Group Term Assignment - File System

Members:
Ulices Gonzalez, Marco Robles, Yash Pachori, Prashrit Magar

Student IDs:
923328897, 921282632, 923043313, 922068027

Github Usernames:
ulicessgg, CaliPrerunner, ypachori0, GameaddictXO

GitHub Repository:

https://github.com/CSC415-2024-Fall/csc415-filesystem-ulicessgg.git

Table of Contents

Description:

Milestone 1

In the first milestone, our file system is now capable of detecting a Volume Control Block in order to determine the need for formatting the current volume. If present the Volume Control Block will be loaded alongside the File Allocation Table to keep track of our free space throughout the file system, and the root directory which contains n amount of entries. If no Volume Control Block is present our file system will then initialize an instance with allocated memory as well as create and allocate an instance of our FAT and our root before writing it to memory. Upon completion, the file system will then terminate and free the allocated memory for the VCB, FAT, and Root. This can be tested and observed through the use of the hexdump utility which will dump the sample volume to the terminal showing the information for all three of the components of our file system.

Milestones 2 and 3

In the second and third milestones, our file system will be capable of handling user commands input through the command line that will initiate the manipulation of directories and their contents. This will be done through the use of the main file system interface with the implementation of the following functions found in the mfs files.

```
// Key directory functions
int fs_mkdir(const char *pathname, mode_t mode);
int fs_rmdir(const char *pathname);
// Directory iteration functions
fdDir * fs_opendir(const char *pathname);
struct fs_diriteminfo *fs_readdir(fdDir *dirp);
int fs_closedir(fdDir *dirp);
// Misc directory functions
char * fs_getcwd(char *pathname, size_t size);
int fs_setcwd(char *pathname);   //linux chdir
int fs_isFile(char * filename);   //return 1 if file, 0 otherwise
int fs_isDir(char * pathname);   //return 1 if directory, 0 otherwise
int fs_delete(char* filename);   //removes a file
```

These make use of the fs_dirItemInfo, fdDir, and fs_stat structures to support interactions within the file system. The support for Key File I/O Operations will be done through the implementation of the following functions found in the b_io files along with a modified file control block structure for reading and writing.

```
b_io_fd b_open (char * filename, int flags);
int b_read (b_io_fd fd, char * buffer, int count);
int b_write (b_io_fd fd, char * buffer, int count);
int b_seek (b_io_fd fd, off_t offset, int whence);
int b_close (b_io_fd fd);
```

Volume Control Block Structure

```
typedef struct volumeControlBlock
{
   uint64_t signature;// Signature to identify the volume // 8 bytes
   uint64_t totalBlocks; // number of blocks in the volume // 8 bytes
   uint64_t blockSize; // Size of each block in bytes // 8 bytes
   unsigned int freeBlockCount; // total free space count // 4 bytes
   unsigned int freeBlockStart; // Start index for free space // 4 bytes
   unsigned int fatSize; // size of the file allocation table // 4 bytes
   unsigned int rootLoc; // Index for the root directory // 4 bytes
   unsigned int fatLoc; // Index for the fat // 4 bytes
   char sysType[20]; // Holds the type of volume/file system // 20 bytes
} volumeControlBlock; // Total 64 Bytes
```

Our implementation of the Volume Control Block works with an identifying signature which is used when creating and or loading the VCB from memory. It contains the number of blocks it is made up of as well as the size of each block in terms of bytes. Besides this it also contains metadata regarding the location of the free space especially the beginning of it, the quantity of free space, the size of our manager of choice in this case a FAT, and the location of the root directory in terms of blocks as well as the location of the fat. Lastly, it has a description of its given system type which helps those unaware of our file system but also to identify it in our hexdump.

FAT / Free Space Tracking

```
extern int* FAT;
#define FAT_FREE -1          // Mark free blocks
#define FAT_EOF -2  // End of a chain (EOF)
#define FAT_RESERVED -3      // Reserved for metadata
```

Our implementation of free space tracking makes use of a File Allocation table which is created globally for use system-wide through the process of declaring it as an external variable. It is an integer pointer and its first two values are set and reserved for the VCB and the FAT itself. Free space will be marked with the value of -1 for now until files are implemented and anything that is occupied will be set with -3 and the end of the file chain will be represented with -2. This is managed through the use of helper functions that keep track of the FAT values as well as help allocate memory for directories.

Directory Entry Structure

```
typedef struct dir_Entry
{
   time_t creation_date; // Saves the creation timestamp, 8 bytes
   time_t last_edited; // When edited saves the timestamp, 8 bytes
   time_t last_accessed; // When accessed saves the timestamp, 8 bytes
   unsigned int blockPos; // Holds index of the entry in blocks, 4 bytes
   unsigned int size; // Stores the size in bytes of the file, 4 bytes
   int is_Directory; // Identifies a directory or file, 4 byte
   char name[60]; // Name of the file or directory being created, 60 bytes
} dir_Entry;
```

Our implementation of a Directory Entry works with the use of three timestamp values in order to provide users details on the creation, last edit made, and the last time a file or directory has been accessed. A directory entry also contains a block index for its respective position in our file system along with its size. In order to differentiate between a file and a directory an integer value was created that will be checked based on its set value, if 0 then it is a file, otherwise it is a directory. Lastly, in order to identify directory entries each has a name that can be set upon creation with a maximum length of 60 characters.

Approach: Milestone 1

In order to begin implementing a working File System we plan on starting by creating global instances of the most important portions of our file system, the Volume Control Block, the File Allocation Table (FAT) for free space tracking, and our root directory. This will ensure easier workflow between files as well as cut down on dependencies for local variables as this can lead to unfortunate mistakes that can make or break our file system. We intended to do this by splitting this phase of the file system into 5 steps. The first is the formatting of the Volume by detecting the presence of the Volume Control Block in our sample volume. The second step will be if there is no volume control block present then we will create an instance of it and initialize it before writing it to disk. The third step will be the creation and initialization of our FAT which will occur if no FAT is present on disk when loaded. The fourth step will be the implementation of free space management through the use of the FAT in order to implement the creation of directories, this will be in the form of an allocateBlocks function which will be used for directory creation in order to allocate the memory needed for the creation of a directory based on the number of entries present in it. Our fifth and final step will be the implementation of directory creation which we intend to support the use of both root and nonroot directories depending on the parent passed into the function. Once this is created we will then test our file system through the use of a hexdump function in order to ensure the creation of our VCB, FAT, and Root.

Volume Formatting & Initialization

To implement the Volume Control Block into our File System we intend to allocate the memory needed for it and upon confirmation of successful allocation we will then read a block from disk if a VCB instance is already present we will then load our FAT and Root from disk as well. This will be done through the use of our volume signature which is set to the first 18 digits of Pi. If the signature is not detected we will then begin to initialize the instance of the VCB we have created. To properly initialize all the values of the VCB we will create our FAT and Root before initialization as this is dependent on the location of our free space, the size of it, and the location of our root directory. Once this is completed we plan to then finally initialize all the values of the VCB and then write the VCB, FAT, and Root to disk.

FAT Creation/ Free Space Initialization - Yash

The File Allocation Table (FAT) is represented as an array of integers. Each index corresponds to a block on the disk. All the blocks are set to `-1` to indicate that they are free. When the file system is initialized, the FAT is allocated to a size that is equal to the total number of blocks with the first block being reserved for the Volume Control Block (VCB). While blocks are being allocated, the `allocateBlock` function iterates through the FAT to find a free block mark it with a special value, and then decrement the count of free blocks in the VCB. For single block allocation, that value is `-2`. For multi-block allocation, the `allocateBlocks` function links consecutive blocks in the FAT and establishes a chain that signifies the blocks being used for a particular file. The last block is marked `-2` to indicate the end of the allocated space.

