File System Project

Team Kentucky Kernels

Phillip Davis

Jared Aung

Preet Vithani

Igor Tello

Github name and link: R3plug

Planning

The planning for this project consisted of three major stages, based on the milestones defined throughout the course. The first stage focused on creating the core structures essential for operating our file system at a fundamental level. The second stage emphasized the development of directory-level functions that utilized those structures. The final stage implemented the broader functionality of file operations such as reading and writing.

In the beginning, our team was optimistic and approached the design in a traditional, handwritten planning format. We identified the three essential components of our system: the Volume Control Block (VCB), the extent-based free space manager, and the Directory Entry (DE) structure. These formed the foundation of our entire file system.

A key decision early on was to ensure that core metadata like the VCB, extent table, and root directory would be kept in memory for fast access. This allowed us to easily retrieve vital properties such as block size or volume layout during runtime. For example, the extent table's block locations and root directory’s start block and size were stored in the VCB and loaded during initialization.

We explored options for space management. Although a FAT-style structure was considered initially, we shifted to using extent tables to handle allocation more efficiently. This decision allowed for tracking contiguous ranges of free or used blocks, simplifying allocation logic and reducing fragmentation.

While we made educated decisions about which properties to include in these structures, we quickly discovered during implementation that some parts needed to be adjusted or fine-tuned. Nevertheless, our original structure and modular approach held strong throughout the development process. We deliberately chose to keep the structures lean and flexible to avoid unnecessary complexity.

The second stage focused on building the functionality to interact with the data structures. Using the provided API function signatures, we implemented directory traversal and manipulation features, including creating, opening, and reading directories.

One major challenge in this stage was designing and implementing a reliable parsePath function. Since path resolution was fundamental to all other operations, we invested time in making it robust. It had to support relative and absolute paths, interpret ".", "..", and tokenize each component effectively. The complexity of corner cases made this a time-consuming part of our development.

Once parsePath was in place, the remaining directory-related functions became manageable. Our path stack and CWD management infrastructure also helped establish consistent navigation behavior throughout the system.

For the final stage, we built support for file handling—opening, reading, writing, and closing files. While we had experience from previous assignments, adapting this to our extent-based system posed new challenges.

We relied heavily on the foundational directory and metadata access functions from Stage 2. The idea was to open a file by locating its directory entry and loading relevant information into a File Control Block (FCB), which maintained metadata like size, current byte position, and extent allocation details.

A critical realization during implementation was that our read/write operations needed to handle non-contiguous extents correctly. We had to compute the correct Logical Block Addresses (LBAs) based on offset and extent position, and adapt our reads and writes to work across potentially multiple extents.

For writing, we attempted a three-phase plan:

1. Write any initial offset into the first block.

2. Write full blocks where possible.

3. Write the remainder into a partial block at the end.

However, the real challenge emerged during allocation. Ensuring enough space was available, growing files as needed, and then cleaning up unused space after closing required more planning than expected. The logic around releasing extra blocks after truncation and managing partial blocks at the end proved especially fragile.

While our initial plan helped structure the implementation, this stage taught us the importance of anticipating edge cases in disk I/O. The write operation, in particular, became a patchwork of fixes as we adapted to emerging issues.

Description of File System

Let's start with the basic structures we have the VCB. The volume control block is created when the volume is formatted during the initialization of our file system. The structure is used to confirm that the volume is formatted (via the signature) and contains the information that our file system will need to work with our volume. The information tracked includes block count and size of the volume as well as several block locations of note for our file system including the root directory and the structure containing our free space information. The volume control block will start at the first logical block in our volume by convention and it will be brought into memory by our file system to begin working with the volume. The volume control block is defined below:

```
typedef struct VolumeControlBlock {
    char     signature[8];
    uint32_t blockSize;
    uint32_t totalBlocks;
    uint32_t extentTableStart;
    uint32_t extentTableBlocks;
    uint32_t rootDirStart;
    uint32_t rootDirBlocks;
    uint32_t freeBlockStart;
    time_t   createTime;
    time_t   lastMountTime;
} VCB;
```

The next core structure we have is the base of our free space system, the Extent Table. The table is implemented as an array of fixed-size extent records. Each extent contains the starting block, count of consecutive blocks, and a flag indicating if it is used. This allows us to quickly allocate

and release large chunks of blocks efficiently. The VCB tracks the starting point and size of this table, and the table is read into memory during initialization for allocation and deallocation operations.

