

# CSC 415 - 01 OPERATING SYSTEM PRINCIPLES Summer 2024

## File System Project

“Team of Three”

Team Member	Student ID
Siarhei Pushkin	
Philip Karnatsevich	
Yahya Obeid	

**GitHub Link:**

<https://github.com/CSC415-2024-Summer/csc415-filesystem-yahyaobeid>

# Table of Contents

<b>1. The plan for each phase and changes made</b>	<b>3</b>
<b>2. A description of the File System</b>	<b>6</b>
<b>3. Issues we had</b>	<b>7</b>
<b>4. Details of how each of the functions works</b>	<b>8</b>
<b>5. Screenshots showing each of the commands</b>	<b>15</b>

# #1. The plan for each phase and changes made

Our work emphasizes teamwork, project planning, and understanding file system structures. To deliver the project on time, we divided our project into four main phases.

**Phase 1:** In this phase, we defined the basic structure and strategies for our file system. The Volume Control Block (VCB) was created to store important information about the volume configuration, including block size, total blocks, available blocks, and the location of the root directory and free space map. We also set up the Directory Entry structure to manage files and directories, including details such as timestamps, file size, and ownership. To efficiently track free space, we planned to use a bitmap in which each bit represents the status of a block (free or occupied). Additionally, we described the inclusion of necessary metadata, such as file permissions, timestamps, and ownership information.

**Phase 2:** In the second phase, we achieved significant progress in developing the core functionality of our file system. Specifically, we successfully wrote the Volume Control Block (VCB) in block 0. The VCB contains essential information about the volume layout, including block size, total blocks, available blocks, and the locations of the root directory and free space map. It includes fields for a unique signature for validity verification, the size of each block in bytes, the total and available number of blocks, and block numbers for the root directory and free space map. Furthermore, we implemented the Free Space management system using a bitmap to track free and allocated blocks across five blocks. Initially, all blocks are marked free except for the

first six system-reserved blocks, including the VCB and the free space map. This bitmap, starting at block one, allows for efficient space management.

We also created and initialized the root directory with essential entries for the current (".") and parent ("..") directories. Our directory system uses a structure called `directoryEntry` to store information such as file names, timestamps, and attributes. It includes functions for creating (`fs_mkdir`) and removing directories (`fs_rmdir`), opening (`fs_opendir`), reading (`fs_readdir`), and closing directories (`fs_closedir`), and managing the current working directory (`fs_getcwd`, `fs_setcwd`). Additionally, it includes tools for checking file types (`fs_isFile`, `fs_isDir`) and deleting files (`fs_delete`).

The `fs_stat` structure provides detailed file information, facilitating directory and file management within the file system. Phase 2 also features a hexdump of the volume file showing the VCB, Free Space, and complete root directory, along with detailed descriptions of the VCB structure, Free Space structure, and Directory system.

**Phase 3:** In the next phase, we successfully integrated the file system shell (`fsshell`) into our file system, making sure that important directory operations were completely functional. This allowed us to perform actions such as creating directories (`md`), listing directories (`ls`), printing the working directory (`pwd`), and changing directories (`cd`). To achieve this goal, we have implemented and refined several key functions.

The functions we implemented include:

- `fs_setcwd`: Changes the current working directory.

- `fs_getcwd`: Prints the current working directory.
- `fs_isFile`: Checks if a given path is a file.
- `fs_isDir`: Checks if a given path is a directory.
- `fs_mkdir`: Creates a new directory.
- `fs_opendir`: Opens a directory for reading.
- `fs_readdir`: Reads the contents of an open directory.
- `fs_closedir`: Closes an open directory.
- `fs_stat`: Provides information about a file or directory.
- `fs_delete`: Deletes a file.
- `fs_rmdir`: Removes a directory.

These functions are necessary for achieving basic file system operations. Their integration into `fsshell` allowed us to communicate with the file system and test these operations. For example, `fs_setcwd` and `fs_getcwd` supported changing and displaying the current directory. `fs_mkdir` and `fs_rmdir` controlled the creation and deletion of directories. Functions like `fs_opendir`, `fs_readdir`, and `fs_closedir` managed reading directory contents.

**Phase 4:** During the final phase, we made sure to thoroughly test and debug our file system to ensure it was operating correctly and reliably. We started with unit testing to validate each function separately. Then, we focused on integrating functions, such as creating a directory, navigating to it, and creating a file within it. Our debugging process involved thorough error handling, step-by-step code inspections, and team code reviews to identify and fix any bugs. This careful process allowed us to manage different issues, including resolving continuous segmentation fault errors in all phases of development.

## **#2. Description of the File System**

This project is a representation of a file system. Our team's primary focus was on developing a simplified version of the file system. Our project includes functionalities for creating, reading, writing, and deleting files. It also can create and remove directories. Additionally, the file system contains advanced data handling, efficient space management, and reliable persistent storage. Our design features have reliable data structures specifically customized for effective file and directory management. Furthermore, we ensured that the system has an intuitive interface to streamline interaction with the file system, making it user-friendly.

