JMWT File System

Student Name:	Student ID:
John Cuevas	920542932
Michael Abolencia	917581956
William Pan	922967228
Tina Chou	922911207

Github for submission:

Name: WilliamPanTW

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

San Francisco State University

Prof. Robert Bierman

May 6 ,2024

1. A description of your file system

The primary objective of this assignment is to develop a C-based file system program that mimics the structure utilized by an operating system for file management on a virtual custom disk, termed "SampleVolume". This assignment encompasses various components, including formatting the volume, establishing and managing a free space management system, initializing a root directory, performing folder and file directory operations, such as creating, reading, writing, and deleting files, and displaying information using the correct commands. Effective communication and error handling are key to ensuring we meet the project requirements.

The core functionality of our program lies in its ability to interact through command-line input, offering a seamless experience akin to the use of a Linux shell. Throughout the project, we implemented key commands such as directory navigation (cd), file listing (ls), creating new directories (md), creating new files (touch), and deleting files or directories (rm) to perform essential file and directory operations. Additionally, we incorporated features like moving files within directories (mv), copying files within the file system (cp), copying files between the file system(cp2fs) and Linux systems (cp2l), and verifying file integrity with the addition of a command for displaying file contents (cat). These functions helped us better understand low-level file functionality, advanced buffering, free space allocation, and release management and ensured the accuracy of tracking file information with persistent directory structures.

This large-scale project demanded careful planning and experimentation as we engaged four individuals in its development. We got set for brainstorming sessions and experienced plenty of trial and error along the way. It was a journey filled with learning opportunities as we aimed to create something outstanding.

2. The plan for each phase and changes made

2.1. Milestone 1 - Plan

The original plan for milestone 1 is relevant, simple and wrong.

1. Volume control block structure contains necessary component members, such as a 'signature' for block verification, if it has already been initiated. Block,root and free_block index indices block counts, and size for block,root and free_block in bytes count.

2. The approach of free space is using a bitmap with helper functions of get set and clear bit. That is indicated if the bits were ‘0’ as free and ‘1’ as occupied or unavailable blocks to use.

3. We initial root directory onto the disk(sample Volume) using 'writeRootDirectory' function.However, we need to address how to handle other directories since our current setup assigns 'dot' and 'dot dot' as it self, which might not work well for directories with different parent-child relation.

Name	**Key**	**Helper**
Michael Abolencia	Volume control block (VCB)	VCB structure
Tina Chou	Free Space managment (bitmap)	initFreeSpace() ,set_bit , get_bit , clear_bit
William Pan	Root Directory	writeRootDirectory()

2.2. Milestone 1 - Change

- To enhance the readability and efficiency of the hexdump display, We have moved the signature right up to the top. Then, add the free block index so you can easily keep track of the latest unused index. After that, we've paired up blocks to represent the VCB, bitmap, and root. Each with an index that tells you where it is on the disk and size to tell how big it is in blocks.

- Change the VCB signature to our own custom 2^128 signature

- Initially, we cannot “LBAread " or it will cause error because we access the VCB directly from the struct variable before initializing. Thus by changing to dynamic allocating memory for struct VCB, we were able to access “LBAread " through the struct pointer. This enables the system to read and display correct content at block 1 of the hexdump.

- Fixed the bitmap and encapsulated with the extent structure with index of start and number of blocks. That was stored in each directory entry as reference to track information on disk correctly.

- By understanding the LABwrite function, we've successfully implemented a root directory, aligning it with the bitmap to safeguard against overwrites. This integration allows us to load directory entries and the Volume Control Block (VCB) as needed,

2.3. Milestone 2 - Plan

Name	Key	Helper
Michael Abolencia	**Md command:** Make Directory	**1. fs_ Mkdir** **1.1. parePath:** check if the path is valid or not **1.2. findUnusedDE:** Find unused entry **1.3. findDirEntry:** find directory by name **1.4. isDirectory:** check if it have marked as directory **1.5. loadDir:** Loads a directory entry from disk into memory
Michael Abolencia	**Rm command** Remove directory	**1. fs_isDir:** Check if it is directory **2. fs_rmdir:** Remove a directory **3. fs_isFile:** Check if it a file **4. Fs_delete:** Remove a file
John Cuevas	**cd command:** Implement **Fs_setcwd** function to change directory	**Step1. take the path specified to temp buffer** **Step2. call parsepath to check if it valid** **Step3. Check if the parent directory doesn't exists** **Step4. look at parent[index], is it a directory** **Step5. Load Directory** **Step6. Updating the string for user** **Step7. Determine if it absolute or relative path** **Step8. Determine the dot and dot dot** **a. Tokenize the substring to ignore it self(dot) and adding "/" for future implement path** **Step9. Load current to temp**

William Pan	**Ls command** List the file in a directory	Have argument -> is dir -> opendir -> display No argument(CWD) -> fs_getcwd -> fs_opendir - > display **1. fs_isDir** Check if it is directory **2. Fs_opendir :** opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory. **3. displayFiles(In shell)** **3.1. Fs_readdir** mfs.helper.c provide the caller with information about each file as it iterated through directory **3.2. Fs_stat** retrieve file status information **3.3. fs_close** If the steam is invalid Deallocates directory memory **4. fs_isFile** Check if it a file **5. fs_getcwd** Provide valid current working directory
Tina Chou	**PWD command** Prints the working directory	**1. Fs_getcwd** Copy the current working directory string back to the buffer with the limit of size

2.4. Milestone 2 change

- **Added bitmap.c and bitmap.h**
  For better clean code we create an additional bimap.c file to store the contain function for our free space management including tracks,set,clear,allocation,release of blocks.

- **Modify dictionary function int initRootDir(uint64_t numberOfBlocks)**
  By reference parse path function we add one more parameter to createDirectory(uint64_t entries_number, struct pp_return_struct* info). Which were able to create a root directory and dictionary accordingly.

- **Modify fs_getcwd parameter**
  After watching the lecture we decide to change the Directory Functions we know that Copy the current working directory string to the buffer with the limit of size, thus we change the origin function of buffer name "pathname" to pathbuffer char * fs_getcwd(char *pathbuffer, size_t size); with better readability and clarity.

- **Fixed bitmap do not load exist directory after system closed**
  We forget to added lbaread after loaded free space if it is initialize, thus our free map only get to where we set in loaded root, by adding the lbaread to fsmap we are able to pass in the correct bitmap and make other code without overwrite after root directory .

- **Added free space and root directory loader**
  Initial we only check if Volume Control Block (VCB) is exits, now we added two proceeds to load bitmap(free space) and root directory information

- **Added global c file and header file**
  After some research, create a global.h file with "extern" of multiple global definitions, that will be included in the needed file. Then assign those global variables in the global.c file to prevent multiple declarations throughout the project.

- **Added mfs_helper.c and mfs_helper.h**

  This file contains functions for multiple file system operations that interact with the file system, including directory navigation, file status retrieval, directory entry parsing, and space deallocation. Which is a helper function file for mfs.c .

2.5. Milestone 3 Plan

Name	Key	Helper
William Pan	**Struct f_cb** Add needed info	1. **int numBlocks;** hold how many block file occupies int 2. **currentBlock;** hold the current block number 3. **int mode;** hold the mode from flag struct 4. **dirEntry * file;** hold the file directory struct 5. **dirEntry * parent;** hold the parent directory
William Pan	**Flag**	1. **O_CREAT** Create a file if the file does not exist. 2. **O_TRUNC** Truncate the file and free block associate to zero. 3. **O_APPEND** Set the position of file
William Pan	**Mod**	1. **O_RDONLY** (0) Open for reading only. 2. **O_WRONLY** (1) Open for writing only 3. **O_RDWR** (2) Open for reading and writing
William Pan	**Touch command**	1. **b_open** Interface to open buffered file a. Initial file control block information 2. **createfile** a. If the file does not exist b. And flag is set then [process create file c. Mark not directory and write back to disk 3. **b_close** Close the associate directory stream

John Cuevas	**MV command**	**1. Setup shell.c** a. Similar to cmd_cp to check argument b. Check if it is file that able to move c. Copy source file to destination d. Remove the origin file **2. moveFile function for part c** a. Copy file to destination directory b. Update time and parent c. Write it back to disk **3. Use fs_deletd for part d** Remove file from origin destination
Tina Chou	**Cp command**	**1. B_open** Interface to open buffered file **a.** Initial file control block information **2. b_read** Interface to read a buffer a. If mod is write only then denied b. Divide in three part as lecture **3. b_write** Interface to write function a. if read only then denied b. Write until b_read stop return c. Refill buffer if necessary d. Update all information e. Write back disk **4. b_close** Close the associate directory stream
TinaChou	**Cp2l command** Copies a file from the test file system to the linux file system	**Step1. testfs_fd= b_open (src, O_RDONLY)** Open source file from our File system **Step2. linux_fd = open (dest,| O_CREAT)** Open linux dictionary and create file **Step3. b_read (testfs, buf,BUFFERLEN)** **a.** Copy source file content from our file system,by filling out the buffer in block. **Step4. write (linux_fd, buf, readcnt)** Write file back to linux file system until is done **Step5. close** Close linux and file system directory stream.

Tina Chou	**Cp2fs command** Copies a file from the Linux file system to the test file system	**Step1. linux_fd = open (src, O_RDONLY);** Open source file from linux file system **Step2. testfs_fd = b_open (dest, O_CREAT)** Open our file system dictionary and create file **Step3. read (linux_fd, buf, BUFFERLEN);** **a.** Copy source file content from linux file system, by filling out the buffer in block. **Step4. b_write (testfs_fd, buf, readcnt);** Write file back to our file system until is done **Step5. close** Close linux and file system directory stream
Michael Abolencia	**Cat command**	**1. b_open (src, O_RDONLY)** **Open source file and set as read only** **2. b_read (testfs_src_fd, buf, BUFFERLEN-1)** **a.** Copy source file content from our file system,by filling out the buffer in block. **3. Print the buffer from read** **4. b_close** Close associate directory stream
Michael Abolencia	**Seek**	**1. setting the file position**

2.6. Milestone 3 Change

- **Added b_io_helper.c and b_io_help.h**
  We create a b_io helper file to contain b_io.c needed operations. Including createFile for b_open , movefile and copyfile for move file command
- **Fixed touch**
  Added lbawrite after allocated space to make sure it write back
- **Added helper function to fixed movefile**
  Added copyfile function to better organized move file function
- **Assign specific mod to the file control block**
  Modular the flag with hexadecimal(16) to get the remain as mod

3. Issues

3.1. Milestone 1

Issues 01 - There is no sample Volume created, thus the hexdump could not execute properly
Resolution 01 - After making a Run, it generates the fsshell and samplevoulme. And the hexdump works properly.

Issues 02- Random number in 56-63 byte

```
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 42 20 74 72 65 62 6F 52  4B 4C 00 00 00 00 00 00 | B treboRKL......
000210: 00 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00 | ................
000220: 01 00 00 00 00 00 00 00  1D 00 00 00 00 00 00 00 | ................
000230: 06 00 00 00 00 00 00 00  C0 88 7B 7C 3B 5B 00 00 | ........{|;[..
```

Resolution 02 - The reason is unclear and needs further investigation to change the way we allocate volume control blocks**.(SOLVED in milestone 2)**

Issues 03 - The free_block_index works correctly with the check of printing out whether is used or free. However, the result in block 2 in hexdump indicates the free space has the wrong output showing our bitmap(fsmap) has not been updated and written to disk.

Resolution 03 - The logic would be to initialize and update the freespace according to the LBAwirte of how many blocks are used and starting from block 1.But need further try and error.

Issues 04 - Someone initial the one variable of volume control block in fsinit.c to test if it writes correctly, but the block 1 hexdump would show the variable away from the 000200 instead located at 000210.
Resolution 04 - After discussion and multiple tries we found out the order and the size in volume control block matter. And by arranging the order the hexdump works correctly.

Issues 05 - Someone initial the signature from the volume control block with their custom signature. **(Wrong and fixed in milestone 2)**

```
#define MINBLOCKSIZE 512
#define PART_SIGNATURE  0x526F626572742042
#define PART_SIGNATURE2 0x4220747265626F52
#define PART_CAPTION "CSC-415 - Operating Systems File System Partition Header\n\n"
```

Resolution 05 - Someone suggests the signature is already set in fslow.h. So changing will probably fit the requirements.

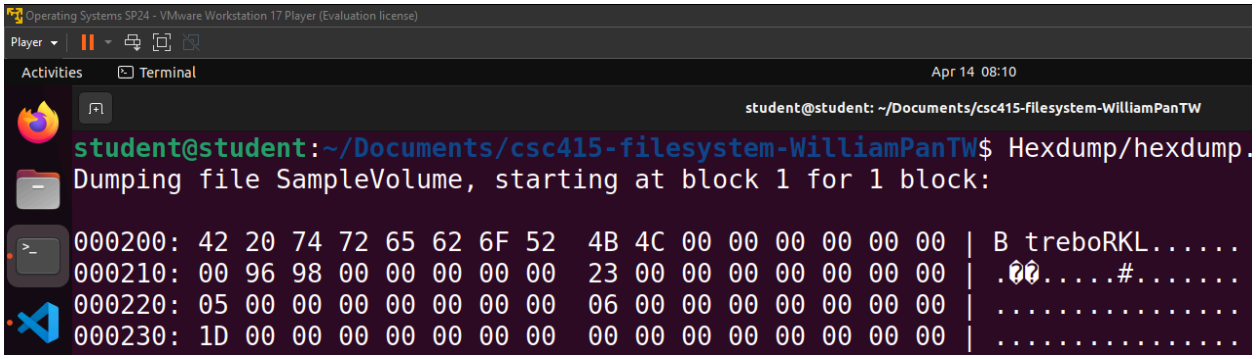
3.2. Milestone 2

Issues 01 - Random number in 56-63 byte (C0 88 7B 7C 3B 5B 00 00)

```
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 42 20 74 72 65 62 6F 52  4B 4C 00 00 00 00 00 00 | B treboRKL......
000210: 00 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00 | ................
000220: 01 00 00 00 00 00 00 00  1D 00 00 00 00 00 00 00 | ................
000230: 06 00 00 00 00 00 00 00  C0 88 7B 7C 3B 5B 00 00 | ........??{|;[..
```

Resolution 01 - The reason probably is because we allocate the volume control block on stack, after we allocate using malloc random number in 53-63 byte does not show up.



Issues 02 - The free_block_index in VCB works correctly to track the use and unuse of bitmap. But the hexdump in block 2 does print out matching results after Root initializes.

