

San Francisco State University

File System Design

MKFS

THE GITHUB USER -> rf922

THE GITHUB LINK -> [filesystem-rf922](https://github.com/rf922/filesystem-rf922)

THE TEAM MEMBERS:

Anthony Silva, Rafael Fabiani, Rafael Sant Ana Leitao, Tyler Fulinara, Vinh Ngo
922907645, 922002234, 922965105, 921919541, 920984945

CSC-415-01 Operating Systems
Professor Robert Bierman

1 Introduction

Our team has successfully completed the CSC415 Group Term Assignment, which involved the design and implementation of a file system written in C. We created the crucial components of a file system, including the definition of objectives, the design of directory entry and volume structures, as well as the demarcation of free space. The project was divided into three distinct stages, each demanding a specific set of tasks and skills, and we have managed to actualize our theoretical designs into practical applications during the stages.

In the first stage we successfully implemented key components of a file system written in C. We focused on establishing the volume control block, managing the free space, defining the directory entry structure, and outlining the file system metadata. Our Volume Control Block serves as the essential repository of information about the volume for the file system. We accomplished this by putting together a structure that has fields for essential volume data storage. In managing free space, we adopted the File Allocation Table (FAT) methodology, drawing inspiration from FAT32 implementation. Our FAT table is embodied in an array, indexing every block within our approximately 10MB volume, with each block sized at 512 bytes. As for the directory entry structure, we devised a structure that efficiently organizes and manages directory entries, enhancing the overall performance of our file system. File System Metadata, an indispensable component of our system, includes various elements. The block size is set at 512 bytes, equating a single sector. Metadata also accounts for the total number of blocks on the system, along with the size of our FAT which has been calculated at 153 blocks. Other metadata elements include an entry signifying the starting block of the root cluster, the amount of free space, a magic number, a cap for entries per directory, a field holding bytes per sector, the number of reserved sectors, and the entries in our root directory.

The second stage focused on the process of volume formatting, covering several critical aspects. The team effectively established the Volume Control Block (VCB) by assigning it to block 0, following standard file system organization procedures. This achievement set the foundation for the next stages of the project. The team implemented the Free Space

management system. This system involved initializing free space and developing an allocation procedure. The allocation mechanism makes sure that free space can be efficiently assigned when necessary, supporting dynamic data management within the file system. Another significant step achieved in this phase was the construction and initialization of the Root Directory. The root directory was effectively developed with the necessary basic entries, including the "." and "..". The second stage of this project provided a robust foundation for the final phase, which will involve implementing file operations.

The final stage was integrating the fsshell into the file system, enabling the accomplishment of various file and directory operations. Throughout this phase, the team has succeeded in implementing several necessary functions in the file system. The incorporation of these functionalities into the file system means that users can now perform key directory operations such as making a directory (md), listing a directory (ls), printing the working directory (pwd), and changing the directory (cd).

To demonstrate the functionality and efficiency of our file system, we are presenting the following:

- All source files (.c and .h) used in the project.
- Modified Driver (fsshell.c) program which utilizes the header file for our file system exclusively.
- A Makefile to build the entire program.
- A comprehensive description of our file system.
- Issues we encountered and how we overcame them.
- Details on how our driver program operates.
- Screen shots demonstrating the execution of each of the commands listed in the readme.
- The volume file of our file system, capped at 10MB

All source files (.c and .h) used in the project.

- b_io.c - b_io.h - FAT.c - FAT.h - fsInit.c
- fsLow.c - fsLow.h - fsshell.c - mfs.c - mfs.h
- root_init.c - root_init.h - vcb_.c - vcb_.h

Core Functions

fsinit.c - Main Driver for File System Functions

The file fsInit.c serves as the main driver for the file system assignment and contains the implementation of functions to initialize and exit the file system.

int initFileSystem(uint64_t number_of_blocks, uint64_t block_size):

- This Function is used to initialize the file system.
- @param number_of_blocks - The number of blocks in the file system.
- @param block_size - The block size of each block in the file system
- @return - the next block in the chain from the fat array

void exitFileSystem():

- This Function is used to exit the file system.
- @return - void

mfs.c - Core Directory Functions

Allows us to affect the file system using a combination of these functions with calls. These functions allow us to move around in the file system and make changes like moving a file or setting the current working directory.

The main core functions in MFS.c are `parse_directory_path` and `get_target_directory()` which allow us to use a string of the path and get the correct information from which gets stored in an entry struct.

char *fs_getcwd(char *path, size_t size):

- This function changes the current working directory to the specified path. It allows us to navigate to a different directory within the file system.
- @param path - A char pointer representing the path given from user.
- @return - On success, this function returns the path of the current working directory.

int fs_setcwd(char *path):

- This function sets the current working to a new directory that is passed through.
- @param path - A char pointer representing the path given from user.
- @return - On success of setting current working directory, return 0
 - If the directory does not exist return -1

Directory_Entry *get_target_directory(Directory_Entry entry):

- This function searches for a directory entry that matches the provided directory name within the given array of directory entries. It iterates through each directory entry in the current directory until a match is found. If a match is found, it allocates memory for a new directory entry, copies the matching entry into it, and returns the new directory entry.

- @param entry - A Directory_Entry representing the directory the function is trying to get
- @return - On success of getting target directory, return 0
 - If the directory can't load return -1

int find_target_entry(Directory_Entry *current_dir_ent, char *token):

- This function gets the target entry
- @param current_dir_ent - A Directory_Entry representing the path for the current directory entry.
- @param token - A char pointer representing the token needed to find target entry
- @return - On success of setting current working directory, return 0
 - If the directory does not exist return -1

int parse_directory_path(char *path, parsed_entry *entry):

- This function is used to parse the given path and set the entry struct variables.
- @param path - A char pointer representing the string that gets parsed as the path.
- @param entry - A parsed_entry pointer representing the entry that holds the parent and name and index.
- @return - On success of parsing through and setting correct entry values, return 0.

get_empty_entry():

- This is a helper function to get a empty entry
- @param parent - A directory_entry that represents the parent of the empty entry you are trying to get.
- @return - On call, return index of parent where the entry is not active.

int fs_mkdir(const char *pathname, mode_t mode):

- This function is used for creating a new directory.
- @param pathname - A char pointer representing the string that gets parsed as the path.

- @param entry - A parsed_entry pointer representing the entry that holds the parent and name and index.
- @return - On success of parsing through and setting correct entry values, return 0, on failure, return -1.

int fs_rmdir(const char *pathname):

- This function is used for removing a directory.
- @param pathname - A char pointer representing the string that gets parsed as the path.
- @return - On success of removing directory, return 0.
 - On failure directory existing, on failure directory has stuff inside that can't be deleted, on file being passed in, return -1.

int fs_isFile(char *filename):

- This function is used to check if the given filename corresponds to a regular file.
- @param filename - A char pointer representing the string that gets parsed as the path (file).
- @return - On success if is a file, return 1. If not a file, return 0

int fs_isDir(char *pathname):

- This Function is used to check if the given pathname corresponds to a directory or not.
- @param pathname - A char pointer representing the string that gets parsed as the path (directory).
- @return - On success if is a directory, return 1. If not a directory, return 0.

int fs_mkfile(char *filename):

- This Function is used to create a new file and return success or not.
- @param filename - A char pointer representing the filename of the

new file.

- @return - On success of creation of file, return 0. If no name given, or if file name already exists, or if no file created, return -1.

int fs_mvFile(char *filename, char *pathname):

- This Function is used to move a file (only a file not a directory) to another directory.

- @param filename - A char pointer representing the filename of the file you want move (src).

- @param pathname - A char pointer representing the pathname of the directory you want to move to (destintation).

- @return - On success of moving a file, return 0. If src is a directory, or if src doesn't exist, or if destintation is a file, or if destintation does not exist, or if failure to write to disk, return -1.

int fs_delete(char *filename):

- This Function is used to deletes a file (can't delete a directory).

- @param filename - A char pointer representing the filename of the file you want to delete.

- @return - On success of deleting a file, return 0. If it is a directory, or if not a valid file, or if failure to write to disk, return -1.

int fs_renameDirectoryOrFile(const char *path, const *newName):

- This Function is used to rename a file or directory.

- @param path - A char pointer representing the path of the file/dir you want to rename.

- @return - On success of renaming a file or directory, return 0. If name already exists, return -1.

***** Directory iteration functions *****

fdDir *fs_opendir(const char *pathname):

- This Function is used to open a directory.

- @param filename - A char pointer representing the pathname of the directory you want to open.

- @return - On success of creation of file, return 0. If it is not a directory, or if directory doesn't exist, or if failure to bring to memory, return -1.