Next, we have the directory system. Each directory is represented as an array of DE (Directory Entry) structures, each of which stores metadata about either a file or a directory. The DE structure is defined as:

```
#define MAX_NAME_LENGTH 255
typedef struct {
    ExtentTable mem;
    time_t creationTime;
    time_t modificationTime;
    time_t lastAccessTime;
    uint32_t size;
    char name[MAX_NAME_LENGTH];
    char isDir;
} DE;
```

Each directory includes references to itself and its parent in the first two entries. Empty slots are marked by null characters in the name field. Upon formatting, the root directory is created and remains loaded in memory as both the root reference and the initial current working directory (CWD).

When the file system is initialized, it checks the first logical block for a valid VCB signature. If one is found, the system loads the volume information and associated metadata (extent table, root directory). If the volume is not yet formatted, it creates the VCB, writes it to disk, initializes the free space extent table, and creates the root directory structure. After this, it updates the VCB with the starting locations of all these components and writes it back to disk.

Once initialized, the shell allows users to issue commands to perform file and directory operations. The CWD keeps track of the current location for all path resolution functions. The file system supports parsing paths relative

to CWD or absolute from root, and loads directories as needed using extent metadata.

Creating a directory involves parsing the path to find the parent, finding a free DE slot in the parent, initializing a new directory (including '.' and '..'), allocating space using free blocks, and updating the parent directory. Deleting a directory checks that it is empty, frees its allocated extents, clears the parent entry, and updates the parent metadata.

Opening a directory loads its DE entries into memory, while reading a directory returns valid entries. Navigation uses CWD, and changing directories sets it to the loaded DE contents of the target.

For files, the system supports open, read, write, and close. Opening a file resolves the path and initializes an in-memory file control structure with size, position, and open flags. Read and write operations adjust the file pointer and perform disk I/O as needed, handling partial and full block reads/writes using buffering.

Write operations grow files dynamically by allocating new extents. If the file shrinks before closing, extra extents are released. Upon closing, metadata is written back to the directory entry.

The entire design follows an LBA model using disk blocks and extents. It supports modular navigation, file creation, and I/O in a structure consistent with real-world file systems.

In conclusion, our system comprises: VCB for volume metadata, Extent Table for block allocation, DE for directory and file metadata, root and CWD for path management, and FCBs for file operations. Together, they enable a functioning shell-based file system with directory and file support.

Issues

- LS was failing to show newly created directory entries. LS and md were also throwing memory errors. The first issue was the isDir variable had been changed from a char to an int, but the checks that check that value had not all been updated so the find free directory entry function was returning an invalid index when md tried to make a new directory. Next the ppInfo structure had a char pointer to hold the last element name. However, the memory for that variable was never allocated to when the information is accessed it throws a memory error. Since we know the max name length we changed it to a fixed sized array and updated the functions that access it to reflect this.

