Debug Thugs

Members:

Michael Thompson - 922707016
Utku Tarhan - 918371654
Eric Ahsue - 922711514
Randy Chen - 922525848

GitHub: **Jasuv**

https://github.com/CSC415-2025-Spring/csc415-filesystem-Jasuv

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

Assignment - File System Milestone One

Description:

In this assignment, we are asked to implement a fully functional file system. This includes formatting a virtual volume, managing free space, handling directory structures, and supporting file operations. This project is done in three phases: volume formatting, directory function implementation, and file operation implementation.

Approach:

File System - Free Space Manager

To manage free disk space in our system, we implement a bitmap-based strategy to track the allocation status of blocks in freeSpace.c. The core idea is to create a char array where each index corresponds to the allocation status of a specific block. A value of zero indicates that a block is free, while a value of one marks it as used. This array is called freeSpaceMap, and its size is dynamic based on the total number of blocks on the disk.

During system initialization, initFreeSpace() initializes the freeSpaceMap using values passed from fsInit.c, specifically the total block count (blockCount) and block size (blockSize). The size of the map is calculated using the formula:

19531 blocks × 1 byte (sizeof(char)) / 512 (block size), which results in roughly 39 blocks.

We then write this freeSpaceMap to the FS_RESERVED space (40 blocks defined in mfs.h) to protect the VCB and free space data from unauthorized writes or corruption. These blocks are cleared (set to 0) during initialization, marking all blocks as free, and we manually mark the reserved blocks as used with a loop until we reach the end of the reserved range. Once initialized, we write the map to disk using LBAWrite().

To support reloads, exits, and general restoration, the loadFreeSpaceMap() function loads the map from disk and reinitializes in-memory structures for use by other parts of the file system.

For dynamic allocation, we use the allocateBlocks() function to find free blocks for general use. This function scans the freeSpaceMap and attempts to locate the requested number of blocks, either contiguous or discontiguous. When it finds enough blocks, it marks them as used in the map and updates the disk. If the full request cannot be satisfied, the function rolls back to the previous state and returns NULL.

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

Pseudocode for Allocation logic:

```
function allocateBlocks(requestedCount){
       foundBlocks = empty list
       availableBlockCount = 0

       // Start searching after the reserved system blocks
       for i from FS_RESERVED_BLOCKS to FreeSpaceMapSize - 1:
             if freeSpaceMap[i] == 0:
                 add i to foundBlocks
                 availableBlockCount += 1
                 // Finish searching when we reach the requested count
                 if availableBlockCount == requestedCount:
                      break;

             if availableBlockCount == requestedCount:
                 for each block in foundBlocks:
                     freeSpaceMap[block] = 1
                 return foundBlocks
             else:
                 return NULL
}
```

Lastly, we implement freeBlocks() to handle block deallocation when files or directories are removed. It takes a list of block numbers and marks each one as free in the freeSpaceMap, after checking for invalid or reserved blocks being fed to the function. Once updated, the map is written back to disk using LBAWrite() to ensure persistence. This helps keep the file system's block usage accurate and prevents wasted space.

File System - Directory Functions

The first directory function to implement will be the mkdir function because our file system initialization will call it to create the root directory. This function takes in two arguments: the desired path (which includes the directory name) and the mode (file permissions). The basic idea is to create a new de_struct directory array, allocate some blocks for it, and assign those blocks to a directory entry in the parent directory. So now the parent directory knows which blocks on disk belong to the newly created directory. Every new directory is initialized with two directory entries: "." (itself) and ".." (its parent). The "." directory entry will hold the blocks allocated for itself and the number of blocks it has, this way it makes it easier to know which blocks to write to when updating the directory on disk. The ".." directory notes the blocks allocated for its parent directory, so now we can load the parent directory using the ".." entry and load the new directory

with its respective entry from the parent directory. This system makes it possible to avoid having a global file system array that notes all of the directories and files. It's sort of like having pointers, but instead of reading from memory, it's reading from the disk.

To delete a directory is as simple as freeing its allocated blocks on the free space map. This will let the free space manager know that those blocks can now be put to use for another file/directory. Then we have set the parent directory's entry for the removed-directory to an empty default state.

To move a file or directory, we'll just copy the directory entry information from the source parent to the destination parent, then set the src parent DE to empty. This command should handle cases of moving files/directories up ro down sub directories, edge cases with unique pathnames (e.g test vs. test/), and renaming files/directories.

For fs_Open, Read, Close we must interact with the directory file descriptor. The basic idea is to open a directory, read the contents, and close it, all for printing the contents of the directory to the terminal. In fs_open we'll create the directory file descriptor and load the directory to list from disk. fs_read will then take that directory descriptor and produce an appropriate output for that directory entry, including its name, size, and mode. And finally, fs_close will free the directory from memory.

For fs_isFile and fs_isDir we want to check if the path is valid. If the path is NULL, we return 0. We then call parsePath() to locate the file or directory and get its position in the parent directory. If the path is invalid, or the item isn't found, we can return 0 to indicate that it isn't a file or directory. For fs_isFile we want to call our is_directory function from our de_struct to verify whether or not it is a directory. If is_directory is 0, we return 1 indicating that it's a file. For fs_isDir, we return 1 immediately if the index is -2 meaning that it's the root directory. If is_directory is 1, we can return 1 since it's a directory. To handle edge cases, we can add a temp path so that parsePath won't trim the original pathname before passing it to fs_isFile and fs_isDir.

For fs_delete we first check if the file name is valid. We then can call parsePath to locate the file and get its parent directory and index. If the file or directory name is not found, we can return -1. If the entry is a directory, we'll load it and make sure it contains . or .., If the entry is a file, we can free the blocks that it uses. For both cases, we will clear the metadata in the directory entry, reset its fields, and the updated directory back to the disk using LBAWrite. Since we don't track the directory's start block in parseInfo, we'll use the first block listed in blocks_allocated. For the root directory we'll hard the known start block.