FAT Allocate Blocks - Marco

To implement this block allocation function for the FAT, it will first initialize a function to find and allocate a single free block from the FAT. This function will iterate over a buffer containing the FAT passed in by the user along with a start index, and a buffer size. While it is iterating over the FAT it will look for an entry marked as free. With a free entry found, it will return the index of this free block, or -1 if no free block is found. Next, I will implement the allocateBlocks function to allocate multiple blocks for a file. The parameters passed into this function will be the amount of total blocks needed to allocate and a minimum number of contiguous blocks needed. The function will then calculate the buffer size needed to hold FAT entries, based on the FAT size taken from the volume control multiplied by the block size. It will then allocate a buffer in memory to hold the FAT. The fat is loaded from disk into the buffer using the FAT's read function. The function will then loop through the buffer and locate a chain of contiguous free blocks. By iterating over the FAT buffer, it will find a starting index where a free block, save it, and then iterate again starting at the saved index to see if there are enough free blocks contiguous. If so this index will then be used as the starting position for block allocation.With the starting index of the contiguous blocks, the function will enter a loop to allocate each required block. It will link each allocated block in the FAT by setting the numbers of that index to the following block. If there are more blocks needed after the contiguous blocks are allocated it will call the allocate blocks function to find the next set of discontiguous blocks. Once all the blocks have been written the end block in the chain will be set to EOF (end of file). The function will then return the index of the first block in the chain.

Directory Entry Creation and Root Directory Initialization

To implement a function for initializing and writing a root directory in a file system, we start by defining the root directory to point to itself and pass NULL as the parent directory. The function needs to track the parent directory and the number of entries. First, we calculate the bytes needed by multiplying the number of directory entries by the size of each entry (size of the directory entry structure), which will tell us the bytes required for all entries. Then, we calculate the number of blocks needed to store this data on disk using the formula (bytes_needed + (block_size - 1)) / block_size. This calculation tells us how many blocks we're going to use. To prevent wasted space we have to recalculate the total number of bytes we are going to use and divide that number by how big the directory entries are to determine if we can fit in more directory entries within the blocks allocated to us. With these values, memory can be allocated for the directory entries using malloc. A helper function, allocate_blocks, finds the block positions on the disk which tells us the starting location for the root directory. The entries are then initialized, starting with the root entry labeled. and set it with its location as the parent if no parent is provided. We label the second entry as .., and, if the parent is null, assign its location to the root entry. Each entry is initialized with metadata such as creation_date, last_edited, last_accessed, and a directory flag. Finally, the directory data is written to disk using LABwrite, passing the allocated buffer to complete the disk write process.

Approach: Milestone 2

fs_setcwd - Prashrit

To implement the fs_setcwd function, I used a straightforward approach to set the current directory by leveraging key building blocks, such as directory checking and path validation. The function ensures the user-provided pathname is valid and updates the global variable tracking the current working directory. To start, I use input validation, where the function begins by checking the pathname parameter. If it is NULL or an empty string, the function returns -1 and prints an error message. This is done to ensure we are not working with invalid or empty inputs, which could lead to undefined behavior or potential crashes in later operations. The next step is a directory check. Here, I use the fs_isDir function to verify that the given pathname corresponds to a valid directory. If the provided pathname does not point to a directory, we return -1 and output an error message. This check is critical to ensure the current working directory is always set to a valid directory, as many file system operations rely on this being correct. Setting the current working directory to a non-directory path would break these operations. Following this, I perform a path length validation to ensure that the pathname does not exceed the maximum allowed path length. If the length of the path exceeds the limit, the function returns -1 and logs an error message. This step is necessary to prevent buffer overflows and to maintain compatibility with the file system constraints. It ensures that the path we work with fits within the defined size and avoids unexpected issues. Once all of these checks are complete, the function moves to update the current working directory. Here, I use strncpy to copy the pathname into the global currentWorkingDir variable, ensuring it is safely null-terminated. This step finalizes the operation by storing the new directory path as the current working directory, making it accessible for subsequent file system operations.

Fs_getcwd - Marco

Since we have a global directory entry variable that tracks the current working directory, this function will simply use the C function strncpy to copy the name from the global current working directory variable into the pathname buffer passed in through the user. The length of the buffer will be specified by the size variable passed in by the user. The function will then return the pathname buffer with its new name initialized.

fs_isFile - Marco

In our directory entry struct we decided to have an int variable called is_Directory, when this variable is set to 1 it is a directory and 0 a file. I will utilize this variable in the directory entry struct to complete this function. The function will first check to see if the parameter passed in is valid by checking if it points to null or the string length is 0, if it is any one of these then the function will return -1. I will then create variables for the directory entry, index and last element name. These will then be passed by reference into parsepath. If parse path cannot find the directory specified the function will return -1. Once parse path finds the correct directory the function will look into the parent directory at the index it found. It will check to see the value at the is_Directory variable, if it is 1, the function will return 0 and 1 if otherwise.

fs_isDir - Marco

In our directory entry struct we decided to have an int variable called is_Directory, when this variable is set to 1 it is a directory and 0 a file. I will utilize this variable in the directory entry struct to complete this function. The function will first check to see if the parameter passed in is valid by checking if it points to null or the string length is 0, if it is either one of these then the function will return -1. I will then create variables for the directory entry, index and last element name. These will then be passed by reference into parsepath. If parse path cannot find the directory specified the function will return -1. Once parsepath finds the correct parent directory the function will look into the parent directory at the index it found and will return the value in the is_Directory variable.

Fs_mkdir - Marco

To implement the make directory function I plan to use the functions I have previously created as building blocks for this function. I plan to use parse path to check if a directory at the specified location is already present and to get the parent directory of the specified location. The second function I will use is create directory to create the new directory with the parent given by parse path. To start off the function I will first check if the path parameter passed in is valid. If the pathname is null it will return -1 and print to the console an error message. Th following variables will be created: directory entry to hold the parent information, index to hold the index of the location within the parent directory, last element to hold the new name of the directory we want to create and a integer to check the return value of parsepath.
With the parent found, It will then check if the found entry at the given index is a file or not. If it is not then the function will iterate through the parent to see where there is an empty directory entry. It will do this by iterating through the parent and checking the names, if the name of the file is “not used” then the function will assign the index to this directory entry. It will then call create directory to create the directory entry.
After the directory entry is created and written to disk by the function createDir, it will then change the name of the directory entry to the name of last element, change the is_Directory variable in the struct to mark it is a directory now and the last edited date the the creation date of the new directory. It will then return 0 for success and -1 for failure.