- CD throws an error when trying to move to a newly created directory

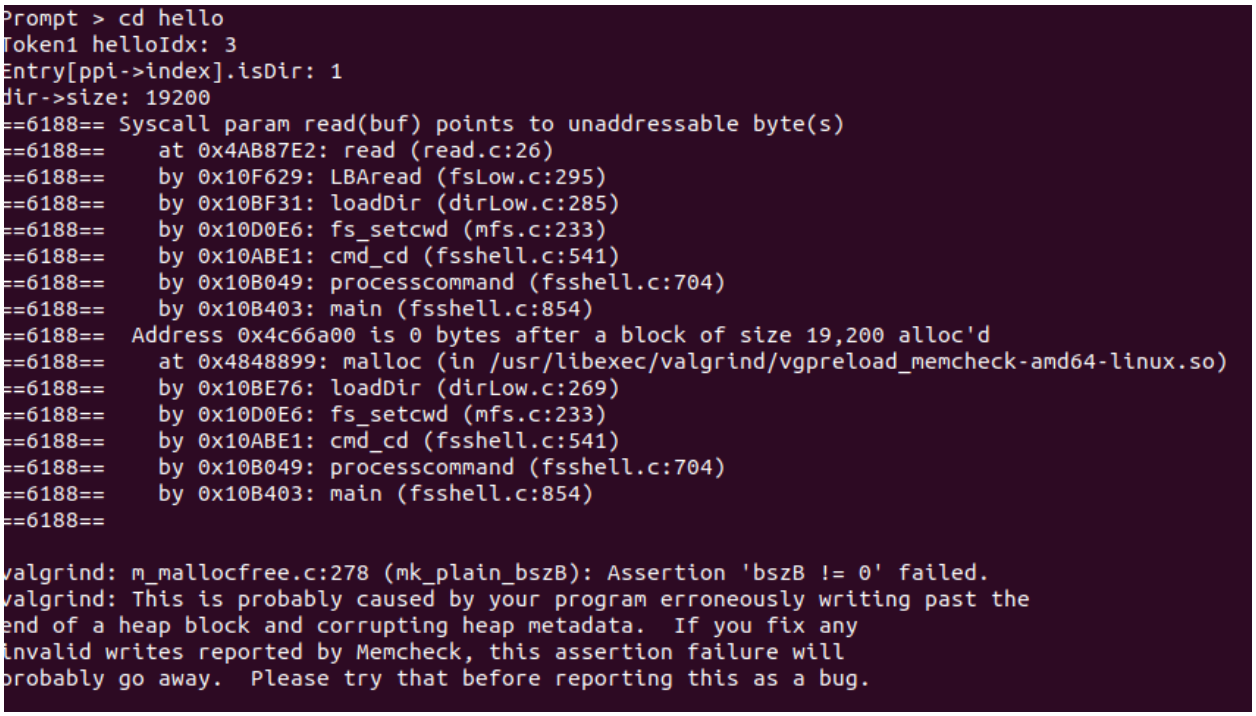
```
Prompt > md test
parsePath completed
blocksNeeded 29696
createDir completed
Prompt > cd test
fs_setcwd: is not a valid path
Could not change path to test
Prompt > cd /test
fs_setcwd: is not a valid path
Could not change path to /test
Prompt >
```

- Using valgrind I saw that in the findInDir function there was an access to an uninitialized value. In fs_setCWD a check to see if a directory is a directory the check return invalid if isDir was marked as true not false. Changed to return false if entry is not a directory. This fixed this error and allowed to move to the newly created directory but an error is thrown now when trying to ls in this new directory.

```
fsshell: malloc.c:2617: sysmalloc: Assertion `(old_top == initial_top (av) && ol
d_size == 0) || ((unsigned long) (old_size) >= MINSIZE && prev_inuse (old_top) &
& ((unsigned long) old_end & (pagesize - 1)) == 0)' failed.
make: *** [Makefile:68: run] Aborted (core dumped)
student@student:~/Documents/csc415-filesystem-R3plug$
```

- When creating the mv function It was getting a segfault while trying to insert the source in the new directory. The for loop that looked for a free directory entry was not exiting once it found it and was trying to access uninitialized areas. Adding a break to the section that runs when a free directory was found fixed that issue, but now it does not move the directory correctly . It writes what looks like an incorrect section of memory so the parent directory still shows it contains the file we want to move but other text is added to the destination directory.

- Keep getting a core dump when trying to run ls or cd inside a newly created directory. Make vrun confirms my suspicions, has to do with memory either reading too much into a small buffer, or allocating way too much space to begin with. When md is run, blocksNeeded is at 19456, not sure if this is meant to be bytes or not. Still trying to figure out the issue.
    - loadDir was the issue
    - Realized malloc(dir->size) was too large, since dir->size was block aligned, not actual number of entries.
    - Used entryCount * sizeof(DE) instead



- There were several issues that needed to be addressed. First the info for newly created directories was not being fully saved in the parent entry for the directory. Next, checks were returning failed because they were checking an entry in the directory to be traversed to instead of checking information about the directory itself causing unneeded failures. The signal flag indicating new directories were directories was not being applied everywhere causing failures.
- Persistence was not happening
    - Turns out we were not loading up the vcb first and comparing signature our file system signature to the VCB signature already on disk. Because of that, we were creating a new volume every time, which would delete the saved data. Fixed solution by first comparing signatures, then if they match, take root from disk and load it into current working directory.