For fs_stat, we start by checking if the input path or the output stat buffer is NULL, since we won't be able to do anything without both. If either is invalid, we can return -1. We then can call parsePath to locate the file or directory to get its parent directory and index. if the item isn't found or the path is invalid, we return -1. We can then pass all the info over to our buffer and fill it up.

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

File System - File operations

Our plan is to test each function one at a time, getting the necessary functions working to try one command, then fixing any issues and implementing the functions needed for the next. For file operations, we will get touch working first, then move on to cat, cp2fs, cp, and cp2l. The first thing to do when implementing the file operations is to implement the b_open() function. I plan to start by checking if the file system has been initialized, and call b_init() if it hasn't. I'll make a copy of the file path so I can safely use it with parsePath() to find the file and its parent directory. If the file doesn't exist and the O_CREAT flag is set, I'll create a new file by finding an empty spot in the parent directory, allocating a block, and filling in the file's metadata like name, size, and timestamps. I'll then write the updated directory back to disk. If the file does exist, I'll make sure it's not a directory, and if the O_TRUNC flag is set, I'll reset the file size and update its modified time. For read or write access, I'll load the first block into a buffer if the file has data. Finally, I'll fill out the file control block (FCB) with the file's info, including buffer, flags, and starting block. Throughout the function, I'll include error checks and free memory as needed to prevent leaks or crashes.

Once b_open() is working, we will implement b_close() because the touch command calls b_open() and b_close(). b_close() will be relatively simple, as it is used for cleaning up the things related to file operations. It will work similarly to how b_close() worked in assignment 5, where it frees the buffer and sets all necessary fields to NULL. We plan to follow this, and as we work on it, we will see if we have to make any changes.

After the touch command is working, we will move on to getting cat, cp, cp2fs, and cp2l working. These require that b_read() and b_write are implemented. In class, the professor showed us a way of doing b_read() in assignment 5 and explained that it was a very effective way to read a file. He broke it into three parts: the first part being copying the data from the internal buffer to the user's buffer, the second being to read full blocks of data from disk to the user's buffer, and the third being copying over the leftover portion of data that is less than 512 bytes. We plan to implement a b_read() function that follows this concept for effective reading of files. Then we will implement b_write(), which is responsible for writing data from a buffer to disk. This will function similarly to b_read(), but will write to disk instead of reading from disk. When we get these functions working, our cat, cp, cp2fs, and cp2l functions should be working as intended.

Step-by-step:

1. Start with b_open() and b_close()
    - These functions are used for the touch command, touch is the starting point for file operations. Once touch is working, we will be able to move on and complete all the other file operations.
2. Then complete b_read() and b_write()
    - These functions are used for cat, cp, cp2fs, and cp2l. These allow us to test our file system and make sure that everything is being written to disk and can be read

properly. cp2fs is a crucial command as it allows us to confirm that our file sizes are working correctly and that the data is being stored as we want it to be. cp2fs is used to copy a file from Linux to our file system. We can confirm any file copied from Linux to our file system using cat.

Changes Made from Planning Stage

- Due to immense complexities arising from implementation, we decided to change from the FAT approach to bitmap based approach for free space management.
- We have also removed the planned applyFSChanges() for our freeSpace prototype, and made changes directly without utilizing any external functions.

Which Shell Commands Work and Don't Work:

The mv command, which is the common preferred method for renaming files/directories, lacks that feature in our filesystem.

Details on Each of Our Functions:

Directory Operation Functions:

1. int * newDir(de_struct * parentDir, mode_t mode);

   This function is responsible for creating a new directory, allocating the blocks needed, initializing the "." and ".." and writing it to disk. The number of blocks required for one directory would be the directory entry size multiplied by the fixed directory size (which we decided on 32 entries) divided by the blocksize (512). Then each directory entry is initialized as empty, except for the first two. The "." entry will keep track of the blocks allocated for itself, and the ".." entry will keep track of the blocks allocated for its parent. The new directory is then saved by writing it to disk, while returning the array of block indexes it was allocated. This function also supports the root initialization, where its ".." entry uniquely stores the same blocks as the "." entry since the root's parent is itself. Then the new directory is assigned to the global rootDir de_struct *.

2. int fs_mkdir(const char * pathname, mode_t mode);

   This function is responsible for handling the "md" command. It first parses the given path and checks if the subdirectory already exists. If it doesn't it will then call newDir() to create the new directory. newDir() returns the array of blocks it has been allocated, and that along with the directory name are stored in a free directory entry in the parent directory. The parent directory is then updated on disk, and is freed (unless it's the root directory or current working directory).

3. int fs_rmdir(const char * pathname);

   This function is responsible for handling the “rm” command. First, it will parse the path and check if the directory-for-removal is in the parent directory. If so, it will then check if you are trying to delete the current working directory (the rootDir check is in the fsshell script) and if the entry is actually a directory and it is empty. If all conditions are met, the blocks_allocated for the directory are freed and the directory entry is reset back to an empty default. Finally, the parent is updated on disk.

4. int fs_mv(const char * srcPath, const char * dstPath);

   This function is responsible for handling the “mv” command. It will parse both the source and destination paths while also making all the edge case checks. When parsing the source, check if it's a full path or just the filename. If just the filename is given, then that means the source file/directory is located in the current working directory, so create and send a concatenated path of cwd/filename to parsePath(). When parsing the destination, check if the filename is at the end of the path, if not, add it and send it to parsePath(). When both source and destination directories are loaded into memory, copy the directory entry information from the source to an empty slot in the destination directory. Finally, set the source directory entry to empty and update both directories on disk.

