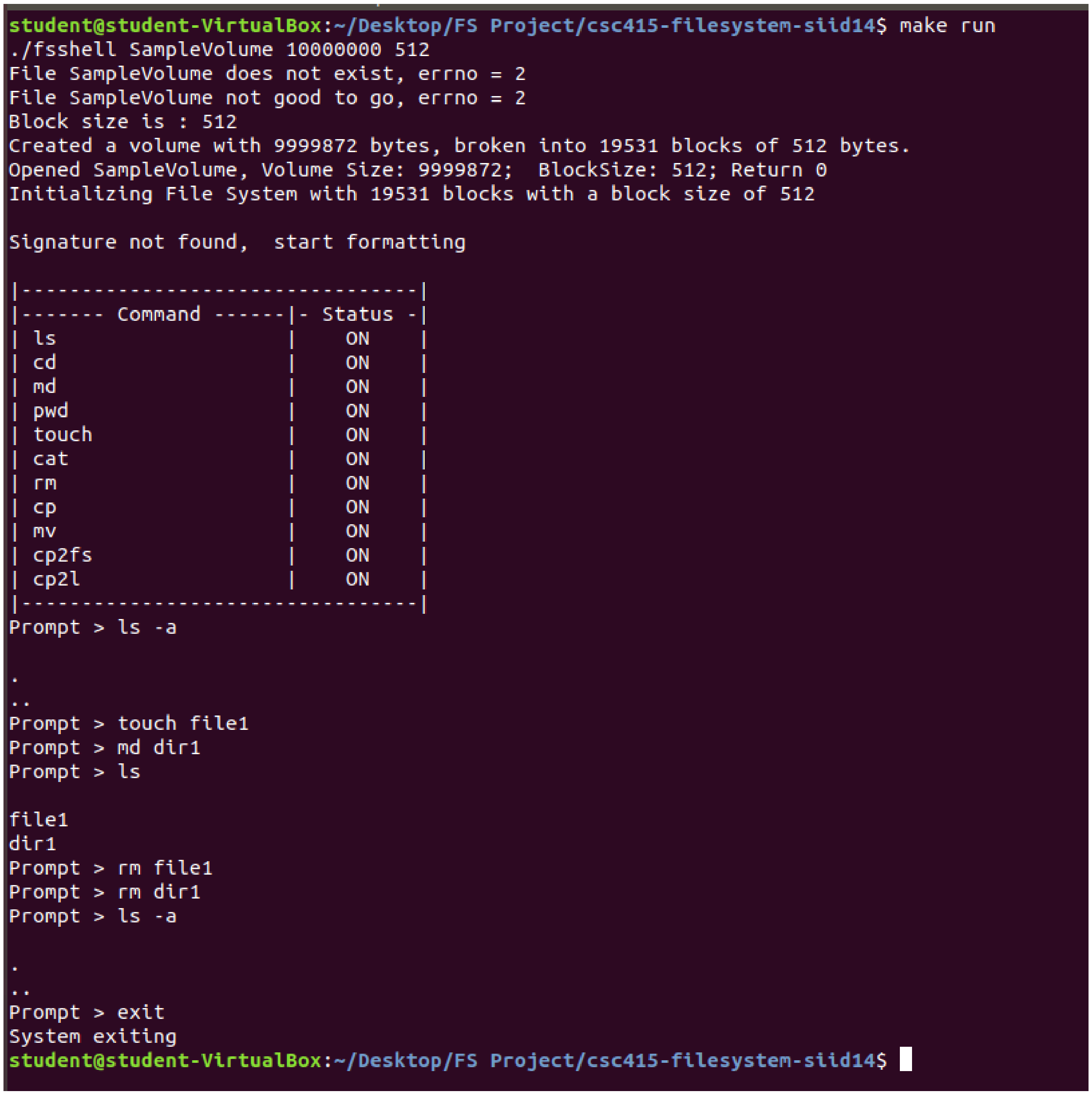
Start with an new sample volume, create “file1” and make “dir” directory in root, then move “file1” to /dir



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

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

.
..
Prompt > touch file1
Prompt > md /dir
Prompt > ls -a

.
..
file1
dir
Prompt > mv file1 /dir
Prompt > ls

dir
Prompt > ls /dir

file1
Prompt > exit
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.4 rm

Start with a new sample volume, create “file1” and make “dir1” directory in the root directory, then remove them.