Parse Path - Marco

To implement the path parsing function I first will start by verifying the path parameter passed in. If the path is null or has a length of zero, the function will return an error code of -1.I will then initialize the following variables: parent to maintain information about the parent directory, index to keep track of the index inside of its parent, and lastElement to store the name of the directory we are looking for. To check if the path the user is looking for is absolute or relative the function will checking the first character of the path parameter. If the first value in the path parameter is a “/” it is an absolute path and will start with the global root directory. If the first value in the path parameter is not a “/”, it is a relative path and starts with the global cwd variable. Tokens will then be parsed from the path string using the '/' delimiter. These tokens will represent each directory in the path. The function will then iterate through the path using the helper function findInDir. This helper function will determine if the path we are looking for is within the current directory. It will do this by iterating through the parent directory, checking to see if the names match. If a directory entry is found, the index is returned. When the function gets to the final token, this signifies that the parent directory of the directory entry has been found. The function will then update the parameters passed in by reference: returnParent, index, and lastElement variables. If any part of the path is invalid is not found the function will return -1. On success it will return 0.

fs_opendir - Prashrit

To implement the fs_opendir function, I used a step-by-step approach to open a directory and prepare it for reading by allocating and initializing a directory stream structure (fdDir). This function ensures that the provided pathname is valid, corresponds to an actual directory, and has accessible entries to iterate through. The function starts with input validation to check if the pathname parameter is NULL or an empty string. If either condition is true, the function returns NULL and logs an error message. This ensures we are working with valid input data and avoids undefined behavior from invalid paths. The next step is a directory check, where the fs_isDir function is used to verify if the provided pathname corresponds to a valid directory. If the path does not point to a directory, the function logs an error and returns NULL. This is done to ensure that only valid directories can be opened. If a file or invalid path were allowed, it would cause issues when attempting to read entries later. Once the validity of the path is confirmed, the function proceeds to retrieve the directory entries. Here, a helper function like fs_getDirEntries is intended to be used to populate an array of fs_diriteminfo structures and determine the total number of entries in the directory. If the retrieval fails (e.g., entries are NULL or entryCount is 0), the function returns NULL and logs an error message. This ensures that we are not working with empty or inaccessible directories. Without this step, the directory stream would not be meaningful.The next step is to allocate memory for the fdDir structure. If the malloc call fails, the function logs an error and returns NULL. Memory allocation is necessary to store information about the open directory, such as the current position and total entries, enabling iteration through the directory. This step ensures the structure is prepared for use. Finally, the function initializes the fdDir structure by setting currentEntry to 0, totalEntries to the number of entries retrieved, and entries to the array of directory items. Once initialized, the function returns the fdDir pointer. This allows the directory to be iterated by using subsequent calls to functions like fs_readdir.

fs_readdir - Yash/Prashrit

The fs_readdir function is designed to read the contents of a directory and return the names of files or subdirectories within it, which makes it analogous to a shell command like ls. In order to implement this, we first had to understand directory structures and how they are stored in the file system, as well as the metadata they contain. Then, we define the parameters of the function. It typically takes a path as an input. Then we would check if the input path exists and is a valid directory before loading the directory block into memory using an appropriate file system function, such as LBAread.Once the directory block is loaded, we need to iterate through the directory entries stored in the block. Each directory entry contains metadata about a file or subdirectory, such as its name, type (file or directory), size, and potentially timestamps. By examining this metadata, we can filter and format the output to include only the necessary information for each entry. For example, we might check if the name of the entry is valid (not an empty or unused entry) and if it should be displayed to the user.To facilitate this process, we use a structure like fs_diriteminfo to store information about each directory entry that we return. The fs_readdir function then retrieves the next directory entry from the preloaded block each time it is called. This involves maintaining a position tracker, such as currentEntry in the fdDir structure, to ensure the function knows where to resume from during the next call.Error handling is a crucial part of this implementation. If the input directory does not exist or cannot be read (e.g., due to invalid metadata or a corrupted block), the function will log an error and return NULL. Similarly, if the directory block cannot be properly loaded into memory (e.g., LBAread fails or returns invalid data), the function will terminate early and release any allocated resources to avoid memory leaks.The logic of this function ensures that it not only retrieves entries from a directory but does so in a structured and repeatable manner. The fs_readdir function is designed to integrate seamlessly with fs_opendir and fs_closedir, creating a consistent interface for working with directories in the file system. This modular design allows the file system to efficiently handle directory operations, maintain state, and avoid redundant operations like reloading directory blocks repeatedly.Lastly, performance considerations are also taken into account. By loading the entire directory block into memory once (in fs_opendir), we minimize repeated disk reads, which can be costly in terms of time. This allows fs_readdir to focus solely on iterating through the entries in memory, making it faster and more efficient for operations involving large directories.

fs_closedir - Prashrit

To implement the fs_closedir function, I used a simple and direct approach to close an open directory stream by freeing allocated resources associated with the fdDir structure. This function ensures that the memory used to store directory entries and the directory descriptor itself is properly released, preventing memory leaks in the file system. The function starts with input validation to check if the dirp parameter is NULL. If it is invalid, the function logs an error message and returns -1. This validation step ensures the function does not attempt to operate on a null pointer, which could lead to undefined behavior or crashes. If the input is valid, the next step is to free the memory allocated for the directory entries. The fdDir structure contains an array of fs_diriteminfo structures (entries), which was likely populated during the fs_opendir call. Freeing this memory ensures that the resources used to store the directory entries are returned to the system. This step is crucial for efficient memory management, as leaving the entries allocated after the directory is no longer needed would result in memory leaks. After releasing the memory for the entries, the function proceeds to free the fdDir structure itself, which was allocated to represent the open directory stream. This step completes the cleanup by releasing all resources associated with the directory stream, ensuring no memory is wasted. Finally, the function returns 0 to indicate success. This confirms that the directory stream was successfully closed and all associated resources were properly released. By following this approach, the fs_closedir function maintains the reliability and efficiency of the file system while preventing resource mismanagement.

fs_stat - Prashrit

To implement the fs_stat function, I followed a structured approach to gather and return metadata about a file or directory. This function ensures the provided path is valid, resolves the path to the correct file or directory entry, and then populates the fs_stat structure with the relevant details such as size, block size, and number of blocks. The function begins with input validation to check if the path or buf parameters are NULL. If either is invalid, the function returns -1 and logs an error message. This step ensures that the function does not attempt to work with invalid inputs, which could lead to crashes or undefined behavior. The next step involves locating the file or directory specified by the path using a helper function like locateDirectory. If the path cannot be resolved (e.g., the file or directory does not exist), the function logs an error and returns -1. This is done to ensure that the fs_stat function only works with valid and existing files or directories. Without this step, the function could attempt to populate metadata for a nonexistent entry, leading to incorrect results .Once the file or directory is located, the function proceeds to populate the fs_stat structure. The size of the file or directory is assigned to st_size, the block size for file system I/O is set to a constant (e.g., 512 bytes), and the number of blocks allocated is calculated by rounding up the size divided by the block size. These fields provide essential information about the file or directory, allowing other parts of the system to determine how much space it occupies and how it is stored. Finally, the function assigns a timestamp to the st_modtime field if a valid "last modified" field is available in the dir_Entry structure. If no explicit last_modified field exists, a default or equivalent field (like creation date) is used. This ensures that the file or directory's metadata includes information about when it was last changed, which is critical for tracking changes and managing dependencies. The function concludes by returning 0 to indicate success. By following this structured approach, the fs_stat function ensures that it reliably provides accurate and essential metadata for files and directories, enabling other parts of the file system to function correctly.

fs_delete - Yash

The purpose of the fs_delete function is to delete a file from the filesystem. It performs the following steps:

1. First, it validates that the provided file name is not NULL
2. Parses the file path to locate the file and its parent directory
3. Validate that the specified path points to a file
4. Remove the file's entry from its parent directory [not implemented yet]
5. Free disk space allocated to the file [not implemented yet]

The first thing it does is check if the filename parameter is NULL to prevent unnecessary operations or crashes. Then, it uses the parsePath function to decompose the file path into its components (parentDir, dirIndex, and lastElement). This ensures that the function can locate the file and its metadata. After that, the file is to be validated. A helper function fs_isFile would be used to confirm that the provided path corresponds to a file and not to a directory or some other invalid type. Then, another helper function called remove_directory_entry will be used to unlink the file from its parent directory and finish with free_disk_blocks to release the disk blocks occupied by the file.

fs_rmdir - Yash

The fs_rmdir function is intended to remove an empty directory from the filesystem. This is done through the following steps:

1. Validate the input directory path to ensure it is not NULL
2. Parse the directory path to locate the directory and its parent directory
3. Ensure that specified path points to a directory
4. Check if directory is empty [not yet implemented]
5. Remove directory entry from parent directory [not implemented]

First, the function checks to make sure the pathname isn't NULL to prevent unnecessary operations or crashes which could be caused by invalid input. Then, it calls the parsePath function to break down the directory path into its 3 components (parentDir, dirIndex, and lastElement). This allows the function to locate the directory and its metadata. It would then use the helper function fs_isDir to check whether the provided path corresponds to a directory. This is done in order to avoid unintended operations on files or invalid paths. We would then use helper functions is_directory_empty and remove_directory_entry to deal with making sure the directory is empty before finally removing it.