5. fdDir * fs_opendir(char *pathname);

   This function opens a directory at a given path and prepares it to be read. It uses parsePath() to find the directory and fill the fdDir struct that the function needs to return. The parseInfo struct that is populated by parsePath contains all of the necessary information to correctly assign every field in fdDir. When assigning the directory field in fdDir, it checks whether the parsePath returned that the path given is the root directory, if it is the root directory it assigns the directory as the parent, if not then it attempts to load the directory using the loadDirectory function and the given parent and index from the parseInfo struct. After fully populating fdDir, the function returns the fdDir struct.

6. struct fs_diriteminfo *fs_readdir(fdDir *dirp);

   This function reads the next valid entry from an open directory. It first skips over all empty entries (entries with the file name ‘\0’), then when it finds a valid entry, the function fills the fs_diriteminfo struct, which is pointed to by dirp->di. After populating the struct, dirp->di will contain the entry’s name, size, and type. Then the dirEntryPosition counter is incremented so that the next call will return the following entry. The function then returns the fs_diriteminfo that was populated, or NULL if no more entries are available.

7. int fs_closedir(fdDir *dirp);

   fs_closedir() is responsible for closing an open directory stream and freeing any resources used while the directory stream was open. It frees the directory entry pointer in the fdDir and fs_diriteminfo that was pointed to by the fdDir passed in. After freeing those, it frees the fdDir pointer itself.

8. char * fs_getcwd(char *pathname, size_t size);

   This function retrieves the current working directory path as a string. It copies the global variable *cwdName* which is used for tracking the active working path into the provided *pathname* buffer variable, while making sure that we do not exceed the determined buffer size and forcing a null terminator at the end of it. If any of these parameters are incorrect, the function returns a NULL to prevent the other functions from breaking. Otherwise, we return a pointer to the updated *pathname* variable enabling commands like pwd to report the current working directory to the user.

9. int fs_setcwd(char * pathname);

   This function updates the current working directory *cwDir* and *cwdName* based on the provided pathname. It handles relative paths and absolute paths, also special cases such as ".." and ".". We handle absolute and relative paths by checking if the given path starts with "/". If it does, it is treated as an absolute path.

   If the path does not start with '/', it is treated as a relative path, meaning navigation is performed starting from the current working directory. For multi-level relative paths like "folder1/folder2", the code mainly uses *strtok_r()* to split the string on '/' and iteratively calls *fs_setcwd()* on each token.

   This lets the function resolve each part in sequence, changing directories step-by-step. If the input is "..", the function removes the last directory component from cwdName using *strrchr()*. If this results in an empty string, it resets back to root. Then, it calls *parsePath()* to locate the parent directory, and loads it using *loadDirectory()* respectively.

   We handle the “.” case, by simply returning by simply returning success as it means to stay in our current working directory.

   In addition to these cases we implemented error handling and memory controls to prevent memory from leaking and preventing segmentation faults that can be induced by other functions who call this function.

10. int fs_isFile(char * filename);
    This function checks if a path points to a file. It return 0 if the path is NULL, invalid, or not found. It uses parsePath() to locate the item, then checks the is_directory flag in our de_struct. If the flag is 0, it returns 1 meaning it is a file. Other it returns 0

11. int fs_isDir(char * pathname);
    Our fs_isDir function does the same as our isFile function except it checks if the index is -2 it returns 1 meaning it's a directory and also checks our de_struct is_directory flag. If either are true, it returns 1. Otherwise it returns 0

12. int fs_delete(char * filename);
    This function removes a file or directory whenever the command rm is used. It returns -1 if the path is invalid or the item is not found. It uses parsePath to locate the entry, then frees is allocated blocks using freeBlocks(). After that, it clears the entry fields and writes the updated parent directory back to the disk using LBAWrite(). If the parent directory is the root directory, it also writes the root directory back to the disk.

13. int fs_stat(const char * path, struct fs_stat * buf);
    fs_stat fills the buffer with information about a file or directory. it returns -1 if the path or buffer is NULL, or if the path is invalid or not found. It uses parsePath to locate the item, then creates a temporary pointer to the entry. It copies the size, block size, number of blocks, and timestamps into the buffer. Since we don't track the time we accessed it, we use the time it was modified.

14. int parsePath(char * pathName, parseInfo * ppi);
    The parsePath function provides a simplified and centralized way to resolve paths within our file system. High level functions such as *fs_mkdir, fs_rmdir, fs_setcwd, and fs_mv* use this specific function to locate specific files and directories which all use this centralized function.

    It determines these key values inside ppi structure to make connections within file structure:

- ppi->index: the index of the final element within the parent directory
- ppi->parent: the directory that contains the final element in the path
- ppi->lastElementName: the name of the final file or folder being referenced

    The function begins to check if the provided pathName is absolute or relative, just like explained above in the *fs_setcwd()* function. We use *strtok_r()* to tokenize every single component at each step and check if the given path is valid. If the subdirectory is found, we call *loadDirectory()* to fetch the contents and load the next token until we reach the end. If any token in between does not exist in the file system, we simply return 0 to indicate the given path is not valid. Once we reach the very last token, we simply return 1 to point out that it exists, which validates the entire path.

15. int findInDirectory(char * name, de_struct * parent);

    This function searches for a file or directory with a given name within a specified parent directory. It iterates through each entry in the parent directory, skipping over any empty entries (entries with a file name beginning with '\0'). If a matching name is found, it returns the index of that entry in the directory. If the entry is not found or either parameter passed in is NULL, the function will return -1.