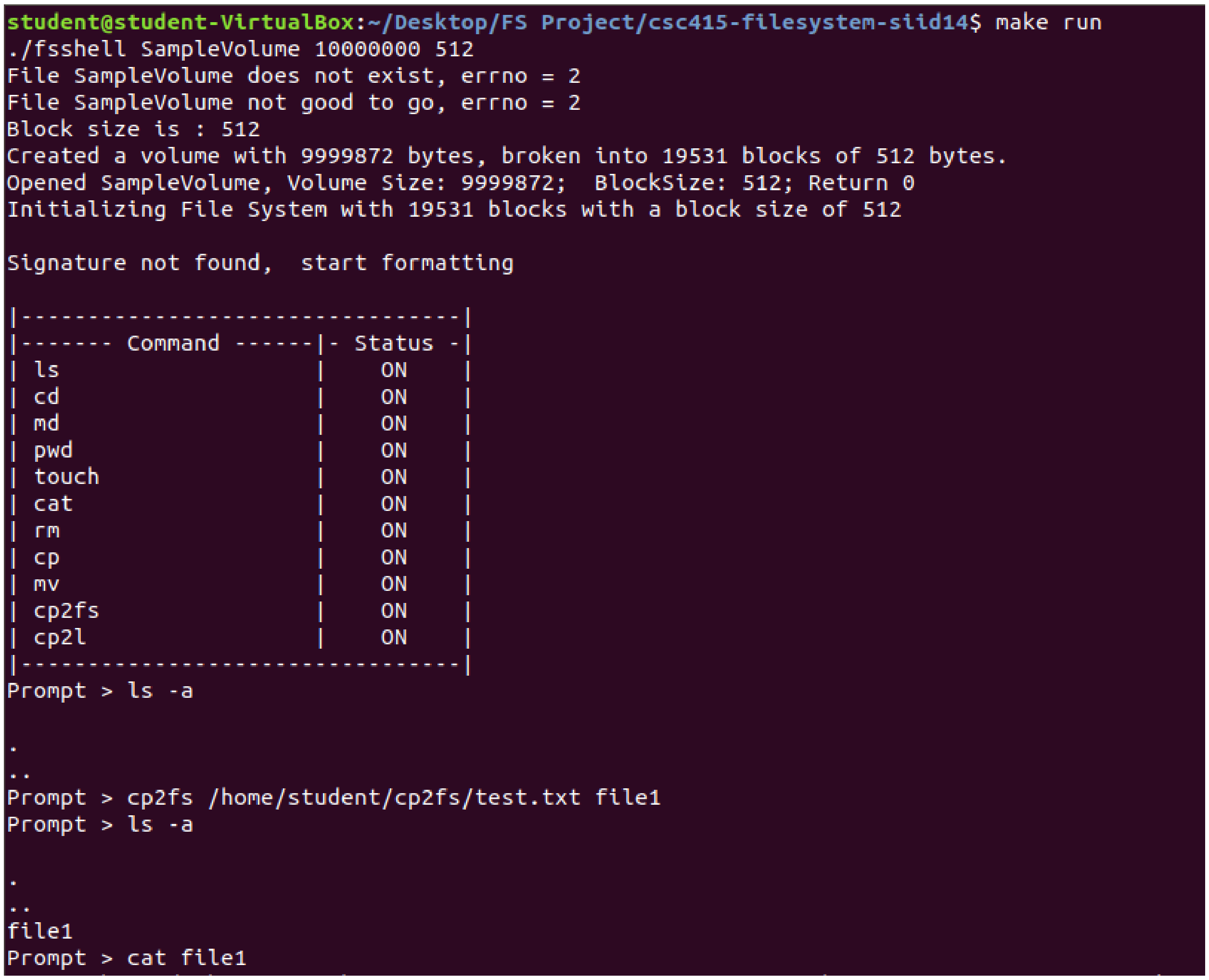


```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

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

.
..
Prompt > touch file1
Prompt > md dir1
Prompt > ls

file1
dir1
Prompt > rm file1
Prompt > rm dir1
Prompt > ls -a

.
..
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

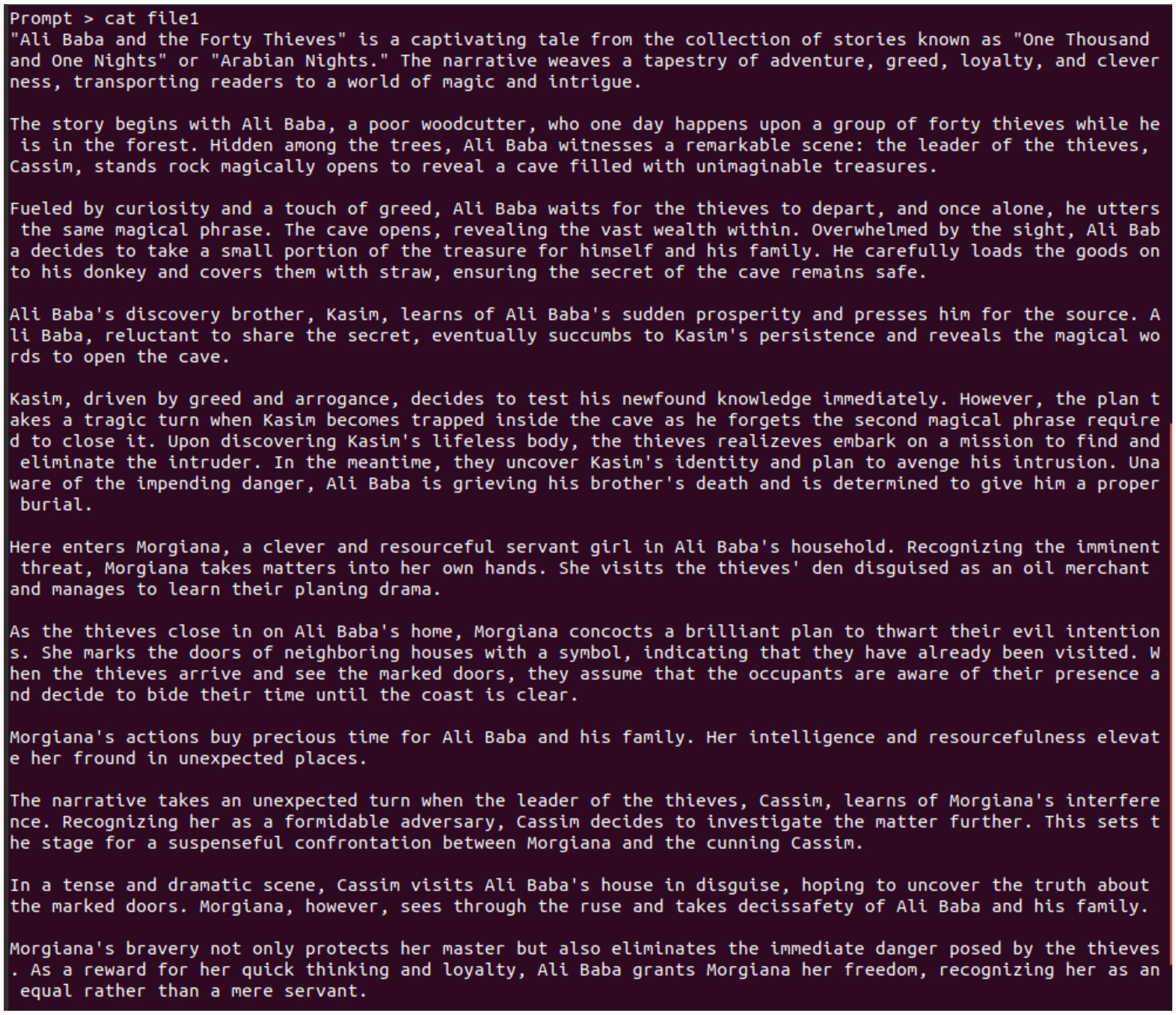
6.5 cp2fs, cp2l, and cat