***** Helper functions for readdir *****

int is_used(Directory_Entry entry):

- This Function is check if directory entry is used.
- @param entry - A directory entry representing the entry that you want to check if used.
- @return - If used, return 1. If not used, return 0.

int is_dir(Directory_Entry entry):

- This Function is check if the directory entry is an directory.
- @param entry - A directory entry representing the entry that you want to check if is a directory.
- @return - If is a directory, return 1. If is not a directory, return 0.

struct fs_direntinfo *fs_readdir(fdDir *dirp):

- This Function is used to read a directory and returns it as a struct to get it item info.
- @param dirp - A fdDir pointer representing the directory that you want to read.
- @return - If a directory item is found, return a pointer to fs_direntinfo containing item info.
- If no more directory items are available to read, return NULL.

int fs_closedir(fdDir *dirp):

- This Function is used to Clean up of a directory.
- @param dirp - A fdDir pointer representing the directory that you want to close.
- @return - Return 0 on sucess, and -1 if NULL.

int fs_stat(const char *path, struct fs_stat *buf):

- This Function is used to update buf sizes.
- @param path - A char pointer of the path we want to use to update.
- @param buf - A struct fs_stat representing the buffer we want to update.
- @return - if success return 0. If entry does not exist, return -1.

int parse_directory_path(char *path, parsed_entry *parent_dir):

- This function is used to parse the given path and set the entry struct variables
- @return - If no matching directory entry is found or an error occurs, it returns NULL.

b_io.c - Core I/O Functions

file_info *get_file_info(char *fname):

- The function retrieves the information of a file given its name.
- @param fname - A pointer to a string containing the name of the file.
- This string is expected to include the file's complete path in the file system.
- @return - On success, this function returns a pointer to a `file_info` struct.
- If fname is a directory, it returns NULL.

void b_init():

- The function initializes the file system by setting up the File Control Block array.

b_io_fd b_getFCB():

- The function returns an available File Control Block (FCB) element index.

- @return - If there is a free FCB element, it returns the index of that element.
- If all FCB elements are in use, it returns -1 to indicate that all FCBs are occupied.

b_io_fd b_open(char *filename, int flags):

- The function writes data from the buffer to the buffered file associated with the given file descriptor.
- @param fd - The file descriptor of the buffered file where data will be written.
- @param buffer - A pointer to the buffer containing the data to be written.
- @param count - The number of bytes to be written from the buffer.
- @return - On success, the function returns the number of bytes written to the file.
- If an error occurs during writing, it returns -1.

int b_write(b_io_fd fd, char *buffer, int count):

- The function writes data from the buffer to the buffered file associated with the given file descriptor.
- @param fd - The file descriptor of the buffered file where data will be written.
- @param buffer - A pointer to the buffer containing the data to be written.
- @param count - The number of bytes to be written from the buffer.
- @return - On success, the function returns the number of bytes written to the file.
- If an error occurs during writing, it returns -1.

int b_read(b_io_fd fd, char *buffer, int count):

- The function reads data from the buffered file associated with the given file descriptor and stores it in the buffer.
- @param fd - The file descriptor of the buffered file from which data will be read.

- @param buffer - A pointer to the buffer where the data will be stored.
- @param count - The number of bytes to be read from the file.
- @return - On success, the function returns the total number of bytes read from the file.
- If the end of file (EOF) is reached during reading, it returns the total number of bytes read until EOF.
- If an error occurs during reading, it returns -1.

int b_close(b_io_fd fd):

- The function closes the buffered file associated with the given file descriptor.
- @param fd - The file descriptor of the buffered file to be closed.
- @return - On success, the function returns 0.
- If the file descriptor is invalid, it returns -1.

int get_last_block(int location):

- The function retrieves the last block number of a file given the location of a block in the file system.
- @param location - The location of a block in the file system.
- @return - The block number of the last block in the file (given its location).

FAT.c - Core FAT Functions

int fat_init(uint64_t number_of_blocks, uint64_t block_size):

- This Function is used to initialize the FAT system
- @param number_of_blocks - A uint64_t of number of blocks needed for the fat
- @param block_size - A uint64_t of block size for each individual block for the fat
- @return - if successfully initialized the fat return 0
- if failed to allocate memory for fat return -1

int fat_read_from_disk():

- This Function is used to read FAT from disk and stores it in memory
- @return - succesfully read fat from disk and allocated memory

return 0

- if failed to allocate memory for fat return -1

uint32_t find_free_block():

- This Function is used to searches the fat array for a free block
- @return - succesful return index
- if otherwise return -1

uint32_t allocate_blocks(int blocks_needed):

- This Function is used to Allocate a given number of blocks, returns the starting block.

- @param blocks_needed - A int the contains how many blocks needed for allocating

- @return - succesfully allocated return the starting block
- if failed to allocate blocks return -1

void allocate_additional_blocks(uint32_t first_block, int blocks_to_allocate):

- This Function is used to Allocate a given number of blocks, returns the void

- @param first_block - A first block in chain
- @param blocks_to_allocate - how many blocks you need to allocate

uint32_t release_blocks(int first_block):

- This Function is used to frees blocks using the index stored at the given position in fat unti EOF

- @param first_block - A first block in chain
- @return - amount of blocks that you freed

uint32_t get_next_block(int current_block):

- This Function is used to retrieve the next block in the chainlocation of a block in the file system.

- @param current_block - THe current block in chain
- @return - the next block in the chain from the fat array

uint32_t get_total_free_blocks():

- This Function is used to retrieve the Count of free blocks in the fat
- @return - the total amount of free blocks in fat

int to_blocks(int bytes):

- This Function is used to compute blocks from blocks given
- @param current_block - The amount of bytes you want to convert
- @return - the converted amount of blocks

void update_fat_on_disk():

- This Function is used to update fat

root_init.c - Core Root Initialization Functions

int load_root():

- The function loads the root directory.
- @return - On success, this function returns a 0.
- If failure on loading root return -1.

Directory_Entry * init_directory(uint64_t block_size, Directory_Entry *parent, char *name):

- The function initalizes a directory.
- @return - On success return the directory initalized.
 - If failure to write to disk return NULL.
 - If path length exceeds limit return NULL.

int read_from_disk(void * buffer, int start_block, int blocks_need, int block_size):

- This Function reads data from disk and stores it in the given buffer.
- @param buffer - A void pointer to the buffer where the read data will be stored.
- @param start_block - The starting block number from where to read data.
- @param blocks_need - The number of blocks to be read from disk.
- @param block_size - The size of each block in bytes.
- @return - If the read operation is successful, return 0.
 - If the read operation fails, return -1.

int write_to_disk(void * buffer, int start_block, int blocks_need, int block_size):

- This Function writes data from the given buffer to the disk.
- @param buffer - A void pointer to the buffer containing the data to be written.
- @param start_block - The starting block number on the disk where data will be written.
- @param blocks_need - The number of blocks to be written to the disk.
- @param block_size - The size of each block in bytes.
- @return - If the write operation is successful, return 0.
 - If the write operation fails, return -1.

vcb.c - Core FAT Functions

int vcb_read_from_disk(VCB *vcb)

- This helper function is used for reading vcb from disk
- @param vcb - A vcb representing which vcb you want to read
- @return - On success of reading the volume control block from disk return 0
- On failure to read VCB from disk, return -1

int vcb_is_init()

- This helper function is used to check if the vcb is initialized
- @return - If vcb is initialized return 0
- If Volume control block is not initialized return -1

Structures for our File System

// Structure to store file information.

```
typedef struct file_info
```

```
{
    char file_name[NAME_MAX_LENGTH]; // file name
    int file_size;                    // file size in bytes
    int location;                     // starting logical block in disk
    int blocks;                       // total blocks of file in disk
    Directory_Entry *de;              // used for correctly updating
    directory information
} file_info;
```

// Definition of the File Control Block structure.

```
typedef struct b_fcb
```

```
{
    file_info *fi;                    // low level system file info
    char *buf;                        // holds the open file buffer
    int index;                        // holds the current position in the buffer
}
```



```
    int buflen;                // holds how many valid bytes are in the
buffer
    int current_location; // current block location
    int blocks_read;      // blocks read so far
    int file_size_index;  // file offset
    int flags;            // mark the purpose when open the file
} b_fcb;
```

```
typedef struct partitionInfo
{
    char volumePrefix[sizeof(PART_CAPTION) + 2];
    uint64_t signature;
    uint64_t volumesize;
    uint64_t blocksize;
    uint64_t numberOfBlocks;
    char *filename; // Never written - always assigned on start
    int fd;         // Never Written - file descriptor
    uint64_t signature2;
    char volumeName[];
} partitionInfo_t, *partitionInfo_p;
```

// This structure is returned by fs_readdir to provide the caller with information

// about each file as it iterates through a directory

```
struct fs_diriteminfo
{
    unsigned short d_reclen; /* length of this record */
    unsigned char fileType;
    char d_name[256]; /* filename max filename is 255 characters
*/
};
```

// This is a private structure used only by fs_opendir, fs_readdir, and fs_closedir