Approach: Milestone 3

b_fcb - Ulices

In order to effectively implement the following functions for file input and output operations I intend to begin with the file control block. To begin, I will add new values to the structure itself so the functions can support flags, our free space tracking through our FAT, and our implementation of a Directory Entry. I intend to do this by including a pointer to a Directory Entry for the file information, the block position in which the file starts, the position it holds in its parent directory(although this is not used for potential future uses), the total size of the file in blocks, and an integer for flag modifiers to support Read, Write, and Read&Write.

```
dir_Entry* fi;          // directory entry info for current file
char * buffer;          //holds the open file buffer
int index;              //holds the current position in the buffer
int bufferUsed;                 //holds how many valid bytes are in the buffer
int blockPosition;      // holds the begining block poisiton
int dirPosition;        // holds position in the parent directory
int totalBlocks;        // holds the total blocks taken up by file
int flags;              // holds flags modifiying file actions
```

b_open - Ulices

For b_open I intend to create a file descriptor after initializing our file system, once sure that the file descriptor is valid I will then use our parsepath function created in Milestone 2 section to gather the Directory Entry index and will then use LBAread to gather the directory entry of the file being opened. If the Directory Entry Returned is valid I will then allocate space within our file control block buffer to support both reading and writing using our defined Block Size. Upon the allocation of the buffer, I will then set the rest of the values in the file control block instance to 0 and using the information present in our Directory Entry set the values for the position of the file using the block position, setting the directory position using the index returned from parsepath by reference, and the total blocks using the file size to calculate the size in blocks using the Block Size. Once this is completed I will then finalize open by setting the file control block flags to the flags input by the user and returning the file descriptor upon successful termination.

b_close - Ulices

Due to the changes made to the file control block, I will add the new struct attributes to close. This will then allow me to follow through and deallocate the file control block attributes starting by freeing the fi directory entry pointer that stores the file information followed by freeing the buffer. Once successfully freed the rest of the attributes will then be set to 0 for future use of the file control block linked to the file descriptor provided. Upon successful termination, I will then return 0 from close to signify that the file is now closed.

b_read - Ulices

For b_read I intend on using logic from our past b_io assignment and begin by limiting the file length to make sure the end of a file is taken into account. From here I will then update the count based on the size of the file and the amount of bytes that have been read if any. I will then check if there are still bytes remaining in our buffer and if they are greater than or equal to the amount requested I will enable the first case of reading to execute later in the function. If the remaining about of bytes is lesser than the amount being requested then the first read case will still be enabled with the second and third cases which will allow for any excess to be handled as well resulting in the calculation of the number of blocks and bytes to copy being performed. In the first case, I will copy from our buffer to that of the users with the first count with the value of the remaining bytes and the original count if the remaining bytes are greater than or equal to the requested count or the total remaining bytes otherwise. I will then update the index of our buffer. The second case will read directly into the user's buffer using the calculated amount of blocks from earlier, this will then be followed by updating the block position by the blocks to copy and the second count used being calculated for future use to return to the user. The third and final case will refill the buffer if needed using a single block and will update the index, the capacity in the buffer, and the current block position, once done I will then update the third count using the total bytes read if lesser than the original third count, if greater than zero the buffer will then be copied to the user one last time. Upon the end of either of the three cases, the bytes copied will then be calculated using each of the three counts and returned to the user.

b_write - Ulices

To implement b_write I will rely on the foundation of our structure and buffer assignment and will begin by creating a character pointer and allocate it using malloc and the defined B_CHUNK_SIZE. Before iterating through the user's buffer I will copy a block of data from their buffer to our temporary buffer and then begin to iterate. I intend to use a while loop as we previously did for our previous assignment. By using a while loop I will be able to monitor the temporary buffer we created and upon successful duplication iteration will begin. Inside the while loop, I will create a count of the total bytes taken up by our buffer and the index of the file buffer combined with the length of our temporary buffer is equal to or exceeds a single block I will copy as much as possible and will then commit the block to memory after finding and allocating a free block. I will then update our buffer to move past the committed contents as well as update our count of bytes copied and reset the length of our temporary buffer and the index of our file buffer. If the temporary buffer does not exceed or meet a block then it will continue to have more data copied into our file buffer from the temporary while incrementing the index with its new length and then copying more from the users into our temporary. This will allow for as many blocks of data to be committed to memory as needed while allowing for smaller portions to be supported until the buffer is full. Upon reaching a NULL status in our temporary buffer we will finally check if there is anything left in our file buffer and if so we will then allocate one final block and write the buffer to memory. I will then free the temporary buffer created and that of the file control block before returning the total bytes copied back to the user upon successful termination.

Issues and Resolutions:

Milestone 1

Although this milestone of our File System Project is straightforward one of the biggest issues we faced was the division of the work as the steps themselves are very much intertwined. For instance the VCB creation can be partially done but is dependent on the FAT and Root being created. The same occurred with the creation of the Root as it relies on a helper function that manages the FAT in order to allocate a block for the directory being created.

This was solved mostly through consistent communication as a group since in order for us to ensure the code we wrote worked we had to update each other based on our completion of our given task. This primarily was the case for the creation of directories and the FAT as they were created alongside one another. This was mediated by having both comment out dependencies until they were fully implemented and then uncommented them and debugged in order to mount them.

Another issue we encountered came about when we moved code regarding the FAT allocation. We decided to move code from our fsInit.c file to fsFAT.c file to keep coherence in our code. Doing so we encountered an error telling us that the function that we were calling fsInit.c to initialize the FAT was not able to be found.

To solve this we looked at how the make file was compiling our program. We then realized it was not compiling the new fsFAT.c to the fsFAT.o file. We then edited the makefile to include this new file along with fsDirEnt., which solved our problem.

Lastly one of the biggest issues we ran into was the creation of the FAT. Due to our misunderstanding of how it would be created we created a node type struct to use for a linked list however this not only failed but led to memory allocation errors resulting in segmentation faults.

After reconfiguring our work and removing the FAT it was reimplemented through the use of an int* which was then supported by helper functions which worked and was able to support the creation of directories from then on as well.

We ran into a few issues with initializing the FAT correctly. The FAT has to be initialized as a linked list in order for it to work with a filesystem. The initial issues were related to malloc and improper memory allocation. The way it works is that the first few blocks are reserved for the VCB. After that, a couple blocks are initialized for the FAT, and the rest of the blocks are initialized as free space for the filesystem. We experienced a few segmentation faults when trying to allocate the different parts of the memory. These were resolved by simply using printf statements to debug and see where the issue was happening, and then fix it by using bounds to check when accessing FAT entries to prevent overflows.