Resolution 02 - The logic in milestone one is wrong which forget to LBAwrite back,the amount of block is used. As screenshot below show 0x07FFFFFFF indicated [0111 1111 1111 1111 1111 1111 1111 1111 1111] of result of bitmap usage VCB(1) + Bitmap(5) + Root Directory (29) in total of 35 block were used.

```
student@student:~/Documents/csc415-filesystem-WilliamPanTW$ Hexdump/hexdump.linux SampleVolume --start 2 --count 1
Dumping file SampleVolume, starting at block 2 for 1 block:

000400: FF FF FF FF 07 00 00 00  00 00 00 00 00 00 00 00 | ????............
```

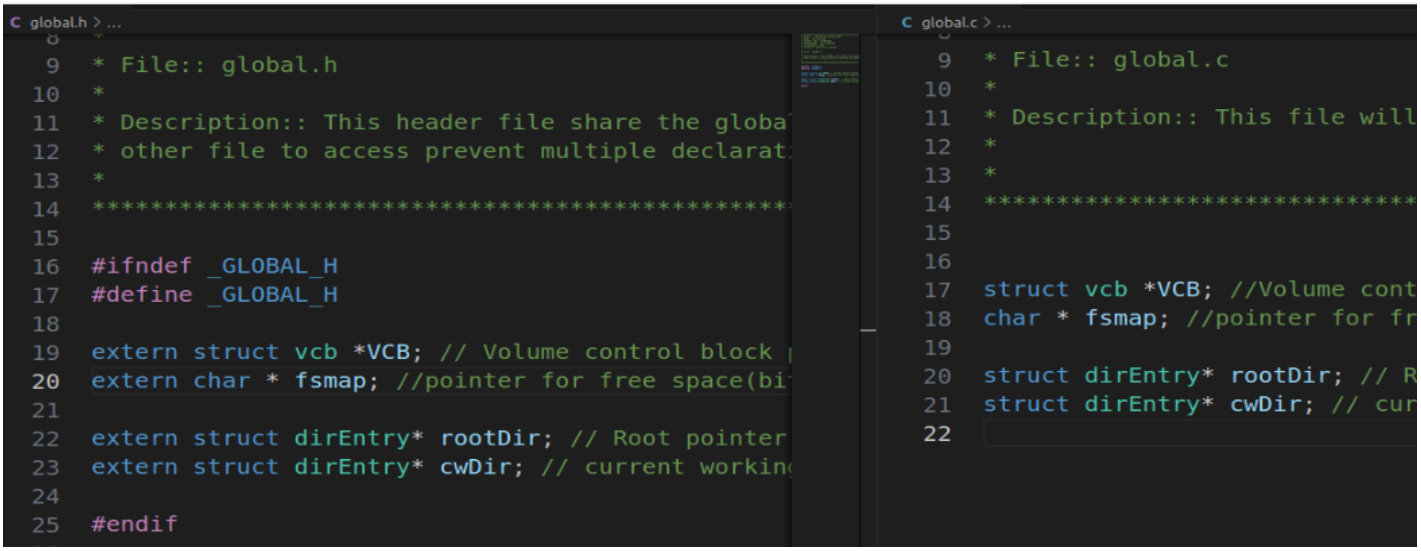
Issues 03 - PART_SIGNATURE and PART_SIGNATURE2 were all declared in block 0, thus we cannot use it as our VCB signature.

Resolution 03- Define our own 2^8 magic number **(Fixed milestone 1 issue 2)**

Issues 04 - Encounter multiple definitions of variable , because defined in header files are included in multiple source files. Resulting in sharing global variables that are not safe.

```
student@student:~/Documents/csc415-filesystem-WilliamPanTW$ make run
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -o fsshell fsshell.o fsInit.o fsLow.o -g -I. -lm -l readline -l pthread
/usr/bin/ld: fsInit.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/fsInit.h:55: multiple definition of
VCB'; fsshell.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/fsInit.h:55: first defined here
/usr/bin/ld: fsInit.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/fsInit.h:56: multiple definition of
fsmap'; fsshell.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/fsInit.h:56: first defined here
/usr/bin/ld: fsInit.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/mfs.h:40: multiple definition of `ro
tDir'; fsshell.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/mfs.h:40: first defined here
/usr/bin/ld: fsInit.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/mfs.h:41: multiple definition of `cw
ir'; fsshell.o:/home/student/Documents/csc415-filesystem-WilliamPanTW/mfs.h:41: first defined here
collect2: error: ld returned 1 exit status
make: *** [Makefile:61: fsshell] Error 1
```

Resolution 04: Create a global.h file that marks global variables as “extern”, for those files needing access to the global variable. Which means telling the compiler that these variables are declared in the global.c and should be linked appropriately during the linking phase(make run).



Issues 05 - After making a new directory with md commands the root blocks on hexdump do not show up the new directory entries info instead of a random 8 bytes number which match with the new directory entries. Thinking the directory entries would only contain the pointer instead of info

Resolution 05- After planning and discussion we found out that the info should update on the root directory entries instead of pointer, eventually adding the lbawrite and parse path info that appears on the root directory.

Issues 06 - While changing directory in root with cd dot dot or cd dot twice with statement of loadedCWD is not equal to loadedRoot, will cause a double free.

```
Prompt > cd ..
/
double free or corruption (!prev)
make: *** [Makefile:67: run] Aborted (core dumped)
student@student:~/Documents/csc415-filesystem-WilliamPanTW$
```

Resolution 06- The reason would likely be that loadedCWD is assigned to temp, which has the same memory location. By adding if loadedCWD equal ppinfo.parent before freeing it, ensuring that it's only freed when necessary.

3.3. Milestone 3

Issues 01 - The second time make directory(md) will not append correctly on hexdump as root dictionary, but it append on directory just created .*For example:*

prompt > md test

Prompt >md dir

```
004A40: 74 65 73 74 00 00 00 00  00 00 00 00 00 00 00 00 | test............
004A50: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A60: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A70: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A80: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004A90: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AA0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AB0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AC0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AD0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AE0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004AF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

004B00: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B10: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B20: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B30: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
004B40: 01 00 00 00 23 00 00 00  1D 00 00 00 33 00 00 00 | ....#.......3...
004B50: C3 6D 25 66 00 00 00 00  C3 6D 25 66 00 00 00 00 | m%f....m%f....
004B60: 64 69 72 00 00 00 00 00  00 00 00 00 00 00 00 00 | dir.............
```

The following hexdump should appear in blocks 7(roots) instead of blocks 36(test directory)

Resolution 01- Because we have the wrong logic while creating a directory, by modifying a statement to check if no parent will be root and initial the necessary dot and dot dot as itself. For the normal directory we need to append cd with the parent.

Issues 02 -LS command cannot display correctly with “ls” without specified anything but ls dot could display correctly .

```
Prompt > ls
check ppinfo last element index: -2
check ppinfo last element name: (null)

Prompt > ls .
check ppinfo last element index: 0
check ppinfo last element name: .

test
dir
nice
Prompt >
```

Resolution 02- When there is a path in a function that handles token null or slash without specified anything and nothing meaning root return -2 without index and name, thus the origin output of “ppinfo.lastElementIndex” will be wrong. So by adding a statement that checks for root and load just parent will make the ls command without specific anything work.

```
Prompt > ls -l

D        14688   test
Prompt >
```

Issues 03 - creating file using torch do not appear on hexdump

Resolution 03 - Add LBAwrite to make sure it write back to disk, because the LBAwrite on the fsInit.c will not write again after process the file system

4. Details of how each of your functions work

To better maintain and collaborate our code , we divide it into multiple files.

File	Description
fsInit.c	Main driver for file system assignment to initialize the system .
fsInit.h	Main driver header file for file system assignment.
bitmap.c	This is a free space system that contains functions * for bitmap which tracks,set,clear,allocation,release of blocks. *
bitmap.h	
global.c	This file will define the global variable
global.h	This header file share the global variable that provide , another file to access prevents multiple declarations throughout the project.
mfs.c	This is the file system interface. * This is the interface needed by the driver to interact with * your filesystem. *
mfs.h	This is the file system interface. * This is the interface needed by the driver to interact with * your filesystem.
mfs_helper.c	This is the file system interface. * This is the interface needed by the driver to interact with * your filesystem. *
mfs_helper.h	This is the file system interface. * This is the interface needed by the driver to interact with * your filesystem.
b_io.c	Basic File System - Key File I/O Operations
b_io.h	Interface of basic I/O Operations
b_io_helper.c	This header file provide necessary function for b_io.c
b_io_helper.h	This header file provide necessary function for b_io.c
fsshell.c	Main driver for file system assignment. (MOVE file)

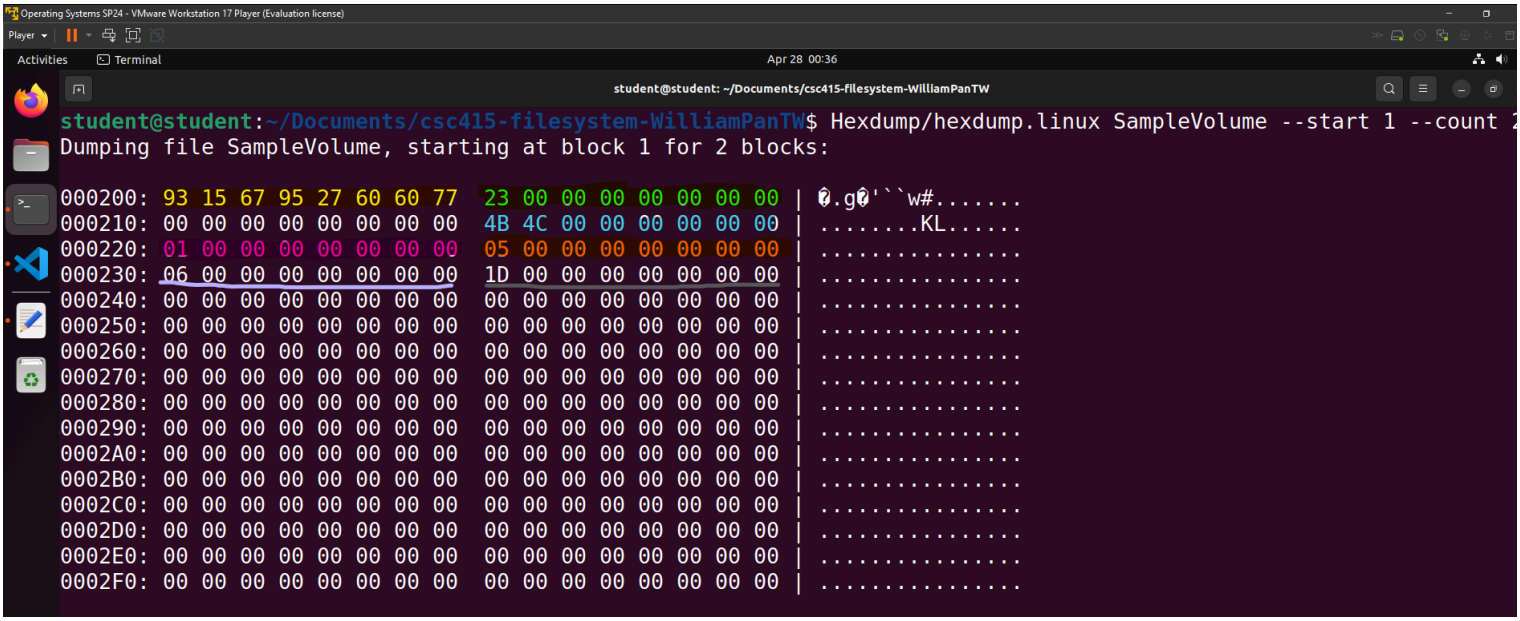
4.1. A description of the VCB structure

Our volume control block have contains information,that help us better control of the file system with following variable:

- A 8 bytes of signature to store the magic number with 2^128 range.

```
29      #define vcbSIG 0x7760602795671593 //magic number signature for VCB
```

- A 8 bytes free_block_index to track location of the free space in bitmap
- A 8 bytes block_index to location the VCB (block 0)
- A 8 bytes block_size to store capacity of the volume blocks (19531 blocks)
- A 8 bytes bit_map_index to store location of the bitmap (index 1)
- A 8 bytes bit_map_size to store the total number of free blocks(bitmap length)
- A 8 bytes root_dir_index to location the root directory in disk
- A 8 bytes root_dir_size to store the total number the root directory use in disk



Hexdump format is in little-endian format of less significance on right to left, and we write it in block 0 starting from 1 . With the order from top to bottom variable in struct VCB .

Bytes 00-07(yellow): signature Indicates 0x7760602795671593 in hexadecimal which is the same as the define vcbSIG from “fsinit.c.h”.

Bytes 08-15(green): Indicates 0x23 in hexadecimal which is equivalent to 35 in decimal that is what we initialize the **free_block_index** from vcb. To keep track of which freespace we locate. For the graph we initial Volume control block with 1 block ,freespace as 5 blocks and 29 blocks for the root directory.

Bytes 16-23(Black): block_index Indicates 0x0 meaning the VCB store in index 0 of the disk

Bytes 14-31(Blue): block_size Indicates 0x4C4B in hexadecimal which is equivalent to 19531 in decimal that the numberOfBlocks pass in initFileSystem, which also is what we initialize to volume control block structure .

Bytes 32-39(purple) : bit_map_index Indicate 0x01 in hexadecimal is equivalent to 1 in decimal, meaning location of the bitmap store after volume control block.

Bytes 40-47(Orange): bit_map_size Indicates 0x05 in hexadecimal is equivalent to 5 in decimal.Is what we initialize the, meaning we need 5 blocks of space to allocate free space using the bitmap method.

Bytes 48-55 (Purple line): ***Root_block_index*** Indicates 0x06 in hexadecimal or 6 in decimal, meaning the position of root directory stored in disk after 1 block of volume control block plus 5 blocks of bitmap.

Bytes 56-63 (gray line): ***Root_block_size*** Indicates 0x1D in hexadecimal which is equivalent to 29 in decimal that we initialize to volume control block. Meaning we allocated a Root directory with 29 blocks.

4.2. A description of the Free Space structure

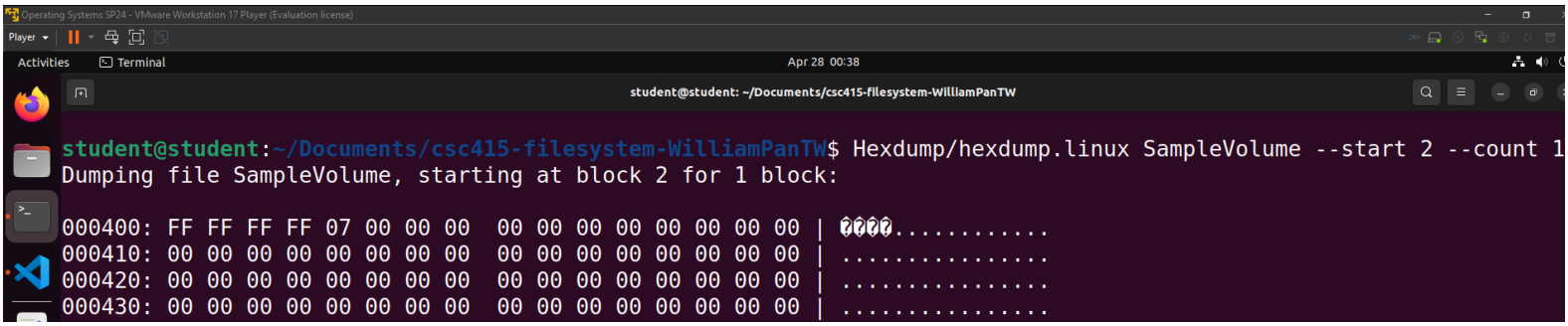
To keep track of the free space we are using the bitmap method, by using 1 bits per block to indicate if it is used as “1” and “0” as free.

First, we initialize the free space for the file system, however the smallest addressable size is byte so we divided the number of blocks from the file system by 8 to get the block we needed. Then, three helper functions(set,get and clear bit) are created to set the bits correctly and efficiently.

Second, after we initialize all the free space in bitmap as zero, we set up(malloc) the total blocks we need in free space, then set the according blocks as used.

Third, we updated the ***free_block_index*** in the volume control block to keep track of which bit is allocated and write it back to disk.