// Think of this like a file descriptor but for a directory - one can only read

```
// from a directory. This structure helps you (the file system) keep track of
// which directory entry you are currently processing so that everytime the
// caller
```

```
// calls the function readdir, you give the next entry in the directory
```

```
typedef struct
```

```
{
    unsigned short d_reclen;    /* length of this record */
    unsigned short dirEntryPosition; /* which directory entry position, like
file pos */
```

```
    Directory_Entry * directory; /* Pointer to the loaded directory you want
to iterate */
```

```
    struct fs_diriteminfo * di;    /* Pointer to the structure you return
from read */
```

```
    } fdDir;
```

```
// helper structure for parse path function
```

```
// this structure contain the directory entry of the parent
```

```
// and the index to the target directory entry
```

```
typedef struct
```

```
{
    Directory_Entry *parent;
    int index;
    char * name;
} parsed_entry;
```

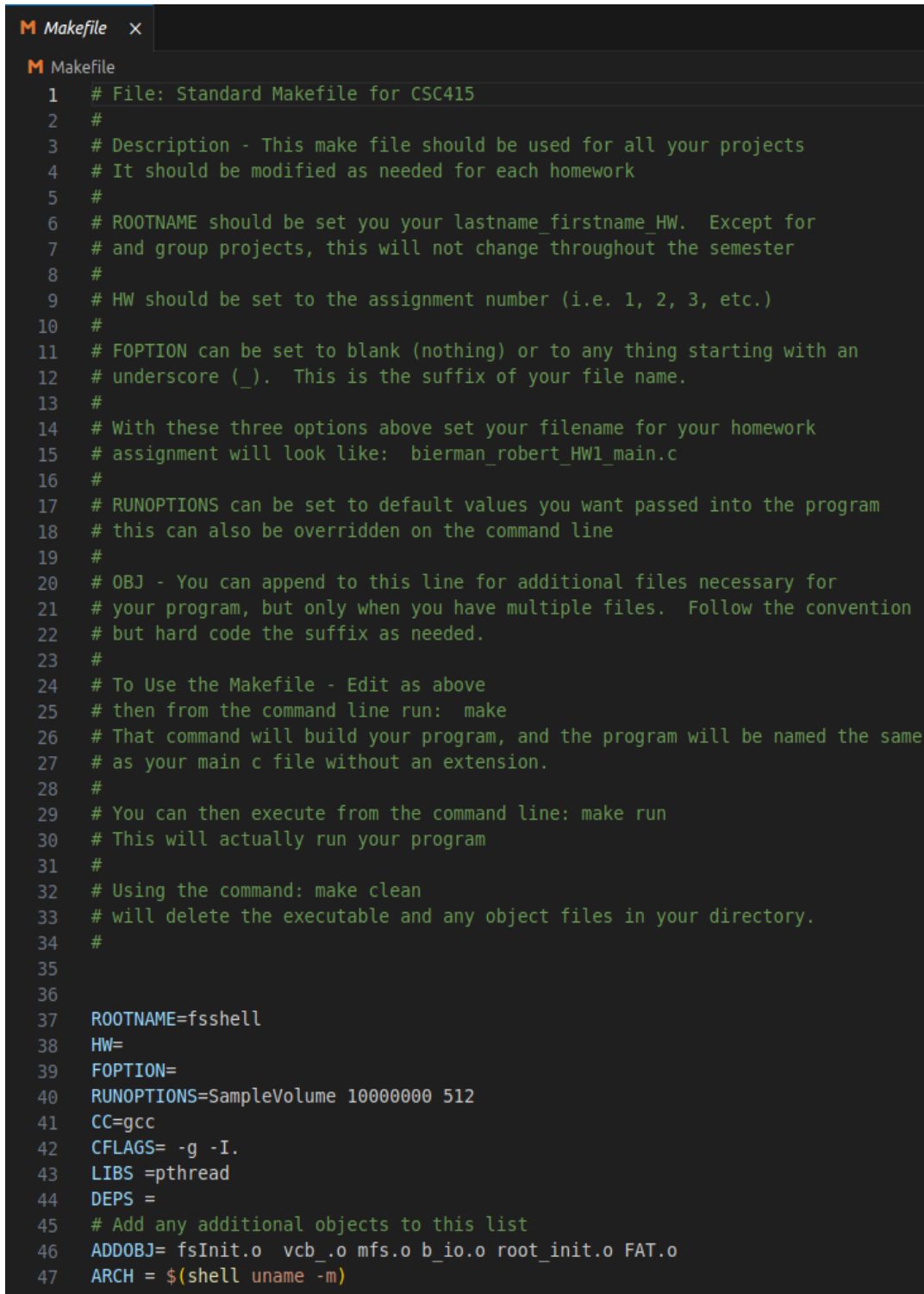
```
// This is the structure that is filled in from a call to fs_stat
```

```
struct fs_stat
```

```
{
    off_t    st_size;                /* total size, in bytes */
    blksize_t st_blksize;           /* blocksize for file system I/O */
    blkcnt_t st_blocks;             /* number of 512B blocks allocated */
    time_t   st_atimes;             /* time of last access */
    time_t   st_mtime;             /* time of last modification */
    time_t   st_createtime;         /* time of last status change */
```

```
    /* add additional attributes here for your file system */  
};  
  
typedef struct Directory_Entry {  
    char dir_name[NAME_MAX_LENGTH];  
    char path[MAX_PATH_LENGTH];  
    uint32_t dir_attr;  
    uint32_t dir_first_cluster;  
    uint32_t dir_file_size;  
} Directory_Entry;  
  
typedef struct VCB { //size of fields, desc of field  
    uint32_t total_blocks_32;    // 4 bytes, the total number of blocks in the  
file system  
    uint32_t FAT_size_32; // 4 bytes, total blocks in the FAT  
    uint32_t root_cluster;    // 4 bytes, location of the root  
    uint32_t free_space;      // 4 bytes, amount of free blocks  
    uint32_t magic_number;    // 4 bytes, Magic Number  
    uint32_t entries_per_dir; // 4 bytes  
    uint16_t bytes_per_block; // 2 bytes, blockSize,  
    uint16_t reserved_blocks_count; // 2 bytes, reserved blocks count  
} VCB;
```

A Makefile to build the entire program.



```
1 # File: Standard Makefile for CSC415
2 #
3 # Description - This make file should be used for all your projects
4 # It should be modified as needed for each homework
5 #
6 # ROOTNAME should be set you your lastname_firstname_HW. Except for
7 # and group projects, this will not change throughout the semester
8 #
9 # HW should be set to the assignment number (i.e. 1, 2, 3, etc.)
10 #
11 # FOPTION can be set to blank (nothing) or to any thing starting with an
12 # underscore (_). This is the suffix of your file name.
13 #
14 # With these three options above set your filename for your homework
15 # assignment will look like: bierman_robert_HW1_main.c
16 #
17 # RUNOPTIONS can be set to default values you want passed into the program
18 # this can also be overridden on the command line
19 #
20 # OBJ - You can append to this line for additional files necessary for
21 # your program, but only when you have multiple files. Follow the convention
22 # but hard code the suffix as needed.
23 #
24 # To Use the Makefile - Edit as above
25 # then from the command line run: make
26 # That command will build your program, and the program will be named the same
27 # as your main c file without an extension.
28 #
29 # You can then execute from the command line: make run
30 # This will actually run your program
31 #
32 # Using the command: make clean
33 # will delete the executable and any object files in your directory.
34 #
35
36
37 ROOTNAME=fsshell
38 HW=
39 FOPTION=
40 RUNOPTIONS=SampleVolume 10000000 512
41 CC=gcc
42 CFLAGS= -g -I.
43 LIBS =pthread
44 DEPS =
45 # Add any additional objects to this list
46 ADDOBJ= fsInit.o vcb_.o mfs.o b_io.o root_init.o FAT.o
47 ARCH = $(shell uname -m)
```

```
M Makefile x
M Makefile
35
36
37 ROOTNAME=fsshell
38 HW=
39 FOPTION=
40 RUNOPTIONS=SampleVolume 10000000 512
41 CC=gcc
42 CFLAGS= -g -I.
43 LIBS =pthread
44 DEPS =
45 # Add any additional objects to this list
46 ADDOBJ= fsInit.o vcb_.o mfs.o b_io.o root_init.o FAT.o
47 ARCH = $(shell uname -m)
48
49 ifeq ($(ARCH), aarch64)
50 |   ARCHOBJ=fsLowM1.o
51 else
52 |   ARCHOBJ=fsLow.o
53 endif
54
55 OBJ = $(ROOTNAME)$(HW)$(FOPTION).o $(ADDOBJ) $(ARCHOBJ)
56
57 %.o: %.c $(DEPS)
58 |   $(CC) -c -o $@ $< $(CFLAGS)
59
60 $(ROOTNAME)$(HW)$(FOPTION): $(OBJ)
61 |   $(CC) -o $@ $^ $(CFLAGS) -lm -l readline -l $(LIBS)
62
63 clean:
64 |   rm $(ROOTNAME)$(HW)$(FOPTION).o $(ADDOBJ) $(ROOTNAME)$(HW)$(FOPTION)
65
66 run: $(ROOTNAME)$(HW)$(FOPTION)
67 |   ./$(ROOTNAME)$(HW)$(FOPTION) $(RUNOPTIONS)
68
69 vrun: $(ROOTNAME)$(HW)$(FOPTION)
70 |   valgrind ./$(ROOTNAME)$(HW)$(FOPTION) $(RUNOPTIONS)
71
```

A description of our file system

The file system our team developed is a project rooted in C, organized into three major stages. The Volume Control Block (VCB), a structured data repository that stores vital data about the volume, is the system's basis. The VCB, which is situated at block 0, is crucial to managing and directing the file system's operations.

