Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471
CSC415 Operating Systems

File System Writeup

Team 3

Github Link: https://github.com/CSC415-2024-Spring/csc415-filesystem-Colorbomb1

Github Name: Colorbomb1

Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471
CSC415 Operating Systems

Approach / Plan:

Our approach and plan for developing the file system was centered around creating directory and file management functionalities, alongside space management within our file system. We start by initializing the file system, setting up essential components such as the volume control block (VCB), which is crucial for managing the overall layout and access to the disk space. This included implementing the signature verification that was required. Required, but important to ensure data integrity and block management. Our directory management strategy involves constructing and manipulating directories to provide a simple web-like structure to hold our files and directories all together with the root directory. We have incorporated functions that allow for the creation of directory entries, searching for free directory blocks, and automatically configuring entries for current and parent directories. This approach ensures that we were able to manage memory dynamically and ensure reliable allocation of blocks. Our system is designed to handle errors strictly, mostly memory allocation failures and exceeding system limits like the maximum number of open files, which are critical for maintaining system stability and reduced risk of memory leaks. Our free space management (FSM)uses a bitmap system, a proven method for efficiently tracking which blocks on the disk are free or used. This enables effective allocation and deallocation of disk space, essential for the scalability and optimization of storage resources. Functions within this system include setting blocks as free or used, calculating the space needed for the free space map, and performing direct disk operations based on these maps. The inclusion of path parsing and manipulation utilities allows for seamless navigation and management within the file system, facilitating the translation between relative and absolute paths and enhancing the file system's usability.

File System Description:

DirectoryEntry.c file contains all the necessary functions to initialize, manage, and manipulate directory entries which are crucial for organizing files and directories within our file system.

At the core of our directory management functionality are functions to create and initialize directory entries. This includes setting up the structure for each directory, defining whether it is a

regular directory or a special one like '.' or '..', and allocating the necessary space in memory. We utilize an array of structures in **DirectoryManager.h and DirectoryManager.c** to dynamically allocate and configure directory structures based on the size calculated from the block size and the number of directory entries, along with keeping track of the starting location of each entry this ensured that each directory is optimally structured for performance and space utilization when its communicating with our Freespace bitmap structure. For our root directory, we initialize it at run-time; this involves setting up a directory entry for the root, allocating space, and writing the initial configuration to disk. This setup not only signifies the starting point of the directory structure but also anchors the entire directory management system, allowing for efficient path resolutions and directory modifications when we got to milestone two. The members of our array of structures in **DirectoryManager** and their purpose are as follows: **index** represents the position or identifier of the DirectoryManager within the DirMan array of structures. It can be used to reference or iterate through directory managers without having to search by memory address or other more confusing means. int **location** serves as the field that stores a reference to the starting block of the directory on the disk that this particular DirectoryManager is responsible for. This is essential for our direct disk operations in fsLow, such as reading or writing the directory data to and from disk, as it tells the file system exactly where the directory's data begins. ***thisDirectory*** *is a* pointer to the DirectoryEntry structure that describes the directory currently managed by our DirectoryManager. This is supposed to handle the metadata about the directory itself, such as its name, size, type, and time variables, which we weren't able to complete. In other words, it's a link between the directory manager and the directory entries it manages. ***directory_managers*** points to an array or linked list of DirectoryEntry structures to represent the actual contents of the directory, allowing the DirectoryManager to manage not just the directory itself but also its contents should the user want to make changes to it.

For the Freespace Manager (**FSM.h/c**), Our implementation begins with the creation and handling of a free space map, a simple but very critical component that tracks which blocks on the disk are available for use. This bitmap allows us to efficiently query and update the status of disk blocks, significantly optimizing our file system operations such as file creation, deletion, and resizing. We compute the necessary size for our bitmap based on the total number of blocks