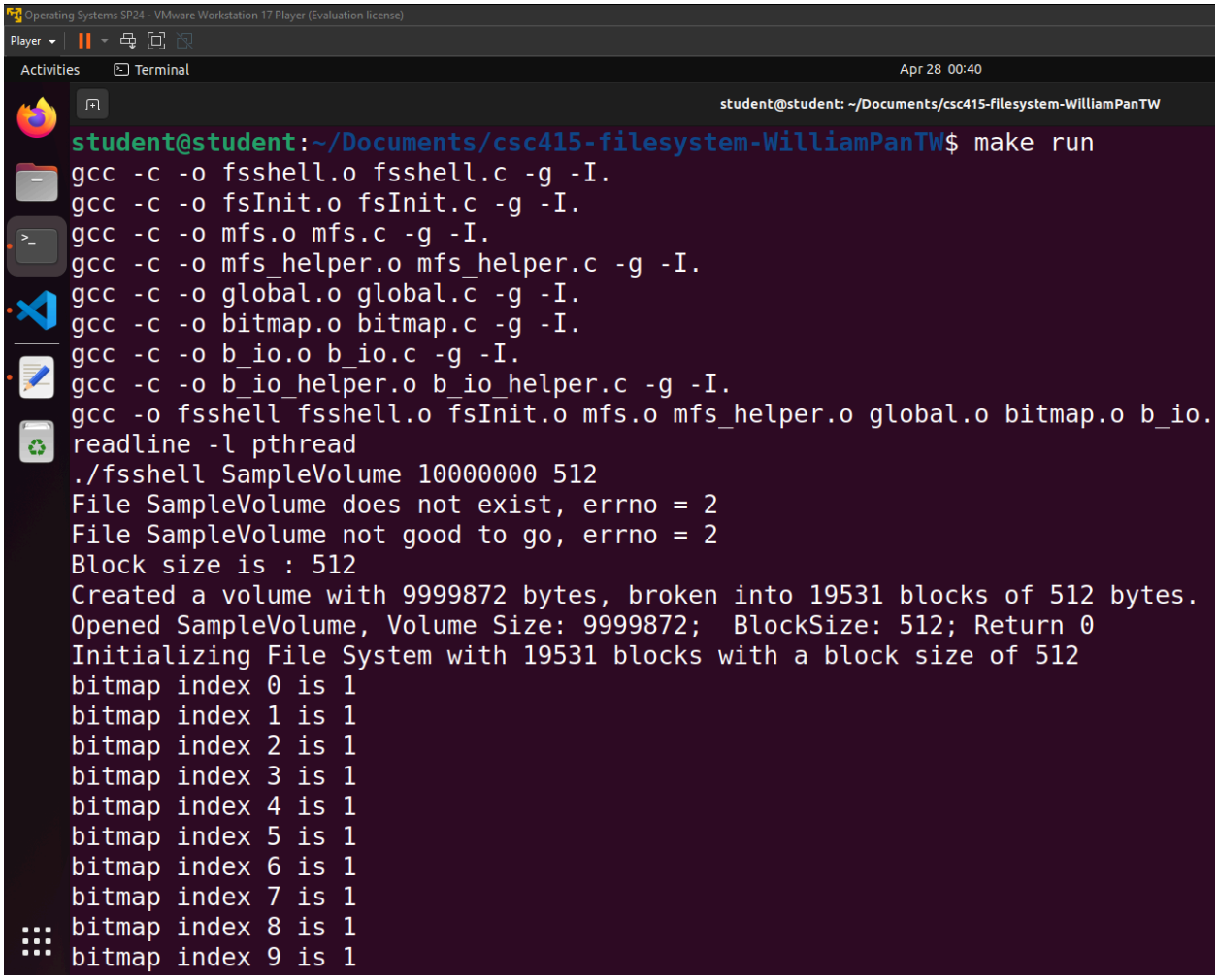
Lastly, we repeat step three where memory is allocated until there is no space left .

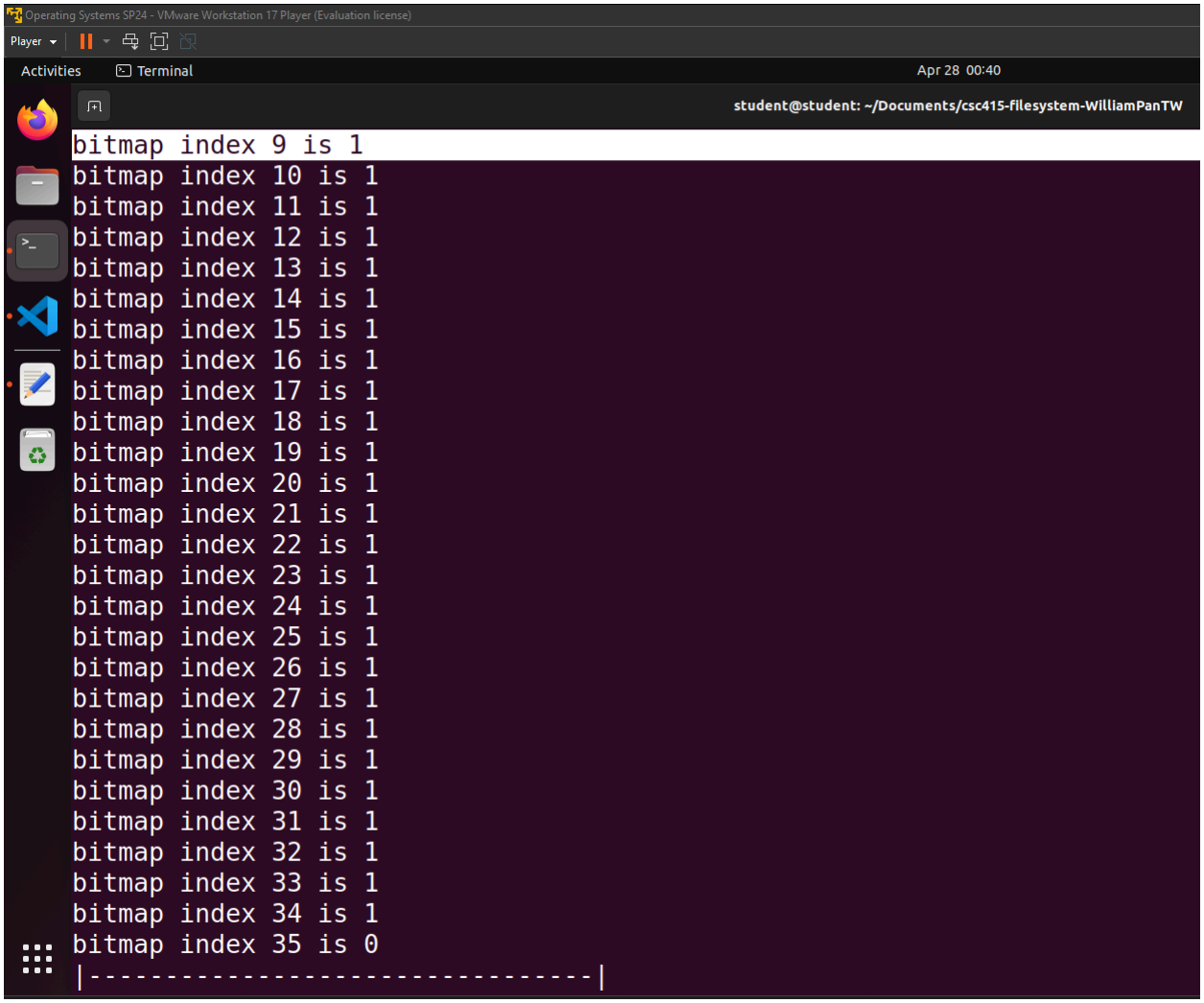


During milestone two we added an extent of **start** and **count** how many blocks were used to encapsulate the functionality for other freespace systems.

Step1. Start from the root directory after vcb and bitmap
Step2. Determine chunk size based on provided minimum block count
Step3. Loop until all blocks are allocated or there's not enough space
Step4. If no enough space found then quit
Step5. Iterate through bitmap to find free space
Step6. Mark allocated blocks as used in the free space map
Step7. Then store extent information for directory needed
Step8. Write updated freespace information back to disk
Step9. Write updated free block information for VCB back to disk

Here is a visual presentation of how bitmap indexes were marked as “1” occupied and 0 as free after the initialization of the file system. And because we only initial 1(VCB) + 5 (bitmap) + 29 (root) blocks , the index will mark as used(1) until index 34.





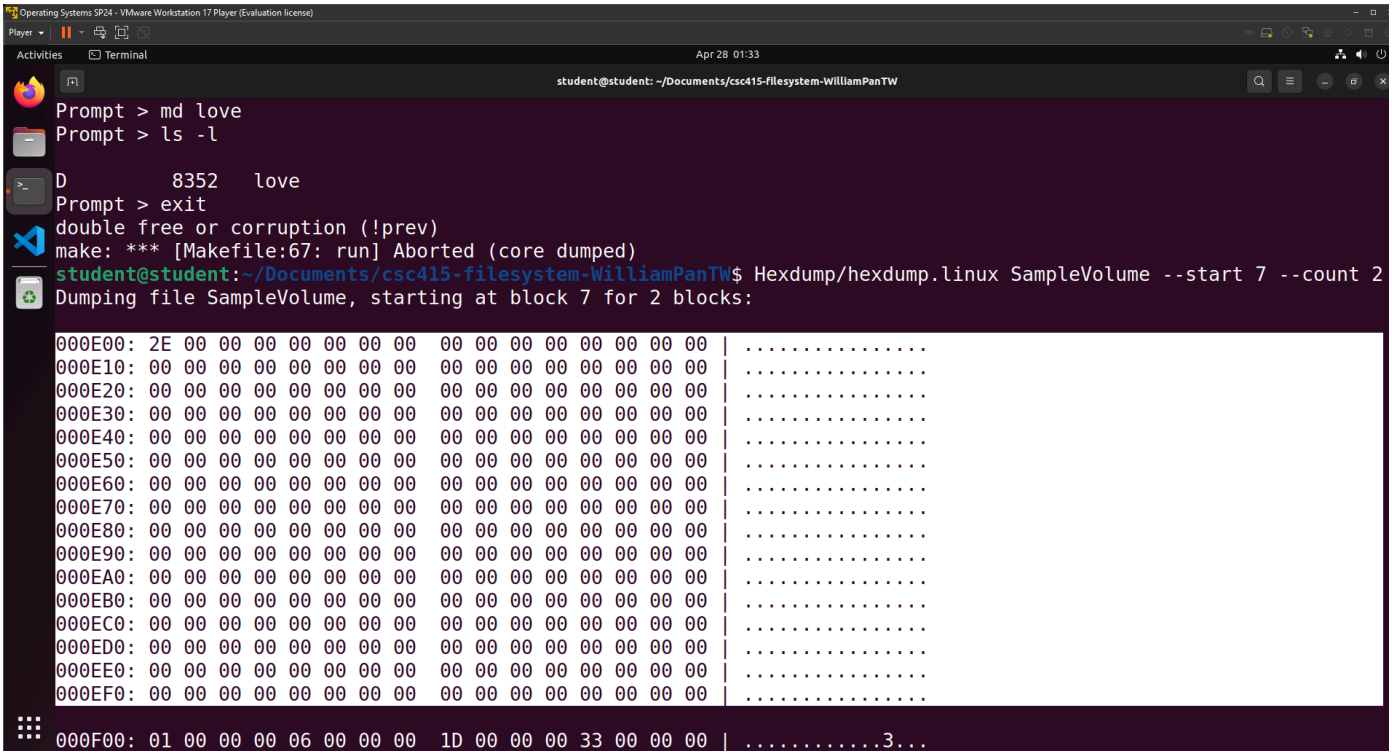
Note : this is just a visual of our bitmap and it will not showup on our filesystem

4.3. A description of the Directory system

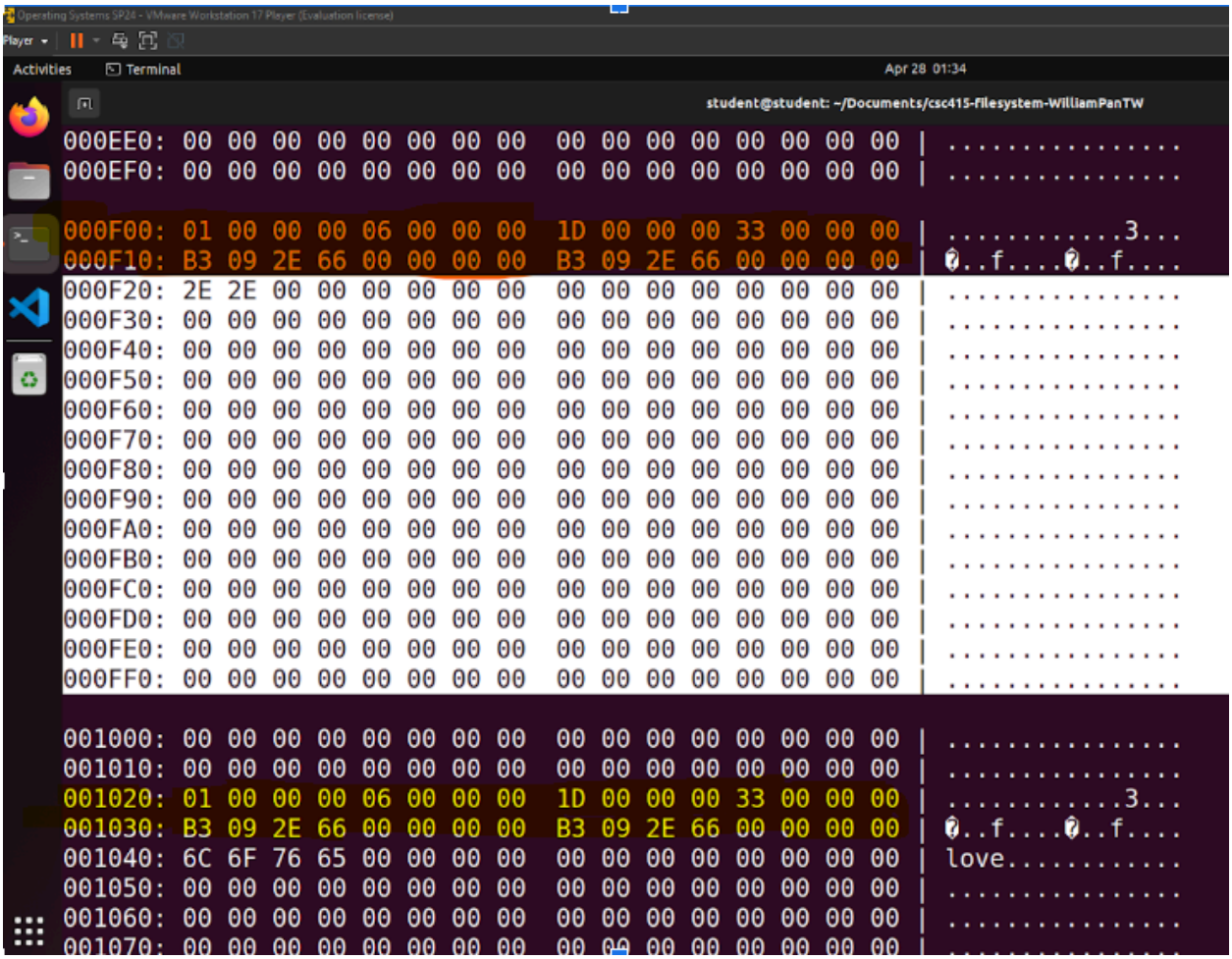
Our directory entries haves this component

- 4 bytes ***fileName*** array with the size of 255 bytes to store name or dot.
- 4 bytes ***isDirectory*** indicate directory as 1 or a file as 0
- 4 bytes ***dir_index*** store the location of directory/file in disk
- 4 bytes ***dir_size*** store the amount of block it use in disk
- 4 bytes ***entry_amount*** to store the amounts of directory it have
- integer or long type depend on implementation ***create Date***
- integer or long type depend on implementation ***modify date***