The second critical component is the Free Space management, which we implemented using the File Allocation Table (FAT) methodology, inspired by the FAT32 architecture. This mechanism is essential for our file system's effective space management, which makes it possible to handle dynamic data. With a block size of 512 bytes, our FAT table is effectively an array that indexes every block on the 10MB volume. Our system also includes a well-structured directory entry mechanism. We have designed this structure to efficiently manage directory entries, enhancing our file system's overall operational efficiency. This is complemented by an array of metadata, including block size, total number of blocks, the size of our FAT, the root cluster's starting block, free space, a magic number, a cap for entries per directory, bytes per sector, the number of reserved sectors, and the entries in the root directory.

In the initial stage, we focused on volume formatting, which included writing and initializing the VCB, Free Space, and the Root Directory. This critical process equipped our file system to support dynamic data management and ensured readiness for subsequent stages of development.

The final stage involved integrating fsshell into the file system. This integration endowed our system with the ability to perform key directory operations, including creating directories (md), listing directories (ls), displaying the working directory (pwd), and changing directories (cd).

In summary, our file system project exhibits a successful implementation of fundamental file system components. With a well-defined VCB, a FAT-based Free Space management system, an efficient directory entry structure, and an array of essential metadata, our file system offers a

comprehensive solution for managing data in a structured, efficient manner. We have successfully navigated the process of volume formatting and effectively integrated the fsshell to enable an array of directory operations. As a result, we have developed a robust and operational file system using C.

Issues we encountered and how we overcame them

Our team faced a number of difficulties while working on this project, which put our confidence, problem-solving abilities, and capacity for collaboration to the test. However, we believe that getting over these challenges was essential to our group's learning process and helped our file system be implemented successfully.

We initially struggled to comprehend and define the project's requirements. Given the complex characteristics and many relationships between different elements, designing a file system from scratch was a difficult task. The choice to use the FAT32 model to manage free space increased our learning curve because we had to comprehend its operation in great detail. Through careful research, creative thinking, and regular team meetings, we were able to overcome this challenge. We carefully examined the specifications and made an organized effort to comprehend the internal operations of each component.

Implementing the Volume Control Block (VCB) was among the major difficulties we encountered. It was a challenge to determine which characteristics to include and how best to handle the VCB. We were able to design a working VCB that successfully saves essential volume information thanks to extensive research and a process of trial and error. Additionally, managing free space proved to be a challenging task. Initializing empty slots and creating an allocation technique required in-depth comprehension and exact execution. We had issues handling block allocation and deallocation as well as appropriately indexing each block in our volume. We were able to create a functional FAT-based free space management system through intensive debugging sessions and frequent iterations.

We also ran into issues with the creation of the directory entry structure. A deep understanding of file systems and data structures were necessary to effectively organize and maintain directory entries. We used a variety of

sources and our combined experience to create a successful directory entry structure that supports the efficient operation of the file system.

Integrating the fsshell into our file system was the final stage of the project, and naturally, the process was very complex. It was difficult to ensure that every important function was successfully implemented because they all depended on one another. To make sure every function operated as intended, we went through numerous iterations of rigorous testing and debugging. The coordination of tasks among team members was still another challenge we encountered, especially given the varied schedules and available time. By creating a clear line of communication and setting up frequent meetings to coordinate our progress and make plans, we were able to overcome this.

So, reflecting on our journey, these challenges turned out to be the stepping stones toward our success. As a result of overcoming these challenges, our journey was even more satisfying and enjoyable. In conclusion, the fact that we were able to finish our file system successfully demonstrates how much we have improved as programmers, and the lessons we have learned will help us in the projects we work on in the future.

Issue #1 Understanding how the FAT implementation works

Microsoft first introduced FAT in Windows 95 OSR2. It supports smaller cluster sizes than FAT16, leading to more efficient space utilization on FAT32 drives. take a look over this as well for ideas

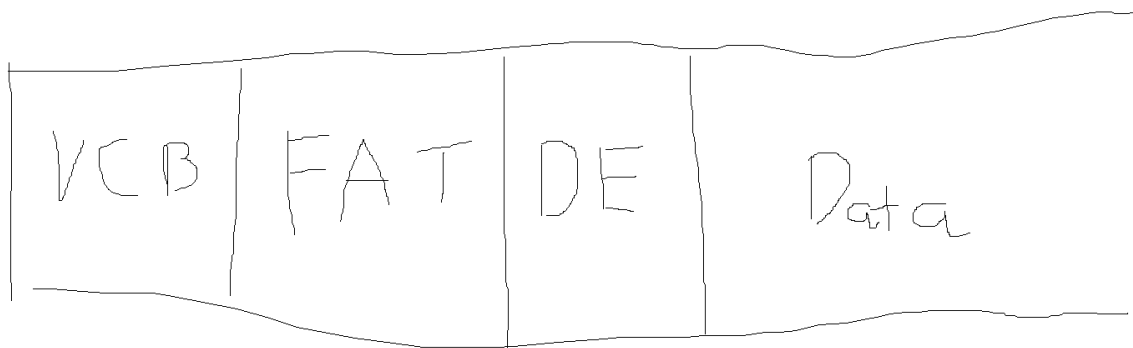
File Allocation Table (FAT): The FAT is a map of the entire file system, it keeps track of which blocks are used and which are free. It also links blocks used by the same file together

Directory Entries (DE): These contain metadata about files and directories, such as name, size, timestamp, and starting block. The root directory entry is special; it's the 'starting point' for finding any file in the system.

Data Blocks: This is where the actual file data is stored. Each block has a unique address, which is used by the FAT and directory entries to reference the data.

Solution: Looked up how the fat32 and fat16 worked and worked off that, we realized that we don't need to reinvent the wheel

Issue #2 Understanding the structure of the file control block and the structure of our Directory entries.



Solution: In person meetings, we drew out how the index and fat would work in our file system

Issue #3 Parse Path issue #1 Changing arguments of parse path and causing issues with other functions that use parse path, Changing parse path return value. Originally for the parse path function we didn't have the entry pointer as a second parameter and just passed in the pathname and returned the directory entry. This caused problems with other functions and would require checking if the entry returned was NULL.

```
//wants to be removed
parsed_entry entry;
if (parse_directory_path(strdup(pathname), &entry) == -1) {
    printf("[RMDIR] invalid path\n");
    free_dir(entry.parent);
    return -1;
}
```

Solution: We fixed this by adding the entry struct as a second parameter and returning a zero for success and a negative one for not successful. The return of an integer also helps keep consistency for all functions in the file system, so there is less confusion.

Issue #4 Parse Path issue #2 Introduction of keeping the parent in addition to the child. The problem we quickly realized in only returning the `Directory_Entry` was that, while we could edit variables in the structure, we needed the parent directory to perform file operations such as delete or move.

```
//We set the start_dir to the parent
//This will allow us to save the parent as move to the new directory
//that will be gotten from the tokenization
Directory_Entry * parent = start_dir;
```

Solution: We created the `parsed_entry` struct that contained the parent `Directory_Entry`, an index to the child entry, and the entry's name.

Issue #5 Parse Path issue #3 Checking if it is absolute or relative. During testing of the parse path function, it sometimes stored the correct entry, and other times did not store an entry at all. We would also sometimes get a segmentation fault. The problem was the logic to check if the path String was absolute or relative was wrong, mainly due to trying to implement the logic from scratch instead of following the example from class.

```
//Creates a start directory that represents the start point
Directory_Entry *start_dir;
// checking for starting point
if ( path[0] != '/' ) { //Checking if the path is the current directory
    start_dir = current_directory;
} else { //otherwise it is the root_directory
    start_dir = root_directory;
}
```

Solution: We adopted the logic which checked to see if the first character was a slash. If a slash, we know the path is absolute, and if not, the path is relative.