Our project focused on building a functional file system, emphasizing on collaboration between team members, project planning, and understanding file system structures. We started by defining the basic structures, including the Volume Control Block (VCB), to store critical information about the volume configuration and the Directory Entry structure for managing files and directories. We used a bitmap to track free space efficiently and included metadata such as file permissions and timestamps.

We have designed core functionalities and implemented the VCB and Free Space management system. In addition, we have created and initialized the root directory with essential entries and developed functions for managing directories and files. These functions include creating, removing, opening, reading, and closing directories, checking file types, and deleting files. Also, we have ensured that key directory operations in the file system shell (fsshell) are functional.

### **#3. Issues we had**

Our team encountered numerous challenges while implementing complex file system functions in C. These challenges were primarily due to the complex nature of file system management and the low-level programming requirements. The project's complexity and long debugging process made it difficult to deliver on time. To overcome these obstacles, we organized regular team meetings to solve problems collaboratively and constantly learn new concepts. We utilized Discord for both asynchronous chats and live meetings, providing real-time support and efficient problem-solving. Flexible scheduling allowed us to reschedule meetings as needed and ensure full participation from all team members. Tasks were divided based on individual strengths, improving efficiency and lowering the time required to complete complex tasks. Our meetings typically lasted about 30 minutes, striking a balance between detailed discussions and avoiding overwhelming the team.

Additionally, we scheduled extra meetings to address urgent problems or brainstorm new ideas. This combination of asynchronous communication and live sessions helped us stay connected, share ideas, and create a supportive environment. Despite the initial delays and challenges, our approach enabled us to manage the project's complexity, address issues promptly, and ultimately achieve our project goals.

## #4. Details of how each of the functions works

### **fs\_setcwd**

The `fs_setcwd` function is used to make changes in the current working directory represented in a file system. It starts by creating space in the computer's memory for a `parentInfo` structure and copying the input pathname. Then, it checks if the provided path is valid. If it's not valid, the function stops and shows an error. If the path is valid, the function verifies whether the cmd is the root directory. If it's not the root directory, the current working directory is freed to prevent memory issues. Then, it loads the new directory using the appropriate parent directory data. Next, the function creates the new current working directory path. If the provided path is absolute (starting from the root directory), it just uses that path. If the provided path is relative (starting from the current working directory), it combines that path with the current working directory path. The combined path is then cleaned to make sure it's in a proper format. If cleaning the path fails, the function stops and shows an error. Lastly, the function frees up the memory it used, updates the current working directory to the new one, and indicates that the process was successful.

### **fs\_getcwd**

The `fs_getcwd` function is used to find out the current working directory and store it in the provided pathname buffer. It first checks if the pathname is NULL or the size is less than or equal to 0. If either condition is valid, it returns NULL to display an error. Otherwise, it uses `strncpy` to copy the current working directory path `cwdPath` into the



pathname buffer, making sure that no more than size - 1 characters are copied. The function makes sure that the string has space for a null termination at the end. Then, it sets the last character of the pathname to "\0" to make sure it is null-terminated. Lastly, it returns the pathname, which now holds the current working directory path.

### **fs\_isFile**

The `fs_isFile` function checks if a given path matches a standard file in the file system. First, it allocates memory for a `parentInfo` structure and duplicates the input pathname string. Then, it calls `parsePath` to parse the provided path and populate the parent structure with appropriate directory data. If `parsePath` returns 0 and the index in the parent is not -1, indicating a valid path, the function extracts the directory entry for the specified path and calls `isFile` to check if it is a regular file. After that, it frees the allocated memory and returns the result of `isFile`. If the path is invalid, the function frees the allocated memory and returns -1 to display failure. This function helps verify the file type before performing operations specific to regular files. If the specified path does not exist or corresponds to a directory or other file type, `fs_isFile` returns false.

### **fs\_isDir**

The `fs_isDir` function checks if a given path is a directory in the file system. It starts by creating a `parentInfo` structure and copying the input path. Then, it uses `parsePath` to analyze the path and fill the parent structure with directory information. If `parsePath` returns a non-zero value or the index in the parent is -1 due to an invalid path, the function frees the allocated memory and returns -1. If the path is valid, the function gets the directory entry for the specified path and uses `isDirectory` to check if it

is a directory. After that, it frees the allocated memory and returns the result of `isDirectory`. This function helps verify a directory's type before performing directory-specific operations. If the path doesn't exist or is a regular file or a different kind of file, `fs_isDir` returns false.

## **`fs_mkdir`**

The `fs_mkdir` function creates a new directory in the file system. To do this, it first allocates memory for a `parentInfo` structure and duplicates the input pathname string. Then, it uses `parsePath` to parse the provided path and fill out the parent structure with directory data. If there's an error or if the directory already exists, the function frees the memory and returns -1. If the path is valid, the function initializes a new directory entry using `initDir` and reads the directory entry from disk using `LBAread`. The new directory's name is set to the last part of the parsed path. The function then finds an empty entry in the parent directory using `findEntryInDir`. If no empty entry is found, the function frees the memory and returns -1. Otherwise, it adds the new directory to the parent directory and writes the updated parent directory to the disk using `LBAwrite`. Lastly, the function frees all allocated memory and returns 0 to indicate success.