Function Details

initFreeSpace: Allocates memory for the ExtentTable, which stores the metadata about which blocks are used or free. Block 0 is initially marked as the vcb, then the following blocks are reserved for the extent table itself, and these blocks are determined by a #define variable. The next blocks that are reserved are for the root directory. After the root directory, the remaining blocks are marked as “free” in one large extent, with the total number of entries being saved as the extent count. Finally, the extent table is written to disk using LBAwrite(), and the allocated memory is freed.

releaseBlocks: Takes in as parameters the starting block and block count for which to be released. Space is allocated for an extent table to be read from disk. Function then iterates through the extent table based on the parameters and sets those blocks to “free”. Finally the extent table is written to disk using LBAwrite() and returns a 0 for success.

createDir: createDir takes two arguments, the number of entries the directory should have and the parent directory. It calculates the amount of memory this will take then converts this into the number of blocks it will need. It then minimally initializes all directory entries except the first two with sentinel values to indicate they are empty. It requests freeBlocks to store the directory from the allocateFreeBlocks function then stores that information along with the creation, modification, and access time to the current time in the first entry of the directory, the “.” entry.

Once the “.” entry is initialized and it checks if the parent argument is NULL. NULL indicates that the directory being created is the root directory. If it is creating the root directory it sets the parent equal to the newly created directory. It then sets the second directory entry to this parent. Finally it calls the writeDir function to write the currently made directory to disk and returns the new directory.

writeDir: Takes one argument, a directory pointer to be written to disk. It then goes through the extents in the self entry of the directory passed to it and writes the directory to disk in chunks relating to the extents in the extent table. It will

return -1 if the number of blocks written for each extent does not match the number that should have been written.

parsePath: Takes two arguments, a string holding a path name a reference to a ppInfo Structure that will hold the directory and and location information of the path target. First it checks if the path is from root or is a relative path. Then it steps down through the file system moving to each place indicated in the path until it reaches the end. If at any point the path is invalid it will return -1. If it reaches the end of the path it will return the information it finds in the ppInfo structure.

loadDir: Takes one argument, a pointer to a directory. It then allocates memory to hold that directory and reads its extent table writing the blocks from disk into memory.

findInDir: Takes two arguments, the parent directory to be searched and a string to be located. It loops through the parent directory comparing the names of the directory entries to the token provided. If located it returns the index in the parent directory where the directory entry is located. If it is not found it returns -1

Fs_mkdir: Takes two arguments, a path where the directory should be made and any permission info to be applied to the directory. Currently it ignores the argument for permissions. It creates the directory then adds the newly created directory information in the first free directory it finds. If The directory is full it calls expandDirectory to add more entries.

findFreeDE: Takes one argument, the directory to be searched. It iterates through the directory checking entries for the empty sentinel value in the directory entry name ‘\0’. If a free entry is found it will return the index of that directory. If the directory is full it will call a function to expand the directory then returns the first free index.

expandDirectory: Takes one argument, the directory to be expanded. It allocates memory for 50 more entries. Then gets free space on the disk to hold these entries. It marks them all as empty. Then it adds the additional allocated extents to the original directories extent table., writes the new blocks to disk

safeFree: Takes one argument, a directory entry pointer. It then checks if this directory is the root directory or the current working directory. If it is not it frees the memory holding the directory.

Fs_delete: Takes in a file name as a parameter. That filename is run through the parsePath() function in order to locate the file within the directory, and fills in the metadata such as parent directory and file's index in that directory. If the file is not found or the path is invalid, an error is returned. Once located, the function checks if the file is actually a file or a directory, since this function is only for file deletion. Once that has been established, the data blocks are freed by using the freeBlocks() function, letting the free space manager know those blocks are good to go. Next, the file's directory entry is cleared using memset(), removing all metadata. Finally, the updated directory is written back to disk with writeDir() function and then the allocated memory is freed with the function returning a 0 if all is successful, or a -1 if anything fails.

Fs_rmdir: This function takes the absolute or relative path of a target directory and removes it from the file system. It begins by parsing the given path using parsePath to locate the parent directory and the index of the directory entry to be removed. The function ensures that the directory is not the root, not the current working directory, and is empty before proceeding. If these checks pass, the directory's extents are released, the directory entry is cleared from the parent, and the updated parent directory is written back to disk.

Fs_stat: This function resolves a given path using parsePath, then copies information from the corresponding DE (Directory Entry) into a user-provided fs_stat structure. The metadata returned includes the file size, number of allocated blocks, timestamps for creation, modification, and last access, as well as the block size used by the file system. This function allows programs and users to inspect and retrieve metadata about a files and directories.