Issue #6 Get Target Directory issue. To separate logic and not have redundant code, we created a `get_target_directory()` that could be used to retrieve a directory entry from a directory. But the function would always return an empty directory. After much troubleshooting, we realized that we used `malloc` to store the entry, which created an empty entry to the found entry. Also that part of the logic needed to be its own function.

```
Directory_Entry *get_t_d(Directory_Entry *current_dir_ent, char *token)
{
    printf("[ GET DIRECTORY INDEX] : Getting target directory, token: %s\n", token);

    Directory_Entry *entries = malloc(DIRECTORY_MAX_LENGTH * 512);
    if (entries == NULL)
    {
        printf("[ GET DIRECTORY INDEX ] : Failed to allocate memory for entries.\n");
        return NULL;
    }

    int readResult = LBaread(entries, 1, current_dir_ent->dir_first_cluster);
    if (readResult < 0)
    {
        printf("[ GET DIRECTORY INDEX] : LBaread failed.\n");
        free(entries);
        return NULL;
    }

    printf("[ GET DIRECTORY INDEX] : Read entries from disk at location: %d\n", current_dir_ent->dir_first_cluster);

    Directory_Entry *retVal = malloc(sizeof(Directory_Entry));
    if (!retVal)
    {
        free(entries);
        return NULL;
    }

    for (int i = 0; i < entries_per_dir; i++)
    {
        printf("[ GET DIRECTORY INDEX ] : Checking entry number: %d, name: %s, attribute: %d\n", i, entries[i].dir_name, entries[i].dir_attr);

        if (strcmp(entries[i].dir_name, token) == 0)
        {
            printf("[ GET DIRECTORY INDEX ] : Found matching directory entry at index: %d\n", i);
            *retVal = entries[i];
            free(entries);
            return retVal;
        }
    }

    printf("[ GET DIRECTORY INDEX ] : No matching directory entry found. Returning NULL.\n");

    free(entries);
    return NULL;
}
```

Solution: We changed `get_target_directory()` to get the parent entry and `find_target_entry()` to get the index of the entry we were looking for.

Issue #7 Parse path still not functioning correctly when testing. After all the changes made to the parse path function, things still weren't working, and other people in the group needed it to test their code in the mfs file.

```
//First Check setup for parse_directory_path return value
//0 Succeeds, -1 fails
if ( parse_directory_path(path, &entry) == -1) {
    printf("[ FS SETCWD ]: Invalid path.\n");
    free_dir(entry.parent);
    return -1;
}
```

Solution: Vinh copied the parse path example given in class and adapted it to our filesystem, which worked correctly.

Issue #8 forgot to use mode in fs_mkdir. Forgetting to use the mode in the fs_mkdir which made have to go back and add it in during debugging at the end of the project

```
int fs_mkdir(const char *pathname, mode_t mode)
{
    //represents the entry that holds entry.parent and entry.name to use
    //to create a new directory
    parsed_entry entry;

    //First Check setup for parse_directory_path return value
    //0 Succeeds, -1 fails
    if ( parse_directory_path(pathname, &entry) == -1) {
        printf(" [MKDIR] invalid path\n");
        return -1;
    }

    //Check for if is the name of the entry is an empty string OR
    //Check for if the index of the entex is not error or -1 OR
    //Check if the parent of the entry is not NULL and exists as
    //either root or another directory
    if ( strcmp(entry.name, "") == 0 || entry.index != -1 || entry.parent == NULL){
        free_dir(entry.parent);
        printf("[MKDIR] error\n");
        return -1;
    }
}
```

Issue #9 Write Problem 1 Originally we thought that we were writing to the block in the hex dump, however we realized that one of our team members made the first index of the entry to hold the file's information. This is set to entry by default to show that it has been properly initialized. This problem came from miscommunication from not writing how everything worked out beforehand.

```
student@student-VirtualBox:~/csc415-filesystem-rf922$ Hexdump/hexdump.linux --start 224 --count 3 SampleVolume
Dumping file SampleVolume, starting at block 224 for 3 blocks:

01C000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0E0: 00 00 00 00 00 00 00 00 65 6E 74 72 79 00 00 00 | .....entry...
01C0F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
```

Issue #10 Write Problem 2 Testing the write function using the cp2fs. We had an issue where we were not getting right values on the hex dump. For some reason the hex dump for cp2fs would work with smaller text files. However, when we put bigger text files the first couple of blocks would be overridden.

```
//grab the current block, store it in our buffer
LBAreadd(fcbArray[fd].buf, 1,fcbArray[fd].current_location );
```

Solution: We realized that we were losing some of the buffer, so we ended up having to do a LBAreadd, to make sure we got the whole buffer and not just the last part, because we are always a full count of 1 block. By doing this LBAreadd in write, we are making sure that the buffer is correctly updated and has everything in it.

Issue #11 Making reading and writing connect to each other. Originally we had done our read and functions separately. Some focused on read and other focused on write. When we compared, the logic of the two were

different, so when we ended up having to debug, we had change some things over and over again.

```
int bytes_written = 0;
int bytes_into_block = 0;
int to_be_written = count; //600
uint32_t initial_block = fcbArray[fd].current_location;
fcbArray[fd].fi->de->dir_first_cluster = initial_block;
int total_blocks = 1;

while (bytes_written < count)
{
    bytes_into_block = B_CHUNK_SIZE - fcbArray[fd].index; //512

    // Check if the buffer can fit entirely in the current block
    if (to_be_written <= bytes_into_block)
    {
        // Write the entire buffer to the current block
        memcpy(fcbArray[fd].buf + fcbArray[fd].index, buffer + bytes_written, to_be_written);

        printf("fcbArray >>> 1 buffer: %d \n", fcbArray[fd].buf);

        // Write the current block to the disk using LBAwrite
        if (LBAwrite(fcbArray[fd].buf + fcbArray[fd].index, 1, fcbArray[fd].current_location) == 0)
        {
            printf("fcbArray index: %d \n", fcbArray[fd].current_location);
            printf("fcbArray buffer: %d \n", fcbArray[fd].buf);
            printf("fcbArray index: %d \n", fcbArray[fd].index);
            printf("[b_write] 1 LBAwrite failed");
            break;
        }

        // Update the buffer index and file offset
        fcbArray[fd].index += to_be_written;
        fcbArray[fd].file_size_index += to_be_written;
        bytes_written += to_be_written;
        to_be_written = 0; // All data is written
    }
    else
    {
```

```
else
{
    printf("Inside the else-----\n");
    // Write as much as possible to the current block
    memcpy(fcbArray[fd].buf + fcbArray[fd].index, buffer + bytes_written, bytes_into_block);

    // Write the current block to the disk using LBAwrite
    if (LBAwrite(fcbArray[fd].buf + fcbArray[fd].index, 1, fcbArray[fd].current_location) == 0)
    {
        printf("[b_write] 2 LBAwrite failed");
        return -1;
    }

    // Update the buffer index, file offset, and bytes_written
    fcbArray[fd].index += bytes_into_block; //512 in the block
    fcbArray[fd].file_size_index += bytes_into_block;
    bytes_written += bytes_into_block; // 512
    to_be_written -= bytes_into_block; // 88

    // Check if we need to allocate more blocks
    if (to_be_written > 0)
    {
        // Find the next block in the file using FAT
        uint32_t next_block = get_next_block(fcbArray[fd].current_location);
        printf("This is the next_block before the if: %d \n", next_block);

        // If there is no next block, we need to allocate a new block
        if (next_block == EOF_BLOCK)
        {
            // ...
        }
    }
}
```

Solution: Our solution was basically to restart read and write all over again. However, we work together as a team to incorporate details that we had. Basically we were on a Zoom call going line by line making sure each one understood how the read and write worked. This allowed us to iron out any problems, since we each did read and write already before. This brought us on one page and made code and debugging much faster.

Issue #12 Dealing with their buffer being limited to 200, so multiple reads and writes would happen and not just one big read or one big write. The assumption we had was that BUFFERLEN would be variable, like in assignment 5, so when making calculations like adjusting the count or using LBAwrite or LBAread. The count value would be greater than the buffer size, and LBAwrite and LBAread would write and read from other parts of memory.

```
#define BUFFERLEN 200
```

Solution: In b_write() the IO buffer first needed to store the block being written to and then memcpy the user buffer to IO buffer from the current position in the block. In b_read(), instead of using B_CHUNK_SIZE we used the count to handle part1, part2, and part3.

Issue #13 File Information not showing up in the hex dump in the first index of the entries that we created in our free space. We initially set the value of

the name of the file name to be entry to show that it can be used, however when we try to use it, only the file contents gets updated and not the file information like the name and size in the hex dump.

```
Prompt > cat a
entry name: a
file name a
file size 550
The file location: 224
The file_size_index: 0
```

```
system exiting
student@student-VirtualBox:~/csc415-filesystem-rf922$ Hexdump/hexdump.linux --start 224 --count 3 SampleVolume
Dumping file SampleVolume, starting at block 224 for 3 blocks:

01C000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C0E0: 00 00 00 00 00 00 00 00 65 6E 74 72 79 00 00 00 | .....entry...
01C0F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C110: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C120: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C140: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C1F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C200: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAA
```

Details on how our driver program operates.

Our driver program, known as fsshell.c, is an instrumental piece of our file system. It serves as the user interface, accepting user inputs and converting them into tasks for our file system to carry out. The fsshell.c program begins by incorporating necessary libraries and files for its functioning, such as "fsLow.h" and "mfs.h". It also defines a set of constants for various features, each corresponding to a command, and a flag CMDXXXX_ON that determines whether the command is currently active. These flags enable us to manage the functionality of the program, enabling certain commands only when they're ready to be tested.