and the block size of the volume, ensuring that every block is accounted for. This calculation considers the entire storage space, dividing it by the number of bits needed per block, which helps in determining how many bytes we need for the bitmap. After calculating the size, the bitmap is allocated enough memory to cover all blocks that are needed for the given blockSize parameter and the volume itself. This step is critical as it sets the stage for tracking the usage of disk blocks throughout the operation of the file system. Our system provides functions to set individual blocks as free or used within the bitmap. This granularity allows us to accurately and efficiently manage disk space, which is essential for maintaining high performance and storage utilization. This implementation includes functions to write the entire bitmap to disk, as well as to read the bitmap from disk. These operations are vital for the persistence of the file system state. Alongside this, we developed helpers within the file system to find free blocks, verify the continuity of our free space and which bits and bytes were being affected in certain instances. As a result this allows us to allocate or deallocate blocks as needed for the entries we create. These tools are integrated throughout the file system to support various file operations that require disk space management. Additional utility functions are included to aid in tasks such as finding the first free block, checking if a range of blocks is free, and setting or clearing blocks in a range. These utilities enhance the flexibility and robustness of our disk space management.

For parsepath.h/c: (**create_abs**) converts a given pathname into an absolute path. If the path is already absolute (starts with '/'), it simply duplicates it. Otherwise, it constructs an absolute path using the current working directory (**CWD**). This is essential for operations that require a full path for accurate file or directory location. (**validate_and_prepare_path**): It checks if the provided pathname is null, creates an absolute path, and prepares it for further processing. This function ensures that all path operations are performed on valid and correctly formatted paths, thereby preventing errors and inconsistencies in path handling. (**tokenize_path**): This function breaks down a path into its constituent components or "tokens'' based on the directory delimiter ('/'). These tokens are used to navigate through the directory structure. (**move_through_directory**): This function navigates through the directory entries based on each token derived from the path. It handles special directory names like "." (current directory) and ".." (parent directory) to navigate. This allows for traversal of directory structures. (**traverse_path**): after breaking a path into tokens, this function sequentially navigates through the filesystem's directory tree, using the tokens to guide the traversal from the root directory to the target directory or file. (**parse_path**) integrates validating, preparing, tokenizing, and traversing into a single operation to resolve a pathname to its corresponding DirectoryEntry. We use this in mfs to manipulate the directory entries that we create. Merging Paths (**merge_paths**)

is used for constructing full file or directory paths by appending a filename to a base path. This function is needed for operations like file creation, which we didn't do but helps, I guess, where a new file path needs to be generated dynamically. **mfs.h/c** takes these path manipulators and implements open_dir set and get_cwd, fs_mkdir to allow us to create directories.

Issues / Problems:

We weren't able to get removal down properly or directory traversal. This file system is only able to create new directories and print the working directory. There are a massive number of issues with our project and the main one is getting the cd command to properly traverse through our created directories, it's because of our implementation of paths, as we were strapped for time, we weren't able to get our paths to work in conjunction with newly made directory so there's no traversal in our manipulation functions. To fix this we felt like we should completely rework the pathing logic to check and monitor which path we are in. our approach of storing the path in an array could have been fortified in a much simpler way like the one shown in class. Another issue is the inability to display our directories metadata in a proper way, we are stuck with a segmentation fault when we attempt to use ls -al in order to read creation time and the other info. A potential fix for this would have been proper manipulation of time_created and so on and provide a method of getting the creation time once the directory is created. In milestone one we had a very disorganized approach to addressing the metadata and decided to focus more on the core components of the entries, this resulted in us losing track of time to get the metadata properly implemented in the entries.The existing testing for our path and directory management functionalities is not complete, we are leaving edge cases untested and that could prove to be very problematic when the user decides to create multiple directories.

Details on Functions:

File: VCB.c/h

int check_signatures(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: int (Returns 1 if signatures are valid, otherwise returns 0)

- Functionality: Checks the digital signatures of the given volume control block. Returns the validation status.

int verify_signatures(const VCB *volume)

- Parameters: const VCB *volume (Pointer to the volume control block structure)
- Return: int (Returns 1 if both digital signatures match predefined constants, otherwise returns 0)
- Functionality: Compares the digital signatures in the volume control block against predefined constants.

void report_signature_status(const VCB *volume, int isValid)

- Parameters: const VCB *volume (Pointer to the volume control block structure), int isValid (Integer treated as a boolean value indicating validity of signatures)
- Return: void
- Functionality: Outputs the status of the digital signatures to the standard output.

