Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

Files System Project

Group Name: RCJ

Github link for group submit:

https://github.com/CSC415-2024-Spring/csc415-filesystem-darren816

Description:

Our task requires developing a file system in C language, divided into three different stages. Initially, we focused on formatting the volume, a process that involves creating the volume control block, designing a mechanism to manage the free space, establishing the directory structure, and initializing the root directory. We then delved into the implementation of directory entry functions and created functions that handle file system input/output operations, including open, close, read, write, and seek operations. Additionally, our archive system has a hex dump tool that allows us to examine the contents of the root directory and VCB. The core component fsshell.c prompts the user to enter commands similar to Linux file systems, such as ls, cd, md, etc.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

Plans and changes at each stage:

Milestone One

Plan:

- Volume Control Block (VCB) Structure: Define the structure containing metadata about the file system.
- Free Space Tracking: Implement a bitmap-based approach to track free space on the disk.
- Directory Entry Structure: Define the structure to store information about files and directories.
- File System Metadata: Include support for extended attributes for files and directories.

Changes Made:

- Description of VCB Structure: Detailed the attributes and initialization functions related to the VCB.
- Description of Free Space Structure: Expanded on bitmap initialization and disk synchronization.
- Directory System Description: Provided details on directory listing, creation, deletion, and navigation functionalities.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

Milestone Two

Plan:

- Implement various directory functions like fs_mkdir, fs_opendir, fs_readdir, etc.
- Define and integrate directory entry structure for managing file attributes.
- Develop basic input/output operations within the file system using b_io.c.
- Enhance the volume control block functionalities.

Changes Made:

- Directory Functions Description: Provided detailed explanations for functions like fs_mkdir, fs_rmdir, etc.
- Volume Control Block: Further elaborated on the initialization process and its significance.

Overall Collaboration and Issue Resolution

- Team Collaboration: Utilized GitHub for version control and held regular meetings for progress updates and task assignments.
- Issues Faced: Encountered challenges in integrating components, managing memory allocation for bitmaps, and accessing directory entries.
- Resolution: Implemented rigorous testing, conducted code reviews, and maintained open communication to address challenges effectively.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

Milestone Three

Plan:

1. Direct Entry Structure:

- Define a structure to represent directory entries, including key details like name, author, size, etc.
- Implement an array of directory entries for holding directory information.
- Develop auxiliary structures for locating directory entries.

2. Free Space Management:

- Use a Bitmap array to manage available space in the file system.
- Implement functions like setBit() and clearBit() for manipulating the bitmap.
- Initialize the bitmap during file system initialization and manage it effectively.

3. Directory Functions:

- Develop functions like parse(), fs_mkdir(), fs_rmdir(), etc., for directory operations.
- Ensure proper parsing of path names and validation of directory operations.

4. Volume Control Block (VCB):

- Define and initialize the VCB structure containing vital file system information.

5. Basic Input/Output Operations (b_io.c):

- Implement functions like b_open(), b_close(), b_read(), b_write(), and b_seek() for file operations.
- Handle file opening, closing, reading, writing, and seeking efficiently.

Changes Made:

- Resolved challenges related to memory allocation for bitmaps and efficiently managing available space.
- Implemented functions for setting and clearing bits in the bitmap.
- Addressed difficulties in locating and loading directory entries into memory.
- Integrated open, read, and write functions into the shell, resolving errors related to accessing b_io.c functions.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

● **Components**:

1. Direct Entry Structure:

The directory project structure consists of several parts: name, author, size, location, last access, creation date, last modification, permissions, description, directory, block, and archive type. These elements store name, creator, size, location, access timestamp, creation, modification, permission settings, description, directory indicator, occupied block, and file type information respectively. In addition, there is an independent structure named locationDE, which contains deAddress and deIndex, which represent the address and index of the directory item respectively. In an archive system, the directory item structure acts as a container for all directories, encapsulating key details such as name, size, timestamp, and description. To enhance accessibility, an array of directory entries is created to hold NUMBER_OF_DE. In addition, auxiliary structures help locate the address and index of each directory entry.

```
#ifndef DIR_ENTRY_H
#define DIR_ENTRY_H

#include <time.h>
#define NUMBER_OF_DE 50
#define DE_SIZE 60

typedef struct{
    char name[50];
    char author[20];
    long size;
    int location;
    time_t last_accessed;
    time_t date_created;
    time_t last_modified;
    int permission;
    char description[255];
    int dir;
    int blocks;
    unsigned char fileType;
} dir_entry;

extern dir_entry DE[NUMBER_OF_DE];  // Declare DE as extern.

typedef struct{
    int deAdress;
    int deIndex;
} locationDE;

#endif // DIR_ENTRY_H
```

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

2. Free Space:

In the provided code, the Bitmap[] array serves as a key element in managing the available space within the file system. This array is used as a mechanism to track the allocation status of blocks, with each bit representing a block - 1 for an allocated block and 0 for a free block. Through the setBit() and clearBit() functions, specific bits can be manipulated to represent the allocation and release of blocks respectively, thereby ensuring effective management of available space. In file systems, the bitmap system plays a key role in determining which blocks can be assigned when creating a new file or directory. It operates in bytes, where each set of 8 bits represents a block, where 1 represents an allocated block and 0 represents a free block. The size of the bitmap is determined by the number of blocks multiplied by the block size and the integer size. Bitmap initialization occurs during file system initialization. Initial bits are assigned to the root and volume control blocks, and then the bitmap is written to disk using LBAwrite().

```
#include <Free_Space.h>
#include <time.h>
//bitwise function for setting a bit to 1
void setBit(int Bitmap[],int n){
    Bitmap[n/8] |= (1 << (n%8));
}
//bitwise function for clearing a bit to 0
void clearBit(int Bitmap[],int n){
    Bitmap[n/8] &= ~(1 << (n%8));
}
```

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

3. Directory Functions:

A. parse():

The function parse() processes the given path name and returns a structure holding the positions of n-1 directory entries and the position of the last element in that entry. Output -1 indicates that the pathname is invalid. This method involves parsing the string pathname into an array and checking whether the first index is "/". If so, it represents an absolute path; otherwise, it is a relative path. For relative paths, LBAread is executed from the current directory location, and for absolute paths, it is executed from the root directory. The data read is stored in the temporary directory project structure. Subsequently, iterate over the tag array containing each path element. At the same time, iterate over the array of directory entries to locate the path. Once the required path is found, the next directory entry is loaded until the last path is reached. At this time, n-1 directory items and the location of the directory item or the files in it are returned. If the path is not found during the iteration, -1 is returned, indicating that the path does not exist in the file system.

B. fs_mkdir():

In the process of creating a new directory, the mk_dir() function plays a pivotal role. Initially, it passes the provided path through the parse() function to ensure its validity. After confirming a valid path, the function will proceed to generate new directory entries. This involves assigning a name and timestamp value to the directory, with the name set to the new directory name and the timestamp reflecting the current time.

C. fs_rmdir():

The function rmdir() has the task of renaming a directory. Like mk_dir(), its initial task is to verify the validity of the provided path. Once confirmed, the function proceeds to create a new copy of the directory entry (DE), changing its name to the new desired name, and updating the last accessed and last modified time values to reflect the current time.

D. fs_opendir():

In the function fs_opendir(), a directory pathname is received as input and, if the directory path exists, a pointer to the fsDir structure is returned. The process first resolves the directory path to verify its existence. If the directory exists, the function proceeds to determine whether it is an archive or a directory. If it does not exist or is a file, the function terminates. Instead, if it is a valid directory, the directory is loaded into memory and a pointer to the structure containing its information is returned.

E. fs_readdir():

In this function, similar to fs_opendir(), our goal is to provide a pointer to a directory structure that contains information about the directory specified as an argument. However, unlike fs_opendir(), we receive a pointer to the directory pathname instead of getting the pathname directly. This change allows for more direct reference to directory information, simplifying the process of accessing and manipulating directory data.

F. fs_closedir():

close_dir() verifies that the pathname exists and, if so, proceeds to release its memory allocation and set it to NULL, effectively closing it. This simple but crucial feature ensures that directory resources are managed correctly and memory is freed for other operations.

G. **fs_getcwd():**

The purpose of this function is to retrieve the current working directory, but unfortunately, it is not implemented yet. One way to implement it might involve printing the name of the current directory, then navigating to its parent directory, and repeating this process iteratively until the root directory is reached. By traversing the directory structure in this way, we can effectively determine and display the current working directory.

H. fs_setcwd():

Although not yet implemented, this function is designed to update our current working directory to the new specified directory. One possible implementation strategy is to parse the provided path and load the directory corresponding to the path after confirming its validity. This process will enable a seamless transition to the newly designated directory in the file system.