The dispatch_t structure is crucial. It connects user-inputted commands to their corresponding functions in the program, along with a description of the command. All of

these are housed within an array `dispatchTable[]`, which acts as the main command registry for the program.

The `fsshell.c` program initiates the file system with the ``initFileSystem`` function in `fsInit.c`, which sets up the required structures and the system for operations. It then enters a loop, waiting for user commands, reading them, parsing the commands and their arguments, and dispatching them to the appropriate functions for execution.

User commands map to specific functions in the `dispatchTable`. For example, when a user enters the `'ls'` command, the `cmd_ls` function is activated. This function lists the contents of a directory by invoking the ``fs_readdir`` function from the file system and displaying each item. If the user inputs the `'cd'` command, the `cmd_cd` function alters the working directory by calling the ``fs_setcwd`` function.

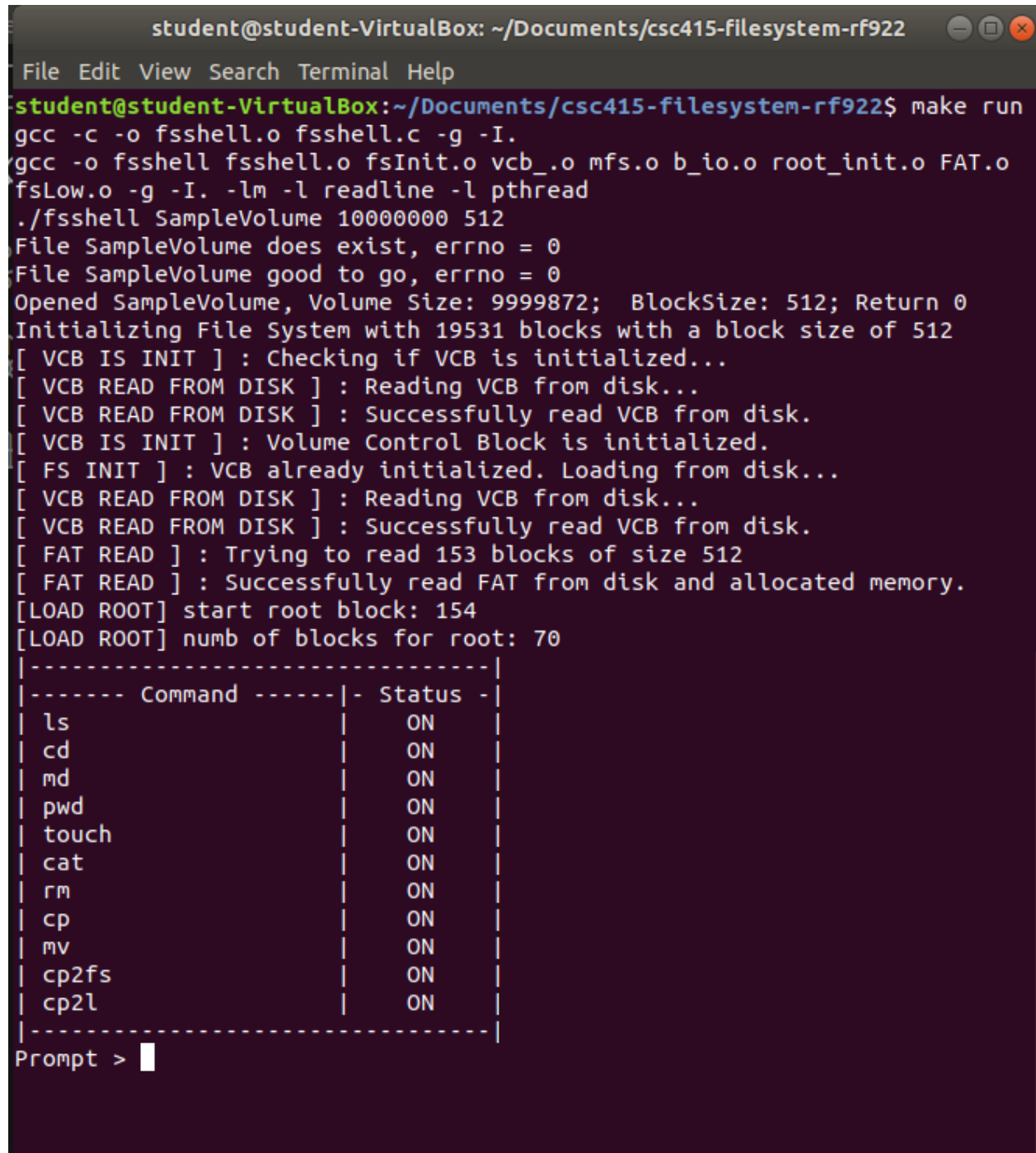
Other commands operate in a similar way. The `'md'` command triggers the `cmd_md` function to create a new directory, `'rm'` prompts the `cmd_rm` function to remove a directory or file, and `'cp'` commands copy files from one location to another. Some commands, like `'mv'`, may be inactive depending on their `CMDXXXX_ON` flags, providing flexibility and control over the features that can be tested. The driver program checks the return values of each function call, and if an error is detected, it informs the user, offering clarity and guidance.

The `dispatchTable` array also has the `'help'` command, which displays a list of available commands and their descriptions. The command serves as a user guide, making the system more accessible to users. The `displayFiles` function is called upon by `cmd_ls` when the user wants to list the files in a directory. It reads directory entries through the ``fs_readdir`` function and then prints out each item. If the `flong` flag is set, it also retrieves and prints additional information such as whether the item is a directory and its size. The code embodies the principle of separation of concerns, keeping the file system logic separate from the user interface logic.

In conclusion, our driver program, `fsshell.c`, offers an efficient interface for interacting with our file system. It reads user commands, dispatches them to the right functions, handles errors, and provides guidance to users, ensuring a great experience.

Screen shots demonstrating the execution of each of the commands listed in the readme.

Compilation:



```
student@student-VirtualBox: ~/Documents/csc415-filesystem-rf922
File Edit View Search Terminal Help
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -o fsshell fsshell.o fsInit.o vcb_.o mfs.o b_io.o root_init.o FAT.o
fsLow.o -g -I. -lm -l readline -l pthread
./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
[ VCB IS INIT ] : Checking if VCB is initialized...
[ VCB READ FROM DISK ] : Reading VCB from disk...
[ VCB READ FROM DISK ] : Successfully read VCB from disk.
[ VCB IS INIT ] : Volume Control Block is initialized.
[ FS INIT ] : VCB already initialized. Loading from disk...
[ VCB READ FROM DISK ] : Reading VCB from disk...
[ VCB READ FROM DISK ] : Successfully read VCB from disk.
[ FAT READ ] : Trying to read 153 blocks of size 512
[ FAT READ ] : Successfully read FAT from disk and allocated memory.
[LOAD ROOT] start root block: 154
[LOAD ROOT] numb of blocks for root: 70
|-----|
|----- 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 - Lists the file in a directory

- ls: List out all the files in the current directory
- ls -l : List out the size of each file
- ls -x: Invalid option

```
Prompt > ls
```

```
fileA.txt
```

```
fileF.doc
```

```
Folder_d
```

```
fileC_copy.
```

```
Prompt > ls -l
```

```
-          0    fileA.txt
```

```
-       28987   fileF.doc
```

```
D       35840   Folder_d
```

```
-          0    fileC_copy.
```

```
Prompt > ls -x
```

```
ls: invalid option -- 'x'
```

```
Usage: ls [--all-a] [--long/-l] [pathname]
```

```
Prompt > 
```

cp Copies a file - source [dest]

Example: Copy and paste fileA.pdf in the same directory as fileB.pdf.