Start with a new sample volume, copy the file with path “/home/student/cp2fs/test.txt” from Linux to “file1” in the root directory of our file system, and then display the contents of the file.

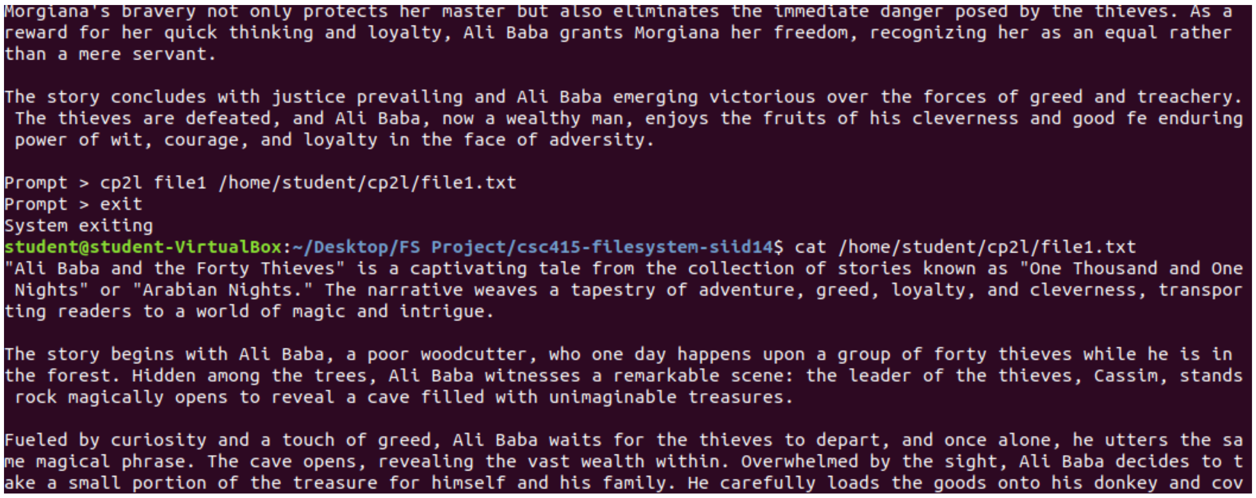
copy “file1” from our file system to the path “/home/student/cp2l/file1.txt” in Linux file system, quit the FS program, and display the contents of file1.txt from Linux terminal.



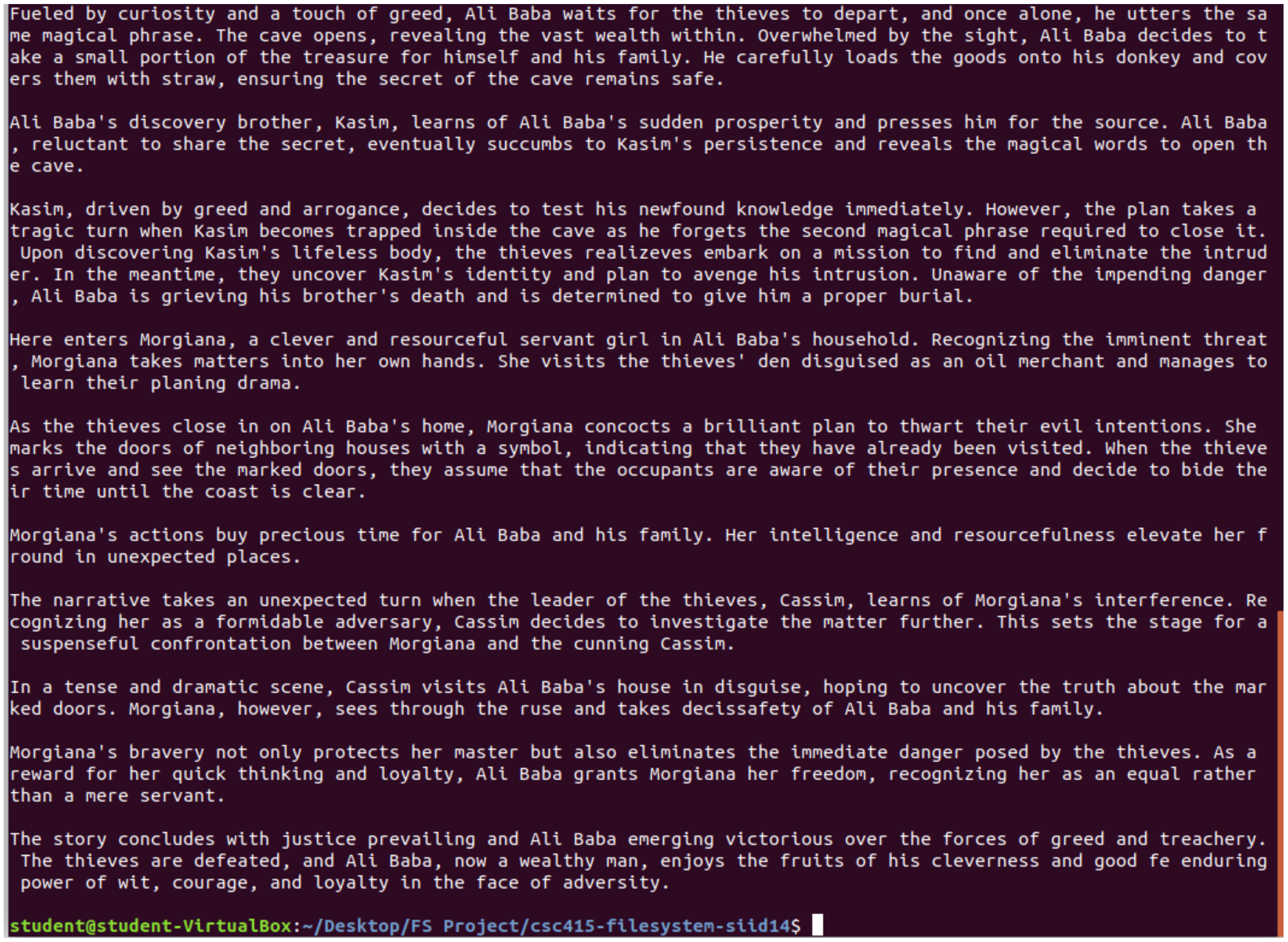
```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