int init_volume_control_block(VCB *volume, uint64_t block_number, uint64_t init_block_size)

- Parameters: VCB *volume (Pointer to the volume control block structure), uint64_t block_number (Index of the block), uint64_t init_block_size (Size of a block in bytes)
- Return: int (Always returns 0)
- Functionality: Initializes a volume control block with digital signatures, block number, and block size. Outputs memory addresses of initialized fields.

uint64_t read_volume(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: uint64_t (Result of reading operation)
- Functionality: Reads data from a volume based on its control block configuration.

uint64_t write_volume(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: uint64_t (Result of writing operation)
- Functionality: Writes data to a volume based on its control block configuration.

void initialization_test()

- Parameters: None
- Return: void
- Functionality: Tests initialization and signature verification processes. Writes to the volume.

File: FSM.c/h

void calculate_free_block_space(VCB* Volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: void
- Functionality: Calculates the amount of free block space available on the volume.

int allocate_free_space_map(VCB* Volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: int (Status of allocation, typically 0 for success, non-zero for failure)
- Functionality: Allocates memory or structure for managing free space on the volume.

void mark_system_blocks_used(VCB* volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: void
- Functionality: Marks system blocks as used in the free space map to prevent reallocation.

int64_t init_free_space_map(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)

- Return: int64_t (Status of initialization, typically 0 for success, non-zero for failure)
- Functionality: Initializes the free space map for the volume.

int64_t write_free_space_map()

- Parameters: None
- Return: int64_t (Status of write operation, typically 0 for success, non-zero for failure)
- Functionality: Writes the current state of the free space map to persistent storage.

-

. void calculate_free_space_blocks(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: void
- Functionality: Calculates and updates the number of blocks needed for the free space map based on the block size and total blocks of the volume.

int allocate_free_space_map(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: int (Returns 0 on success, -1 if memory allocation fails)
- Functionality: Allocates memory for the free space map and initializes all bits to 1 (indicating blocks are free).

void mark_system_blocks_used(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: void
- Functionality: Marks the initial blocks (system-reserved plus those used for the free space map) as used in the free space map.

int64_t create_map(VCB *volume)

- Parameters: VCB *volume (Pointer to the volume control block structure)
- Return: int64_t (Returns 0 on success, -1 on error)
- Functionality: Initializes the free space map by calculating block requirements, allocating memory, marking system blocks, and writing the map to disk.

int64_t write_map()

- Parameters: None
- Return: int64_t (Result of write operation)
- Functionality: Writes the free space map to disk using the LBAwrite function.

u_int8_t *getFreeSpaceMapOffset(uint64_t index)

- Parameters: uint64_t index (Index of the block in the free space map)
- Return: u_int8_t * (Pointer to the specific byte in the free space map)
- Functionality: Computes the offset within the free space map for a given block index.

int64_t write_map_block(uint64_t index)

- Parameters: uint64_t index (Index of the block in the free space map to write)
- Return: int64_t (Result of write operation)
- Functionality: Writes a specific block of the free space map to disk.

int64_t read_map(uint64_t position, uint64_t count)

- Parameters: uint64_t position (Starting position on disk), uint64_t count (Number of blocks to read)
- Return: int64_t (Result of a read operation)
- Functionality: Reads the free space map from disk into memory.

int64_t set_free(uint64_t count, uint64_t startingBlock)

- Parameters: uint64_t count (Number of blocks to set free), uint64_t startingBlock (Starting block index)
- Return: int64_t (Number of blocks successfully set to free)
- Functionality: Sets a range of blocks as free in the free space map.

int64_t free_blocks_at_position(uint64_t count, uint64_t startingBlock)

- Parameters: uint64_t count (Number of blocks to free), uint64_t startingBlock (Starting block index)
- Return: int64_t (Number of blocks successfully freed)
- Functionality: Frees up blocks at specified positions and writes changes to disk.

int are_blocks_free(uint64_t start, uint64_t count)