I. fs_isFile():

The objective of this function is to determine if the provided pathname corresponds to a file. It accomplishes this by parsing the pathname and checking for indicators of a file. If the pathname signifies a file, the function returns 1; otherwise, it returns 0 to denote that it is not a file.

J. fs_isDir():

To determine if something is a directory, we can use the fs_isFile() function. If the function returns true, it means it is a file, and we also return true; otherwise, if it returns false, it means it is not a file, then it returns the negation of fs_isFile(), which means it is indeed a directory.

K.fs_delete():

The purpose of this function is to delete a directory. The process first resolves the path to the directory. It then searches the directory structure for a directory that matches the resolved path. After finding the directory, it sets its contents to NULL and 0, effectively deleting it. If parsing the path or finding a matching directory in the structure fails, the function returns -1 to indicate the error.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

4. Volume Control Block:

The volume control block (VCB) holds important information about the file system, including total block count, block size, free blocks, root location, and root size. During the first operation of our file system, VCB is initialized. The first time we run the file system, we use LBAread to read block 0 and verify its signature number. If the signature matches our magic number, it means the VCB has been initialized. On the contrary, if the VCB is not initialized, we initialize it by loading parameters such as root size, total block and block size. We then use LBAWrite() to write the initialized VCB to memory.

5. b_io.c:

In b_io.c, the goal is to develop functions responsible for basic input/output operations within the file system. The main functions include b_open(), b_close(), b_read(), b_write(), and b_seek(), each of which plays a vital role in facilitating file processing and operations within the system.

A. b_open():

This function acts as an interface for opening buffered files, handling flags similar to those in the linux open() function, such as O_CREAT, O_TRUNC, and O_APPEND. When using O_CREAT, the goal is to create a regular archive if the specified path name does not exist. Initially, the path is parsed and a new archive named with the pathname is created. For O_TRUNC, open the file for writing and perform an LBAwrite operation on the returned file descriptor (fd) and the new scratch directory entry. O_APPEND, on the other hand, opens the file for update and b_seek()s the fcbarray using the SEEK_END flag.

B. b_close():

The purpose of this function is simple: close the specified file. It locates the file using the provided path, then frees it from memory and sets its buffer back to NULL. This procedure ensures that resources associated with the file are properly released and buffers are reset for future operations.

C. b_read():

The function b_read() is used to read the buffer in the file system and is divided into three different parts. First, part 1 involves filling as much content as possible from the buffer. If there is any remaining data, Part 2 handles the remaining data by performing an LBAread() operation on it. This is done by combining the remaining contents of the buffer with the contents read in Part 1, "Refill". After completing these three parts, the total number of bytes read is returned.

D. b_write():

The goal of b_write() is to write to a buffer within the file system. Like b_read(), the function is divided into three parts, each dealing with a specific aspect of the buffer. However, unlike b_read() which uses LBAread(), b_write() uses LBAwrite() to perform its operation. This difference ensures that data is written to the file system efficiently and accurately.

E. b_seek():

The goal of b_seek() is to relocate the current file's position in the stream to a new specified position in the file. This function handles three cases: SEEK_SET, SEEK_END and SEEK_CUR. With SEEK_SET, the file's position is set to the offset's position. In contrast, SEEK_END adjusts the position to be equal to the negative value of the offset. For SEEK_CUR, the file's position is set to the current position plus the selected offset. This comprehensive approach ensures precise navigation in the file flow.

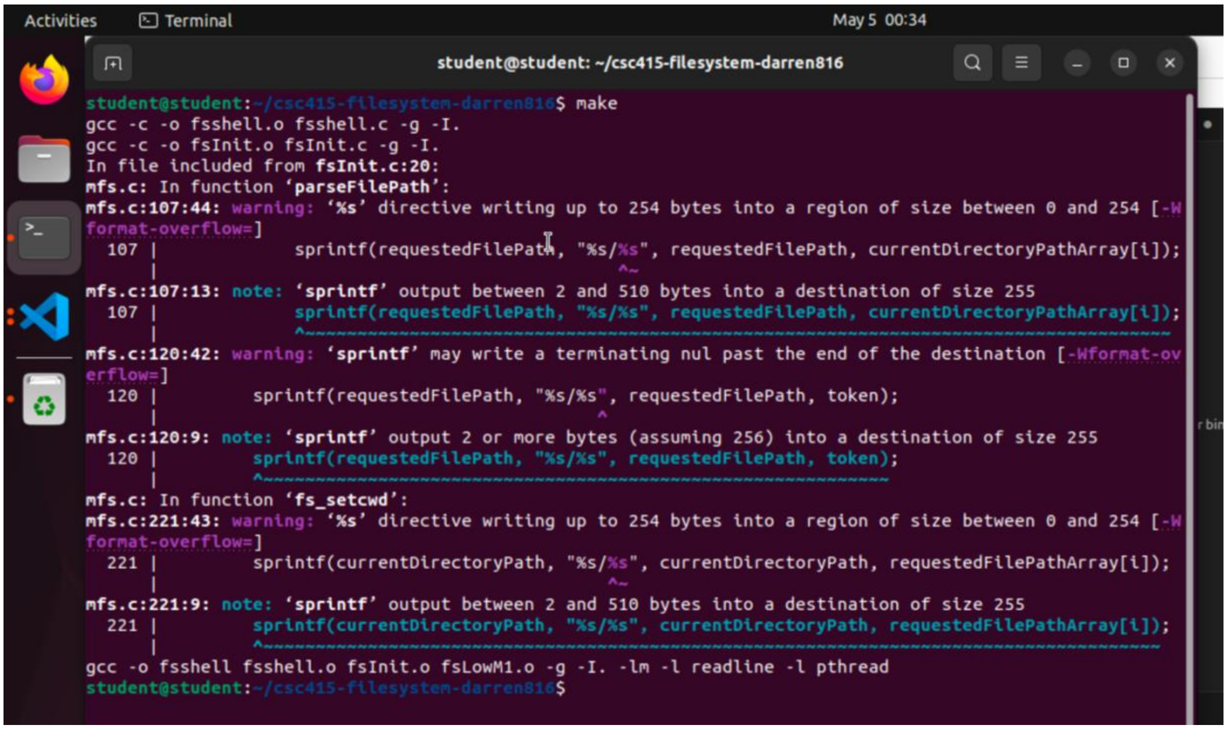
Issues and Resolutions:

We encountered some challenges when dealing with the file system. A major obstacle is determining the appropriate memory allocation for bitmaps, leading to problems in efficiently managing the available space. Additionally, creating functions for setting and clearing bits proved problematic. After you create a bitmap and its helper functions, it can be difficult to access and update it from other functions, especially when you add more space or set bits during the creation of a new directory entry. Another challenge involves locating the directory entry being parsed and loading it into memory. Similarly, accessing the array of directory entries in the b_io.c function encounters similar difficulties as the directory functions. Additionally, integrating the open(), read(), and write() functions in the shell proved challenging, resulting in errors when trying to access commands related to b_io.c functions.

Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

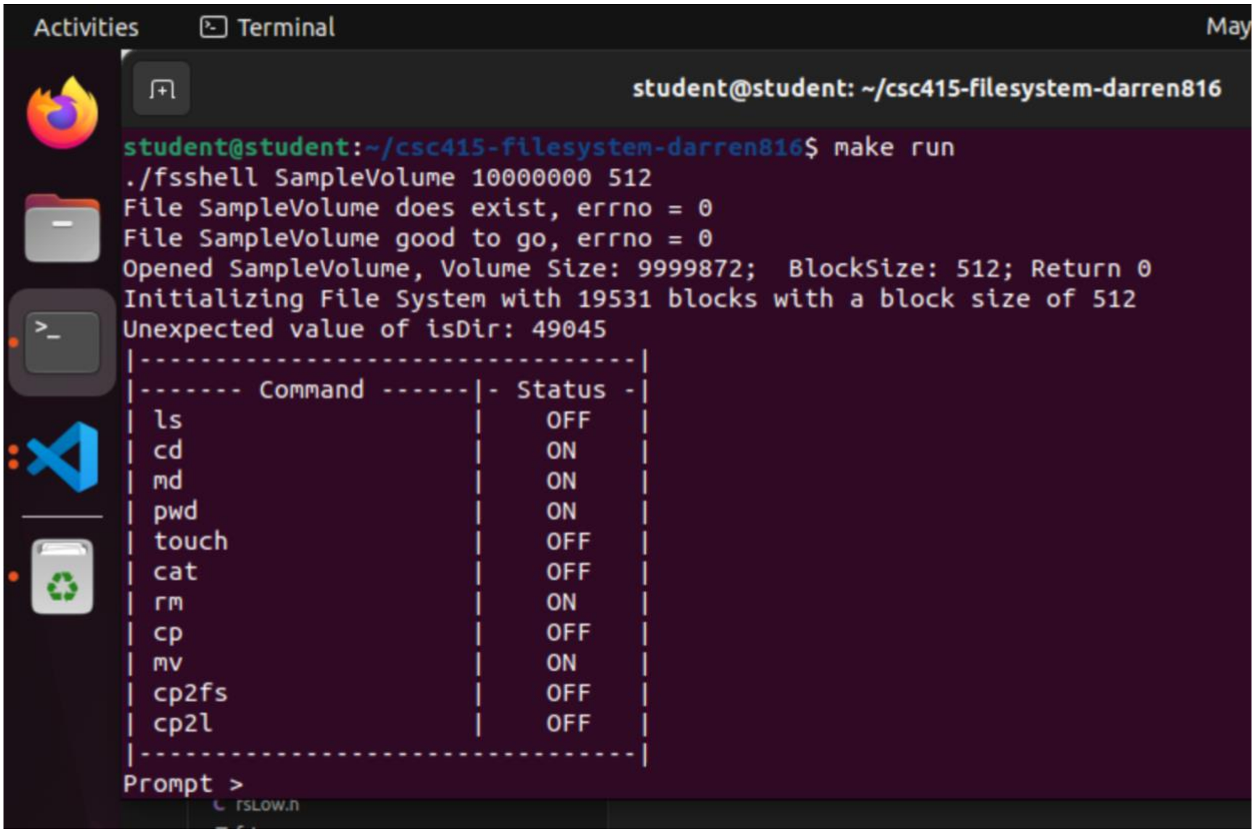
Screenshot of compilation:

make:



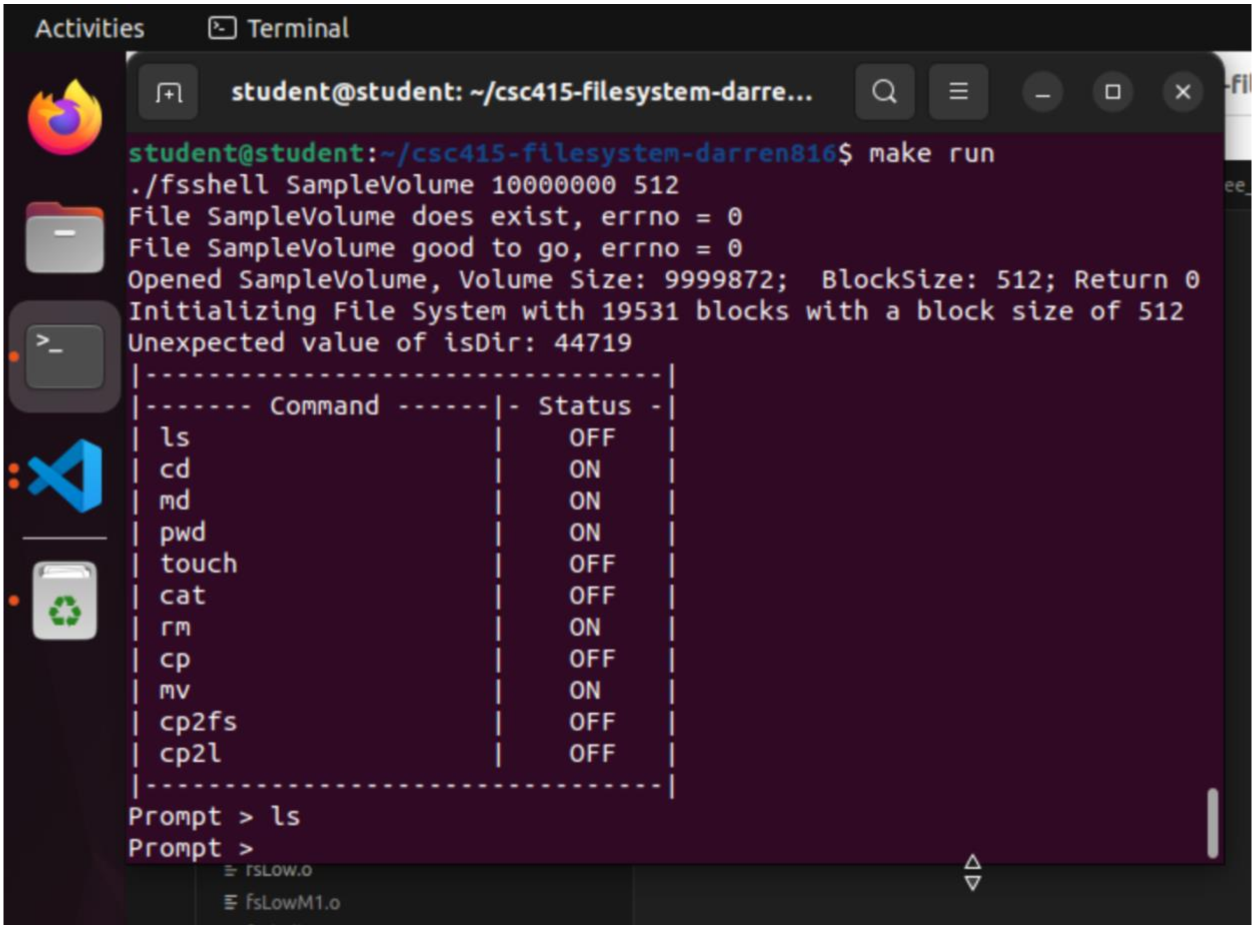
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

make run:



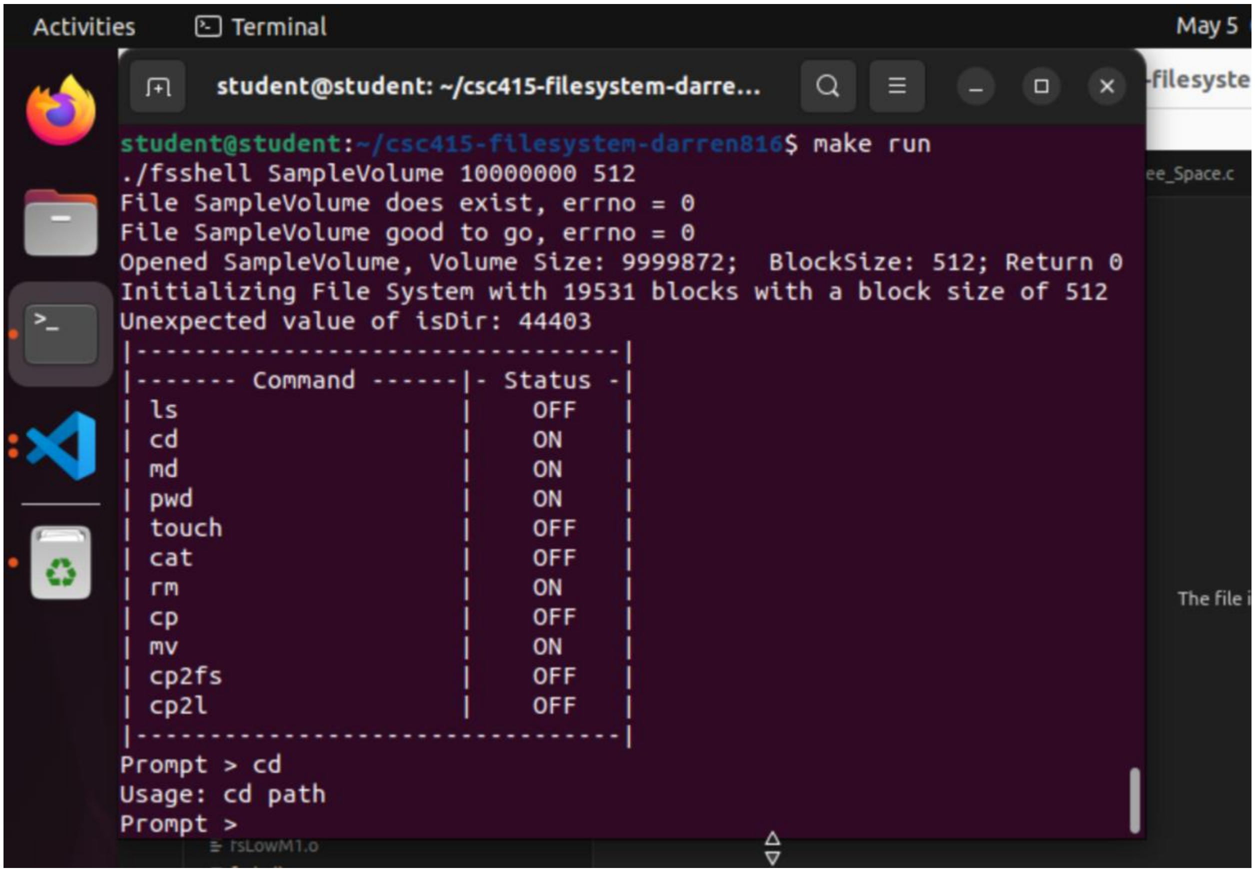
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

ls:



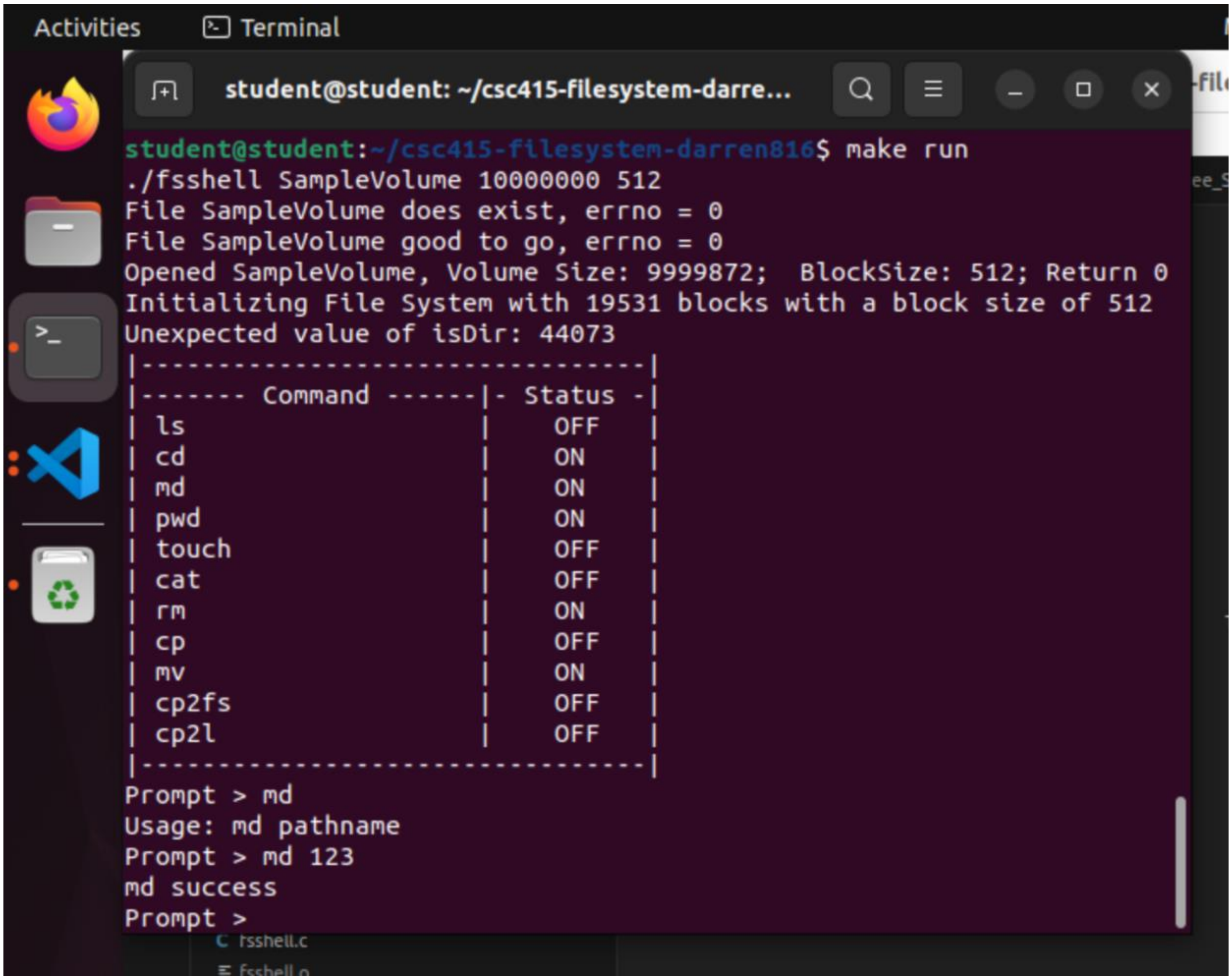
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

cd:



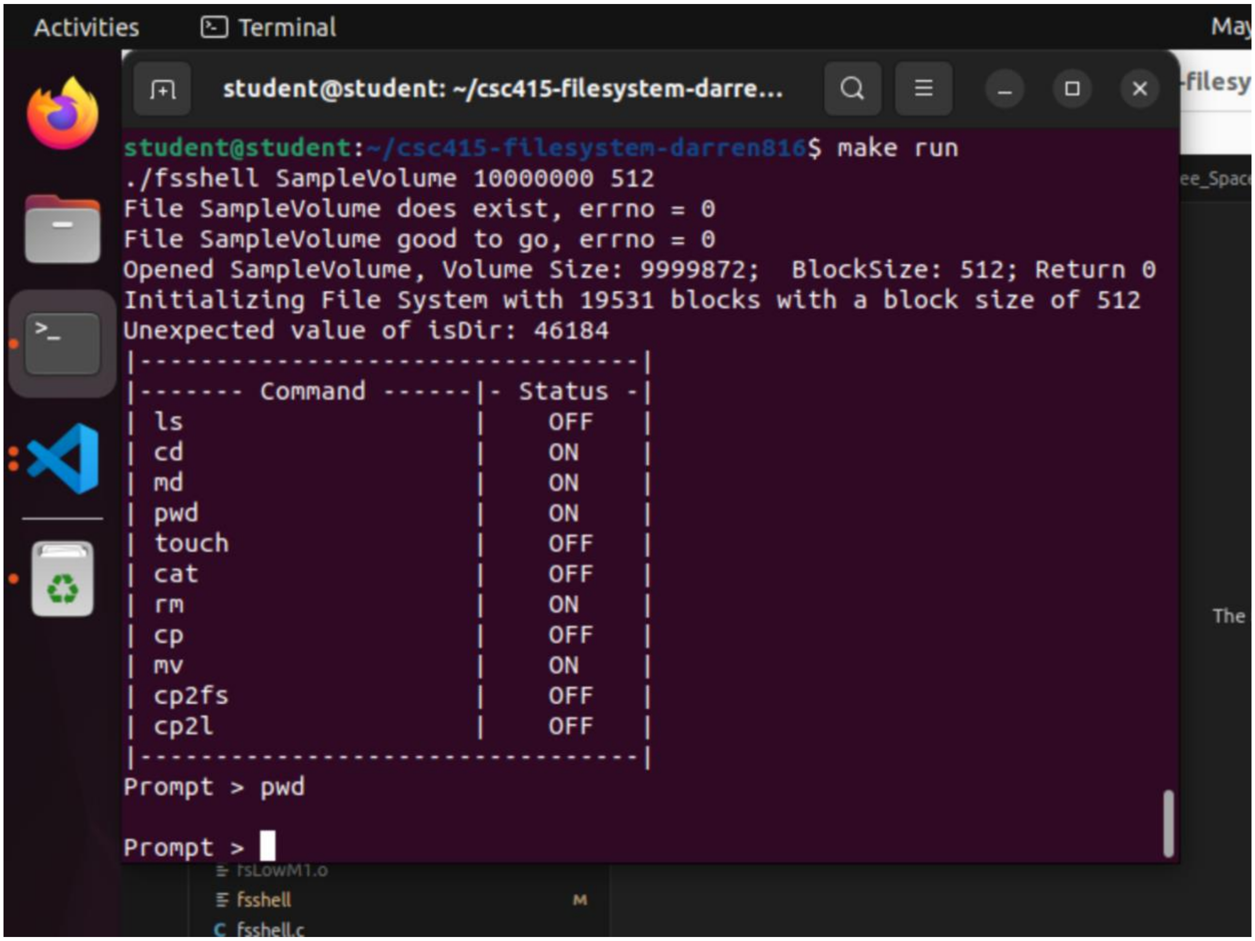
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

md:



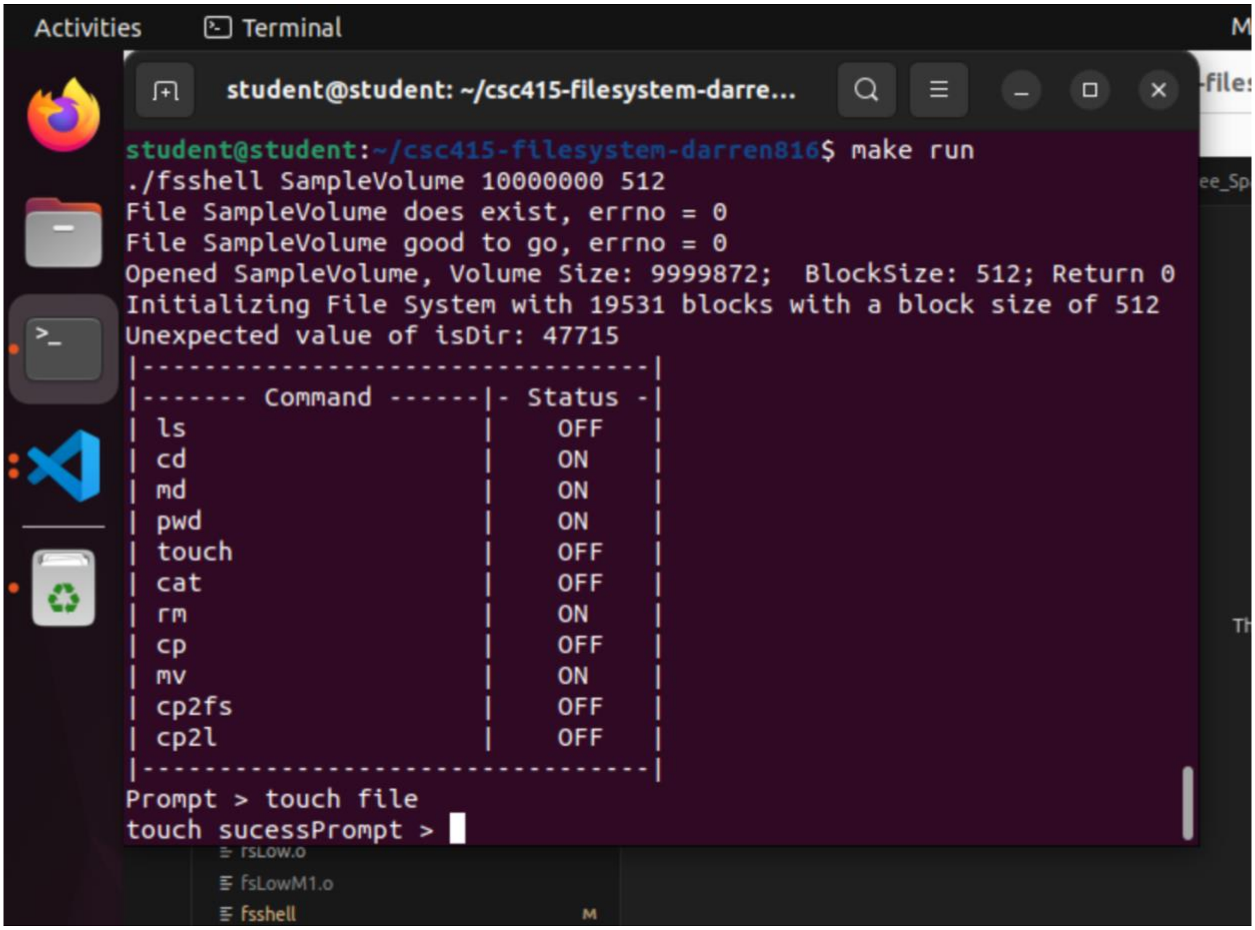
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

pwd:



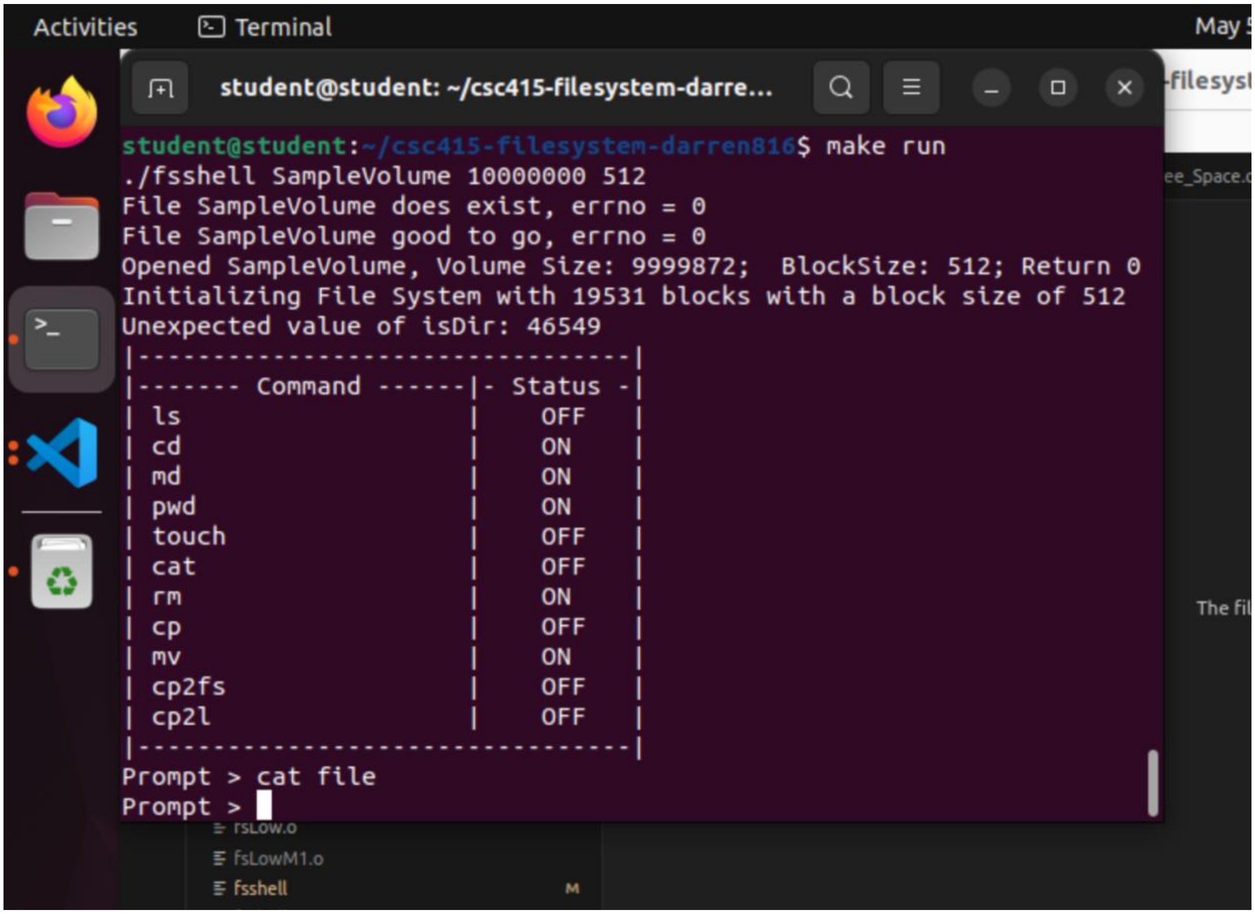
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

touch:



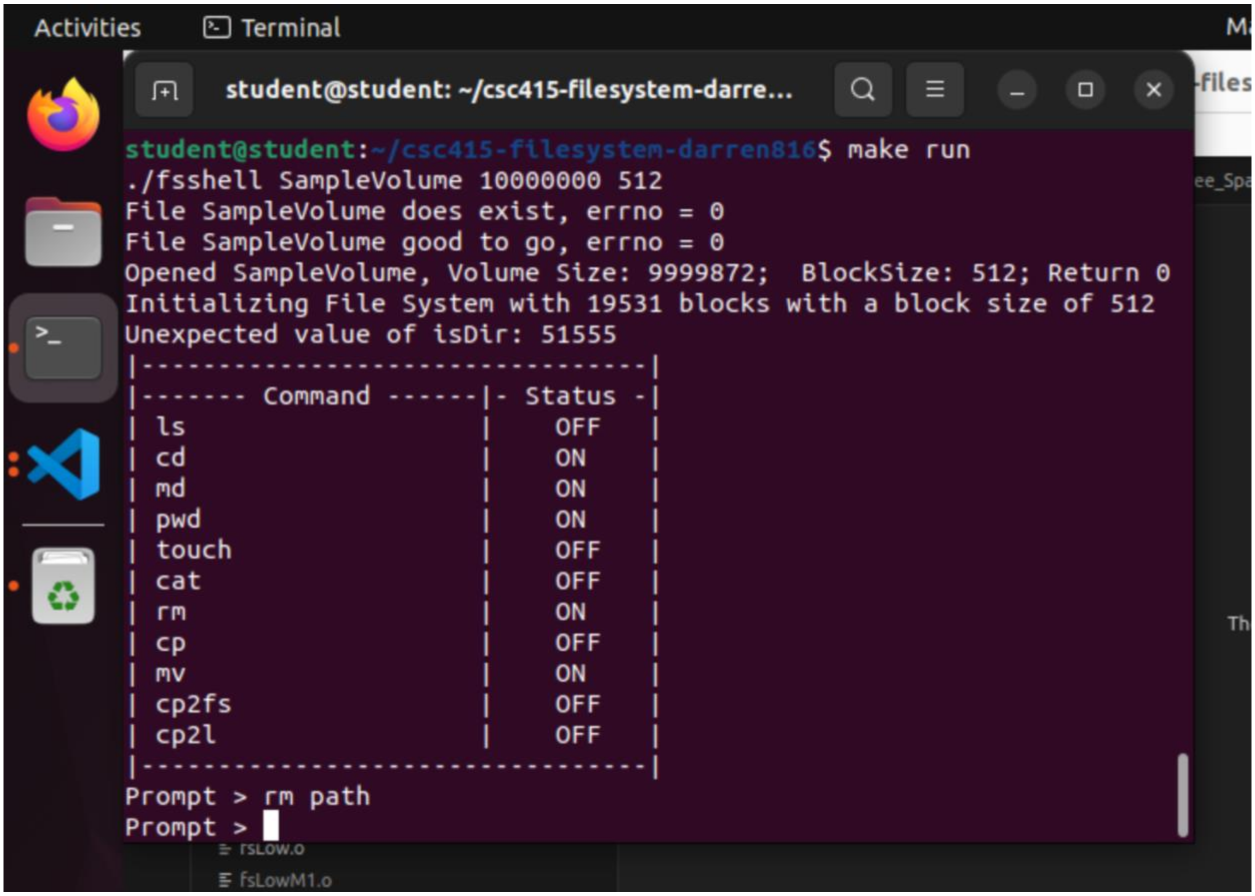
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