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

.
..
Prompt > cp2fs /home/student/cp2fs/test.txt file1
Prompt > ls -a

.
..
file1
Prompt > cat file1
```



```
Prompt > cat file1
"Ali Baba and the Forty Thieves" is a captivating tale from the collection of stories known as "One Thousand
and One Nights" or "Arabian Nights." The narrative weaves a tapestry of adventure, greed, loyalty, and clever
ness, transporting readers to a world of magic and intrigue.

The story begins with Ali Baba, a poor woodcutter, who one day happens upon a group of forty thieves while he
 is in the forest. Hidden among the trees, Ali Baba witnesses a remarkable scene: the leader of the thieves,
Cassim, stands rock magically opens to reveal a cave filled with unimaginable treasures.

Fueled by curiosity and a touch of greed, Ali Baba waits for the thieves to depart, and once alone, he utters
 the same magical phrase. The cave opens, revealing the vast wealth within. Overwhelmed by the sight, Ali Bab
a decides to take a small portion of the treasure for himself and his family. He carefully loads the goods on
to his donkey and covers them with straw, ensuring the secret of the cave remains safe.

Ali Baba's discovery brother, Kasim, learns of Ali Baba's sudden prosperity and presses him for the source. A
li Baba, reluctant to share the secret, eventually succumbs to Kasim's persistence and reveals the magical wo
rds to open the cave.

Kasim, driven by greed and arrogance, decides to test his newfound knowledge immediately. However, the plan t
akes a tragic turn when Kasim becomes trapped inside the cave as he forgets the second magical phrase require
d to close it. Upon discovering Kasim's lifeless body, the thieves realizeves embark on a mission to find and
 eliminate the intruder. In the meantime, they uncover Kasim's identity and plan to avenge his intrusion. Una
ware of the impending danger, Ali Baba is grieving his brother's death and is determined to give him a proper
 burial.

Here enters Morgiana, a clever and resourceful servant girl in Ali Baba's household. Recognizing the imminent
 threat, Morgiana takes matters into her own hands. She visits the thieves' den disguised as an oil merchant
and manages to learn their planing drama.

As the thieves close in on Ali Baba's home, Morgiana concocts a brilliant plan to thwart their evil intention
s. She marks the doors of neighboring houses with a symbol, indicating that they have already been visited. W
hen the thieves arrive and see the marked doors, they assume that the occupants are aware of their presence a
nd decide to bide their time until the coast is clear.