To summarize, the `fs_mkdir` function creates a new directory with the specified name and handles errors such as those in an existing directory. If the directory does not exist, it initializes and adds the new directory into the parent directory, updating the file system structure accordingly. This function makes sure proper error handling and resource management throughout the directory creation process.

## **fs\_opendir**

The `fs_opendir` function opens a directory for reading in the file system. First, it duplicates the input pathname string and checks if the duplication was successful. Then, it initializes a `parentInfo` structure and calls `parsePath` to parse the provided path and populate the parent structure with directory data. If `parsePath` fails, the parent directory is `NULL`, or the path does not conform to a directory, the function returns `NULL`.

If the path is valid and corresponds to a directory, the function allocates memory for a `fdDir` structure representing the directory to be opened. If memory allocation fails, the function returns `NULL`. The `fdDir` structure is initialized with the directory entry status set to 0. It also allocates memory for a `fs_diriteminfo` structure, which holds directory item information. If this allocation fails, the function frees the previously allocated memory and returns `NULL`. The `fs_diriteminfo` structure is initialized with the record length and the directory name copied from the parent directory's file name. Lastly, the function returns the initialized `fdDir` structure, allowing the directory to be read. The `fs_opendir` function opens a selected directory for reading, given its path. It initializes necessary structures and verifies that the path corresponds to a valid directory. If successful, it returns a directory structure ready for reading. Otherwise, it handles errors and returns `NULL`, making sure proper error handling and resource management throughout the process.

## **fs\_readdir**

The `fs_readdir` function reads an entry from an open directory in the file system. It takes a pointer to a `fdDir` structure, representing the open directory, as input. First, it checks for errors by making sure that the input pointer `dirp` or the current working directory is not `NULL`. If either is `NULL`, it returns an error. Next, the function calculates the total number of entries in the current working directory by separating the directory's file size in bytes by the size of a `directoryEntry`. If the position of the directory entry in `dirp` is greater than or equal to the number of entries, it means there are no more entries to read, and the function returns `NULL`.

If there are entries to read, the function gets the current directory entry at the position specified by `dirEntryPosition` from the `cwd` directory entries. It then updates the `dirEntryPosition` to point to the next entry for the next reads. The function populates a `fs_diriteminfo` structure with the appropriate data from the directory entry, such as the record length, file type, and name, and returns this structure.

## **fs\_closedir**

The `fs_closedir` function closes an open directory in the file system. It takes a pointer to a `fdDir` structure, representing the open directory as input. First, the function checks if the input pointer `dirp` is `NULL`. If it is, the function returns -1 to display an error. If the pointer is valid, the function checks if the `di` component of the `fdDir` structure (which holds directory item information) is not `NULL`. If `di` is not `NULL`, the function frees the memory allocated for `di` and sets the pointer to `NULL` to prevent hanging references. Lastly, the function frees the memory allocated for the `fdDir` structure and returns 0 to

display success. The `fs_closedir` function makes sure that all resources associated with the open directory are correctly released and handles errors such as trying to close a directory that is not open.

### **fs\_stat**

The `fs_stat` function extracts information about a file or directory in the file system and saves it in a provided `fs_stat` structure. It begins by allocating memory for a `parentInfo` structure and copying the input path string. The function then uses `parsePath` to analyze the provided path and fills the parent structure with appropriate directory data. If `parsePath` returns an error, or if the path is invalid or not a directory, the function frees the allocated memory and returns -1. If the path is valid, the function fills the `fs_stat` structure with the file size in bytes, the block size, and the number of blocks the file uses. Lastly, it frees the allocated memory and returns 0 to indicate success. The `fs_stat` function is critical for getting detailed information about a file or directory in the file system. It requires the path of the file or directory as input and provides a structure containing metadata such as file size, block size, and the number of blocks. This function is important for users to get complete details about specific files or directories, helping in file system operations and management.

### **fs\_delete**

The `fs_delete` function deletes a selected file from the file system by taking the file name as input. It begins by allocating memory for a `parentInfo` structure and duplicating the input filename string. The function then uses `parsePath` to parse the provided path and fill the parent structure with directory data. If `parsePath` displays an