16. de_struct * loadDirectory(de_struct * target)

    This function has the biggest responsibility of loading the directories from disk into memory. It is given the directory entry from the parent to load, and then checks if that entry is indeed a directory. If so, it'll allocate space in memory for the directory and read the blocks_allocated for the directory from disk (one at a time; as specified in our LBAread/write solution). Finally, the loaded directory is returned and is freed later in the function that called loadDirectory().

Free Space Functions:

1. initFreeSpace(int blockCount, int blockSize):

   This function initializes the freeSpaceMap in memory by allocating space which is one byte per block.It marks all blocks free first, then reserves system blocks. Finally, it writes the map to the disk via LBAWrite() to the FS_RESERVED sector.

2. allocateBlocks(int count):

   This function finds available blocks in the freeSpaceMap for a specified number. If enough blocks are found, they are marked as used. If not, it is rolled back, and returns NULL. Assuming blocks are found, it returns a list of blocks for the rest of the system to reference for that given operation. There are also measures against allocating VCB or FreeSpaceMap, the scanning logic is designed to skip these sectors to prevent accidental allocations to these blocks.

3. freeBlocks(int* blockArray, int count):

   This function takes a list of blocks called blockArray, and how many blocks was fed. It first validates that these blocks are safe to deallocate, such as VCB, FreeSpaceMap, already free etc. Then it simply sets the block in freeSpaceMap as 0, and wipes the contents of the block using LBAWrite() which makes the wiped data persistent.

4. checkBlockAvailability(int blockIndex)

This is a simple function for the rest of the system to use for extra layers of validation and debugging purposes to check whether a given block is free or not. It returns 0 for free, and 1 for used blocks.

5. loadFreeSpaceMap(int blockSize, int startBlock, int totalBlockCount)

   This function is responsible for restoring the freeSpaceMap from disk during system reinitialization or volume mounting. It first ensures any previously allocated freeSpaceMap memory is freed to avoid leaks. It then calculates how many blocks need to be read based on the total number of blocks in the system and the block size, using this to determine the correct read size. The function allocates enough memory to hold the full bitmap, then reads it from the specified startBlockusing LBAread(). If the read is successful, the free space map is now live in memory and ready to be used for block allocation and deallocation. If the read fails or memory allocation fails, it reports the error and ensures the system doesn't proceed with an invalid map.

File Operation Functions:

1. b_open():
   Our b_open() starts similarly to the assignment 5 b_open(). It starts by getting a free file control block and allocating the file buffer, then using parsePath to find whether the file already exists or not. The use of parsePath in this function allows us to know whether the file needs to be created or already exists. We can use this in conjunction with the O_CREAT flag to know when we need to create a file.

   Creating a new file involves finding an empty slot in the parent directory, allocating disk space for the file, and initializing the directory entry with the needed metadata (name, size, permissions, timestamps). Once the file is created, it then writes the updated parent directory back to disk so that the file is persistent, and populates all the necessary info in the file control block. After finishing this, the file is now created.

   The other case in b_open() is that the file already exists and must be read from disk. The use of parsePath allows us to get access to the information about the file to populate the file control block. If the file already exists, we must account for the Linux open flags, which include O_RDONLY, O_WRONLY, O_RDWR, and O_TRUNC. If the flag O_TRUNC is applied, it resets the file size to 0 and sets the modification timestamp to the current time, then updates the parent directory. If the file is opened for reading or writing, it loads the first block of the file from disk into the buffer. After the checks for these flags, it populates the file control block with all the needed information.

   Now that b_open() has handled all cases, the file has been opened correctly, and it can return the file descriptor.

2. b_close():

The b_close() function is also similar to how the b_close() function needed to be implemented for assignment 5. b_close() is responsible for closing an open file, this involves cleaning up all of the used resources for b_open(), b_read(), and b_write(). The main purpose of b_close() is to flush any remaining unwritten data in the file buffer to disk, free the file control block buffer, and set the file info and parent directory pointer to NULL to mark the slot as available.

3. b_read():

   The purpose of b_read() is to read data from an open file into the user's buffer. It does this in three parts:
   a. Part 1: Handles copying any data in the file's buffer to the user's buffer, which may or may not be enough to fulfill the read request. It copies this directly from the file buffer to the user's buffer and updates all variables to keep track of positions and counts.
   b. Part 2: Attempts to read as many full blocks of data as possible from disk into the user's buffer. It does this by calculating how many full blocks fit into the remaining request count and reads them from disk using LBAread().
   c. Part 3: Reads any remaining data that is less than a full block. After the first two parts, the only remaining data should be less than a full block, so part 3 copies the remaining required amount into the user's buffer.

   After doing this, b_read() returns the total number of bytes successfully read, which has been tracked by the bytesReturned variable.