Morgiana's actions buy precious time for Ali Baba and his family. Her intelligence and resourcefulness elevat
e her fround in unexpected places.

The narrative takes an unexpected turn when the leader of the thieves, Cassim, learns of Morgiana's interfere
nce. Recognizing her as a formidable adversary, Cassim decides to investigate the matter further. This sets t
he stage for a suspenseful confrontation between Morgiana and the cunning Cassim.

In a tense and dramatic scene, Cassim visits Ali Baba's house in disguise, hoping to uncover the truth about
the marked doors. Morgiana, however, sees through the ruse and takes decissafety of Ali Baba and his family.

Morgiana's bravery not only protects her master but also eliminates the immediate danger posed by the thieves
. As a reward for her quick thinking and loyalty, Ali Baba grants Morgiana her freedom, recognizing her as an
 equal rather than a mere servant.
```



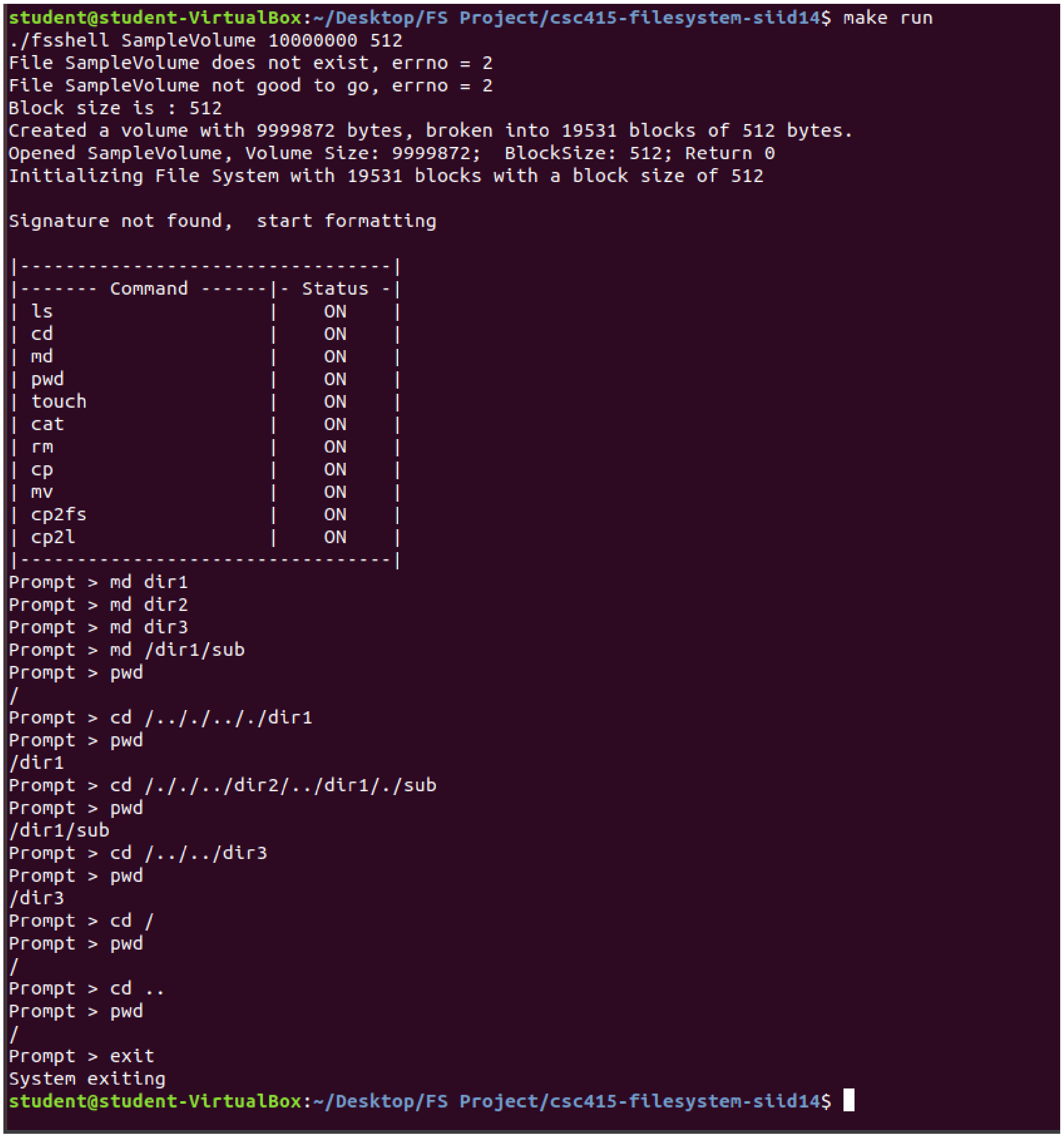
```
Morgiana's bravery not only protects her master but also eliminates the immediate danger posed by the thieves. As a
reward for her quick thinking and loyalty, Ali Baba grants Morgiana her freedom, recognizing her as an equal rather
than a mere servant.

The story concludes with justice prevailing and Ali Baba emerging victorious over the forces of greed and treachery.
 The thieves are defeated, and Ali Baba, now a wealthy man, enjoys the fruits of his cleverness and good fe enduring
 power of wit, courage, and loyalty in the face of adversity.