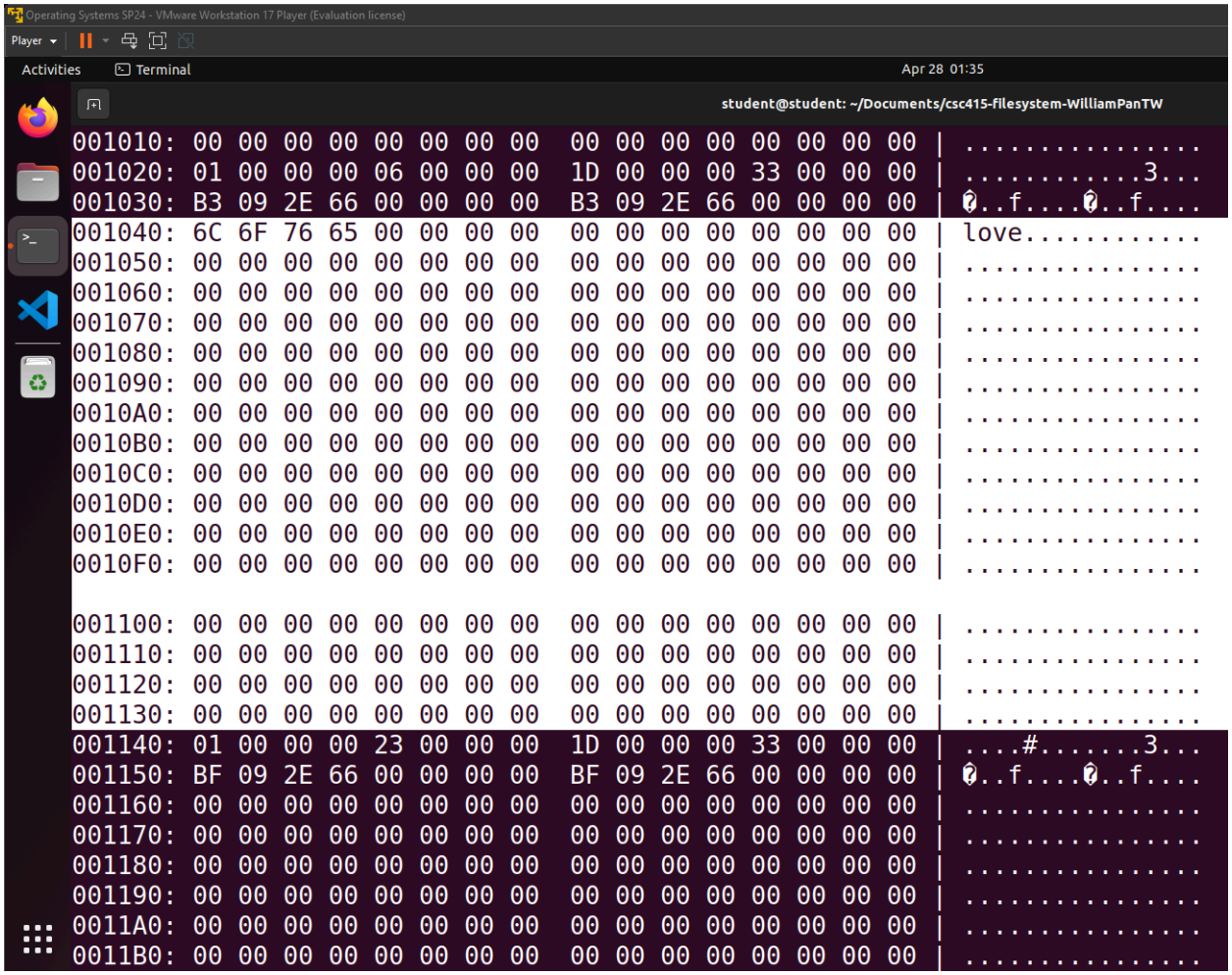
And you can imagine the directory is an array. For example, below is a root directory hexdump



As the hexdump shows, we write the root directory from block 7 using LBAwrite because vcb and bitmap take up the 6 blocks. The display order will be the same order as our directory structure. From **000E000** to **000EF0** is a block of 256 bytes of row(16) times columns(16). Meaning the first variable of directory entries array of ***fileName*** with the size of 256(16X16) bytes from the root directory. With it initialized with a dot for it itself, with the following **000F00** indicated 0x01 meaning it is **isDirectory** as root is a directory. Then 0x06 meaning **dir_index** of location where root is stored in disk. With 0x1D meaning ***dir_size*** the blocks of the root directory used. Follow with 29 ***entry_amount*** the root directory has as 0x33. Lastly, we have the same create date and modify date with 8 bytes of the second screen shot.



From 000F20 to 001010 is the second dictionary entry of the root directory,which store dot dot , which yellow parts are the same as dot (orange part) because it pointing to itself.



So, while we make a new directory called love in the root directory, it will be the same as the previous structure, and it will appear on the next array of root directory. With the evidence of hexdump of 001040 to 001130 are written as love for filename , then with the difference timestamp in 001150.

4.4. How **int loadFreeSpace(uint64_t numberOfBlocks);** work in fsInit.c

This function will be called if the volume control block signature is initial.By first , malloc the needed buffer for our free space system, with the corrected size from volume control block and read the location of the bitmap from disk(simpleVolume), to the buffer where we malloc.

4.5. How **int loadRootDirectory(uint64_t numberOfBlocks);** work in fsInit.c

This function will be called if the volume control block signature is initial.By first, malloc the needed buffer for our root directory, with the corrected size from volume control block and read the location of the root directory from disk(simpleVolume), to the buffer where we malloc.

4.6. How **int initVolumeControlBlock(uint64_t numberOfBlocks);** work in fsInit.c

If the volume control block is not initialized then we create one with this function. By initializing with signature , location and amount of the volume control block.

4.7. How **void set_Dir(char *name, int index, int dirEntryAmount, struct dirEntry *dirEntries, struct extent *location);** work in fsInit.c

Function set up directory entries, with first index file name to dot, set the attribute to is directory(1) , give the correct size from struct extent start and count. Then by getting the current time we could set the modification and create a date.

4.8. How **struct extent* allocateSpace(uint64_t block_amount, uint64_t min_block_count);** work in bitmap.c

Step1. Start from the root directory after vcb and bitmap

Step2. Determine chunk size based on provided minimum block count

Step3. Loop until all blocks are allocated or there's not enough space

Step4. If no enough space found then quit

Step5. Iterate through bitmap to find free space

Step6. Mark allocated blocks as used in the free space map

Step7. Then store extent information for directory needed

Step8. Write updated freespace information back to disk

Step9. Write updated free block information for VCB back to disk

4.9. How **int createDirectory(uint64_t entries_number, struct pp_return_struct* ppinfo); work** in fsInit.c

Step1. Get directory entry size in bytes.
Step2. Determine bytes needed for root directory (entry size times amounts)
Step3. Calculate number of blocks needed.(divided 512 with floor operator)
Step4. Calculate allocate block size.(blocks times number)
Step5. Get the actual allocatable entries amount
Step6. Calculate actual number of directory entries in bytes.
Step7. Allocate free space for directory with allocateSpace function
Step8. Allocate a pointer in heap to keep track directory entries
Step9. Initialized directory entries with null terminated from index 2
Step10. Set root directory index 0 as dot
Step11. create a dirEntry buffer of parent point to directory entries
Step12. Determine the buffer of its parent for dot dot.
Step13. If parent is provided set directory as correspond parent
Step14. Write directory entries to disk.
Step15. Set current working directory if root.
Step16. Update VCB with root directory info.
Step17. Free allocated space.
Step18. Return start index of allocated space.

4.10. How **initFreeSpace(numberOfBlocks) work** in bitmap.c

If the volume control block is not initialized meaning we do not load the free space system then we create one bitmap system with this function.

Step1. We used 1 bit to mark if the block is used or not but the smallest addressable is byte, so we divide the parameter numberofblock with 8.
Step2. Then use the floor operation of calculate the needed block for bitmap
Step3. Allocated a buffer from heap with the needed blocks times blocks size
Step4. Return error if fail to create the buffer
Step5. Initial all free space as zero using memset indicated as free
Step6. Check the free location and mark 1 bit used for volume control block
Step7. Iterate and set bits for bitmap as used.
Step8. Initial bit map size and index back to volume control block
Step9. Write the information of free space back to disk

4.11. How **void set_bit(char* bitmap, int position);** work in bitmap.c

Set the bit for block in bitmap

Step1. By divided 8 bits we know what bytes this blocks it is alloacted
Step2. Use module to get remaining tell us the bit of this bytes
Step3. Set origin bits with a left shift or operation of position according step 1 and 2 , without changing others values of bits

4.12. How **int get_bit(char* bitmap, int position);** work in bitmap.c

Step1. By divided 8 bits we know what bytes this blocks it is alloacted
Step2. Use module to get remaining tell us the bit of this bytes
Step3. Move bit according step 1,2 to the least significant position.
Step4. Isolate the least significant bit and other using AND operation
Step5. Return the least significant of used(1) or unused(0)

4.13. How **void clear_bit(char* bitmap, int position);** work in bitmap.c

Step1. By divided 8 bits we know what bytes this blocks it is alloacted
Step2. Use module to get remaining tell us the bit of this bytes
Step3. Change the specific index from remaining of bits by left shift not operator, without changing others values of bits

4.14. How **void releaseBlock(uint64_t startBlock, uint64_t block_amount);** work in bitmap.c

Free the specific range of bits(blocks) in free space system (bitmap)

Step1. Check if startBlock is within the amount of file system
Step2. If it out of bounds, return without performing any action
Step3. Loop bitmap buffer with clear bit function of the range number
Step4. Update/write bitmap back to disk(sample volume)

4.15. How **int fs_mkdir(const char *pathname, mode_t mode);** work in mfs.c

1. check if the path is valid
2. compare the new directory is exist or not
3. find free directory entries
4. update info index where free space locate
5. Call create directory with the parse path info

4.16. How **int fs_rmdir(const char *pathname);** work in mfs.c

1. check if the path is valid
2. Check if the parent directory doesn't exists
3. prevent user to remove dot and dot dot
4. Check if it a directory
5. Load directory to check if directory empty
6. Free blocks associated with directory
7. Set directory as unused
8. Update the modify time of the deleted parent
9. Write directory entities changes back disk

4.17. How **fdDir * fs_opendir(const char *pathname);** work in mfs.c

1. check path name is valid or not
2. If the pare path index equal to specified root from parepath function and it last elements name is set to NULL, load the root directory itself
3. Else we load the specified parent index
4. allocate memory for directory item info to provide caller with information
5. allocate memory for fdDir structure to keep track current process
6. initial struct file descriptor information
7. returns a pointer to the directory stream.

4.18. How **char * fs_getcwd(char *pathbuffer, size_t size);** work in mfs.c

1. Copy the current working directory string back to the buffer with the limit of size

4.19. How **int fs_setcwd(char *pathname);** work in mfs.c

1. take the path specified to temp buffer
2. call parsepath to check if it valid
3. Check if the parent directory doesn't exists
4. Look at parent[index], is it a directory
5. Load Directory
6. updating the string for the use
7. If start with absolute path("/"), copy the path
8. Else the path is relative
9. Track position of token
10. If token is ".", ignore it
11. build the new path based on processed
12. Ensure the current working directory ends with "/
13. If last character is not "/", append "/"
14. Check current working directory is neither root directory nor parent directory
15. Load the current working directory

4.20. How **int fs_isFile(char * filename);** work in mfs.c

1. Return -1 indicated error , if the path is invalid
2. Return -1 indicated error , if the parent directory doesn't exists
3. Return 1 indicated it is file, if it is not a directory
4. Otherwise return 0

4.21. How **int fs_isDir(char * pathname);** work in mfs.c

1. Return -1 indicated error , if the path is invalid
2. if it root then load the the root directory as parent
3. Else Load directory entry for specified parent index
4. Return 1 if it is a directory
5. Otherwise return 0

4.22. How **int fs_delete(char* filename);** work in mfs.c

1. Return error , if the path is invalid
2. Return error if the parent directory does exist
3. prevent user to remove dot and dot dot
4. Free extents associated with file
5. Set directory as unused (NULL terminated)
6. Updated the parent modify date
7. Write directory entities changes back to disk

4.23. How work **struct fs_diriteminfo *fs_readdir(fdDir *dirp);** work in mfs_helper.c for display file function in ls command

1. Check if directory stream handle is valid
2. Iterate until reaching the end or finding a valid entry
    a. Get the directory entry at the next position
    b. move to the next directory entry position
    c. Skip empty directory entries if it null terminated
    d. Copy directory entry information to fs_diriteminfo structure
    e. set the file type
    f. Return next directory entry information
3. Return Null meaning End of file or error

4.24. How work **int fs_stat(const char *path, struct fs_stat *buf);** work in mfs_helper.c for display file function in ls command

1. Get the directory entry corresponding to the last element in the path
2. Set file size base of directory size or directory with number of entries
3. Initial the size of buffer
4. return success if retrieve file information

4.25. How work **int fs_closedir(fdDir *dirp);** work in mfs_helper.c

1. Free associated directory entries stream

4.26. How work **int isDirEntryEmpty(struct dirEntry* dir);** work in mfs_helper.c for Rm command

1. check if directory entry is empty

4.27. How work **void deallocateSpace(uint64_t startBlock, uint64_t numberOfBlocks);** work in mfs_helper.c for Rm command

1. Check if the range to clear is valid
2. Clear specified range of bit by iterated amount of size suing clear_bit function
3. Write the updated bitmap system back to disk

4.28. How work **int freeExtents(struct dirEntry* entry);** work in mfs_helper.c for Rm command

1. Free associated extents with file

4.29. How **int findUnusedDE(struct dirEntry* entry);** work in mfs_helper.c

1. iterate through all directory entires
2. if found null ternimate then return index

4.30. How **int findDirEntry(struct dirEntry* entry, char* name);** work in mfs_helper.c

1. find directory by it name
2. iterate all directory entries if any name match return index
3. Otherwise return error if no directory found

4.31. How **int parsePath(char* path, struct pp_return_struct* ppinfo);** work in mfs_helper.c

1. Return error, if path and parse path info are not NULL.
2. Copy path to a temporary buffer.
3. If it start with ‘/ meaning it is absolute then load root to temporary buffer
4. Otherwise, it is relative then load the current working directory to buffer
5. Tokenize path using "/" delimiter.
6. Handle case where tokenOne is NULL or no path specified
7. Iterate through tokens in the path to find it in directory
8. Tokenize the next element of the path.
9. Find directory entry for tokenOne.
10. Set parse path info values for parent, lastElementName, and lastElementIndex.
11. Return success if it reaches the end of file.
12. Check if a directory entry exists for tokenOne.
13. Check if the directory entry is actually a directory.
14. Load directory entry using parent index for tokenOne.
15. Set parent to the loaded directory
16. Update tokenOne for next iteration.
17. Return failure if path is invalid.
18. Return -2 if no error, but path does not exist.