```
Prompt > touch fileA.doc
[IS DIR] fileA.doc does not exist
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 224
Prompt > cp2fs test.txt fileB.doc
[IS DIR] fileB.doc does not exist
3
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 225
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
Prompt > cp fileB.doc fileA.doc
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 227
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
Prompt > exit
System exiting
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$
```

```
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$ Hexdump/hexdump.linux --start 225 --count 2 SampleVolume
Dumping file SampleVolume, starting at block 225 for 2 blocks:

01C200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C400: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C410: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C420: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C430: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C440: 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42 42 | AAABBBBBBBBBBBBB
01C450: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C460: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C470: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C480: 42 42 42 42 42 42 42 42 42 42 42 42 43 43 43 43 | BBBBBBBBBBBBCCC
01C490: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C4A0: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C4B0: 43 43 43 43 43 43 43 43 43 43 44 44 44 44 44 44 | CCCCCCCCCDDDDDDD
01C4C0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C4D0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C4E0: 44 44 44 44 44 44 44 44 44 44 45 45 45 45 45 45 | DDDDDDDDDDEEEEEEE
01C4F0: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 | EEEEEEEEEEEEEEEE
```

```
01C400: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C410: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C420: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C430: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C440: 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42 | AAAABBBBBBBBBBBBBB
01C450: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBBBB
01C460: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBBBB
01C470: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBBBB
01C480: 42 42 42 42 42 42 42 42 42 42 42 43 43 43 43 | BBBBBBBBBBBBBBCCC
01C490: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C4A0: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C4B0: 43 43 43 43 43 43 43 43 43 44 44 44 44 44 44 | CCCCCCCCCDDDDDDD
01C4C0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C4D0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C4E0: 44 44 44 44 44 44 44 44 44 44 45 45 45 45 45 | DDDDDDDDDDEEEEEEE
01C4F0: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 | EEEEEEEEEEEEEEEEE

01C500: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 | EEEEEEEEEEEEEEEEE
01C510: 45 45 45 45 45 45 45 45 45 45 45 45 45 46 46 | EEEEEEEEEEEEEEEF
01C520: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFFFF
01C530: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFFFF
01C540: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFFFF
01C550: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFFFF
01C560: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFFFF
01C570: 46 46 46 46 46 46 46 46 46 46 46 46 46 47 47 | FFFFFFFFFFFFFFFFFG
01C580: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C590: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5A0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5B0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5C0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5D0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5E0: 47 47 47 47 48 48 48 48 48 48 48 48 48 48 48 | GGGGHHHHHHHHHHHH
01C5F0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH

student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$
```



```
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$ Hexdump/hexdump.linux --start 228 --count 2 SampleVolume
Dumping file SampleVolume, starting at block 228 for 2 blocks:

01C800: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C810: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C820: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C830: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAAAA
01C840: 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42 42 | AAABBBBBBBBBBBBB
01C850: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C860: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C870: 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 | BBBBBBBBBBBBBBBB
01C880: 42 42 42 42 42 42 42 42 42 42 43 43 43 43 43 43 | BBBBBBBBBBBBBCCC
01C890: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C8A0: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 | CCCCCCCCCCCCCCCC
01C8B0: 43 43 43 43 43 43 43 43 43 43 44 44 44 44 44 44 | CCCCCCCCCDDDDDDD
01C8C0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C8D0: 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 44 | DDDDDDDDDDDDDDDD
01C8E0: 44 44 44 44 44 44 44 44 44 44 45 45 45 45 45 45 | DDDDDDDDDDEEEEEEE
01C8F0: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 | EEEEEEEEEEEEEEEE

01C900: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 | EEEEEEEEEEEEEEEE
01C910: 45 45 45 45 45 45 45 45 45 45 45 45 45 45 46 46 | EEEEEEEEEEEEEEEF
01C920: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFF
01C930: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFF
01C940: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFF
01C950: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFF
01C960: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 46 | FFFFFFFFFFFFFFFF
01C970: 46 46 46 46 46 46 46 46 46 46 46 46 46 47 47 | FFFFFFFFFFFFFFFFG
01C980: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C990: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA00: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA10: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA20: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA30: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA40: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA50: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA60: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA70: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA80: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CA90: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAA0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAB0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAC0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAD0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAE0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01CAF0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG

01CA00: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA10: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA20: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA30: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA40: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA50: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA60: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA70: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA80: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CA90: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAA0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAB0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAC0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAD0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAE0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01CAF0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
```

mv Moves a file - source dest

Example: Make a new directory named directory and move fileB.doc into directory. Then command “ls” to show fileB.doc is no longer in the current directory. Lastly move into directory and show fileB.doc is in directory.

```
Prompt > touch fileA.doc
[IS DIR] fileA.doc does not exist
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 224
Prompt > cp2fs test.txt fileB.doc
[IS DIR] fileB.doc does not exist
3
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 225
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
Prompt > md directory
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 70.
[ ALLOCATE_BLOCKS ] : Updating FAT.
hex values of dir_name: 2E
hex values of block location: E3
hex values of file size: 8C00
hex values of : 18000000
[INIT ROOT] root start block: 227
Prompt > ls

fileA.doc
fileB.doc
directory
Prompt > mv fileB.doc directory
Prompt > ls

fileA.doc
directory
Prompt > cd directory
Prompt > ls

fileB.doc
Prompt > █
```


md Make a new directory

Example: show contents of directory. Make directoryC and show it's in the current directory.

```
Prompt > ls

fileA.doc
fileB.txt
Prompt > md directoryC
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 70.
[ ALLOCATE_BLOCKS ] : Updating FAT.
hex values of dir_name: 2E
hex values of block location: E2
hex values of file size: 8C00
hex values of : 18000000
[INIT ROOT] root start block: 226
Prompt > ls

fileA.doc
fileB.txt
directoryC
Prompt >
```

rm Removes a file or directory

Example: show files in the current directory. Remove fileA.doc in the current directory and show the file was removed.

```
Prompt > ls

fileA.doc
fileB.txt
directoryC
Prompt > rm fileA.doc
Prompt > ls

fileB.txt
directoryC
Prompt > rm directoryC
Prompt > ls

fileB.txt
Prompt >
```

touch Touches/Creates a file

Example: Show the files in the current directory. Then create fileD.txt and “ls” to show the file was created and in the directory.

```
Prompt > ls

fileB.txt
Prompt > touch fileD.txt
[IS DIR] fileD.txt does not exist
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 226
Prompt > ls

fileD.txt
fileB.txt
Prompt > 
```

cat Limited version of cat that displace the file to the Console

Example: First copy contents from test.txt to fileA.doc in our file system. Then run the “cat” command to show the file contents.

```
Prompt > cp2fs test.txt fileA.doc
[IS DIR] fileA.doc does not exist
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 224
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
Prompt > cat fileA.doc
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH
Prompt > █
```

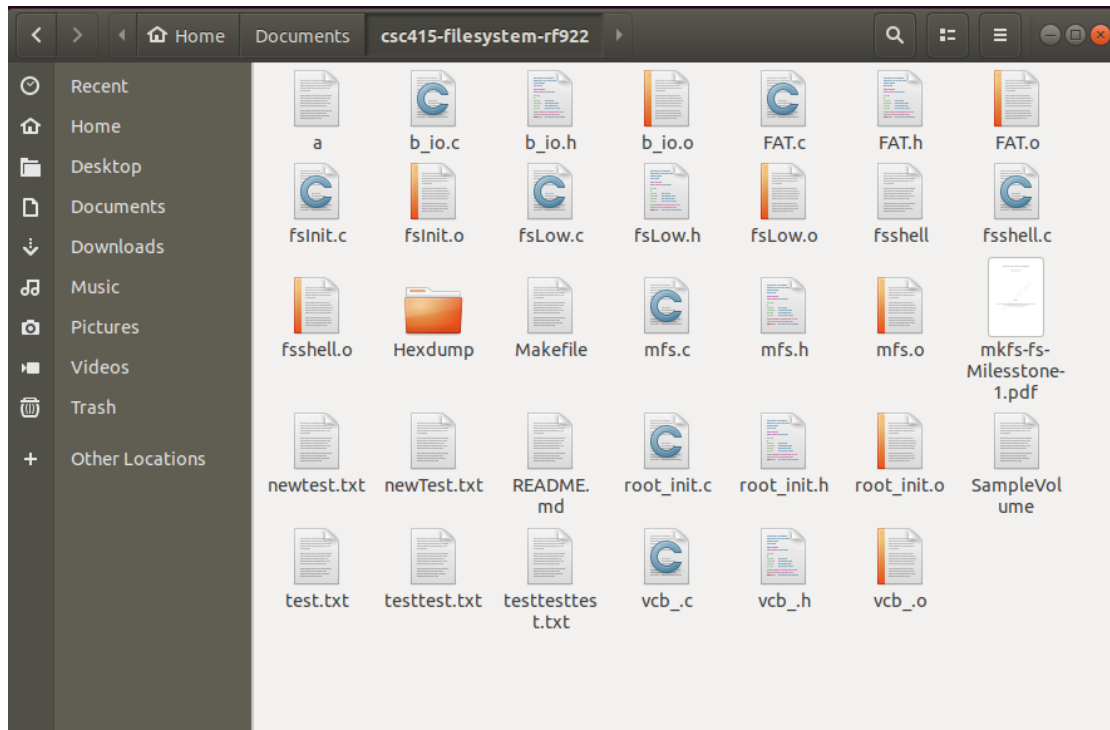
cp2l Copies a file from the test file system to the linux file system

Example: show the contents of the file system project directory where newfile.txt is not present. Then copy to location file newfile.txt from our filesystem SampleVolume to the file system project directory on our actual machine.

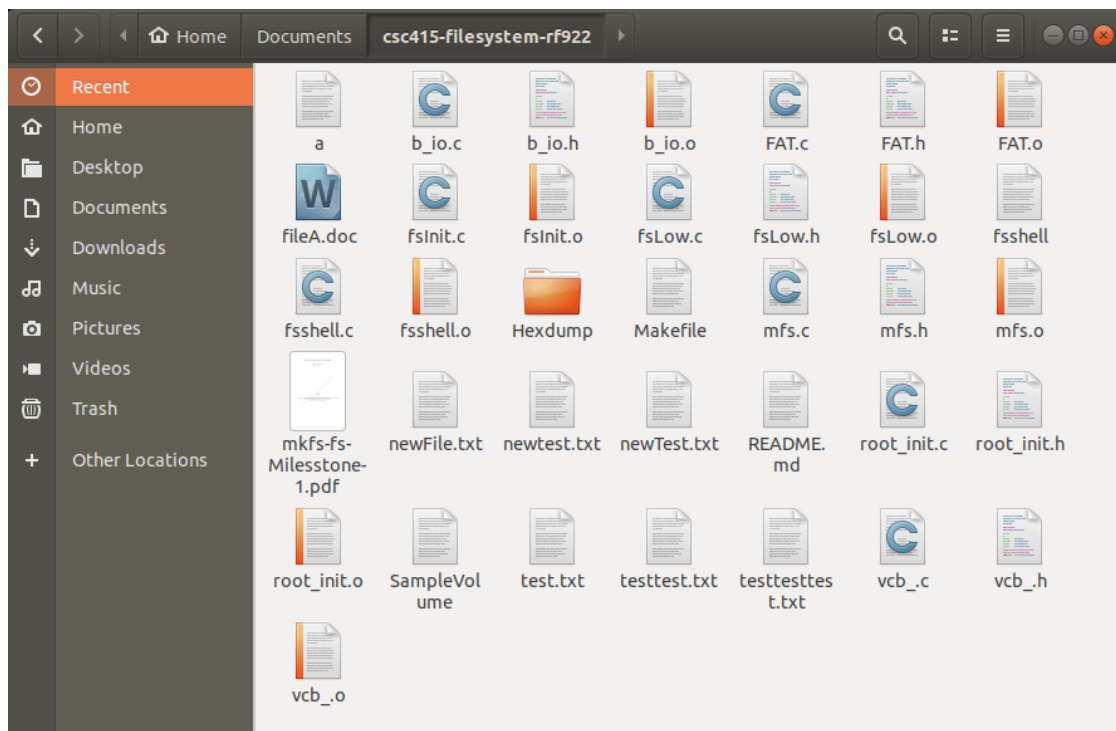
Group name: MKFS
Github: rf922

ID: Shown Below
CSC415 Operating Systems

Before:



After:



[illegible]