Once the FAT was correctly initialized, we ran into another error that we worked a long time on to resolve. This was related to the free space. We wanted the FAT to be initialized as a linked list and then link to the free space with each block pointing to the next block, each ending with an EOF marker. The FAT didn't accurately reflect which blocks are free, which led to allocation failures and overwriting of data. We originally had free free blocks marked as 0, reserved blocks as 1, and EOF as -1. The reserved blocks would correctly be marked as reserved, but then everything else for the FAT and free space would be marked as free without the EOF markers. We were able to somewhat fix it, the FAT would have the EOF markers before going into free space. We then modified our approach a little bit and defined the markers before the

function definitions and that helped resolve many of the issues. However, the EOF markers were still not correctly being written. We suspect the issue lies with LBAwrite. For some reason, it is not writing to all the blocks. When debugging, we discovered that it expected 19531 blocks but only wrote to 410, and then with more debugging we found it's not even writing to all 410 of those blocks, only to 407. Unfortunately we were unable to resolve that issue before submission, but the rest of the FAT still works.

Milestone 2

Yash

The main issue with fs_delete is that the helper functions were not able to be implemented in time. The portion of the function where these would have been used are commented out, but the work is still there. We also did not have time to extensively test the function. The resolution to this would have been to implement the functions in time and test edge cases as well.

The same goes for fs_rmdir as well. The helper functions is_directory_empty and remove_directory_entry were not able to be implemented in time, and we did not have the time to test it with edge cases. The way to resolve this would have been to better manage time and implement the necessary functions.

Prashrit

The implementation of readdir gave me a lot of issues as well. While trying to implement the function we used an incorrect return type. I was using **const char *dirPtah** instead of **fdDIr *dirp.** Another issue being that I ran an integer instead of a pointer to **fs_disriteminfo.** There was memory allocation in the wrong function, and there was confusion between the two functions **fs_opendir** and **fs_readdir**.

I did not have many issues with fs_stat although I wasn't using all of the implantations that we had on our struct into my function. The issue in the code:

```
// Populate the fs_stat structure
 buf->type = entry->is_Directory ? FS_TYPE_DIR : FS_TYPE_FILE;
 buf->size = entry->size; return 0; // Success }
```

My fix:

```
buf->st_size = entry->size;             // File or directory size in bytes
   buf->st_blksize = 512;                // Assuming a fixed block size
   buf->st_blocks = (entry->size + 511) / 512; // Number of blocks (round up)
   buf->st_accesstime = entry->last_accessed; // Last accessed time
   buf->st_createtime = entry->creation_date; // Creation time
   return 0; // Success
}
```

We can see that before I was not populating the fs_stat structure properly and all of the implantations that was in my .h file was not used.

I did not have that many issues when it came to functions such as fs_opendir and fs_closedir because they were pretty straight forward. The main fucntion that was very hard to grasp was fs_readdir.

Marco

Milestone 3

Ulices

Due to issues some work being neglected especially when it came to overhauling previous sections of the code many were left non-functional making it difficult for most if not all of the file input and output to be implemented. Implementing b_read and b_write was the extremely straightforward using the foundations we covered in the solutions for our second and fifth assignments. This was especially true for b_open and b_close as they were extremely straightforward and mostly relied on the editing of the fcb struct. However due to the overhauling of the fat being neglected and a large portion of the functions of Milestone to needing to be reworked by myself and my fellow groupmates and led to little to no time to implement b_seek. This also bled into the handling of file operation flags as currently the file input and output only supports Reading Writing and reading & writing.

Besides this the other major issue that was caused because of the time constraints placed upon our group due to issues with coordination during Milestone 2 was the fact that the move support was not implemented either and was left out of the final project.

Milestone 3 was especially difficult for myself as a majority of my time was spent debugging code for Milestone 2 as a lot of it was marked as finished and working even though it had never been tested nor implemented properly.

When attempting to debug issues throughout the file system concurrently with implementing file operations I found many issues with the fat that we're still present since they have been ignored specifically the fact that helpers in order to update the fat we're not implemented nor planned, so when attempting to create a file let alone read and write from it it became nearly impossible since this was only found shortly before the due date.

Following on this it also translated into new issues being present in the initialization of the file system as some of our code was edited without the entire group's knowledge leading to loading the file system now causing a segment fault upon loading.

Hexdump Analysis:

Volume Control Block

```
000200: FB A0 9E F6 2F 1E 5C 04  4B 4C 00 00 00 00 00 00 | ....../.\.KL......
000210: 00 02 00 00 00 00 00 00  4A 4C 00 00 04 00 00 00 | ........JL......
000220: 04 00 00 00 00 00 00 00  00 00 00 00 54 68 65 20 | ............The
000230: 47 75 6E 6E 65 72 73 00  00 00 00 00 00 00 00 00 | Gunners.........
000240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x200-0x207 represent the VCB block signature, highlighted in Yellow. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x045C1E2FF69EA0FB which represents: 314159265358979323 in decimal, this is our VCB signature.

Bytes address 0x208-0x20F represent the total number of blocks, highlighted in Orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000000004C4B which represents: 19531 in decimal. Meaning we have 19,531 blocks in our volume.

Bytes address 0x210-0x217 represent the size of each block, highlighted in Blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000000000200 which represents: 512 in decimal, the number of bytes in one block.

Bytes address 0x218-0x21B represent the count of free block, highlighted in Purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00004C4A which as an unsigned int represents: 19530 in decimal, the total number of free blocks in our volume.

Bytes address 0x21C-0x21F represent the free block start, highlighted in Pink. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000004 which as an unsigned int represents: 4 in decimal, index of the first free block in the volume.
Bytes address 0x220-0x223 represent the size of the file allocation table (FAT), highlighted in Red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000004 which as an unsigned int represents: 4 in decimal,the amount of entries in the FAT
Bytes address 0x224-0x227 represent the index of the fat, highlighted in Green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002 which as an unsigned int represents: 0 in decimal, index of the fat due to issues it is not updated.
Bytes address 0x228-0x22B represent the index of the root directory, highlighted in Neon Green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002 which as an unsigned int represents: 0 in decimal, index of the root directory due to issues it is not updated.
Bytes address 0x22C-0x23F represent the name of the volume, highlighted in Dark Blue. Since this data is stored linearly we can read it from left to right. We get: 0x5468652047756E6E6572730000000000000000 which when converted to ASCII we get: “The Gunners”.

File Allocation Table - Yash

```
000400: FD FF FF FF FD FF FF FF  FD FF FF FF FD FF FF FF | ????????????????
000410: FE FF FF FF 06 00 00 00  07 00 00 00 08 00 00 00 | ????............
000420: 09 00 00 00 0A 00 00 00  0B 00 00 00 0C 00 00 00 | ................
000430: 0D 00 00 00 0E 00 00 00  0F 00 00 00 10 00 00 00 | ................
000440: 11 00 00 00 12 00 00 00  13 00 00 00 14 00 00 00 | ................
000450: 15 00 00 00 16 00 00 00  17 00 00 00 18 00 00 00 | ................
000460: 19 00 00 00 1A 00 00 00  1B 00 00 00 1C 00 00 00 | ................
000470: 1D 00 00 00 1E 00 00 00  1F 00 00 00 20 00 00 00 | ............ ...
000480: 21 00 00 00 22 00 00 00  23 00 00 00 24 00 00 00 | !..."...#...$...
000490: 25 00 00 00 26 00 00 00  27 00 00 00 28 00 00 00 | %...&...'...(...
0004A0: 29 00 00 00 2A 00 00 00  2B 00 00 00 2C 00 00 00 | )...*...+...,...
0004B0: 2D 00 00 00 2E 00 00 00  2F 00 00 00 30 00 00 00 | -......./...0...
0004C0: 31 00 00 00 32 00 00 00  33 00 00 00 34 00 00 00 | 1...2...3...4...
0004D0: 35 00 00 00 36 00 00 00  37 00 00 00 38 00 00 00 | 5...6...7...8...
0004E0: 39 00 00 00 3A 00 00 00  3B 00 00 00 3C 00 00 00 | 9...:...;...<...
0004F0: 3D 00 00 00 3E 00 00 00  3F 00 00 00 40 00 00 00 | =...>...?...@...