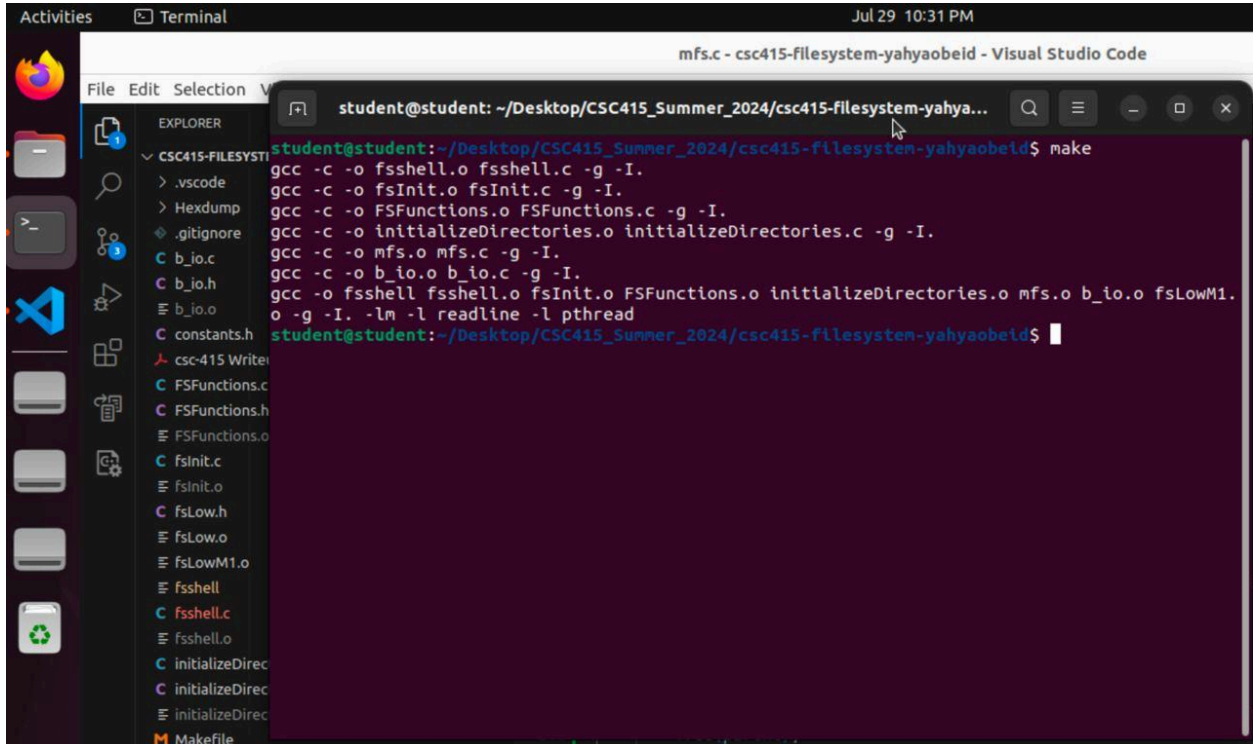
error, the file does not exist, or the path is not a valid file, the function dismisses the allocated memory and returns -1. If the path is valid and conforms to a file, the function fetches the space data for the file using loadSpace. It then marks the file as deleted by setting its name to an empty string. The function goes through the space blocks, clearing the bits for each block to mark them as free. The updated parent directory is then written back to disk using LBAwrite. Lastly, the function frees all allocated memory and returns 0 to indicate success. The function ensures proper error handling and resource management throughout the file deletion.

### **fs\_rmdir**

The fs\_rmdir function removes an empty directory from the file system. First, it allocates memory for a parentInfo structure and duplicates the input pathname string. Then, it calls parsePath to analyze the provided path and fill the parent structure with directory information. If parsePath indicates an error, such as the directory not existing, not being valid, or not being empty, the function frees the allocated memory and returns -1. If the path is valid and the directory is empty, the function uses loadSpace to extract the space information for the directory and then goes through the space blocks, clearing the bits for each block to mark them as free. The directory entry is then marked as deleted by setting its file name to an empty string. The updated parent directory is written back to disk using LBAwrite. Lastly, the function frees all allocated memory and returns 0 to indicate success. The fs\_rmdir function makes sure valid error handling and resource management throughout the directory removal process.

## #5. Screenshots showing each of the commands listed in the readme

Snapshot of compilation:



```
student@student: ~/Desktop/CSC415_Summer_2024/csc415-filesystem-yahyaobeld$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o FSFunctions.o FSFunctions.c -g -I.
gcc -c -o initializeDirectories.o initializeDirectories.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o FSFunctions.o initializeDirectories.o mfs.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
student@student:~/Desktop/CSC415_Summer_2024/csc415-filesystem-yahyaobeld$
```

Screenshot of the execution:

```

gcc -c -o FSFunctions.o FSFunctions.c -g -I.
gcc -c -o initializeDirectories.o initializeDirectories.c -g -I.
gcc -c -o mfs.o mfs.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o FSFunctions.o initializeDirectories.o mfs.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
student@student: ~/Desktop/CSC415_Summer_2024/csc415-filesystem-yahyaobeid$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already formatted
initFileSystem: rootDirectory 0xaaaaaf2f86b80, cwd 0xaaaaaf2f86b80
----- 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 >
  