cat:



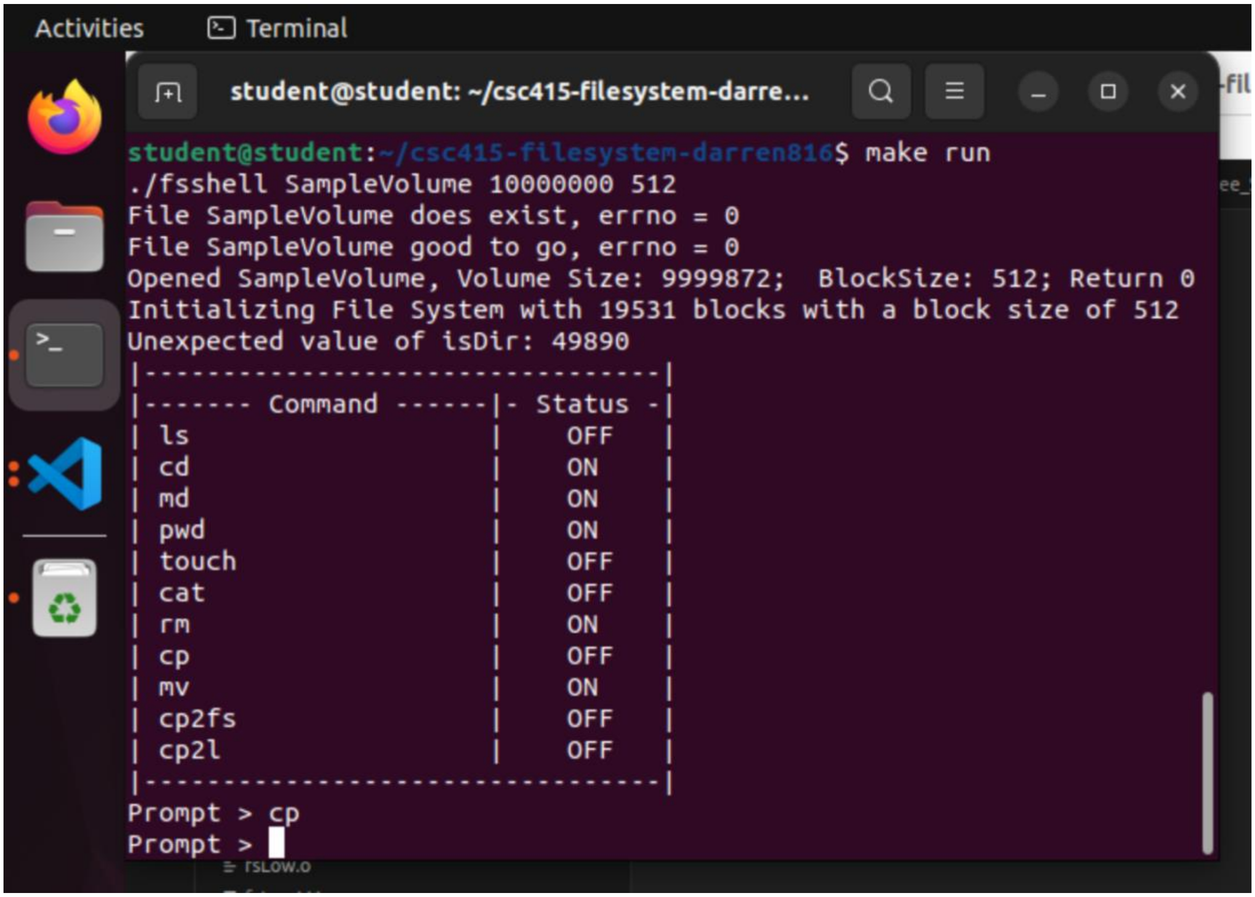
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

rm:



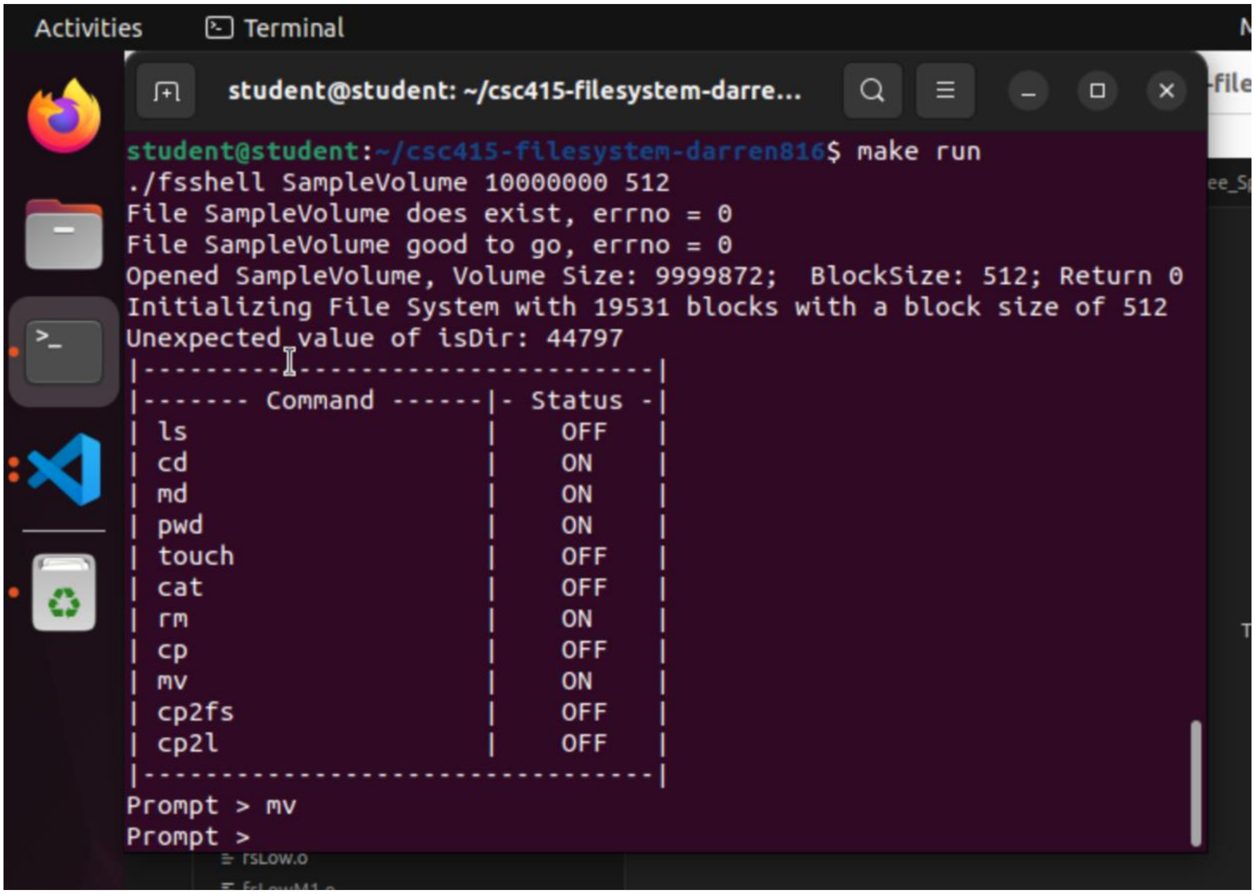
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

cp:



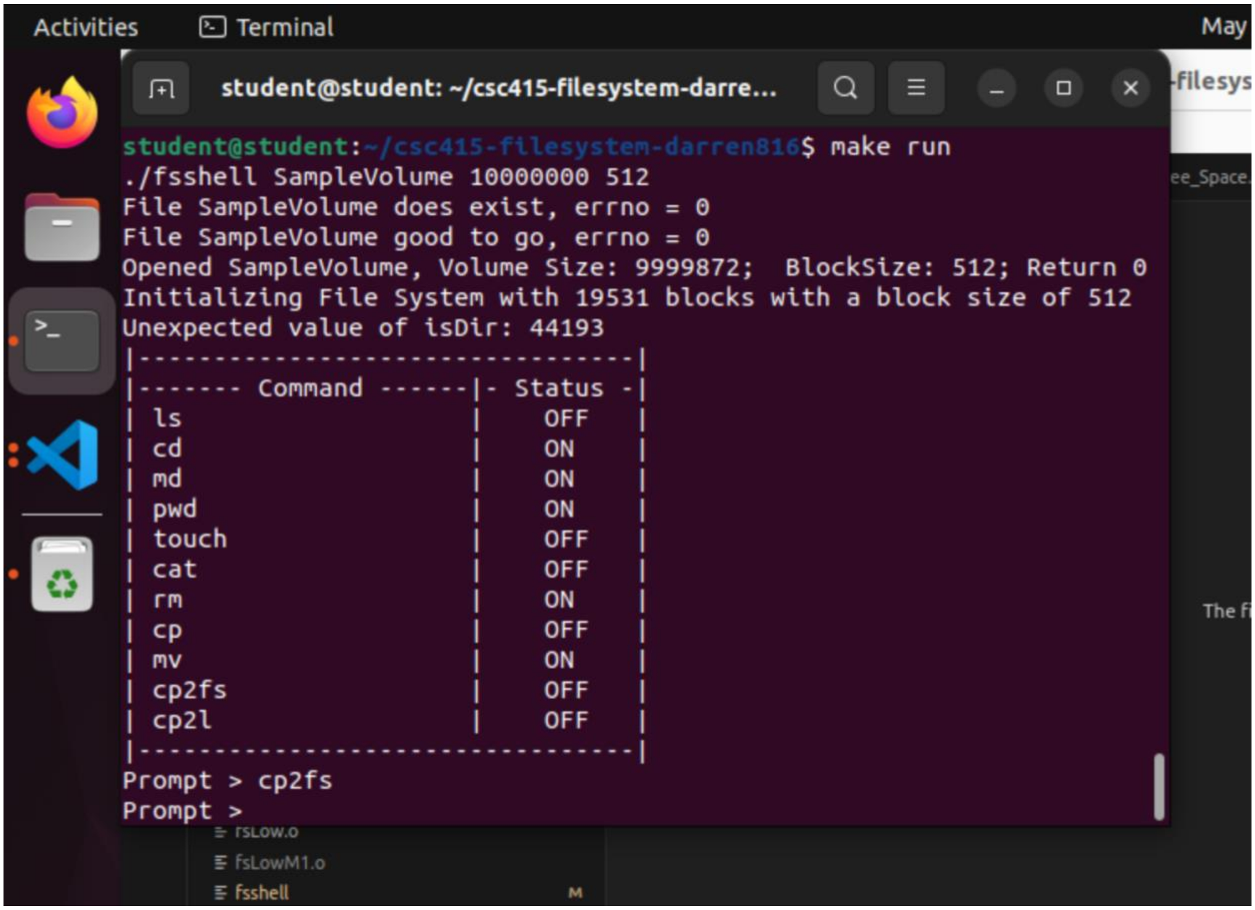
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

mv:



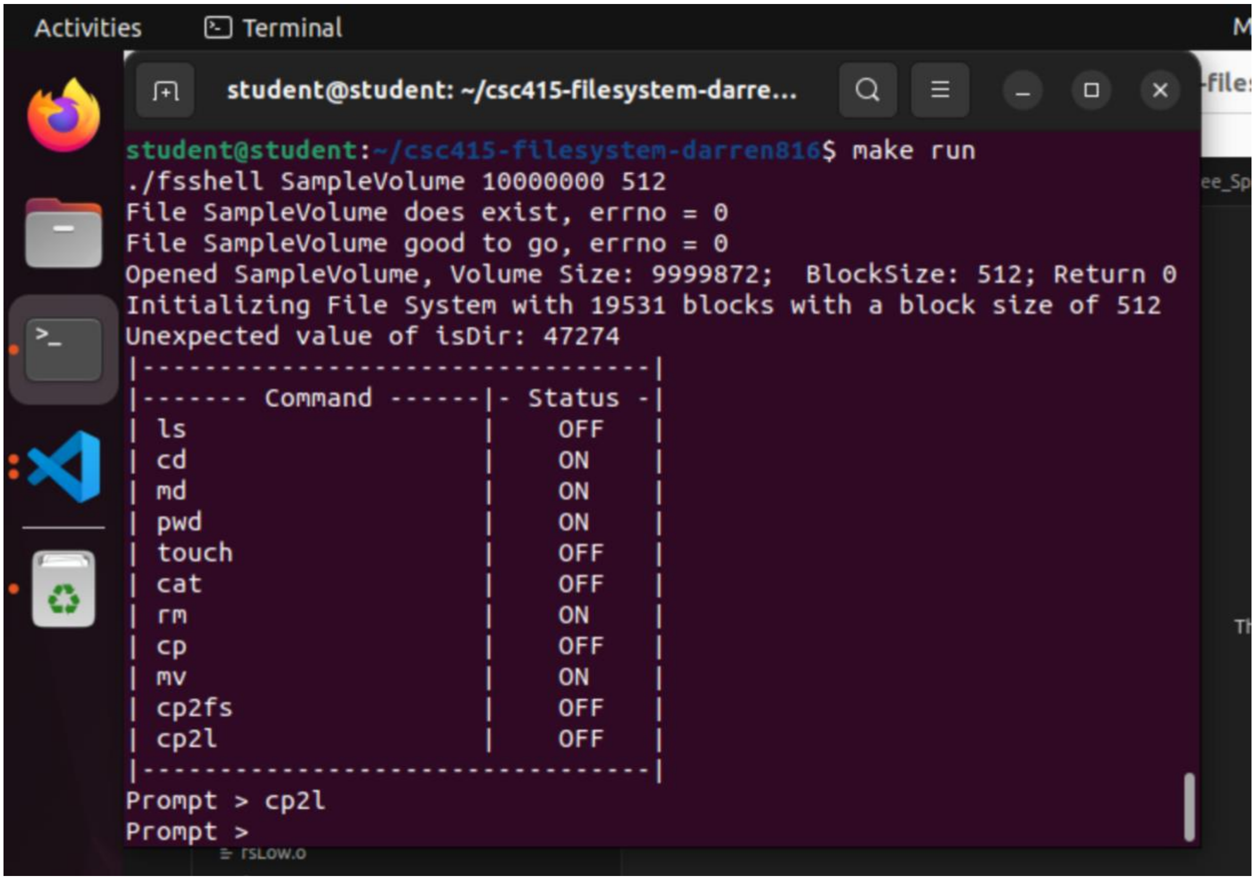
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

cp2fs:



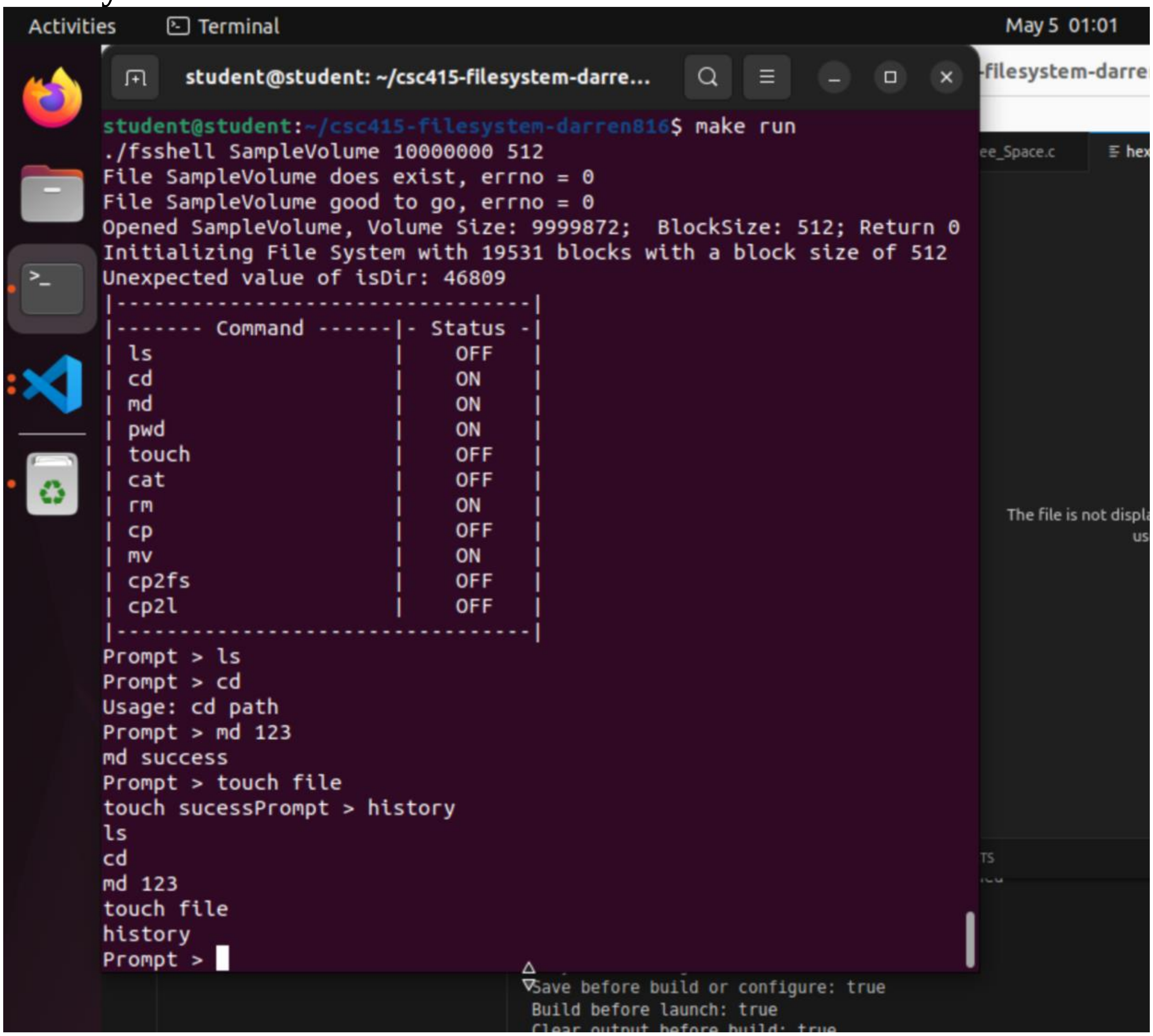
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

cp2l:



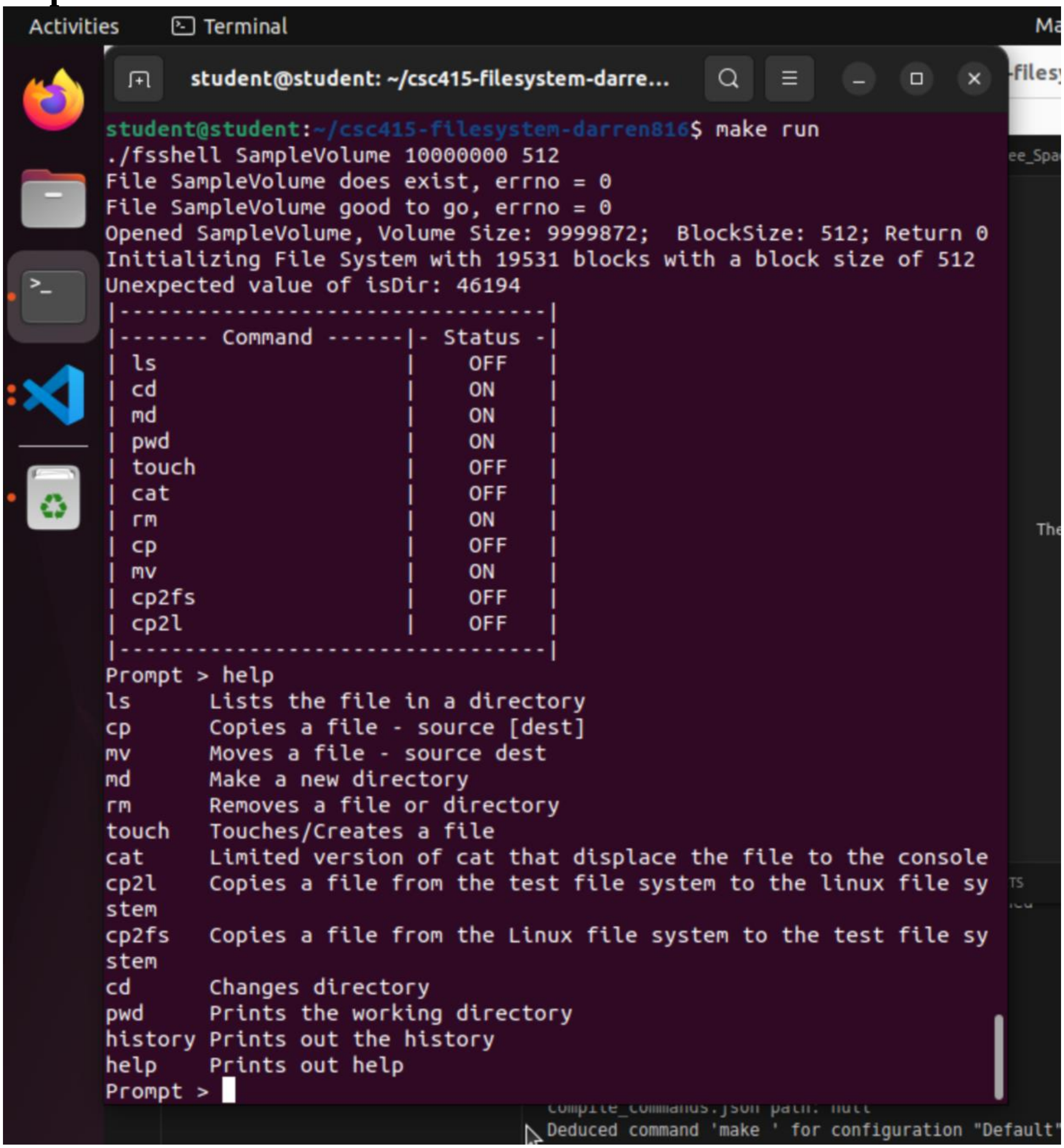
Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

history:



Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

help:



Name:	ID:	Github:
Po Han Chen	923446482	eric915c
Hsin Ying Tsai	923503916	Golden1018
Hsueh Ta Lu	923409640	darren816
Tung Ying Lee	923407911	nlty0122

hex dumps:

```
Dumping file ./hexdump, starting at block 0 for 36 blocks:

000000: 7F 45 4C 46 02 01 01 00  00 00 00 00 00 00 00 00 | ^?ELF...........
000010: 03 00 3E 00 01 00 00 00  E0 09 00 00 00 00 00 00 | ..>.....�.......
000020: 40 00 00 00 00 00 00 00  40 3F 00 00 00 00 00 00 | @.......@?......
000030: 00 00 00 00 40 00 38 00  09 00 40 00 23 00 22 00 | ....@.8...@.#.".
000040: 06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00 | ........@.......
000050: 40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00 | @.......@.......
000060: F8 01 00 00 00 00 00 00  F8 01 00 00 00 00 00 00 | �.......�.......
000070: 08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00 | ................
000080: 38 02 00 00 00 00 00 00  38 02 00 00 00 00 00 00 | 8.......8.......
000090: 38 02 00 00 00 00 00 00  1C 00 00 00 00 00 00 00 | 8...............
0000A0: 1C 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00 | ................
0000B0: 01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00 | ................
0000C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0000D0: 70 1C 00 00 00 00 00 00  70 1C 00 00 00 00 00 00 | p.......p.......
0000E0: 00 00 20 00 00 00 00 00  01 00 00 00 06 00 00 00 | .. .............
0000F0: 48 1D 00 00 00 00 00 00  48 1D 20 00 00 00 00 00 | H.......H. .....

000100: 48 1D 20 00 00 00 00 00  98 03 00 00 00 00 00 00 | H. .....�.......
000110: B0 03 00 00 00 00 00 00  00 00 20 00 00 00 00 00 | �......... .....
000120: 02 00 00 00 06 00 00 00  58 1D 00 00 00 00 00 00 | ........X.......
000130: 58 1D 20 00 00 00 00 00  58 1D 20 00 00 00 00 00 | X. .....X. .....
000140: 00 02 00 00 00 00 00 00  00 02 00 00 00 00 00 00 | ................
000150: 08 00 00 00 00 00 00 00  04 00 00 00 04 00 00 00 | ................
000160: 54 02 00 00 00 00 00 00  54 02 00 00 00 00 00 00 | T.......T.......
```