4.32. How **int isDirectory(struct dirEntry* entry);** work in mfs_helper.c

1. Using AND operator if it is directory will return 1 as directory, elsewise.

4.33. How **struct dirEntry* loadDir(struct dirEntry* entry);** work in mfs_helper.c

1. Loads a directory entry from disk into memory

4.34. How **void freeLastElementName();** work in mfs_helper.c

1. Free parse path info

4.35. How **void freePathParent();** work in mfs_helper.c

1. Free parse path parent info

4.36. How **void freeppinfo();** work in mfs_helper.c

1. Free last element and free path parent by calling those function

4.37. How **b_io_fd b_open (char * filename, int flags);** work in b_io.c

Get everything

1. Initialize our system if we have not start our file system
2. Return error if the filename is invalid using parse path function
3. Retrieved parent directory information
4. Truncate the file and free block associate to zero if TRUNC flag is set
5. Create a new file if create flag is set
    a. Only create file if file does exist
    b. Find unused directory in the parent directory
    c. Return error, if no space left
    d. Assign new index
    e. Return error , if cannot create fail
6. Return error, If the parent information name is still zero
7. Retrieve file parent information
8. set B_open first get file info in case file info fails and forget to free getFCB.
9. Because file descriptorGet has a limit, in case we forget to free it and get file info faults, we get our own file descriptor after the above step success .
10. Return error if no FCB available
11. Allocate memory for the file buffer
12. Initialize FCB with file information
13. Set file position to end if O_APPEND flag is set
    a. iterate to find an available block
    b. Check if the current block is empty
    c. Set the current block to the empty block index
    d. Set the num of blocks of the FCB to the end of the file
14. Return file descriptor

4.38. How **int b_read (b_io_fd fd, char * buffer, int count);** work in b_io.c
Filling the caller's request is broken into three parts

1. Return error if the file descriptor is invalid
2. Return error if the file mode if write only
3. Return error if it is directory
4. Calculate number of bytes available to copy from buffer
5. Calculate amount have given to user
6. Limit count to file length to handle End of file by checking if amount they ask(count) plus already given exceed the file size
    a. reduce count size and we could not go beyond file size
    b. is the first copy of data which will be from the current buffer
    c. IF we have enough buffer just fill in part 1
    d. Else , we just fill in part 1 with the remaining \
        i. Set part 3 to amount to fill minus part 1 taken
        ii. Calculate how many 512 bytes chunks it needed
        iii. Reduce part 3 by the number of bytes that can be copied in chunk by equal to part3 % B_CHUNK_SIZE
7. Move the buffer plus index by adjust it started if part 1 is greater than zero
8. If part 2 greater than zero meaning copy direct to callers buffer
    a. Write back to disk
    b. Update current blocks index
9. If part 3 greater than zero meaning we need to refill out buffer to copy more bytes
    a. try to read B_CHUNK_SIZE bytes into our buffer
    b. reset the offset and buffer length
    c. If the bytes read is greater than part3 assign part3 with bytes read
    d. If part 3 is still greater than zero
        i. Copy part 1 2 and buffer to disk
        ii. Update the file control block index
10. Return part 1 , 2 and 3 of bytes

4.39. How **int b_write (b_io_fd fd, char * buffer, int count);** work in b_io.c

1. Initialize our system if we have not start our file system
2. Return error if the file descriptor is invalid
3. Declared three variable part 1, 2, 3
4. Retrieve file control block parent information
5. Update file's modification date
6. Return error if the file mod is read only
7. Loop until all data is writte
    a. Calculate parts of the buffer to handle
    b. If at start of buffer, create a new extent
    c. Adjust parts if necessary
8. Write remaining bytes from buffer
9. Refill buffer if necessary
    a. If unable to allocate space, return error
    b. Update file properties for new extent
    c. Copy remaining data to buffer
    d. Update buffer index and write to disk
    e. Update count for remaining bytes
10. Calculate total bytes written
11. Update and write back parent directory
12. Write the parent buffer back to disk
13. Return the total number of bytes actually written

4.40. How **int b_seek (b_io_fd fd, off_t offset, int whence);** work in b_io.c

1. Initialize our system if we have not start our file system
2. Get file entry from file control block array
3. Determine new position based on 'whence'
    a. If we finding the start position of file , Set new position to given offset from start of file
    b. If we finding current position of the file , Set new position to current position plus given offset
    c. If it finding the end of the file , Set new position to given offset from end of file
    d. Otherwise no case return error
4. Return error if new position is invalid
5. Update current file position
6. Update file size if necessary
7. Calculate current block location
8. Return the new current position

4.41. How **int b_close (b_io_fd fd);** work in b_io.c

1. Close the associate directory stream

4.42. How **int createFile(struct pp_return_struct* info);** work in b_io_helper.c for b_open

1. Get the parent directory entry where the new file will be created
2. Copy the name of the new file to the directory entry
3. Initialize size and type of the new file
4. Get current time and update according
5. Update parent directory's modification date
6. Write changes back disk

4.43. How **int moveFile(const char *src, const char *dest);** work in b_io_helper.c for move file command

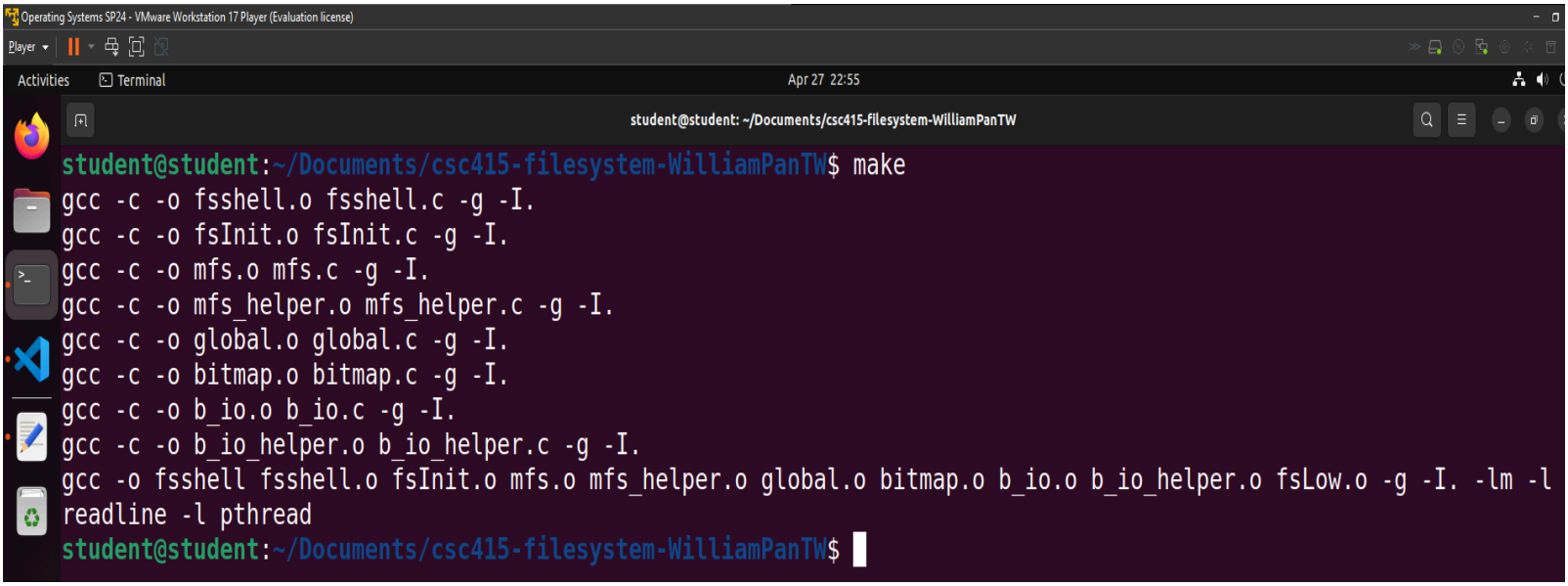
1. Check if the destination path is valid
2. If the parse path index equal to specified root from parepath function and it last elements name is set to NULL, load the root directory itself
3. Else we load the specified parent index
4. copy the file to the destination directory
5. Update modification time of the parent directory
6. write it back to disk
7. Return success

4.44. How **int copyFile(struct pp_return_struct* info, struct dirEntry* src);** work in b_io_helper.c

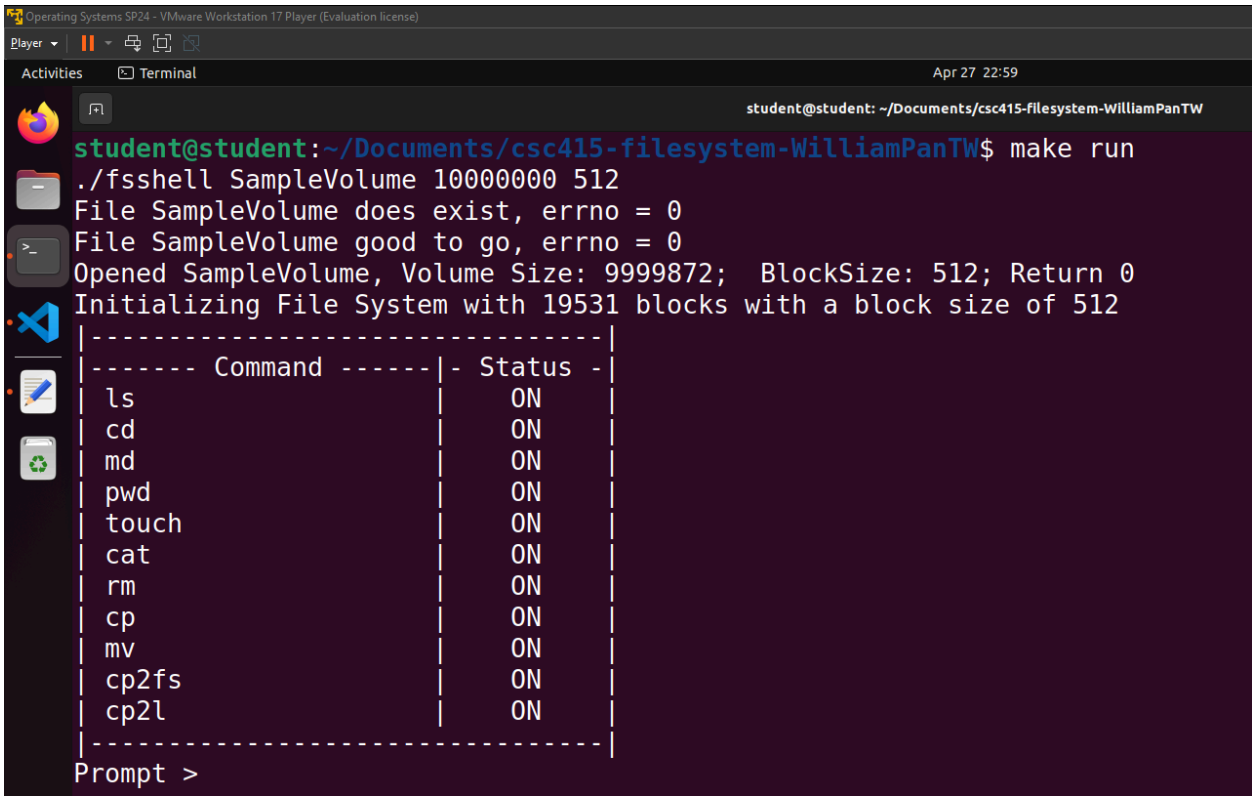
1. Get the parent directory entry where the file will be copied
2. Copy the file name from source to destination directory entry
3. Copy other attributes from source to destination directory entry
4. Get current time and Update parent directory's modification date
5. Write the parent directory entry to disk

5. Screenshots showing each of the shell commands listed above

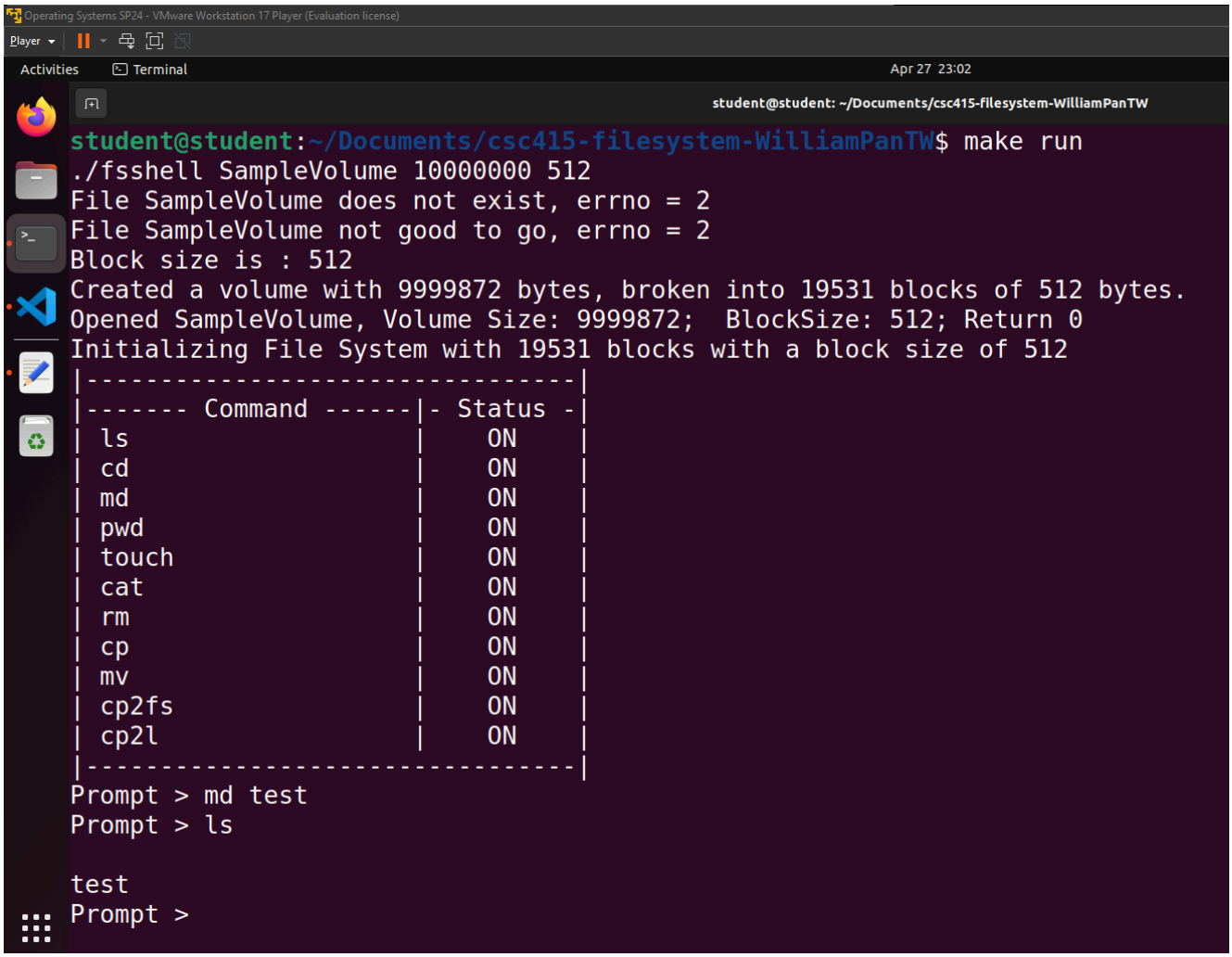
5.1. Screenshot of compilation:



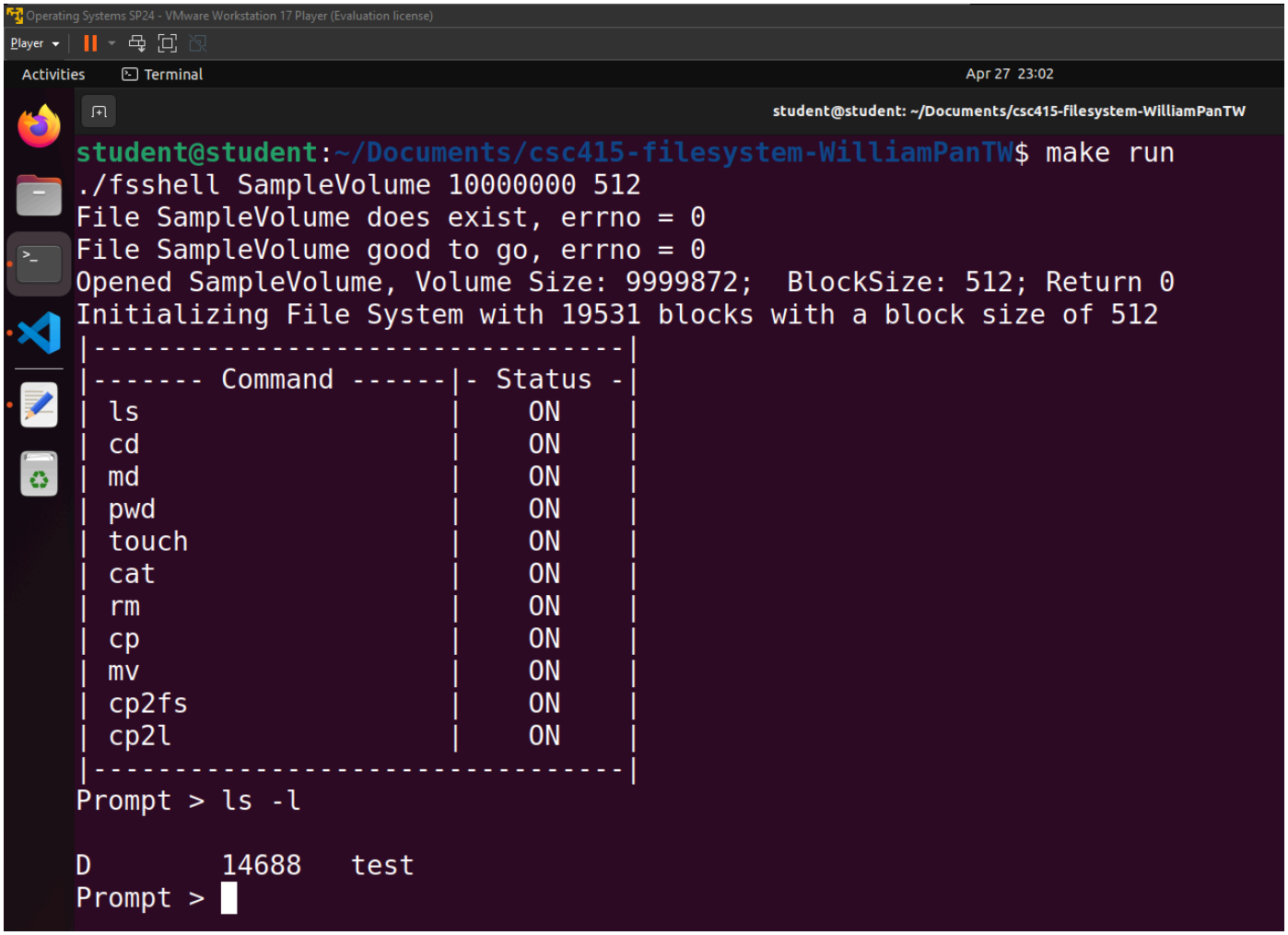
5.2. Screen shot(s) of the execution of the program:



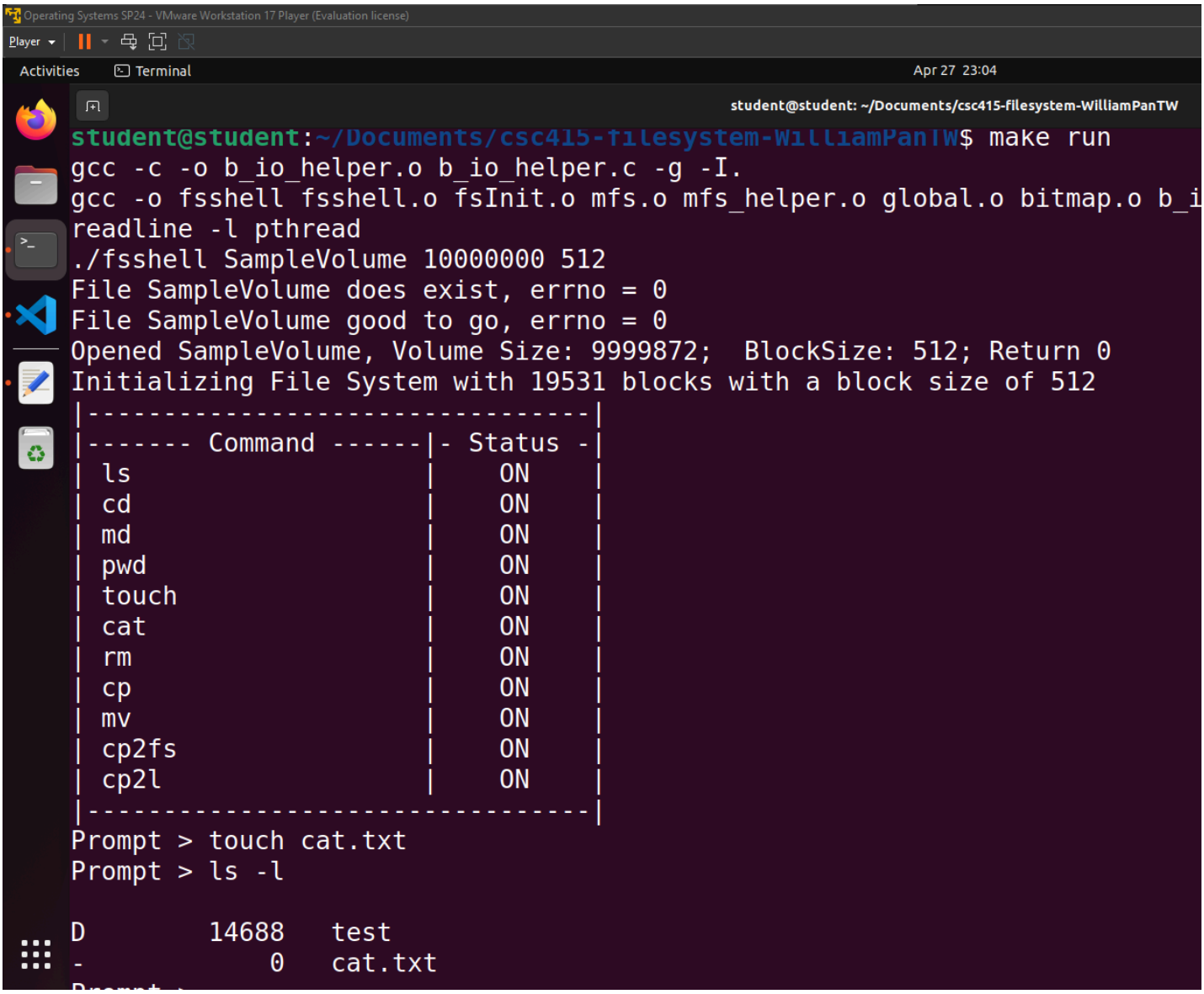
5.3. md - Make a new directory



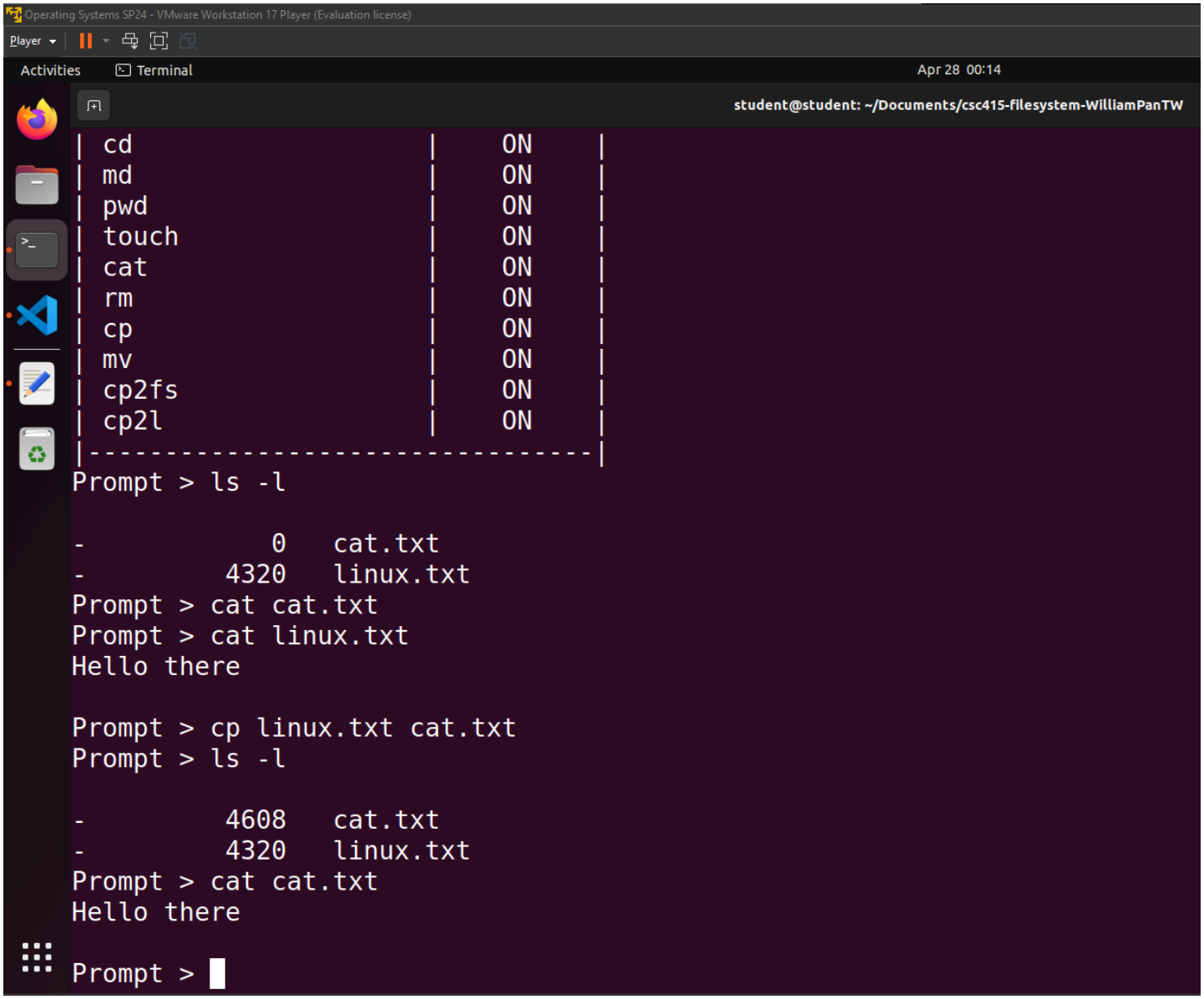
5.4. ls - Lists the file in a directory



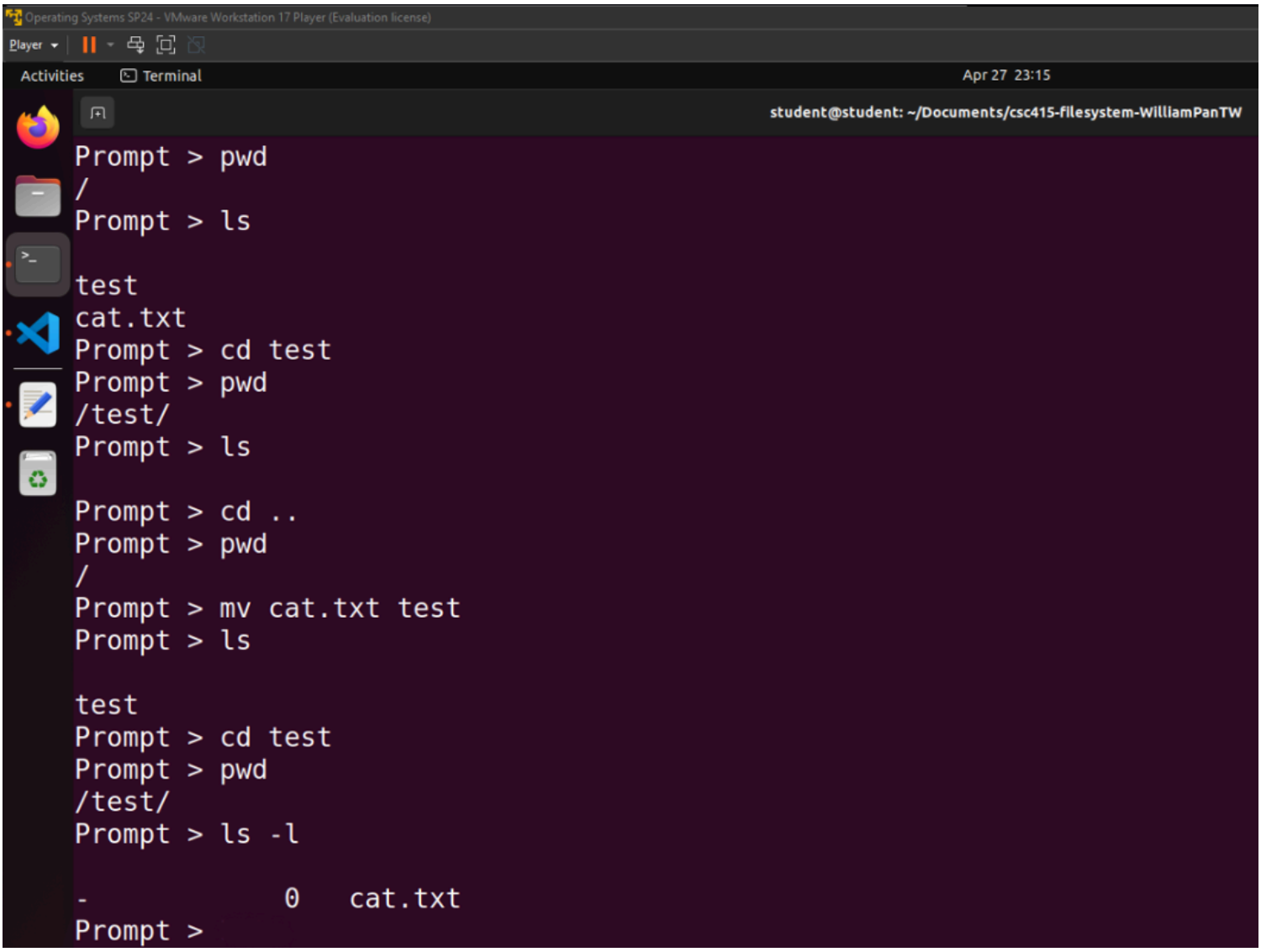
5.5. touch - creates a file



5.6. cp - Copies a file - source [dest]