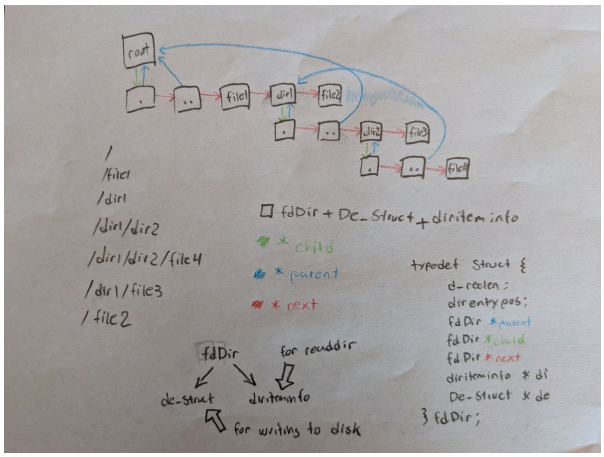
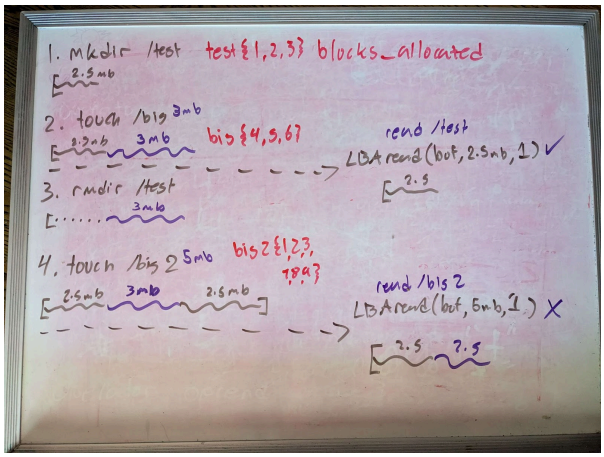
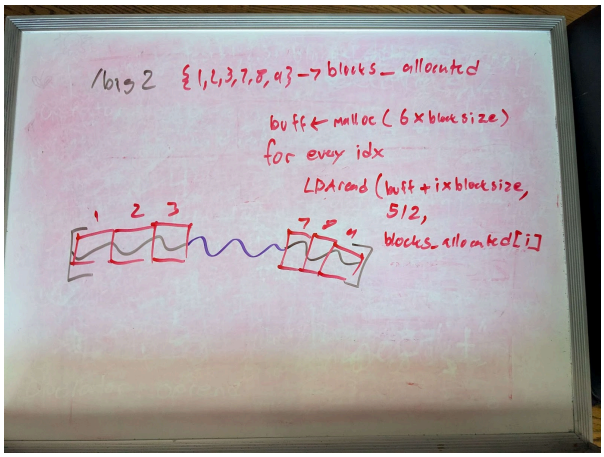
4. b_write():

   b_write() is responsible for writing data from a buffer to a file on disk, which allows the data to be persistent and be stored for later access. It begins by checking if there is any data in the file buffer; if there is, it then copies as much data from the given buffer as it can into the file buffer, and once it is full, it writes the data to disk. After writing this data, if there is still data to write and it is more than a block's worth, it will write the full blocks directly from the given buffer to disk. Then, once all of the full blocks have been written, it copies any remaining bytes to the file's buffer to be written when the file is closed.

   Once this is done, the file metadata is updated to reflect the new size, and the modified timestamp is updated to the current time. If this new size is larger than the current amount of blocks allocated for the file, more blocks are allocated to ensure that there is enough space on disk for the file. Then the parent directory is updated with the new information after the write. The function returns the total number of bytes written, which has been tracked by the bytesWritten variable.

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

Issues and Resolutions:

V1 of File System File Structure 	Our initial approach to the file system project was a pointer based structure. directory entries pointing to its neighbors, children, and parents. However this design was flawed because iterating through a large path would be incredibly slow. Also I (Eric) made incorrect assumptions about what the directory descriptors were actually supposed to do. We would later come up with our current implementation, directory entries noting their blocks_allocated for sub directories, files, etc.
LBA write/read problem 	About part way through milestone 2, we noted a major flaw with how we were writing/reading from disk. Since we wanted to go with a discontiguous approach, we couldn't just call LBAread/write normally. That would assume directories/files are stored continuously on one disk. The picture illustrates a problematic situation where LBAread would start incorrectly accessing something it's not supposed to.
LBA write/read posed solution 	To solve this issue, we decided to switch from reading/writing all the allocated blocks at once, to just one at a time. This way we can account for files/directories who's blocks were allocated discontinuously.

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

LBA write/read final solution

```
// write each block individually based on . blocks_allocated array
for (int i = 0; i < blocksNeeded; i++) {
    void * dirToBlocks = (void *)((char *)dir + i * BLOCK_SIZE);
    if (LBAwrite(dirToBlocks, 1, dir[0].blocks_allocated[i]) != 1) {
        printf("Error writing block %d for new directory\n", i);
        free(dir);
        return NULL;
    }
}
```

because dir is a de_struct pointer, when u increment it, it increments by de_struct size. To fix this, create a NEW void pointer that will increment by blocksize instead

```
// write the updated parent directory to disk
for (int i = 0; i < parentDir[0].blocks_count; i++) {
    void * dirToBlocks = (void *)((char *)parentDir + i * BLOCK_SIZE);
    if (LBAwrite(dirToBlocks, 1, parentDir[0].blocks_allocated[i]) != 1) {
        printf("Error writing updated block %d for parent directory\n", i);
        return -1;
    }
}
```

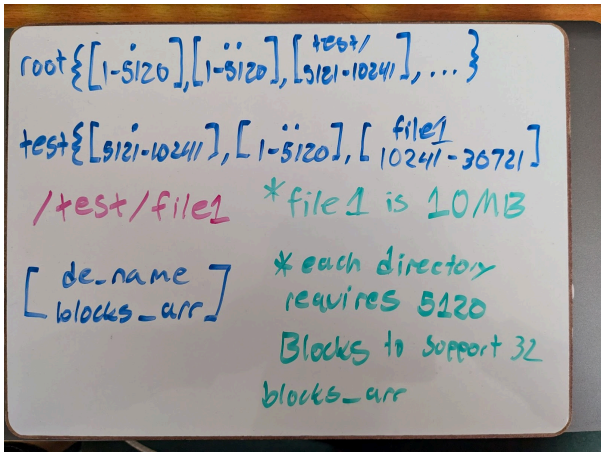
the . directory entry's blocks_allocated holds the indexes of all the blocks used by the directory, so we can load each one individually. Now we can read/write in a discontiguous manner

```
// read each block indtividually based on the blocks_allocated array
for (int i = 0; i < target->blocks_count; i++) {
    void *blockData = (void *)((char *)entries + i * BLOCK_SIZE);
    if (LBAread(blockData, 1, target->blocks_allocated[i]) != 1) {
        printf("Error reading block %d for directory\n", i);
        free(entries);
        return NULL;
    }
}
```

loading is very similar

The final implementation of the solution has all the basic fixes posed by our original draft, but with an added char pointer to buffer characters one at a time instead of whole directory entry lengths.