000500: 41 00 00 00 42 00 00 00  43 00 00 00 44 00 00 00 | A...B...C...D...
000510: 45 00 00 00 46 00 00 00  47 00 00 00 48 00 00 00 | E...F...G...H...
000520: 49 00 00 00 4A 00 00 00  4B 00 00 00 4C 00 00 00 | I...J...K...L...
000530: 4D 00 00 00 4E 00 00 00  4F 00 00 00 50 00 00 00 | M...N...O...P...
000540: 51 00 00 00 52 00 00 00  53 00 00 00 54 00 00 00 | Q...R...S...T...
000550: 55 00 00 00 56 00 00 00  57 00 00 00 58 00 00 00 | U...V...W...X...
000560: 59 00 00 00 5A 00 00 00  5B 00 00 00 5C 00 00 00 | Y...Z...[...\...
000570: 5D 00 00 00 5E 00 00 00  5F 00 00 00 60 00 00 00 | ]...^..._...`...
000580: 61 00 00 00 62 00 00 00  63 00 00 00 64 00 00 00 | a...b...c...d...
000590: 65 00 00 00 66 00 00 00  67 00 00 00 68 00 00 00 | e...f...g...h...
0005A0: 69 00 00 00 6A 00 00 00  6B 00 00 00 6C 00 00 00 | i...j...k...l...
0005B0: 6D 00 00 00 6E 00 00 00  6F 00 00 00 70 00 00 00 | m...n...o...p...
0005C0: 71 00 00 00 72 00 00 00  73 00 00 00 74 00 00 00 | q...r...s...t...
0005D0: 75 00 00 00 76 00 00 00  77 00 00 00 78 00 00 00 | u...v...w...x...
0005E0: 79 00 00 00 7A 00 00 00  7B 00 00 00 7C 00 00 00 | y...z...{...|...
0005F0: 7D 00 00 00 7E 00 00 00  7F 00 00 00 80 00 00 00 | }...~.......?...
```

Byte addresses 0x400 - 0x403 highlighted in light green. Since it is in little endian format, when translated in hex it is 0xFFFFFFFD which equal to -3, telling us that the first block is used. This is used by the VCB.

Byte addresses 0x408 - 0x413 in hex highlighted in dark green. When converted form little endian format it has the values of: 0xFFFFFFFD, 0xFFFFFFFD, 0xFFFFFFFD and 0xFFFFFFFE. 0xFFFFFFFD which equal to -3, telling us that the first block is used. The last int, 0xFFFFFFFE is equal to -2 indicating that it is the end of file for that chain.

Byte addresses 0x410 - 0x5FF are the FAT table chain links. You can see the block points to the the block that follows and increments up

We are missing the EOF marker at the end that should be at 0x5FC - 0x5FF.

Root Directory and Directory Entries - Prashrit

```
000600: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000610: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 06 00 00 | .bNg............
000620: 01 00 00 00 2E 00 00 00  00 00 00 00 00 00 00 00 | ................
000630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x600-0x607 represent the name creation date of the this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x608-0x60F represent the name last edited date of the this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x610-0x617 represent the name last accessed date of the this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to a int to get the raw data we are unable to translate the hex.

Bytes address 0x618-0x61B represent the name last edited date of the this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002, when translated to decimal: 2, which means it is in the second block.

Bytes address 0x61B-0x61F represent the block position of the this particular directory entry, highlighted in purple Since this data is stored in little-Endian format we need to read the bytes

from right to left. We get: 0x000003C0, when translated to decimal: 920, meaning the directory entry is 920 bytes big.

Bytes address 0x620-0x623 represent the size of the this particular directory entry, highlighted in red Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000001, when translated to decimal: 1, meaning the directory entry is a directory.

Bytes address 0x624-0x65F represent the name of the this particular directory entry, highlighted in yellow. The hex value of this data is 0x2E2E00000000...., when translated to ASCII: "..', meaning the .. of the root directory.

```
--------------------------------------------------------------------------------
000660: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000670: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 06 00 00 | .bNg............
000680: 01 00 00 00 2E 2E 00 00  00 00 00 00 00 00 00 00 | ................
000690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x660-0x667 represent the name creation date of this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000674E621C, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x668-0x66F represent the name last edited date of this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000674E621C, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x670-0x677 represent the name last accessed date of this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the

bytes from right to left. We get: 0x00000000674E621C, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x678-0x67B represent the block position date of this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000002, when translated to decimal: 2, which means it is in the second block.

Bytes address 0x67C-0x67F represent the size of this particular directory entry, highlighted in purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000600, when translated to decimal: 1536, meaning the directory entry is 920 bytes big.

Bytes address 0x680-0x683 represent the type of this particular directory entry, highlighted in red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000001, when translated to decimal: 1, meaning the directory entry is a directory.

Bytes address 0x684-0x6BF represent the name of this particular directory entry, highlighted in yellow. The hex value of this data is 0x2E2E0000000000...., when translated to ASCII is "..", meaning the second entry of the root directory.

```
--------------------------------------------------------------------------------
0006C0: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
0006D0: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 00 00 00 | .bNg............
0006E0: 01 00 00 00 64 69 72 00  00 00 00 00 00 00 00 00 | ....dir.........
0006F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Bytes address 0x6C0-0x6C7 represent the name creation date of this particular directory entry, highlighted in light blue. Since this data is stored in little-Endian format we need to read the

bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6C8-0x6CF represent the name last edited date of this particular directory entry, highlighted in dark blue. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6D0-0x6D7 represent the name last accessed date of this particular directory entry, highlighted in green. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x0000000067244FB3, since this data is not able to translate to an int to get the raw data we are unable to translate the hex.

Bytes address 0x6D8-0x6DB represent the block position date of this particular directory entry, highlighted in light orange. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0xFFFFFFFF, when translated to decimal: -1.

Bytes address 0x6DC-0x6F represent the size of this particular directory entry, highlighted in purple. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000, when translated to decimal: 0.

Bytes address 0x6E0-0x6E3 represent the type of this particular directory entry, highlighted in red. Since this data is stored in little-Endian format we need to read the bytes from right to left. We get: 0x00000000, when translated to decimal: 1, meaning the directory entry is a file.

Bytes address 0x6E4-0x71F represent the name of this particular directory entry, highlighted in yellow. The hex value of this data is 0x6E756C6C46696C65000000...., when translated to ASCII: "nullFile", giving us the name of this directory entry.