Fs_getcwd: This function converts the internal CWD stack which tracks each directory level from the root to the current point and returns it into a string path. This allows the programs to understand their location in the file system hierarchy in a UNIX path format.

Fs_setcwd: This function takes a path string, and uses parsePath to resolve it into a valid directory entry. It then loads the new directory into memory, updates the internal CWD pointer to this new location, and adjusts the CWD string stack to show the new path. This function is important for directory navigation within the shell.

PathCleaner: This function normalizes a path string by removing relative path tokens like "." and resolving ".." by adjusting the stack of the current path. It simplifies the input path string into a format, making it easier for parsePath to resolve it correctly. This function ensures that the internal CWD stack and any incoming paths are clean, and ready for parsing or command execution.

Fs_isFile: This function determines whether the provided path refers to a file. It first makes a copy of the path and uses parsePath() to resolve it to a valid directory entry. If the path exists and corresponds to a valid entry, the function checks the isDir return of the corresponding directory entry. If the return is not set to '1', the path is considered to be a file. The function returns 1 if it is a file, 0 if it's not, and -1 on error.

Fs_isDIr: This function checks whether a given path refers to a directory. It copies the path and uses parsePath() to locate the target in the directory structure. If found, it checks the isDir flag of the corresponding directory entry. If the return is '1', it is a directory. The function returns 1 if it is a directory, 0 if not, and -1 on error.

***toString():** This function builds a full path string from the components stored in the global stack (current working directory path). Each component is joined together by a / starting from the root. The path string is allocated or reallocated with enough space to hold all directory names with each having an additional / and a null terminator \0.

b_open: Starts off by ensuring the system is initialized by calling b_init(). Next the flags are checked in order to verify proper access to file, followed by an attempt to locate the file control block and will return an error if no file control blocks are available. Once the flags and the FCB are established, the filename gets run through parsePath() in order to locate the file. If the file does not exist and the O_CREAT flag is present, then createFile() gets called to make a new file

and will store the new directory entry. If the file is successfully found or created, then a buffer is allocated and sets the access mode. Finally, a file descriptor is returned for the newly opened/created file.

b_read: Takes in three parameters, a file descriptor, a buffer and a count. First verifies the file descriptor is valid and points to an open file. If the position is already at end of file, the function returns 0 to indicate no data can be read. Next it calculates how many bytes can actually be read based on file size and current position. If there is left over data in the internal buffer from the previous read, it copies as much from that buffer first. If the read is more than the size of the buffer, the buffer is bypassed and data is read as full blocks directly into user's buffer. Any remaining bytes left over after that are loaded one block at a time from disk and user is given only what is needed. For return, the function will show the number of bytes successfully read.

b_seek: The functions provides a way to reposition the file offset of an open file in the buffered I/O system. It takes in 3 parameters (file descriptor, offset and reference point). The reference can be SEEK_SET, SEEK_CUR or SEEK_END. The function firstly validates the file descriptor and offset, and afterwards calculates the new file position based on the reference point. If calculated position is within a valid bound, it updates the internal file position, invalidates the current buffer by resetting the index and length, and finally returns the new position.

b_write: This function allows buffered writing to a file. It takes in three parameters (file descriptor, buffer provided by user, and length of the buffer). Firstly, it validates the file descriptor and buffer. Afterwards, it writes data from the buffer provided by the user into an internal buffer. When the buffer is filled up, it is written to disk. The function updates the current file position and file size while the data is being written. It ultimately returns the total number of bytes written successfully.

b_close: The function ends operations on a buffered file and clean up resources. It first validates the file descriptor and ensures the internal buffer. If the internal buffer isn't empty, the buffer is flushed to disk and file position and file size are updated. At the end, the internal buffer is freed and all the fields in the file control block (FCB) are reset.

cmd_mv(): The function implements the move command which is used to change the location of a file or directory to a destination directory. The function takes in 3 command line arguments (“mv” itself, source directory, destination directory). The function validates these arguments and ensures that the source exists and destination directories exist and is a directory. It then loads the destination directory and its contents. It finds a free slot in and copies the source into the destination directory. The original source is cleared and both directories are written to disk.

fs_opendir: The function takes one parameter (pathname). The pathname is parsed to ensure the it is a valid directory. Once validates, DirHandle structure is allocated and initialized. DirHandle is then used to load all the directory entries from disk into memory. The entries are stored in a buffer in the handle, which is then wrapped within a fdDir structure. The initialized fdDir* is returned.

fs_readdir(): This function reads the next valid directory entry from a opened directory. DirHandle inside the fdDir is used to iterate through the directory entries. It skips empty entries. For each entry, a fs_diriteminfo is allocated and populated with the entry’s name and type (directory or regular file) and a pointer of the structure is returned.

fs_closedir(): The function cleans up resources associated with an open directory. It frees all the fields in the fdD ir that needed memory allocation, such as buffer and DirHandle.

mc(): The function takes one parameter (size) calls the malloc function with that size and returns the result. This is done to ensure that mc() in cmd_pwd works without making direct changes to fsshell.c.