- Parameters: uint64_t start (Starting block index), uint64_t count (Number of blocks to check)
- Return: int (Returns 1 if all blocks are free, 0 otherwise)
- Functionality: Checks if a range of blocks starting from a given position are all free.

int64_t find_first_free_block(int64_t map_len)

- Parameters: int64_t map_len (Total length of the free space map in blocks)
- Return: int64_t (Index of the first free block, or -1 if no free block found)
- Functionality: Scans the free space map to find the first free block.

int verify_contiguous_free_blocks(int64_t start_index, uint64_t count, int64_t map_len)

- Parameters: int64_t start_index (Starting block index), uint64_t count (Number of blocks to verify), int64_t map_len (Total length of the free space map in blocks)
- Return: int (Returns 1 if all blocks in the range are free, 0 otherwise)
- Functionality: Verifies if a range of blocks starting from start_index for count blocks are contiguous and free.

int64_t find_free_blocks(uint64_t count)

- Parameters: uint64_t count (Number of contiguous free blocks needed)
- Return: int64_t (Starting index of the contiguous free blocks, or -1 if not found)
- Functionality: Finds a contiguous range of free blocks of the specified count.

void mark_blocks_and_fill_positions(int64_t start_block, uint64_t count, int64_t *positions)

- Parameters: int64_t start_block (Starting block index), uint64_t count (Number of blocks), int64_t *positions (Array to fill with block positions)
- Return: void
- Functionality: Marks blocks starting from start_block as used and records their positions.

int64_t *freeblocks_lookup(uint64_t count)

- Parameters: uint64_t count (Number of free blocks needed)
- Return: int64_t * (Pointer to an array of block positions, or NULL if allocation fails)
- Functionality: Looks up a specified number of contiguous free blocks, marks them as used, and returns their positions.

- **File: DirectoryManager.c/h**

void initialize_DirMan()

- Parameters: None
- Return: void

- Functionality: Initializes the global array of DirectoryManager structures. Sets the location of all managers to -1 (indicating unused) and their index sequentially. The first entry is explicitly set to location 0 (indicating it's in use).

DirectoryManager *manager_start(int location)

- Parameters: int location (The starting block location to find the manager for)
- Return: DirectoryManager * (Pointer to the DirectoryManager structure, or NULL if not found)
- Functionality: Searches for a directory manager that manages a directory at a specific location. Returns the manager if found; otherwise, returns NULL.

DirectoryManager *get_free_man_block()

- Parameters: None
- Return: DirectoryManager * (Pointer to a free DirectoryManager structure, or NULL if none are available)
- Functionality: Searches for an unused directory manager in the global array. Returns the first unused manager found, or NULL if all are in use.

DirectoryManager *migrate_directories(DirectoryEntry *newEntry)

- Parameters: DirectoryEntry *newEntry (Pointer to the DirectoryEntry that needs a directory manager)
- Return: DirectoryManager * (Pointer to the DirectoryManager that will manage the newEntry, or NULL on failure)
- Functionality: Attempts to find an existing directory manager for the given directory entry's starting block or allocates a new one if necessary. Returns the manager handling the entry or NULL if unable to find or allocate a manager.

uint64_t calculate_size_in_bytes(uint64_t num_entries)

- Parameters: uint64_t num_entries (Number of directory entries)
- Return: uint64_t (Size in bytes required to store the given number of directory entries)
- Functionality: Calculates the total size in bytes needed to store a specified number of DirectoryEntry structures.

uint64_t calculate_size_in_blocks(uint64_t size_in_bytes, uint64_t blockSize)

- Parameters: uint64_t size_in_bytes (Total size in bytes), uint64_t blockSize (Size of a block in bytes)
- Return: uint64_t (Number of blocks required to store the given size in bytes)
- Functionality: Calculates how many blocks are needed to store the given size in bytes, based on the block size.

uint64_t recalculate_num_dir_entries(uint64_t size_in_blocks, uint64_t blockSize)

- Parameters: uint64_t size_in_blocks (Number of blocks), uint64_t blockSize (Size of a block in bytes)
- Return: uint64_t (Number of directory entries that can be stored in the given number of blocks)
- Functionality: Calculates the number of DirectoryEntry structures that can be stored within the specified number of blocks, based on the block size.