```

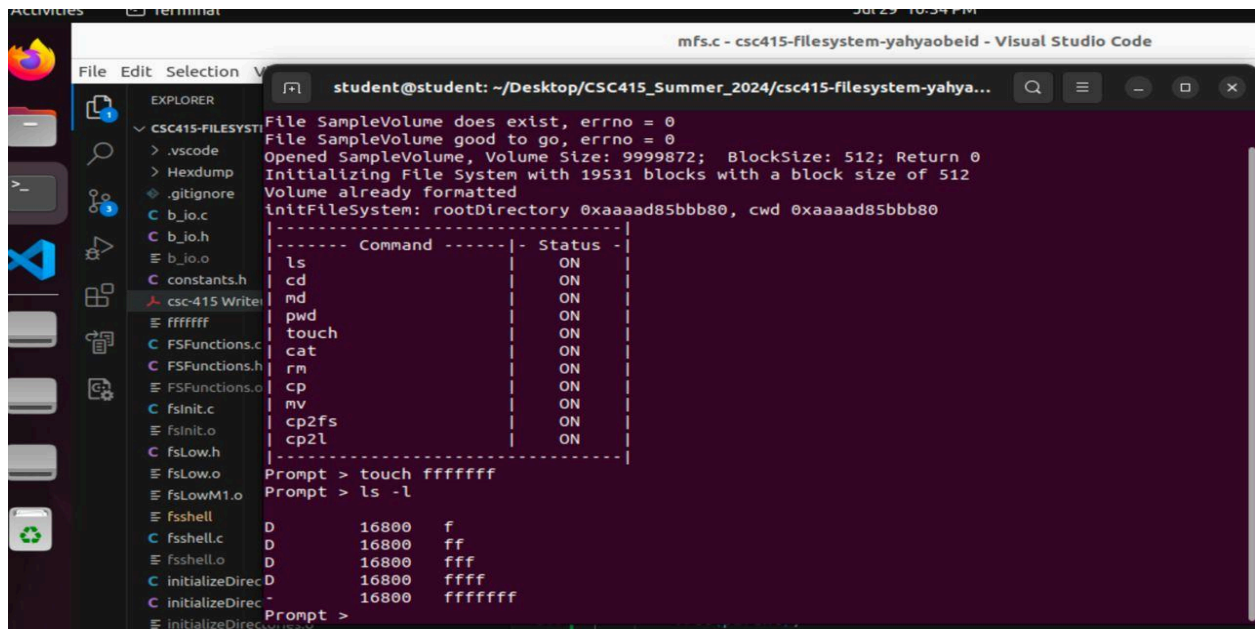
ls:

```

./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already formatted
initFileSystem: rootDirectory 0xaaaaaf2f86b80, cwd 0xaaaaaf2f86b80
----- 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
f
ff
fff
ffff
ffffff
Prompt >
  
```



touch:

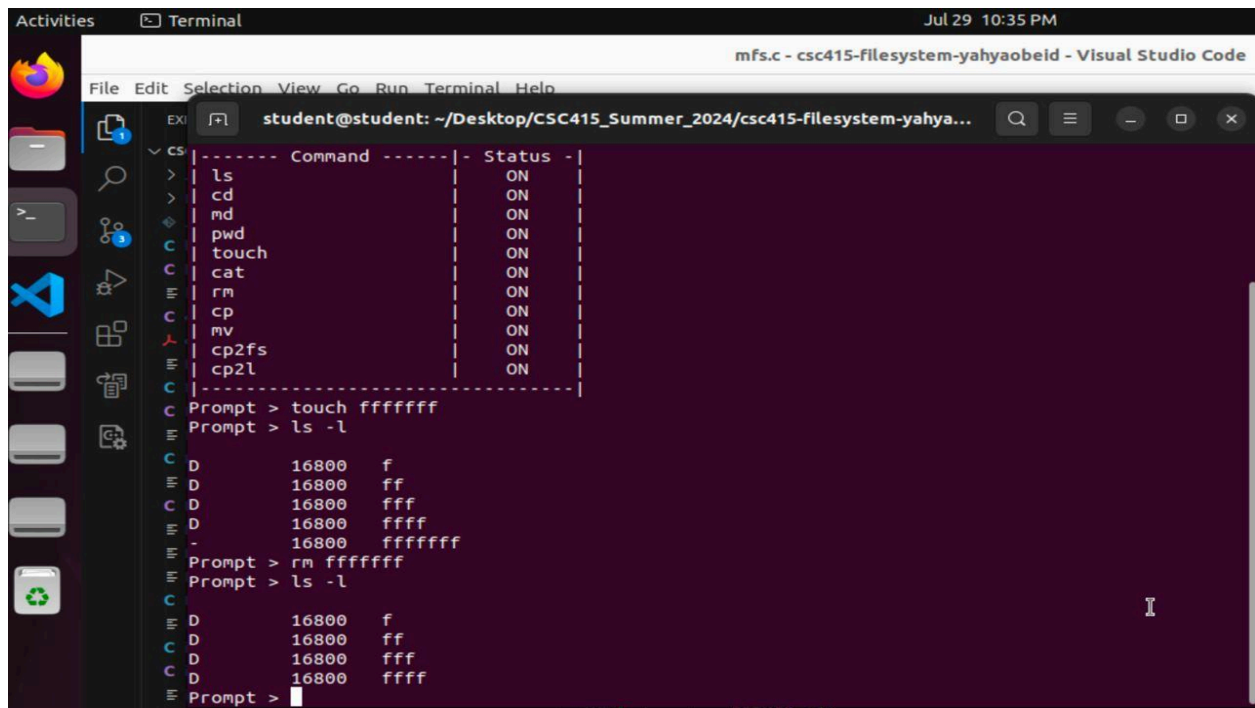


```
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
Volume already formatted
initFileSystem: rootDirectory 0xaaaaad85bbb80, cwd 0xaaaaad85bbb80

----- 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 fffffff
Prompt > ls -l
D      16800  f
D      16800  ff
D      16800  fff
D      16800  ffff
-      16800  fffffff
Prompt >
```