File system supporting large files



This picture illustrates what the blocks_allocated in the directory entries would look like if we needed to support a 10MB file. Each directory entry would need to support up to 20480 block indexes to keep track of a 10MB file. This would make just one directory entry ~80KB and one whole directory 2.5MB, very inefficient. For this reason we have decided to stick with supporting relatively small (~90KB) data.

Memory Leaks in Valgrind

```
==124722== Conditional jump or move depends on uninitialised value(s)
==124722==    at 0x10E178: findInDirectory (mfs.c:940)
==124722==    by 0x10BD13: fs_mkdir (mfs.c:125)
==124722==    by 0x109ED7: cmd_md (fsshell.c:396)
==124722==    by 0x10A65B: processcommand (fsshell.c:703)
==124722==    by 0x10A9AB: main (fsshell.c:852)
==124722==
==124722== Invalid read of size 1
==124722==    at 0x488FE90: strcmp (in /usr/libexec/valgrind/vgpreload_memcheck
==124722==    by 0x10E173: findInDirectory (mfs.c:940)
==124722==    by 0x10BD13: fs_mkdir (mfs.c:125)
==124722==    by 0x109ED7: cmd_md (fsshell.c:396)
==124722==    by 0x10A65B: processcommand (fsshell.c:703)
==124722==    by 0x10A9AB: main (fsshell.c:852)
==124722==  Address 0x4c66570 is 256 bytes inside an unallocated block of size
==124722==
==124722== Syscall param write(buf) points to unaddressable byte(s)
==124722==    at 0x4A849DC: __internal_syscall_cancel (cancellation.c:64)
==124722==    by 0x4A849FB: __syscall_cancel (cancellation.c:75)
==124722==    by 0x1105D7: LBAwrite (fsLowM1.c:250)
==124722==    by 0x10C003: fs_mkdir (mfs.c:159)
==124722==    by 0x109ED7: cmd_md (fsshell.c:396)
==124722==    by 0x10A65B: processcommand (fsshell.c:703)
==124722==    by 0x10A9AB: main (fsshell.c:852)
==124722==  Address 0x4c66000 is 0 bytes after a block of size 16,384 alloc'd
==124722==    at 0x48854DC: malloc (in /usr/libexec/valgrind/vgpreload_memcheck
==124722==    by 0x10E217: loadDirectory (mfs.c:963)
==124722==    by 0x10D907: fs_setcwd (mfs.c:684)
==124722==    by 0x10A14B: cmd_cd (fsshell.c:540)
==124722==    by 0x10A65B: processcommand (fsshell.c:703)
==124722==    by 0x10A9AB: main (fsshell.c:852)
==124722==
```

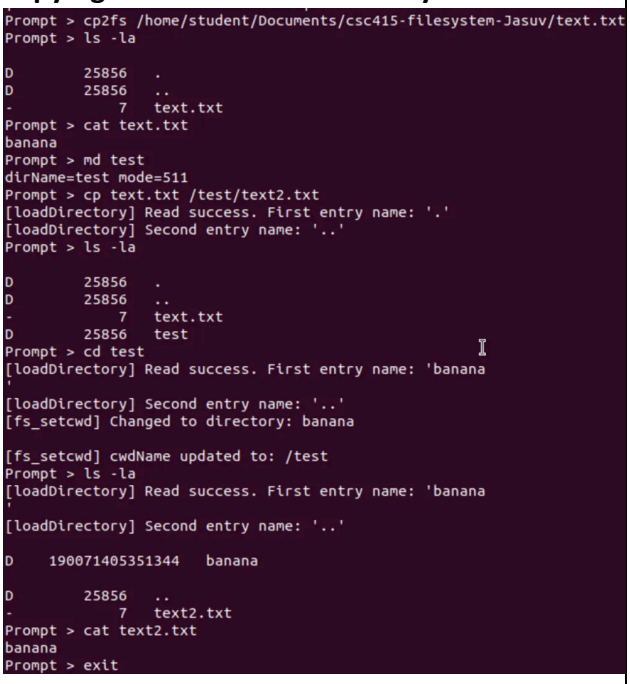
While interacting with the File System we encountered severe memory issues, heap corruptions and incorrectly freed memory caused stability problems including growing memory footprint, segmentation failures and other problems. We resolved these problems by investigating all malloc() calls and making sure we have appropriate and corresponding free() statements.

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

Persistence issues with removing a folder after quitting the program

When we created a directory, exit the program, and attempt to erase the previously created directory, it would not be able to remove the directory. The program would return a problem indicating the block that is corresponding to that respective directory is already free.
This was root caused to be a problem with FreeSpace manager implementation. The free space map was not being correctly written to disk, and the file system would not correctly recover the state where it was left.
We resolved this problem by validation of loadFreeSpaceManager() implementation and updating the vcb struct item called *freespace_list_start* so that it could reference the freeSpaceMap properly upon relaunching the program.

Copying a file from one directory to another



When implementing the cp command, we encountered a bug when copying a file from one directory to another. The first issue was incorrectly setting the file's parent directory, so when it would write the updated metadata to the parent on disk, it would write to the wrong place. To fix this, we added a parent directory pointer in the b_fcb struct. After this was fixed, we realized that when copying a file to a different directory, it would correctly copy the file; however, erroneously overwrite the "." entry in that directory with the contents of the file. We found that the issue was due to the way we were attempting to write any remaining data in the buffer in b_close. There was no check to see whether the file had read or write permissions; this should only write when the file has write permissions. After adding this check, it was no longer incorrectly overwriting the "." entry, and cp was working correctly.