Prompt > cp2l file1 /home/student/cp2l/file1.txt
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ cat /home/student/cp2l/file1.txt
"Ali Baba and the Forty Thieves" is a captivating tale from the collection of stories known as "One Thousand and One
 Nights" or "Arabian Nights." The narrative weaves a tapestry of adventure, greed, loyalty, and cleverness, transpor
ting readers to a world of magic and intrigue.

The story begins with Ali Baba, a poor woodcutter, who one day happens upon a group of forty thieves while he is in
the forest. Hidden among the trees, Ali Baba witnesses a remarkable scene: the leader of the thieves, Cassim, stands
 rock magically opens to reveal a cave filled with unimaginable treasures.

Fueled by curiosity and a touch of greed, Ali Baba waits for the thieves to depart, and once alone, he utters the sa
me magical phrase. The cave opens, revealing the vast wealth within. Overwhelmed by the sight, Ali Baba decides to t
ake a small portion of the treasure for himself and his family. He carefully loads the goods onto his donkey and cov
```



```
Fueled by curiosity and a touch of greed, Ali Baba waits for the thieves to depart, and once alone, he utters the sa
me magical phrase. The cave opens, revealing the vast wealth within. Overwhelmed by the sight, Ali Baba decides to t
ake a small portion of the treasure for himself and his family. He carefully loads the goods onto his donkey and cov
ers them with straw, ensuring the secret of the cave remains safe.

Ali Baba's discovery brother, Kasim, learns of Ali Baba's sudden prosperity and presses him for the source. Ali Baba
, reluctant to share the secret, eventually succumbs to Kasim's persistence and reveals the magical words to open th
e cave.

Kasim, driven by greed and arrogance, decides to test his newfound knowledge immediately. However, the plan takes a
tragic turn when Kasim becomes trapped inside the cave as he forgets the second magical phrase required to close it.
 Upon discovering Kasim's lifeless body, the thieves realizeves embark on a mission to find and eliminate the intrud
er. In the meantime, they uncover Kasim's identity and plan to avenge his intrusion. Unaware of the impending danger
, Ali Baba is grieving his brother's death and is determined to give him a proper burial.

Here enters Morgiana, a clever and resourceful servant girl in Ali Baba's household. Recognizing the imminent threat
, Morgiana takes matters into her own hands. She visits the thieves' den disguised as an oil merchant and manages to
 learn their planing drama.

As the thieves close in on Ali Baba's home, Morgiana concocts a brilliant plan to thwart their evil intentions. She
marks the doors of neighboring houses with a symbol, indicating that they have already been visited. When the thieve
s arrive and see the marked doors, they assume that the occupants are aware of their presence and decide to bide the
ir time until the coast is clear.

Morgiana's actions buy precious time for Ali Baba and his family. Her intelligence and resourcefulness elevate her f
round in unexpected places.

The narrative takes an unexpected turn when the leader of the thieves, Cassim, learns of Morgiana's interference. Re
cognizing her as a formidable adversary, Cassim decides to investigate the matter further. This sets the stage for a
 suspenseful confrontation between Morgiana and the cunning Cassim.

In a tense and dramatic scene, Cassim visits Ali Baba's house in disguise, hoping to uncover the truth about the mar
ked doors. Morgiana, however, sees through the ruse and takes decissafety of Ali Baba and his family.

Morgiana's bravery not only protects her master but also eliminates the immediate danger posed by the thieves. As a
reward for her quick thinking and loyalty, Ali Baba grants Morgiana her freedom, recognizing her as an equal rather
than a mere servant.

The story concludes with justice prevailing and Ali Baba emerging victorious over the forces of greed and treachery.
 The thieves are defeated, and Ali Baba, now a wealthy man, enjoys the fruits of his cleverness and good fe enduring
 power of wit, courage, and loyalty in the face of adversity.