Eight of these directory entries are created because even though we specified 6 entries, we had to allocated blocks to write them on disk of size 512. To use up all the space allocated in the blocks for the disk we were able to create two more, totaling it to eight directory entries.

```
--------------------------------------------------------------------------------
000720: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000730: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 00 00 00 | .bNg............
000740: 00 00 00 00 66 69 6C 65  00 00 00 00 00 00 00 00 | ....file........
000750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Communication and Coordination:

To try and coordinate with each other the best we can we have been making use and relying on our discord channel where we communicate frequently throughout a given week to update each other on progress as well as check in if our work is dependent on someone else. However, due to contrasting work and academic schedules, it has been difficult to meet both in person as well as virtually. We created a when2meet to see our overlapping schedules and find time when we can all join a discord call to work on a given task or follow up on a topic that was mentioned in our chat. However over the course of the project this quickly became more difficult due to the lack of communication present making it difficult to adjust previous mistakes in our code but also making sure that tasks were being completed on time. This then led to some instances where gaps in communication developing leading to instances where work had to be redivided especially towards the due date.

Division of Work:

Milestone 1

Ulices Gonzalez	Check if volume is present or not(if so initialize it) VCB Hexdump
Marco Robles	Create and initialize root directory, Hexdump revision
Yash Pachori	Initialize the FAT (free space tracking)
Prashrit Magar	Analyzed hexdump
Everyone	Debugging and Write Up

Milestones 2 and 3

Ulices Gonzalez	b_open, b_close, b_read, b_write, b_seek, makefile, and shell linking
Marco Robles	fs_getcwd, fs_isDir, fs_isFile, fs_mkdir, and parsepath, help fix allocate blocks function
Yash Pachori	FAT Overhaul, fs_readdir, fs_rmdir, and fs_delete
Prashrit Magar	fs_stat, fs_opendir, fs_closedir, andfs_setcwd
Everyone	Debugging and Write Up

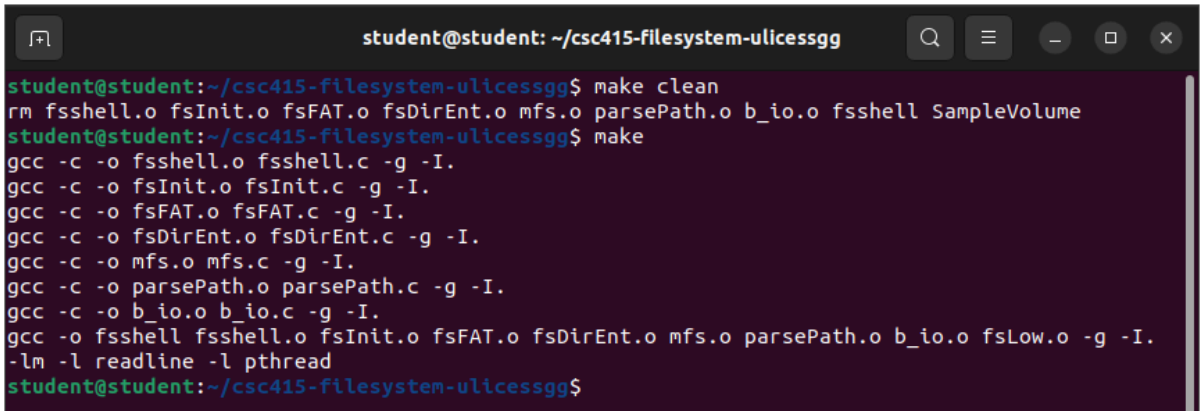
System Hexdump:

```
000200: FB A0 9E F6 2F 1E 5C 04  4B 4C 00 00 00 00 00 00 | ****/.\.KL......
000210: 00 02 00 00 00 00 00 00  4A 4C 00 00 04 00 00 00 | ........JL......
000220: 04 00 00 00 00 00 00 00  00 00 00 00 54 68 65 20 | ............The
000230: 47 75 6E 6E 65 72 73 00  00 00 00 00 00 00 00 00 | Gunners.........
000240: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000250: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000260: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000270: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0002F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000300: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000310: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000320: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000330: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000340: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000350: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000360: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000370: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000380: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000390: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0003F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000400: FD FF FF FF FD FF FF FF  FD FF FF FF FD FF FF FF | ****************
000410: FE FF FF FF 06 00 00 00  07 00 00 00 08 00 00 00 | ****............
000420: 09 00 00 00 0A 00 00 00  0B 00 00 00 0C 00 00 00 | ................
```

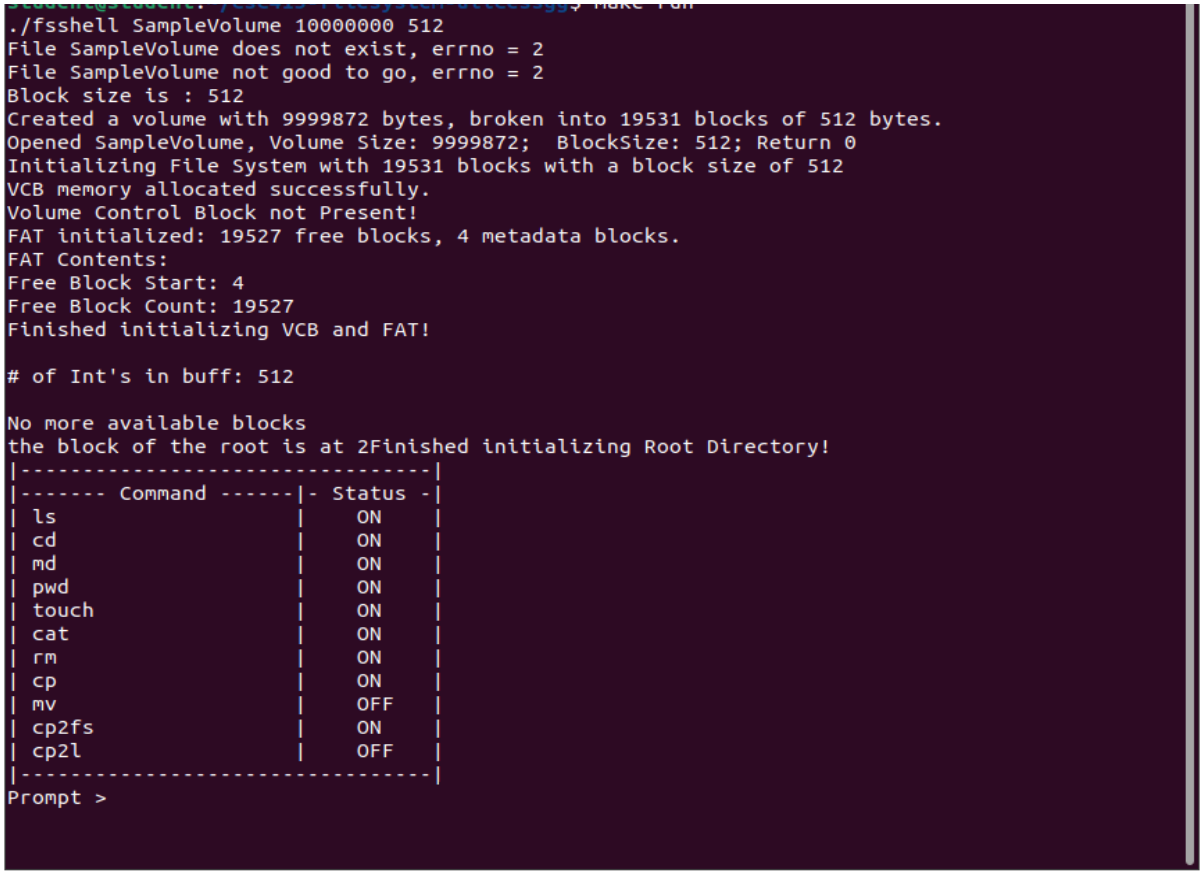
```
000430: 0D 00 00 00 0E 00 00 00  0F 00 00 00 10 00 00 00 | ................
000440: 11 00 00 00 12 00 00 00  13 00 00 00 14 00 00 00 | ................
000450: 15 00 00 00 16 00 00 00  17 00 00 00 18 00 00 00 | ................
000460: 19 00 00 00 1A 00 00 00  1B 00 00 00 1C 00 00 00 | ................
000470: 1D 00 00 00 1E 00 00 00  1F 00 00 00 20 00 00 00 | ............ ...
000480: 21 00 00 00 22 00 00 00  23 00 00 00 24 00 00 00 | !..."...#...$...
000490: 25 00 00 00 26 00 00 00  27 00 00 00 28 00 00 00 | %...&...'...(...
0004A0: 29 00 00 00 2A 00 00 00  2B 00 00 00 2C 00 00 00 | )...*...+...,...
0004B0: 2D 00 00 00 2E 00 00 00  2F 00 00 00 30 00 00 00 | -......./...0...
0004C0: 31 00 00 00 32 00 00 00  33 00 00 00 34 00 00 00 | 1...2...3...4...
0004D0: 35 00 00 00 36 00 00 00  37 00 00 00 38 00 00 00 | 5...6...7...8...
0004E0: 39 00 00 00 3A 00 00 00  3B 00 00 00 3C 00 00 00 | 9...:...;...<...
0004F0: 3D 00 00 00 3E 00 00 00  3F 00 00 00 40 00 00 00 | =...>...?...@...