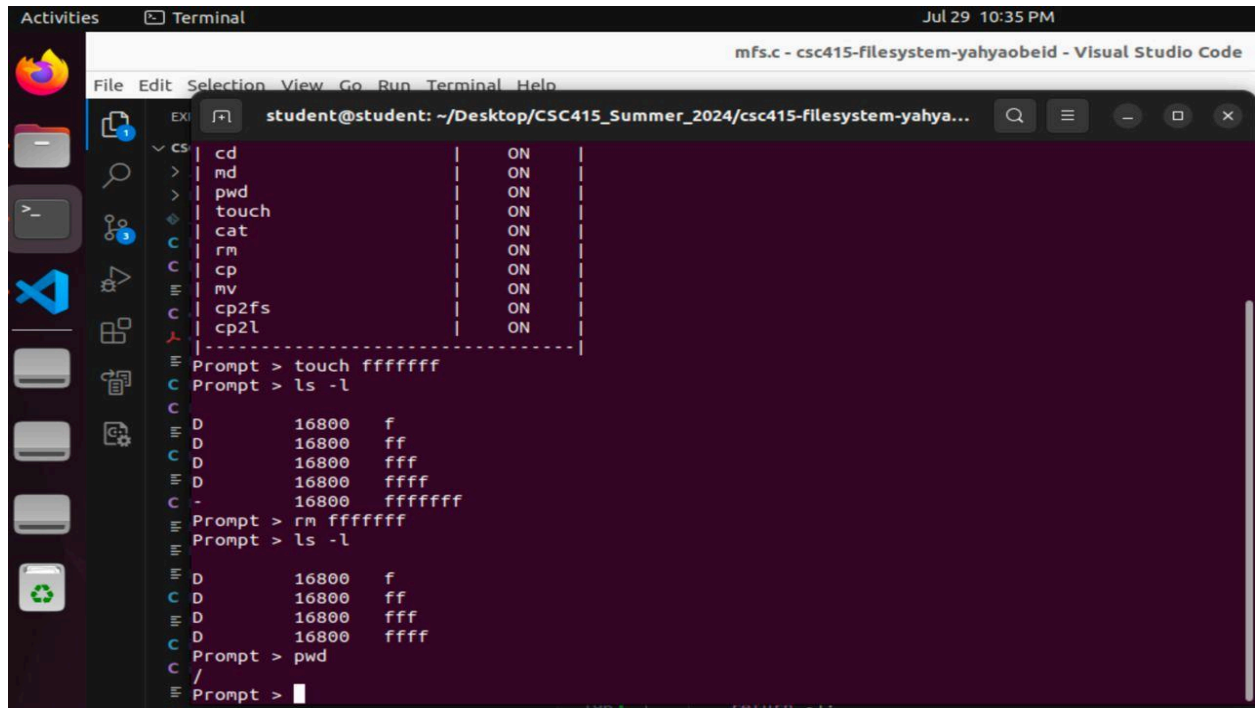
rm:



```
----- 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 fffffff
Prompt > ls -l
D      16800  f
D      16800  ff
D      16800  fff
D      16800  ffff
-      16800  fffffff
Prompt > rm fffffff
Prompt > ls -l
D      16800  f
D      16800  ff
D      16800  fff
D      16800  ffff
Prompt >
```

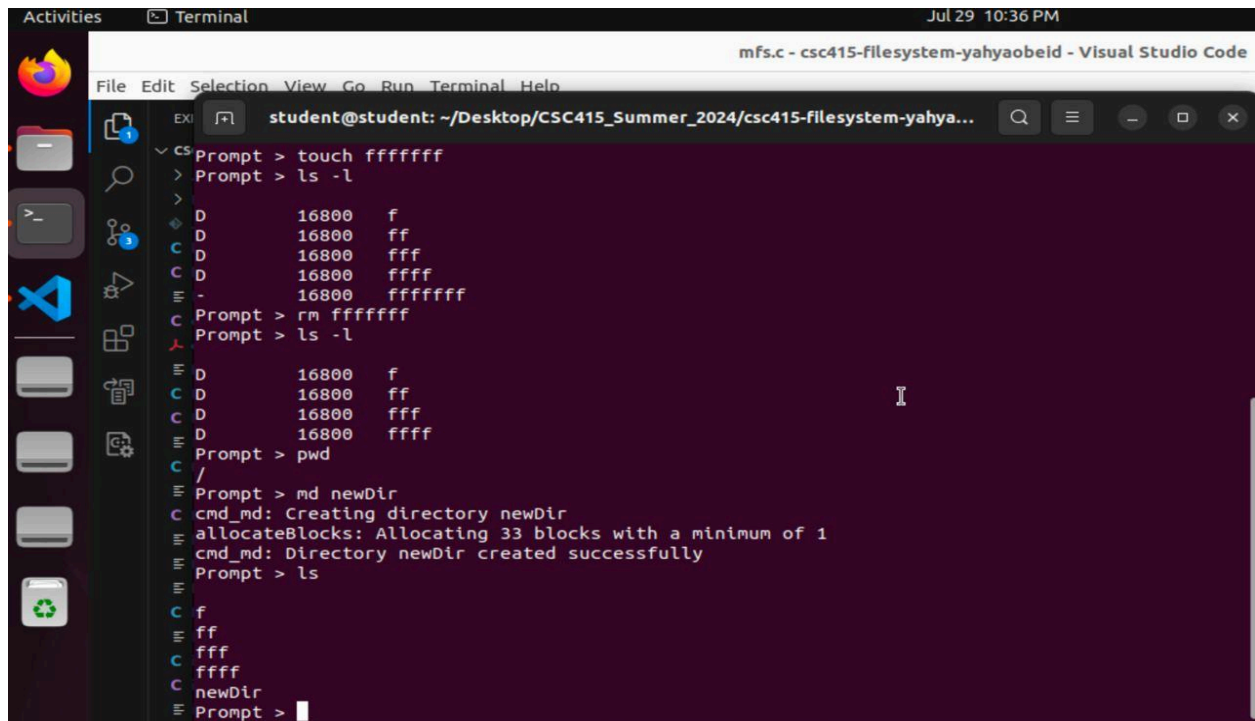
pwd:



A terminal window in Visual Studio Code showing a series of commands and their outputs. The user is in a directory on the desktop. The commands executed are: `cd`, `md`, `pwd`, `touch`, `cat`, `rm`, `cp`, `mv`, `cp2fs`, and `cp2l`. A table is displayed with three columns: file type (D for directory, C for file), permissions (16800), and file name (f, ff, fff, ffff, fffffff). The prompt is `Prompt >`.

```
cd
md
pwd
touch
cat
rm
cp
mv
cp2fs
cp2l
-----
Prompt > touch fffffff
C Prompt > ls -l
D      16800  f
D      16800  ff
C D      16800  fff
C D      16800  ffff
C -      16800  fffffff
Prompt > rm fffffff
Prompt > ls -l
D      16800  f
C D      16800  ff
D      16800  fff
D      16800  ffff
Prompt > pwd
/
Prompt >
```

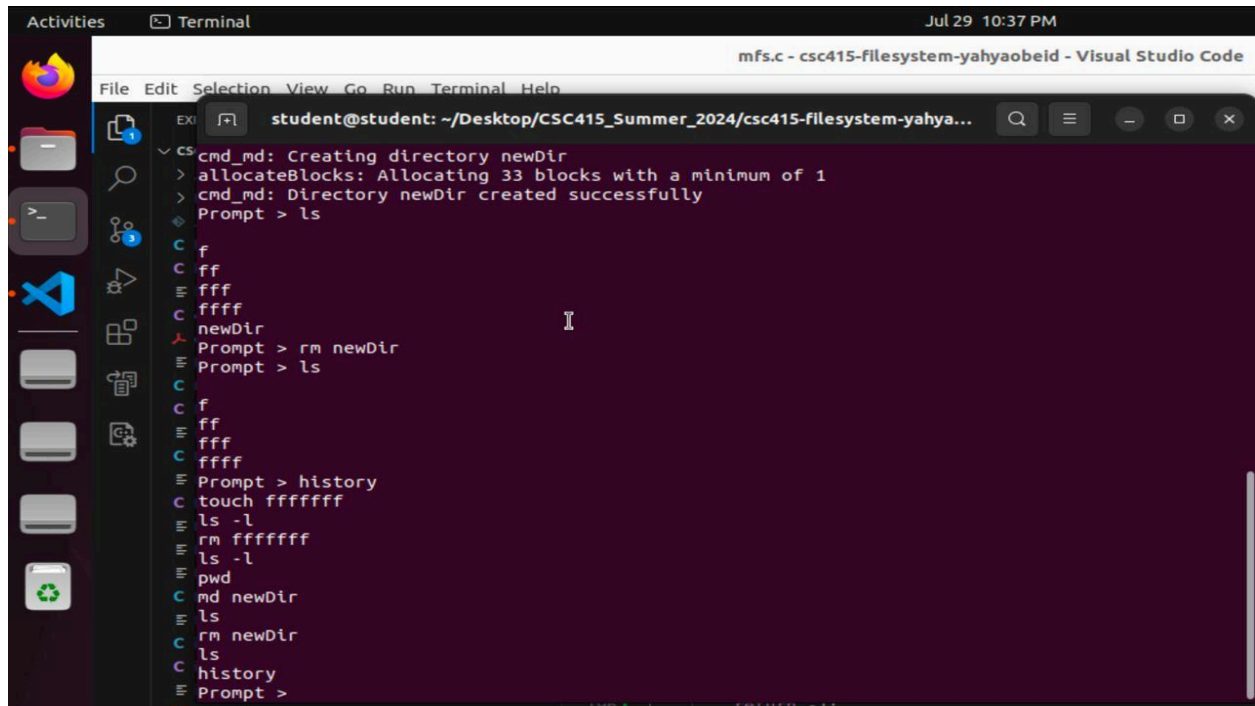
md:



A terminal window in Visual Studio Code showing the execution of `touch`, `rm`, `ls`, `pwd`, and `md` commands. The output shows the creation of a new directory `newDir` and the allocation of blocks. The prompt is `Prompt >`.

```
Prompt > touch fffffff
> Prompt > ls -l
D      16800  f
D      16800  ff
C D      16800  fff
C D      16800  ffff
C -      16800  fffffff
Prompt > rm fffffff
Prompt > ls -l
D      16800  f
C D      16800  ff
C D      16800  fff
D      16800  ffff
Prompt > pwd
/
Prompt > md newDir
cmd_md: Creating directory newDir
allocateBlocks: Allocating 33 blocks with a minimum of 1
cmd_md: Directory newDir created successfully
Prompt > ls
f
ff
fff
ffff
newDir
Prompt >
```

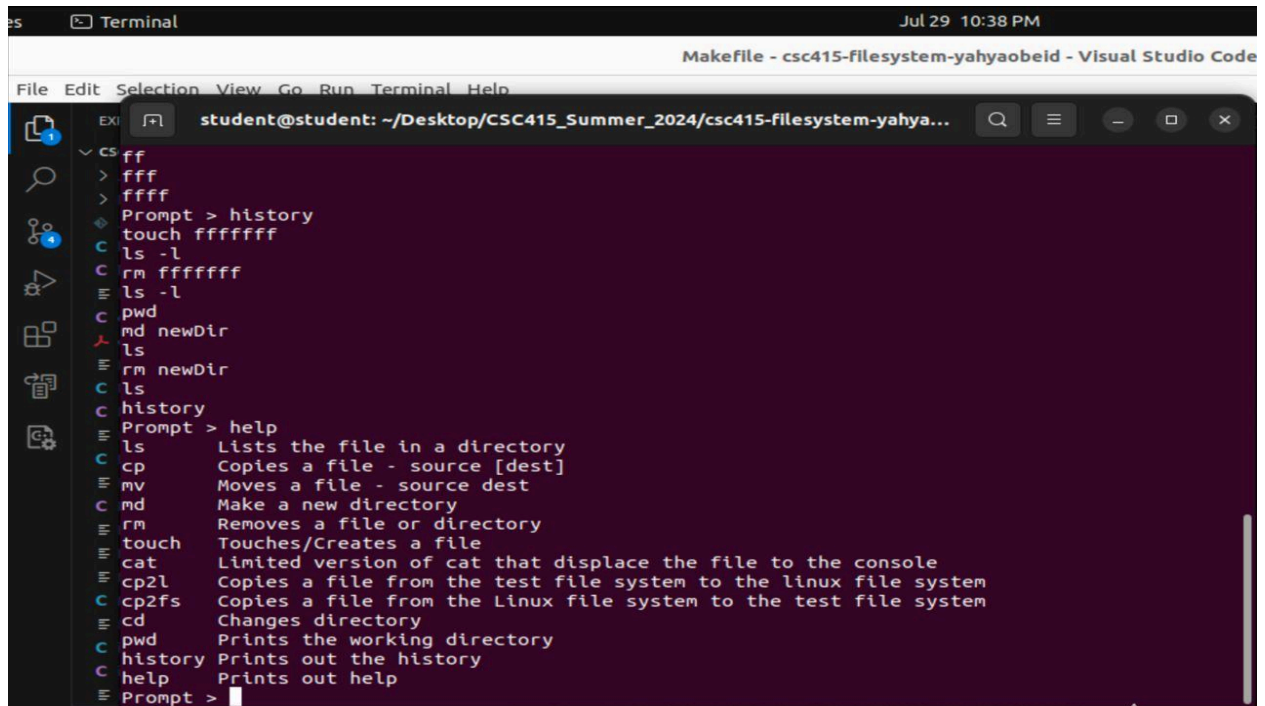
history:



A terminal window titled 'student@student: ~/Desktop/CSC415\_Summer\_2024/csc415-filesystem-yahya...' displays a series of commands and their outputs. The commands include creating a directory 'newDir', allocating blocks, listing files, removing 'newDir', touching 'ffffff', and using 'ls -l' to view file details. The 'history' command is used to list previous commands.

```
cmd_md: Creating directory newDir
> allocateBlocks: Allocating 33 blocks with a minimum of 1
> cmd_md: Directory newDir created successfully
Prompt > ls
f
ff
fff
ffff
newDir
Prompt > rm newDir
Prompt > ls
f
ff
fff
ffff
Prompt > history
touch fffffff
ls -l
rm fffffff
ls -l
pwd
md newDir
ls
rm newDir
ls
history
Prompt >
```

help:



A terminal window titled 'student@student: ~/Desktop/CSC415\_Summer\_2024/csc415-filesystem-yahya...' shows the output of the 'help' command. It lists various Linux commands and their functions, such as 'ls' for listing files, 'cp' for copying, 'mv' for moving, 'md' for making directories, 'rm' for removing, 'touch' for creating files, 'cat' for displaying file contents, 'cp2l' and 'cp2fs' for copying between file systems, 'cd' for changing directories, 'pwd' for printing the working directory, 'history' for printing command history, and 'help' for printing this help message.

```
ls
ls
ls
ls
Prompt > history
touch fffffff
ls -l
rm fffffff
ls -l
pwd
md newDir
ls
rm newDir
ls
ls
history
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 >
```