student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

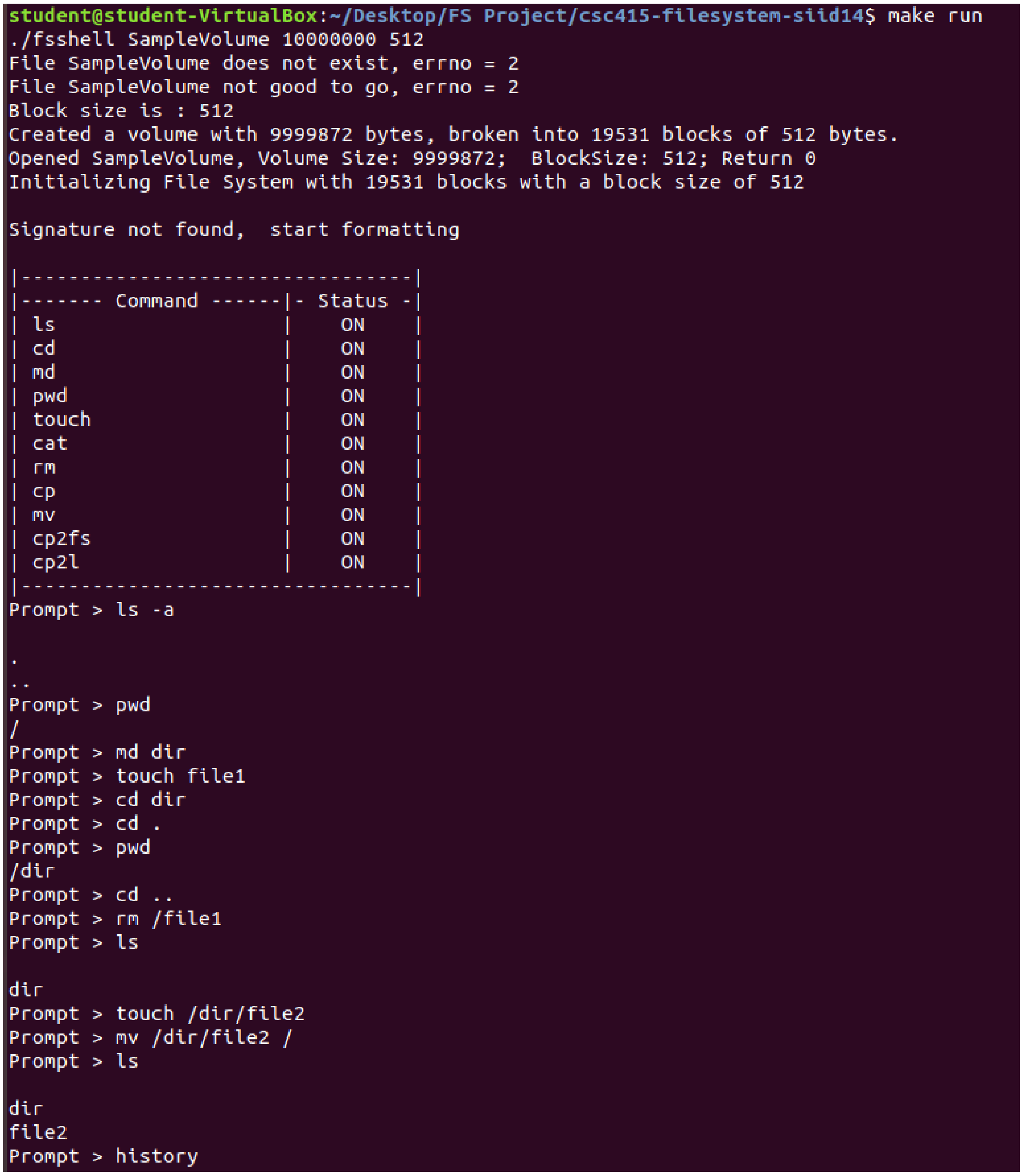
6.6 cd and pwd

Start with an new sample volume, make “dir1”, “dir2”, “dir3” directories in root directory, and then make “sub” directory in /dir1
Execute pwd for each cd: cd to /dir1,cd to /dir1/sub, cd to /dir3, cd to root directory, and cd to parent of root (itself).