```
*newFile.txt  
~/Documents/csc415-fileSystem-rf922  
1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
2 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB  
3 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  
4 DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD  
5 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE  
6 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
7 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
8 GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG  
9 GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG  
10 HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHIIIII
```

```
1
Prompt > touch fileA.doc
[IS DIR] fileA.doc does not exist
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 224
Prompt > cp2fs test.txt fileA.doc
2
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
[MKFILE] file location: 225
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 1.
[ ALLOCATE_BLOCKS ] : Updating FAT.
Prompt > █
```

```
student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$ Hexdump/hexdump.linux --start 225
--count 3 SampleVolume
Dumping file SampleVolume, starting at block 225 for 3 blocks:

01C200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C2F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C3F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
```

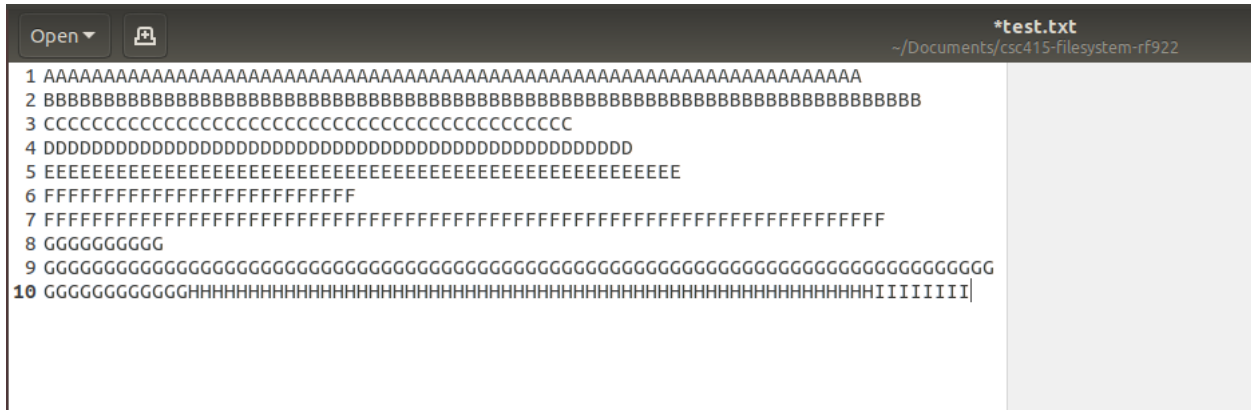
48


```
01C570: 46 46 46 46 46 46 46 46 46 46 46 46 46 46 47 | FFFFFFFFFFFFFFFF
01C580: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C590: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5A0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5B0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5C0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5D0: 47 47 47 47 47 47 47 47 47 47 47 47 47 47 47 | GGGGGGGGGGGGGGGG
01C5E0: 47 47 47 47 48 48 48 48 48 48 48 48 48 48 48 | GGGGHHHHHHHHHHHH
01C5F0: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH

01C600: 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 | HHHHHHHHHHHHHHHH
01C610: 48 48 48 48 48 48 48 48 48 48 48 49 49 49 49 | HHHHHHHHHHHHHHII
01C620: 49 49 49 49 49 0A 00 00 00 00 00 00 00 00 00 | IIIII.....
01C630: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C660: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C690: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C6F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

01C700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C710: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C720: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C730: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C740: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C750: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C760: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C770: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
01C7F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

student@student-VirtualBox:~/Documents/csc415-filesystem-rf922$
```



```
Open ▾ [icon] *test.txt ~/Documents/csc415-filesystem-rf922
1 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
3 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
4 DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
5 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
6 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
7 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
8 GGGGGGGGGG
9 GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG
10 GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGIIIIIIII|
```

cd Changes directory

Example: make directoryA, show it now exists in the current directory. Then move into the directory and show the contents of the directory.

```
Prompt > ls

fileA.doc
Prompt > md directoryA
[ ALLOCATE_BLOCKS ] : Allocating blocks_needed = 70.
[ ALLOCATE_BLOCKS ] : Updating FAT.
hex values of dir_name: 2E
hex values of block location: E2
hex values of file size: 8C00
hex values of : 18000000
[INIT ROOT] root start block: 226
Prompt > ls

fileA.doc
directoryA
Prompt > cd directoryA
Prompt > ls

Prompt > cd ..
Prompt > ls

fileA.doc
directoryA
Prompt > █
```

pwd Prints the working directory

Example: Prints out the current working directory.

```
Prompt > ls

fileA.doc
directoryA
Prompt > cd ..
Prompt > ls

fileA.doc
directoryA
Prompt > cd directoryA
Prompt > pwd
/directoryA
Prompt > 
```

history Prints out the history

Example: This prints out the history of commands.

```
Prompt > cd ..  
Prompt > ls  
  
fileA.doc  
directoryA  
Prompt > history  
cp2fs test.txt fileA.doc  
cat fileA.doc  
ls  
md directoryA  
ls  
cd directoryA  
ls  
cd ..  
ls  
cd ..  
ls  
cd directoryA  
pwd  
cd ..  
ls  
history  
Prompt > █
```

help Prints out help

Example: Run help to show the file system commands.

```
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 >
```

The volume file of our file system, capped at 10MB

2 Tracking Free Space

To handle free space management for our file system we decided to use a File Allocation Table (FAT), since our implementation is based on FAT32. The following is our proposed memory layout.

Our FAT table will be represented by an array which will index every block in our volume. Our volume will be approximately 10MB in size and our block size is 512 bytes, so we will have a total of:

$$\begin{aligned}\text{Total Blocks} &= \frac{\text{Total Bytes}}{\text{Bytes per Block}} \\ &= \frac{10,000,000 \text{ bytes}}{512 \text{ bytes/block}} \\ &= 19,531 \text{ blocks}\end{aligned}$$

Each of our blocks will correspond to an integer index in the table. Given that an integer is 4 bytes, our FAT should need:

$$\begin{aligned}\text{FAT Size} &= \text{Total Blocks} \times \text{Bytes per Block} \\ &= 19,531 \text{ blocks} \times 4 \text{ bytes/block} \\ &= 78,124 \text{ bytes}\end{aligned}$$

With this value we then see that:

$$\begin{aligned}\text{FAT Blocks} &= \frac{\text{FAT Size}}{\text{Bytes per Block}} \\ &= \frac{78,124 \text{ bytes}}{512 \text{ bytes/block}} \\ &= 153 \text{ blocks}\end{aligned}$$

So, our FAT will span 153 blocks.