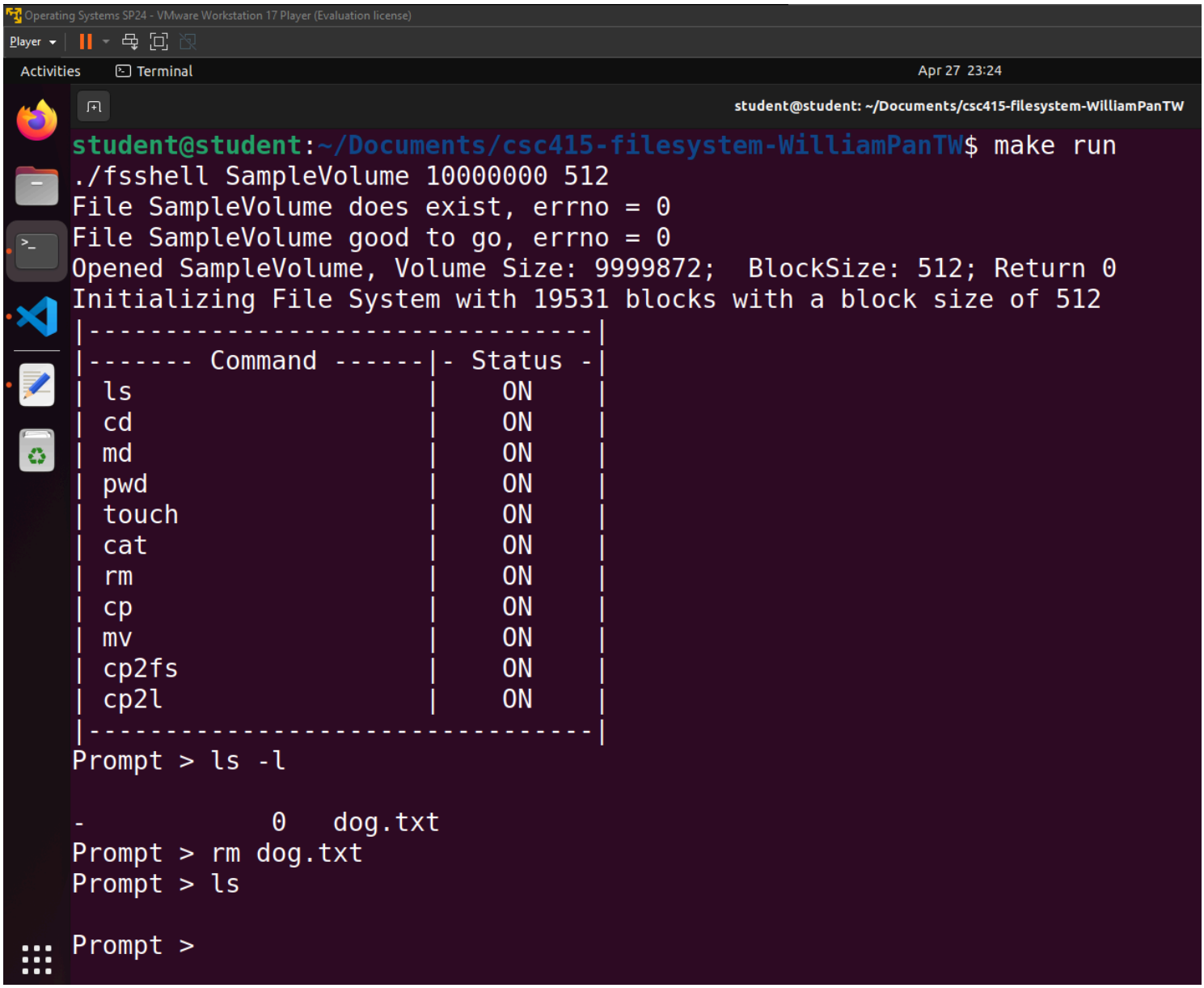


5.7. mv - Moves a file - source dest

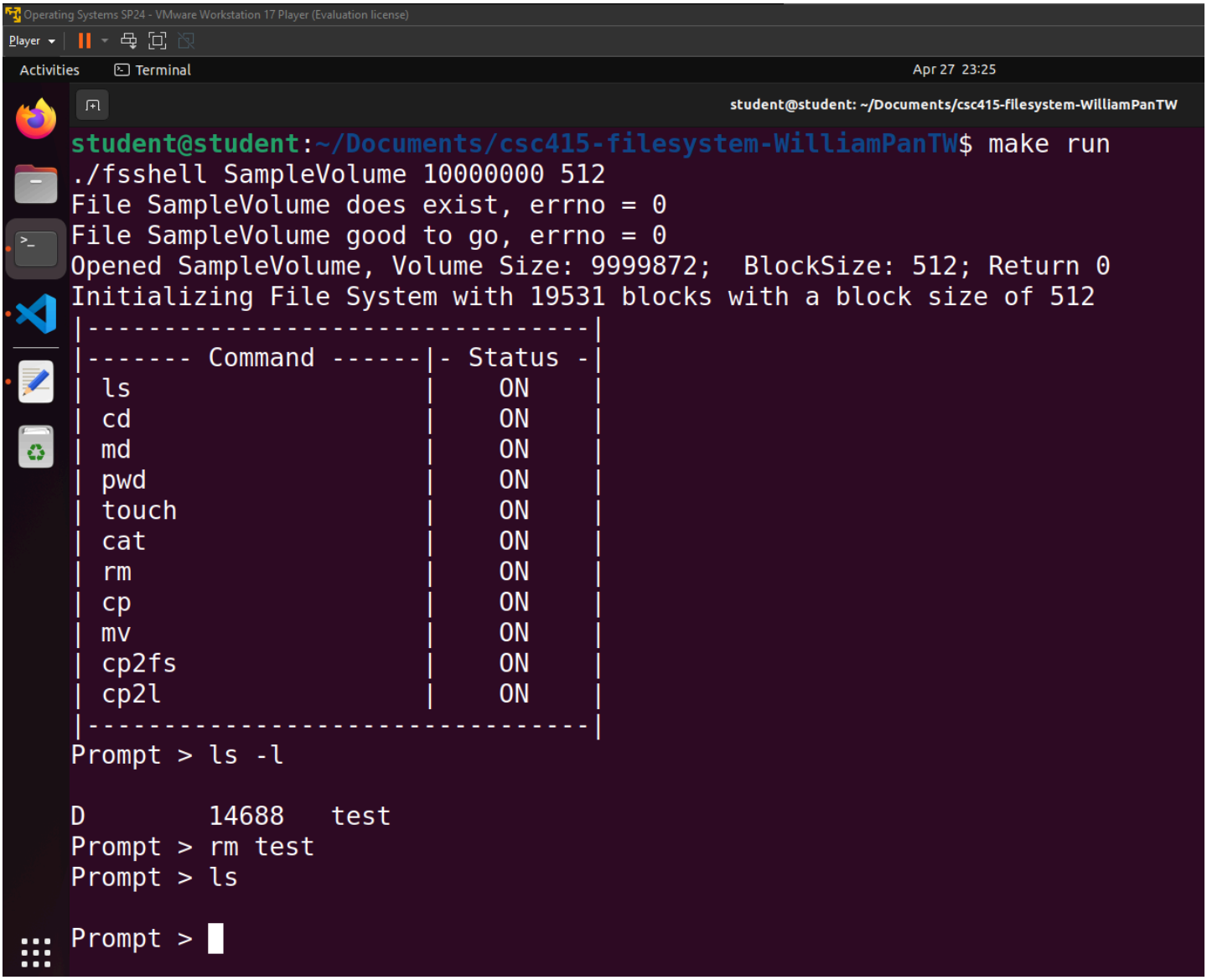


5.8. rm command

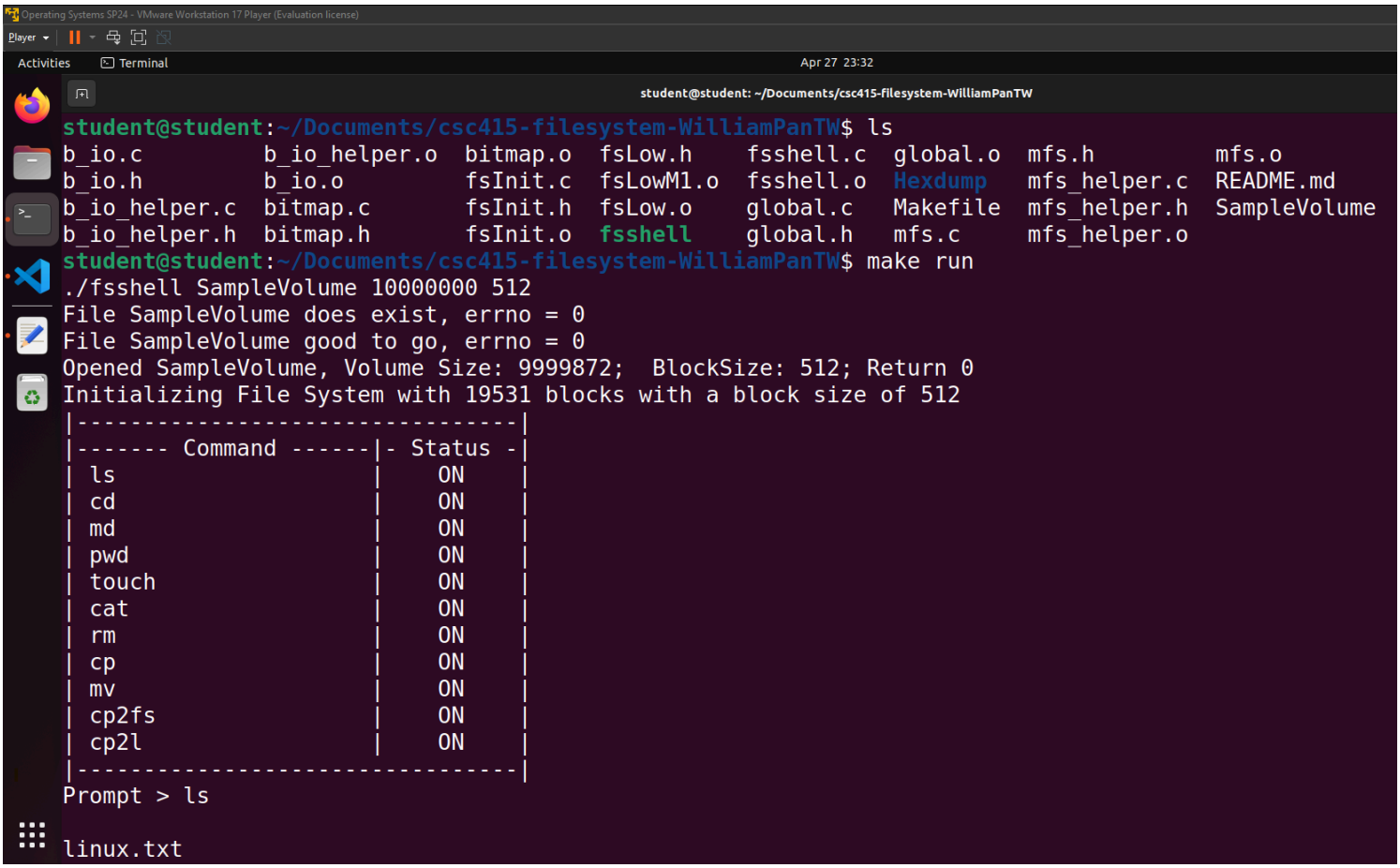
5.8.1. Removes a file

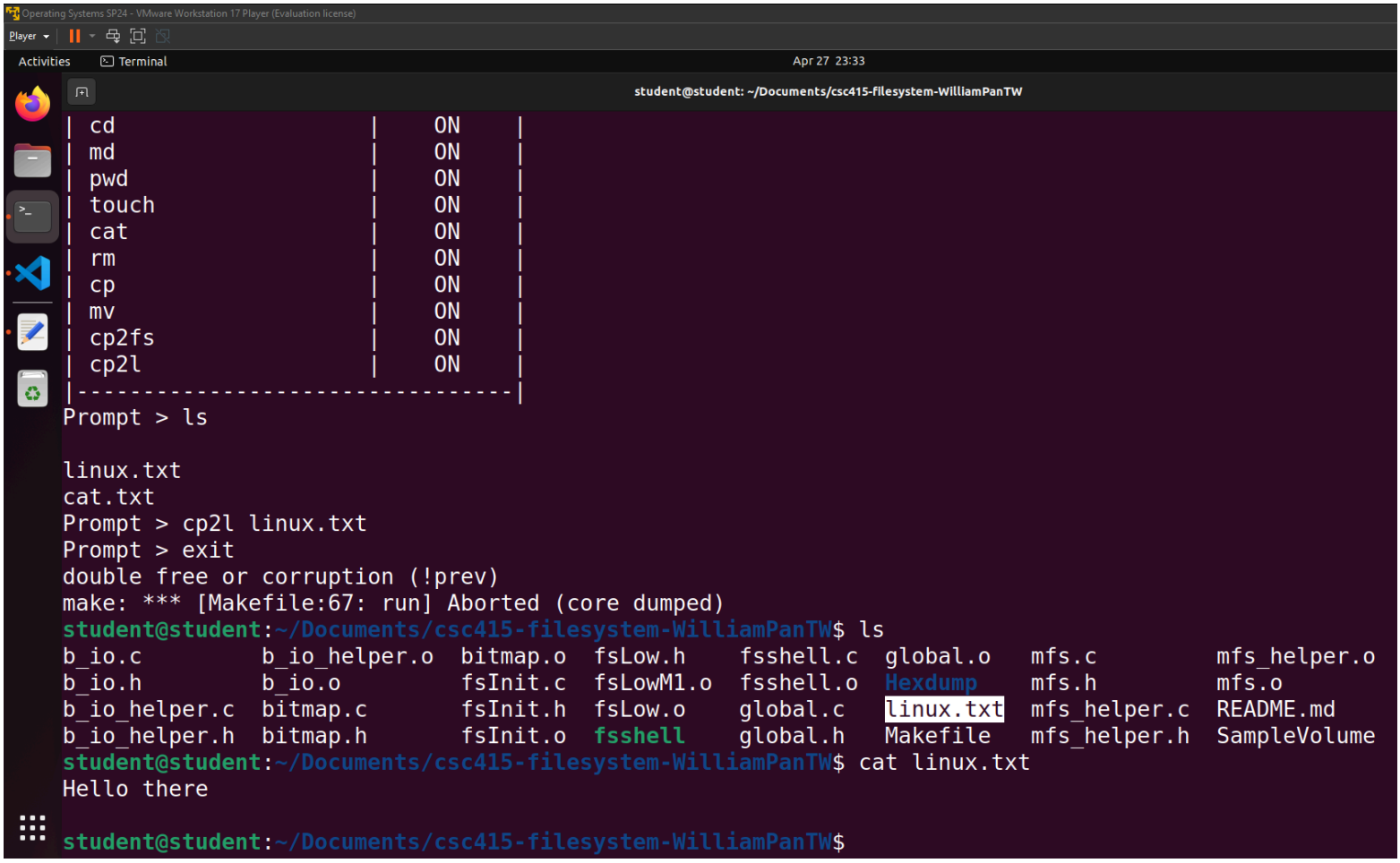


5.8.2. directory

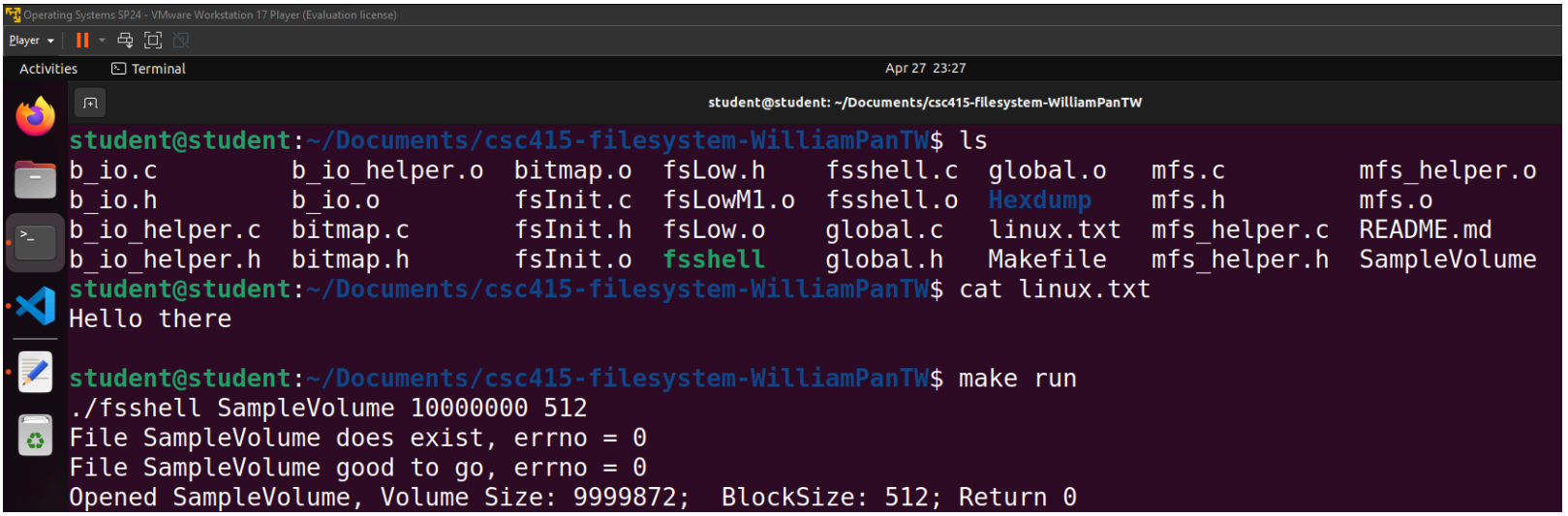


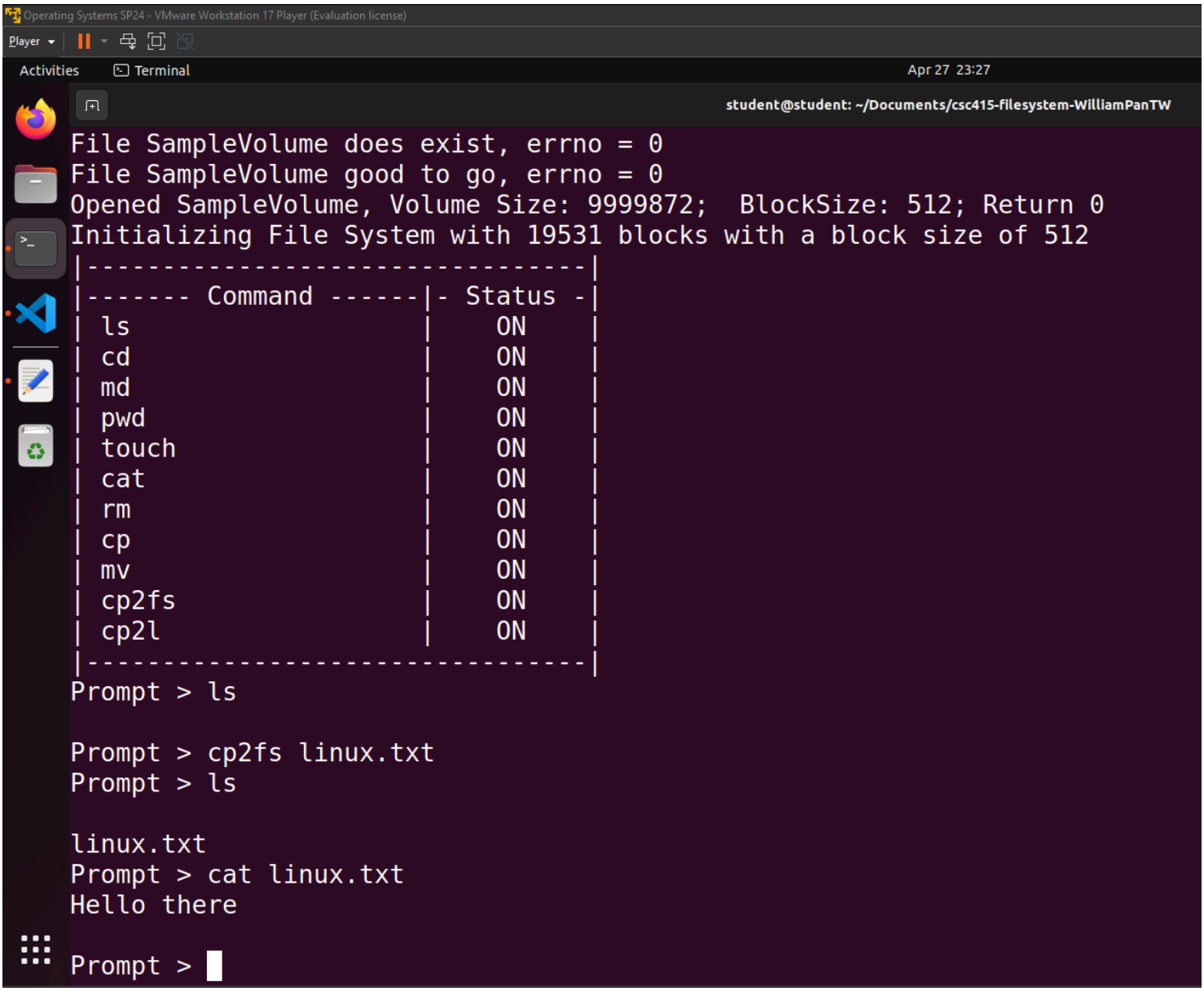
5.9. cp2l - Copies a file from the test file system to the linux file system



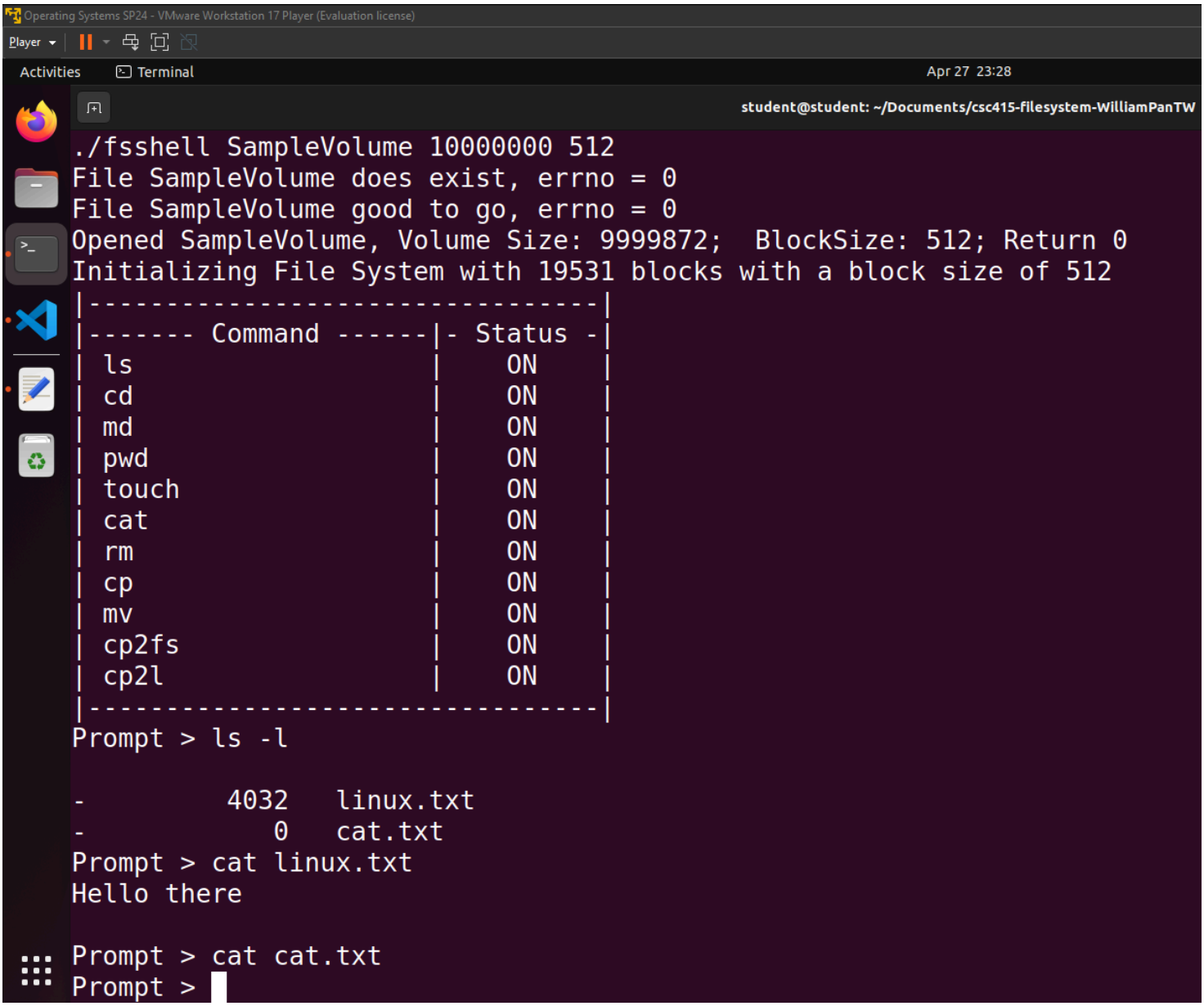

Operating Systems SP24 - VMware Workstation 17 Player (Evaluation license)
Player
Activities
Terminal
Apr 27 23:33
student@student: ~/Documents/csc415-filesystem-WilliamPanTW
cd	ON
md	ON
pwd	ON
touch	ON
cat	ON
rm	ON
cp	ON
mv	ON
cp2fs	ON
cp2l	ON
Prompt > ls
linux.txt
cat.txt
Prompt > cp2l linux.txt
Prompt > exit
double free or corruption (!prev)
make: *** [Makefile:67: run] Aborted (core dumped)
student@student:~/Documents/csc415-filesystem-WilliamPanTW$ ls
b_io.c b_io_helper.o bitmap.o fsLow.h fsshell.c global.o mfs.c mfs_helper.o
b_io.h b_io.o fsInit.c fsLowM1.o fsshell.o Hexdump mfs.h mfs.o