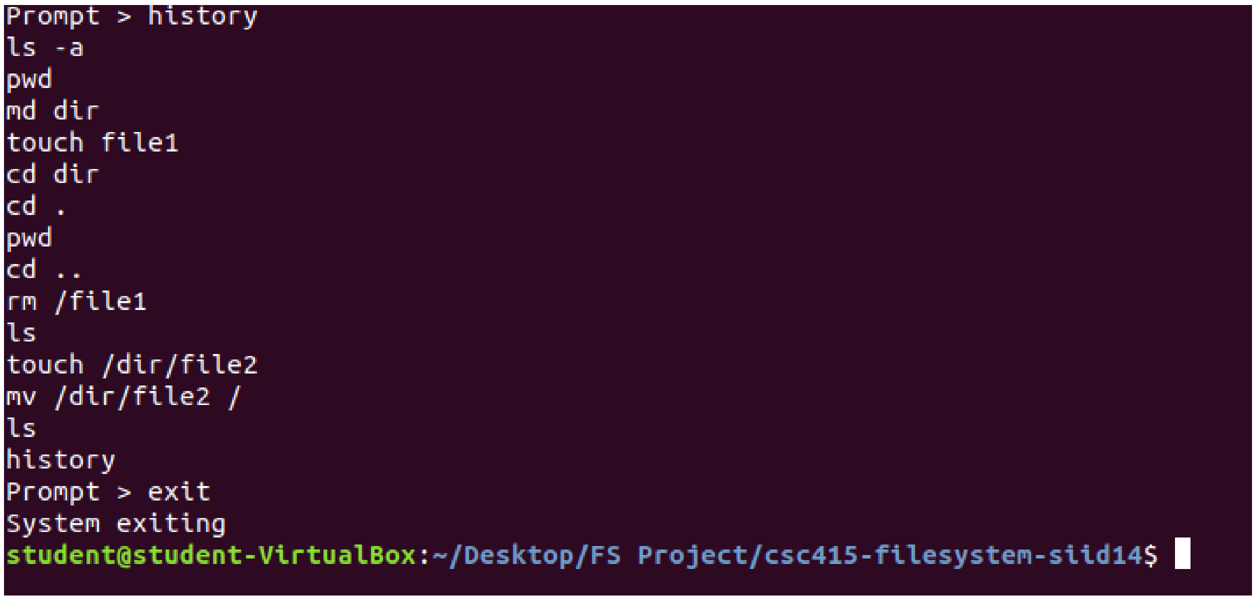


```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|-------------------------------|
Prompt > md dir1
Prompt > md dir2
Prompt > md dir3
Prompt > md /dir1/sub
Prompt > pwd
/
Prompt > cd /../../.././dir1
Prompt > pwd
/dir1
Prompt > cd /./../dir2/../dir1/./sub
Prompt > pwd
/dir1/sub
Prompt > cd /../../dir3
Prompt > pwd
/dir3
Prompt > cd /
Prompt > pwd
/
Prompt > cd ..
Prompt > pwd
/
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.7 history

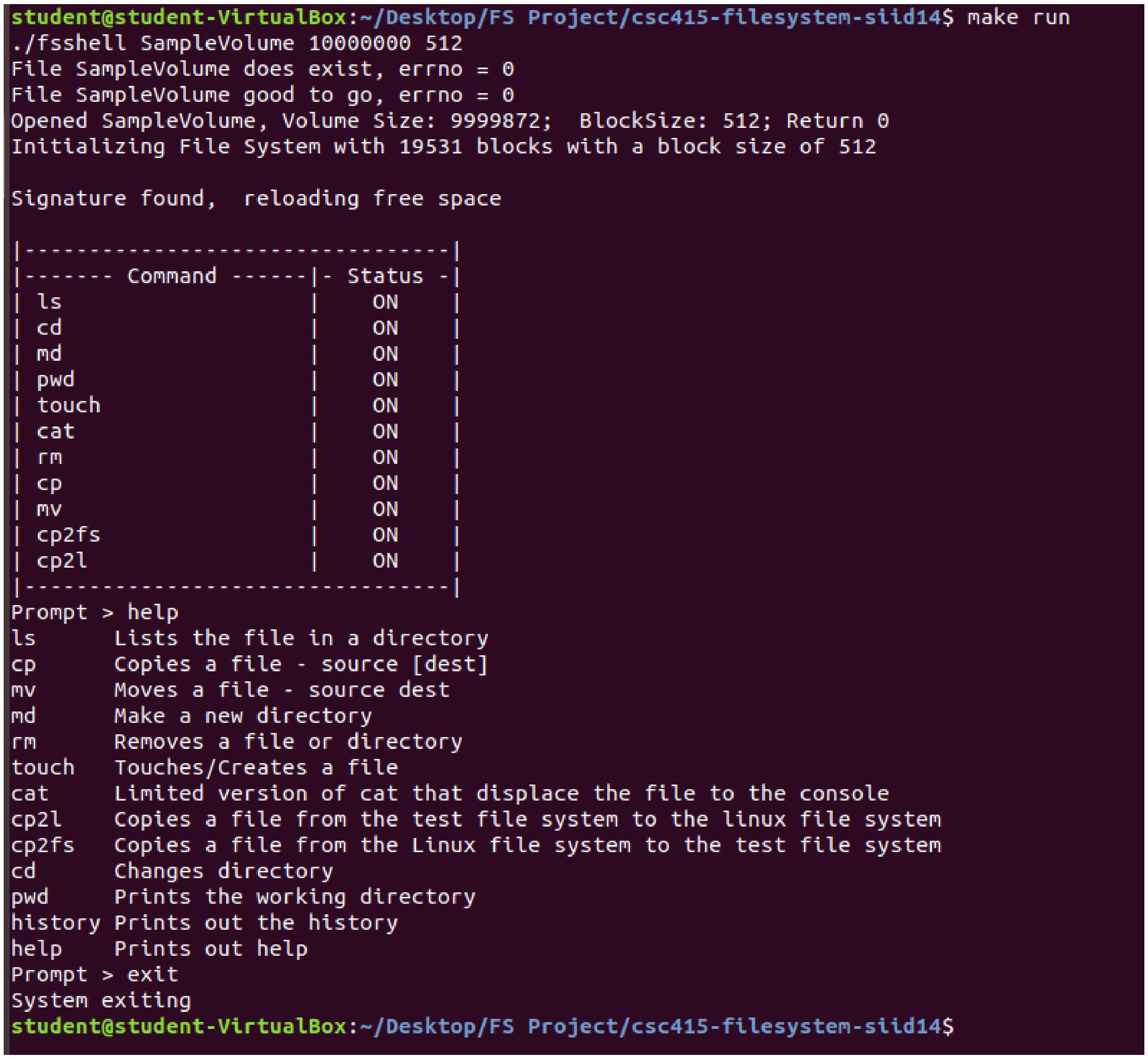


```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512

Signature not found,  start formatting

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

.
..
Prompt > pwd
/
Prompt > md dir
Prompt > touch file1
Prompt > cd dir
Prompt > cd .
Prompt > pwd
/dir
Prompt > cd ..
Prompt > rm /file1
Prompt > ls

dir
Prompt > touch /dir/file2
Prompt > mv /dir/file2 /
Prompt > ls

dir
file2
Prompt > history
```



```
Prompt > history
ls -a
pwd
md dir
touch file1
cd dir
cd .
pwd
cd ..
rm /file1
ls
touch /dir/file2
mv /dir/file2 /
ls
history
Prompt > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```

6.8 help



```
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$ 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

Signature found,  reloading free space

|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > 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 > exit
System exiting
student@student-VirtualBox:~/Desktop/FS Project/csc415-filesystem-siid14$
```