000500: 41 00 00 00 42 00 00 00  43 00 00 00 44 00 00 00 | A...B...C...D...
000510: 45 00 00 00 46 00 00 00  47 00 00 00 48 00 00 00 | E...F...G...H...
000520: 49 00 00 00 4A 00 00 00  4B 00 00 00 4C 00 00 00 | I...J...K...L...
000530: 4D 00 00 00 4E 00 00 00  4F 00 00 00 50 00 00 00 | M...N...O...P...
000540: 51 00 00 00 52 00 00 00  53 00 00 00 54 00 00 00 | Q...R...S...T...
000550: 55 00 00 00 56 00 00 00  57 00 00 00 58 00 00 00 | U...V...W...X...
000560: 59 00 00 00 5A 00 00 00  5B 00 00 00 5C 00 00 00 | Y...Z...[...\...
000570: 5D 00 00 00 5E 00 00 00  5F 00 00 00 60 00 00 00 | ]...^..._...`...
000580: 61 00 00 00 62 00 00 00  63 00 00 00 64 00 00 00 | a...b...c...d...
000590: 65 00 00 00 66 00 00 00  67 00 00 00 68 00 00 00 | e...f...g...h...
0005A0: 69 00 00 00 6A 00 00 00  6B 00 00 00 6C 00 00 00 | i...j...k...l...
0005B0: 6D 00 00 00 6E 00 00 00  6F 00 00 00 70 00 00 00 | m...n...o...p...
0005C0: 71 00 00 00 72 00 00 00  73 00 00 00 74 00 00 00 | q...r...s...t...
0005D0: 75 00 00 00 76 00 00 00  77 00 00 00 78 00 00 00 | u...v...w...x...
0005E0: 79 00 00 00 7A 00 00 00  7B 00 00 00 7C 00 00 00 | y...z...{...|...
0005F0: 7D 00 00 00 7E 00 00 00  7F 00 00 00 80 00 00 00 | }...~.......◆...

000600: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000610: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 06 00 00 | .bNg............
000620: 01 00 00 00 2E 00 00 00  00 00 00 00 00 00 00 00 | ................
000630: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000640: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000650: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000660: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000670: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 06 00 00 | .bNg............
```

```
000680: 01 00 00 00 2E 2E 00 00  00 00 00 00 00 00 00 00 | ................
000690: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0006C0: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
0006D0: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 00 00 00 | .bNg............
0006E0: 01 00 00 00 64 69 72 00  00 00 00 00 00 00 00 00 | ....dir.........
0006F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

000700: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000710: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000720: 1C 62 4E 67 00 00 00 00  1C 62 4E 67 00 00 00 00 | .bNg.....bNg....
000730: 1C 62 4E 67 00 00 00 00  02 00 00 00 00 00 00 00 | .bNg............
000740: 00 00 00 00 66 69 6C 65  00 00 00 00 00 00 00 00 | ....file........
000750: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000760: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000770: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000780: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
000790: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0007F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
```

Screenshot of compilation:



Screenshot(s) of execution:



```
student@student:~/csc415-filesystem-ulicessgg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > ls


----here
cwd
Parent name: .
token1: c
token2: c
 enter loop
Error: Failed to read directory entries
----here2hiPrompt > exit
System exiting
Freed vcb
Freed FAT
Freed Root
Freed cwd
student@student:~/csc415-filesystem-ulicessgg$
```

```
student@student:~/csc415-filesystem-ulicessgg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > cd ..


----here
cwd
Parent name: .
token1: ..
token2: ..
 enter loop

working

 ., 1, ..
still working2
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicessgg$
```

```
student@student:~/csc415-filesystem-ulicesgg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > md newDir

cwd
Parent name: .
token1: newDir
token2: newDir
 enter loop
----here2
# of Int's in buff: 512
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicesgg$
```

```
student@student:~/csc415-filesystem-ulicesggg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > pwd
c
Prompt > exit
System exiting
Freed vcb
Freed FAT
Freed Root
Freed cwd
student@student:~/csc415-filesystem-ulicesggg$
```

```
student@student:~/csc415-filesystem-ulicesggg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > touch file.txt

cwd
Parent name: .
token1: file.txt
token2: file.txt
 enter loop
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicesggg$
```

```
student@student:~/csc415-filesystem-ulicesggg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > cat file1.txt

cwd
Parent name: .
token1: file1.txt
token2: file1.txt
 enter loop
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicesggg$
```

```
student@student:~/csc415-filesystem-ulicessgg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > rm file.txt


----here
cwd
Parent name: .
token1: file.txt
token2: file.txt
 enter loop
----here2hi
cwd
Parent name: .
token1: file.txt
token2: file.txt
 enter loop
Error: Failed to parse directory path.
Prompt > exit
System exiting
Freed vcb
Freed FAT
Freed Root
Freed cwd
student@student:~/csc415-filesystem-ulicessgg$
```

```
student@student:~/csc415-filesystem-ulicessgg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > cp file.txt newFile.txt

cwd
Parent name: .
token1: file.txt
token2: file.txt
 enter loop
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicessgg$
```

```
student@student:~/csc415-filesystem-ulicesggg$ 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
VCB memory allocated successfully.
Volume Control Block not Present!
FAT initialized: 19527 free blocks, 4 metadata blocks.
FAT Contents:
Free Block Start: 4
Free Block Count: 19527
Finished initializing VCB and FAT!

# of Int's in buff: 512

No more available blocks
the block of the root is at 2Finished initializing Root Directory!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    OFF   |
| cp2fs                |    ON    |
| cp2l                 |    OFF   |
|---------------------------------|
Prompt > cp2fs file.txt other.txt

cwd
Parent name: .
token1: other.txt
token2: other.txt
 enter loop
make: *** [Makefile:67: run] Segmentation fault (core dumped)
student@student:~/csc415-filesystem-ulicesggg$
```