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

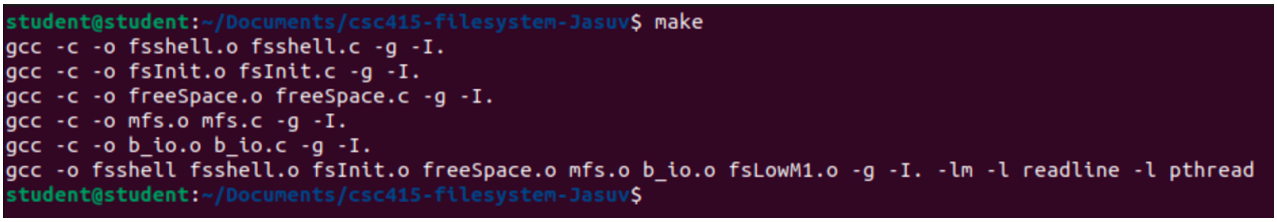
A table of who worked on which components:

Michael	**Utku**	**Eric**	**Randy**
• openDir • readDir • closeDir • findInDirectory • b_read • b_close • b_write	• fs_setcwd() • fs_getcwd() • ParsePath() • freeSpace.c • freeSpace.h	• newDir • fs_mkdir • fs_rmdir • fs_mv • loadDir	• fs_delete • fs_isFile • fs_isDir • fs_stat • b_open

How did your team work together, how often you met, how did you meet, how did you divide up the tasks?

Our team worked well together throughout the project. We were able to split up the tasks fairly so that everyone had something to do. Whenever someone ran into a problem or had a question, we talked about it right away and helped each other out. Everyone stayed involved and showed a lot of effort, which made things go smoothly. Most of our team discussions happened right after class, which was convenient for all of us. We only scheduled full team meetings when we had to make bigger decisions like choosing whether to use an array or a linked list, which we mentioned earlier. Overall, we had good teamwork and communication.

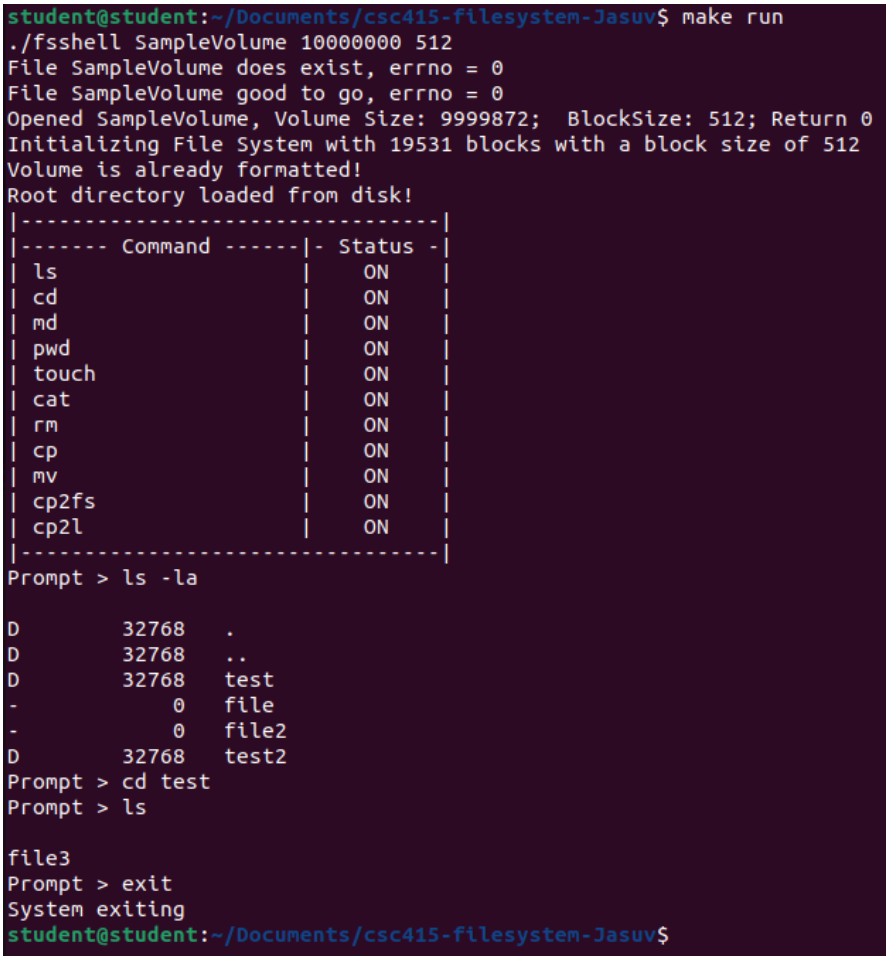
Screenshot of compilation:



Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

Screenshots of the working commands:

- ls command



Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

- cd Command

```
utku@utku-VMware20-1:~/git/csc415-filesystem-Jasuv$ 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
blocksNeeded=64 should be 64
Volume formatted!
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md test
dirName=test mode=511
blocksNeeded=64 should be 64
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > exit
System exiting
utku@utku-VMware20-1:~/git/csc415-filesystem-Jasuv$
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

- pwd command

```
utku@utku-VMware20-1:~/git/csc415-filesystem-Jasuv$ 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
blocksNeeded=64 should be 64
Volume formatted!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > md test
dirName=test mode=511
blocksNeeded=64 should be 64
Prompt > pwd
/
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > pwd
/test
Prompt > exit
System exiting
utku@utku-VMware20-1:~/git/csc415-filesystem-Jasuv$
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