Shell Commands(Working/Non-Working)

LS:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o dirLow.o dirLow.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o fsPath.o fsPath.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o dirLow.o fsFreeSpace.o mfs.o fsPath.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
gcc -o hexdump Hexdump/hexdump.c -g -I.
./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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
Prompt >
```

CP:

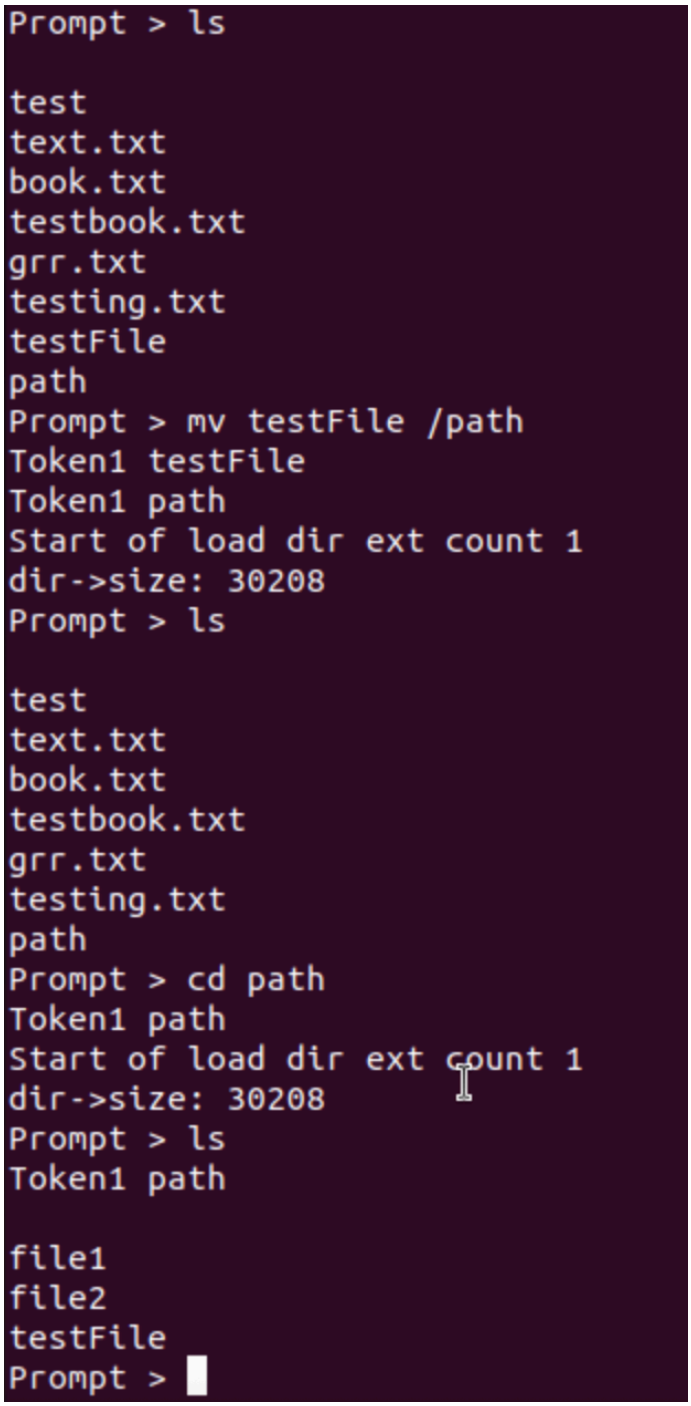
```
student@student:~/Desktop/csc415-filesystem-R3plus$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o dirLow.o dirLow.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o fsPath.o fsPath.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o dirLow.o fsFreeSpace.o mfs.o fsPath.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
gcc -o hexdump Hexdump/hexdump.c -g -I.
./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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
TESTING.txt
Prompt > cd TESTING
Prompt > ls