void get_sizes_for_entries(uint64_t blockSize, uint64_t *size_in_bytes, uint64_t *size_in_blocks, uint64_t *num_dir_entries)

- Parameters: uint64_t blockSize (Size of a block in bytes), uint64_t *size_in_bytes (Pointer to store size in bytes), uint64_t *size_in_blocks (Pointer to store size in blocks), uint64_t *num_dir_entries (Pointer to store number of directory entries)
- Return: void
- Functionality: Calculates and updates the values for size in bytes, size in blocks, and number of directory entries for a given block size, ensuring the size in bytes fits an exact number of DirectoryEntry structures.

Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471
CSC415 Operating Systems

**char *create_abs(const char pathname)*

Parameters:

const char *pathname: The path to be converted to an absolute path.

Return:

Returns an absolute path as a string.

Functionality:

Converts a given relative or absolute path into an absolute path based on the current working directory (CWD). If the path is already absolute, it returns a duplicate; otherwise, it constructs a full path using the CWD.

DirectoryEntry validate_and_prepare_path(const char pathname, char** prepared_path)**

Parameters:

const char* pathname: The path to validate and prepare.

char** prepared_path: Pointer to store the prepared absolute path.

Return:

Returns a pointer to a DirectoryEntry if the path points directly to the root directory, otherwise NULL.

Functionality:

Validates and converts a path to an absolute path, checks for special cases like root directory access, and prepares the path for further processing.

void tokenize_path(char path, char tokens[], int* token_count)**

Parameters:

char* path: The path to tokenize.

char* tokens[]: Array to store the path tokens.

int* token_count: Pointer to store the number of tokens generated.

Return:

void.

Functionality:

Splits a path into tokens based on the '/' delimiter, facilitating directory navigation.

DirectoryEntry move_through_directory(DirectoryEntry current_dir, const char* token, DirectoryEntry** pathgiven, int* pathgiven_index)**

Parameters:

DirectoryEntry* current_dir: Current directory to start from.

const char* token: Next token or part of the path to process.

DirectoryEntry** pathgiven: Array of directory entries for path navigation.

int* pathgiven_index: Pointer to the current index in the path navigation array.

Return:

Returns the next directory entry based on the token or NULL if not found.

Functionality:

Navigates through directories based on the given token, handling special directories like '.' and '..'.

DirectoryEntry traverse_path(char tokens[], int token_count)**

Parameters:

char* tokens[]: Array of path tokens.

int token_count: Number of tokens.

Return:

Returns the final directory entry after traversing all tokens or NULL if any part of the path is invalid.

Functionality:

Uses the tokens to navigate through the file system and find the target directory or file.

DirectoryEntry parse_path(const char pathname)**

Parameters:

const char* pathname: The full path to parse and resolve.

Return:

Returns the DirectoryEntry corresponding to the pathname or NULL if the path does not exist.

Functionality:

Combines validation, tokenization, and traversal to resolve a pathname to its corresponding directory entry.

**char* merge_paths(char *base, const char filename)*

Parameters:

char *base: Base path to which the filename will be appended.

const char *filename: Filename or relative path to append to the base.

Return:

Returns the newly constructed path combining the base and filename or NULL if memory allocation fails.

Functionality:

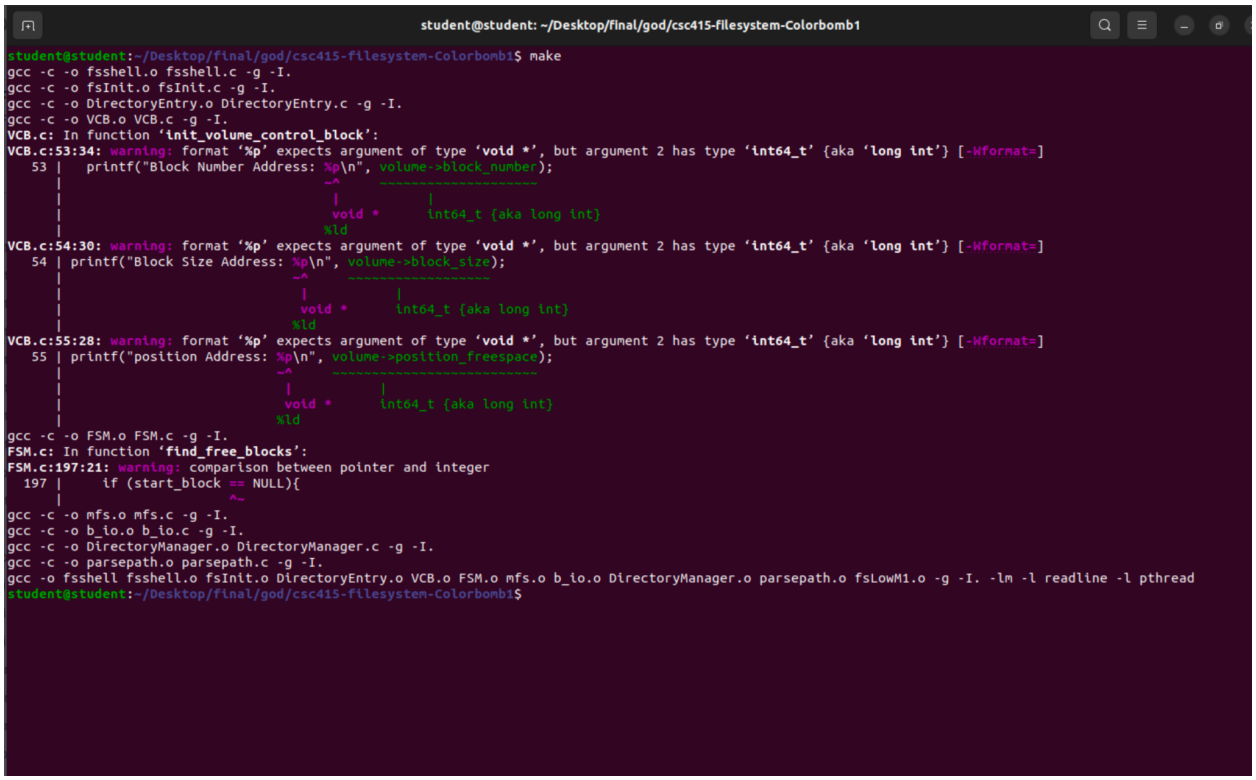
Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471

CSC415 Operating Systems

Constructs a new path by appending a filename or relative path to a base path, used for creating new file or directory paths.

Screenshots:

Compilation:



Initial Execution:

Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471
CSC415 Operating Systems

```
student@student:~/Desktop/final/god/csc415-filesystem-Colorbomb1$ 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
Size of VCB: 8
Digital signatures are valid.
SIGNATURES PASSED
Block Number Address: 0x4c4b
Block Size Address: 0x200
position Address: 0x1
Free space blocks: 5
writing volume
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    OFF   |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    OFF   |
| cat                  |    OFF   |
| rm                   |    OFF   |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|--------------------------------|
Prompt >
```

Execution of ls and md commands:

```
./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
Size of VCB: 8
Digital signatures are valid.
SIGNATURES PASSED
Block Number Address: 0x4c4b
Block Size Address: 0x200
position Address: 0x1
Free space blocks: 5
writing volume
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    OFF   |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    OFF   |
| cat                  |    OFF   |
| rm                   |    OFF   |
| cp                   |    OFF   |
| mv                   |    OFF   |
| cp2fs                |    OFF   |
| cp2l                 |    OFF   |
|--------------------------------|
Prompt > ls

Prompt > md testDirectory
Directory created at path: testDirectory /
Prompt > ls

testDirectory
Prompt > md anotherTestDir
Directory created at path: anotherTestDir /
Prompt > ls

testDirectory
anotherTestDir
Prompt > md oneMoreDir
Directory created at path: oneMoreDir /
Prompt > ls

testDirectory
anotherTestDir
oneMoreDir
Prompt >
```

Cd command and pwd issue:

Cole Brausen ID: 918418987
Gursimran Singh ID: 922759471
Jose Rios ID:921160471

CSC415 Operating Systems