- rm (dir) command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
Initializing File System with 19531 blocks with a block size of 512
Volume is already formatted!
----- PRINTING ROOTDIR -----
entry[0] in rootDir: . 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
entry[1] in rootDir: .. 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
entry[2] in rootDir: test 1
        entry[i].blocks_allocated[0]: 105
        entry[i].blocks_allocated[1]: 106
        entry[i].blocks_allocated[2]: 107
----- END PRINTING -----
Root directory loaded from disk!
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > ls

test
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > ls
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'

test2
Prompt > cd ..
setcwd pathname=..
Prompt > rm test/test2
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > ls
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'

Prompt > cd ..
setcwd pathname=..
Prompt > rm test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
Prompt > ls

Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

- md command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freeSpace.o freeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freeSpace.o mfs.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
./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
blocksNeeded=64 should be 64
Volume formatted!
|--------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|--------------------------------|
Prompt > md test
dirName=test mode=511
blocksNeeded=64 should be 64
Prompt > ls

test
Prompt > md test/test2
dirName=test/test2 mode=511
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
blocksNeeded=64 should be 64
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > ls
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'

test2
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

- mv command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
Volume is already formatted!
----- PRINTING ROOTDIR -----
entry[0] in rootDir: . 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[1] in rootDir: .. 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[2] in rootDir: test 1
        entry[i].blocks_allocated[0]: 105
        entry[i].blocks_allocated[1]: 106
        entry[i].blocks_allocated[2]: 107
        entry[i].blocks_allocated[3]: 108
entry[3] in rootDir: file 0
        entry[i].blocks_allocated[0]: 169
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
        entry[i].blocks_allocated[3]: 0
----- END PRINTING -----
Root directory loaded from disk!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls

test
file
Prompt > mv file test
parsing /file
parsing test/file
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
writing file to 2 in parent dir
Prompt > cd test
setcwd pathname=test
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
[fs_setcwd] Changed to directory: .
[fs_setcwd] cwdName updated to: /test
Prompt > ls
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'

file
Prompt > mv file ..
parsing /test/file
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
parsing ../file
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'
writing file to 3 in parent dir
Prompt > ls
[loadDirectory] Read success. First entry name: '.'
[loadDirectory] Second entry name: '..'
[loadDirectory] blocks_allocated count: '64'

Prompt > cd ..
setcwd pathname=..
Prompt > ls

test
file
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

- touch command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
blocksNeeded=64 should be 64
Volume formatted!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > touch text.txt
Prompt > ls -la

D        32768   .
D        32768   ..
-            0   text.txt
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

- cp2fs command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
blocksNeeded=64 should be 64
Volume formatted!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls -la

D        32768   .
D        32768   ..
Prompt > cp2fs /home/student/Documents/csc415-filesystem-Jasuv/text.txt text.txt
Prompt > ls -la

D        32768   .
D        32768   ..
-            7   text.txt
Prompt > cat text.txt
banana
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

- cp command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
Volume is already formatted!
----- PRINTING ROOTDIR -----
entry[0] in rootDir: . 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
entry[1] in rootDir: .. 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
entry[2] in rootDir: text.txt 0
        entry[i].blocks_allocated[0]: 105
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
----- END PRINTING -----
Root directory loaded from disk!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls -la

D        32768   .
D        32768   ..
-            7   text.txt
Prompt > cp text.txt text2.txt
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

- cat command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
Volume is already formatted!
----- PRINTING ROOTDIR -----
entry[0] in rootDir: . 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[1] in rootDir: .. 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[2] in rootDir: text.txt 0
        entry[i].blocks_allocated[0]: 105
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
        entry[i].blocks_allocated[3]: 0
entry[3] in rootDir: text2.txt 0
        entry[i].blocks_allocated[0]: 106
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
        entry[i].blocks_allocated[3]: 0
----- END PRINTING -----
Root directory loaded from disk!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > ls -la
```

Michael Thompson 922707016 | **Utku Tarhan** 918371654 | **Eric Ahsue** 922711514 | **Randy Chen** 922525848

```
Prompt > cp text.txt text2.txt
Prompt > ls -la

D        32768    .
D        32768    ..
-            7    text.txt
-            7    text2.txt
Prompt > cat text2.txt
banana
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

```
Prompt > ls -la

D        32768    .
D        32768    ..
-            7    text.txt
-            7    text2.txt
Prompt > cat text.txt
banana
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-Jasuv$
```

Michael Thompson	**Utku Tarhan**	**Eric Ahsue**	**Randy Chen**
922707016	918371654	922711514	922525848

- cp2l command

```
student@student:~/Documents/csc415-filesystem-Jasuv$ 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
Volume is already formatted!
----- PRINTING ROOTDIR -----
entry[0] in rootDir: . 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[1] in rootDir: .. 1
        entry[i].blocks_allocated[0]: 41
        entry[i].blocks_allocated[1]: 42
        entry[i].blocks_allocated[2]: 43
        entry[i].blocks_allocated[3]: 44
entry[2] in rootDir: text.txt 0
        entry[i].blocks_allocated[0]: 105
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
        entry[i].blocks_allocated[3]: 0
entry[3] in rootDir: text2.txt 0
        entry[i].blocks_allocated[0]: 106
        entry[i].blocks_allocated[1]: 0
        entry[i].blocks_allocated[2]: 0
        entry[i].blocks_allocated[3]: 0
----- END PRINTING -----
Root directory loaded from disk!
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > cat text.txt
```

Michael Thompson 922707016
Utku Tarhan 918371654
Eric Ahsue 922711514
Randy Chen 922525848