copy_text.txt
Prompt > cp book.txt /TESTING/copy_book.txt
Index: 3
Prompt > ls

copy_text.txt
copy_book.txt
Prompt >
```

MV:


Prompt > ls
test
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
testFile
path
Prompt > mv testFile /path
Token1 testFile
Token1 path
Start of load dir ext count 1
dir->size: 30208
Prompt > ls
test
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
path
Prompt > cd path
Token1 path
Start of load dir ext count 1
dir->size: 30208
Prompt > ls
Token1 path
file1
file2
testFile
Prompt >

MD:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o dirLow.o dirLow.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o fsPath.o fsPath.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o dirLow.o fsFreeSpace.o mfs.o fsPath.o b_io.o fsLow.o -g -I. -lm -l readline -l pthread
gcc -o hexdump Hexdump/hexdump.c -g -I.
./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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
Prompt > md TESTING
Index: 9
Prompt > ls

test
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
Prompt >
```

RM:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
TESTING.txt
Prompt > rm testing.txt
Prompt > ls

test
testbook.txt
grr.txt
wow.txt
TESTING
TESTING.txt
Prompt >
```

Touch:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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 TESTING.txt
Index: 10
Prompt > ls

test
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
TESTING.txt
Prompt >
```

Cat:
(Not working)

Cp2l:
(Not working)

Cp2fs:
(Not working)

Cd:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
TESTING.txt
Prompt > cd TESTING
Prompt > ls

Prompt >
```

Pwd:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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
text.txt
book.txt
testbook.txt
grr.txt
testing.txt
wow.txt
TESTING
TESTING.txt
Prompt > cd TESTING
Prompt > ls

Prompt > pwd
/TESTING/
Prompt >
```

History:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- 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 > history
history
Prompt > ls

test
testbook.txt
grr.txt
wow.txt
TESTING
TESTING.txt
Prompt > history
history
ls
history
Prompt > pwd
/
Prompt > history
history
ls
history
pwd
history
Prompt >
```

Help:

```
student@student:~/Desktop/csc415-filesystem-R3plug$ 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
Kentucky Kernels
System already initialized
|---------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|---------------------------------|
Prompt > help
ls      Lists the file in a directory
cp      Copies a file - source [dest]
mv      Moves a file - source dest
md      Make a new directory
rm      Removes a file or directory
touch   Touches/Creates a file
cat     Limited version of cat that displace the file to the console
cp2l    Copies a file from the test file system to the linux file system
cp2fs   Copies a file from the Linux file system to the test file system
cd      Changes directory
pwd     Prints the working directory
history Prints out the history
help    Prints out help
Prompt >
```

Work Allocation Table

Function Name	**Work Performed By**
createDir	Phillip Davis
writeDir	Phillip Davis
Extent Table Creation	Igor Tello
allocateFreeBlock	Preet Vithani
Volume Control Block Creation	Jared Aung
parsePath	Phillip Davis
loadDir	Phillip Davis
findInDir	Phillip Davis
fs_mkdir	Phillip Davis
findFreeDE	Phillip Davis
expandDirectory	Phillip Davis
safeFree	Phillip Davis
fs_delete	Igor Tello
freeBlocks	Igor Tello
b_open	Igor Tello
b_read	Igor Tello
initFreeSpace	Igor Tello
releaseBlocks	Igor Tello
getVCB	Igor Tello

Debugging all files	Igor Tello, Phillip Davis
Fs_rmdir	Preet Vithani
Fs_stat	Preet Vithani
Fs_getcwd	Preet Vithani
Fs_setcwd	Preet Vithani
PathCleaner	Preet Vithani
Fs_isFile	Preet Vithani
Fs_isDir	Preet Vithani
*toString()	Preet Vithani
Worked on PDF (WriteUp)	Igor Tello, Preet Vithani, Phillip Davis, Jared Aung
b_seek()	Jared Aung
b_write()	Jared Aung
b_close()	Jared Aung
VCB initialization	Jared Aung
cmd_mv()	Jared Aung
fs_opendir()	Jared Aung
fs_readdir()	Jared Aung
fs_closedir()	Jared Aung
mc()	Jared Aung
DirHandle Structure	Jared Aung