b_io_helper.c bitmap.c fsInit.h fsLow.o global.c linux.txt mfs_helper.c README.md
b_io_helper.h bitmap.h fsInit.o fsshell global.h Makefile mfs_helper.h SampleVolume
student@student:~/Documents/csc415-filesystem-WilliamPanTW$ cat linux.txt
Hello there
student@student:~/Documents/csc415-filesystem-WilliamPanTW$

5.10. cp2fs - Copies a file from the Linux file system to the test file system

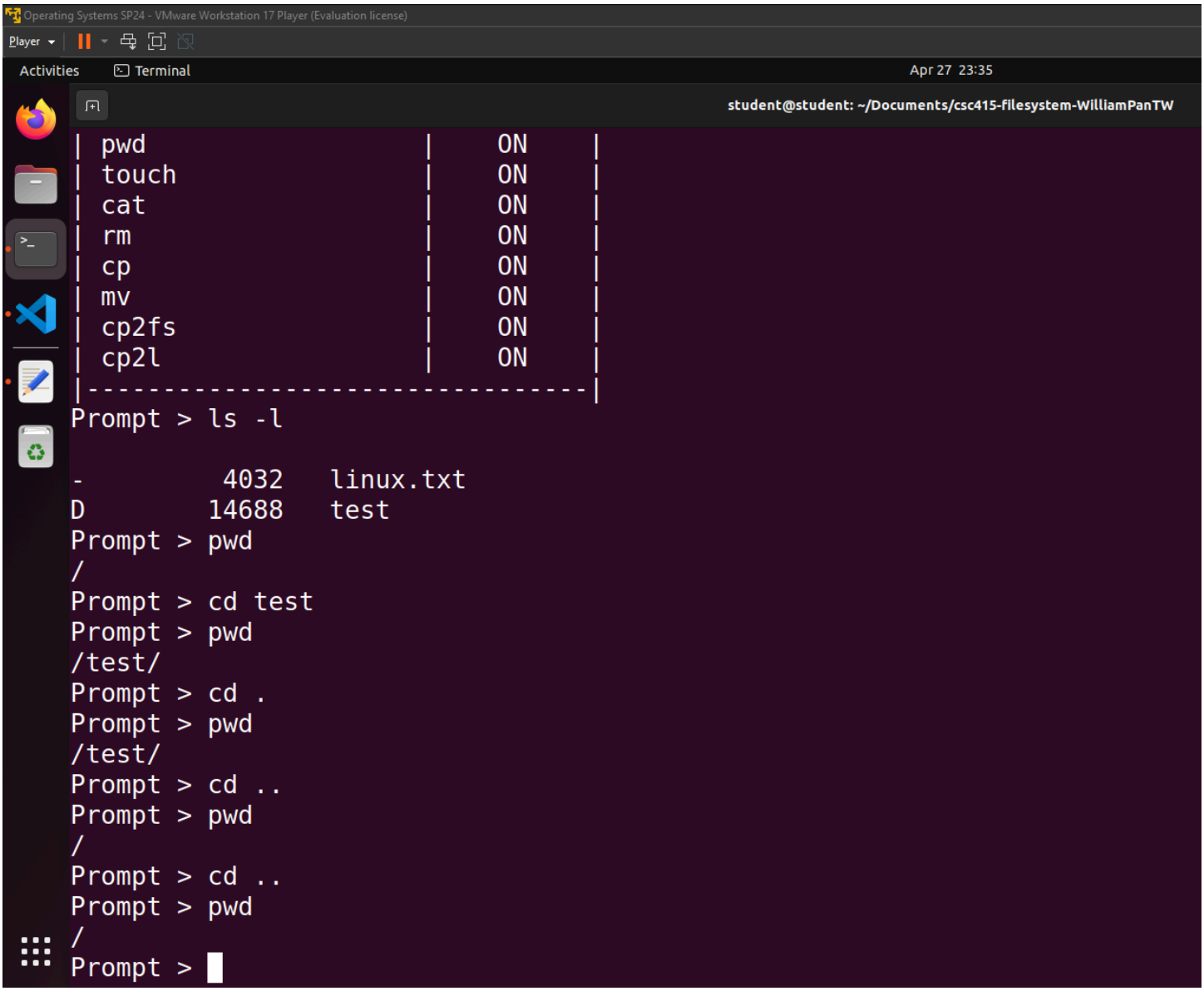



Operating Systems SP24 - VMware Workstation 17 Player (Evaluation license)
Player
Activities
Terminal
Apr 27 23:27
student@student: ~/Documents/csc415-filesystem-WilliamPanTW
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; Return 0
Initializing File System with 19531 blocks with a block size of 512
---------- 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
Prompt > cp2fs linux.txt
Prompt > ls
linux.txt
Prompt > cat linux.txt
Hello there
Prompt >

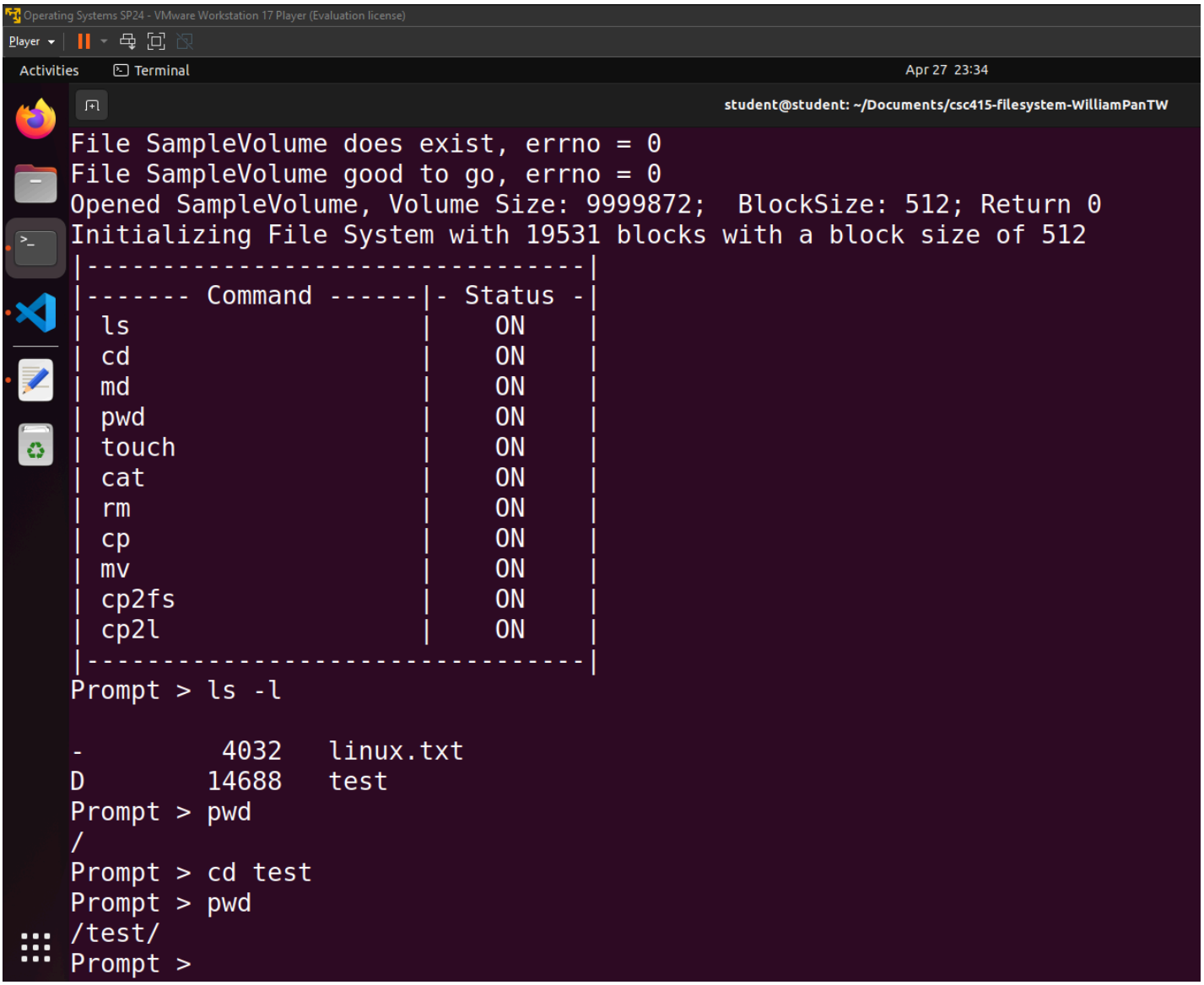
5.11. cat - (limited functionality) displays the contents of a file



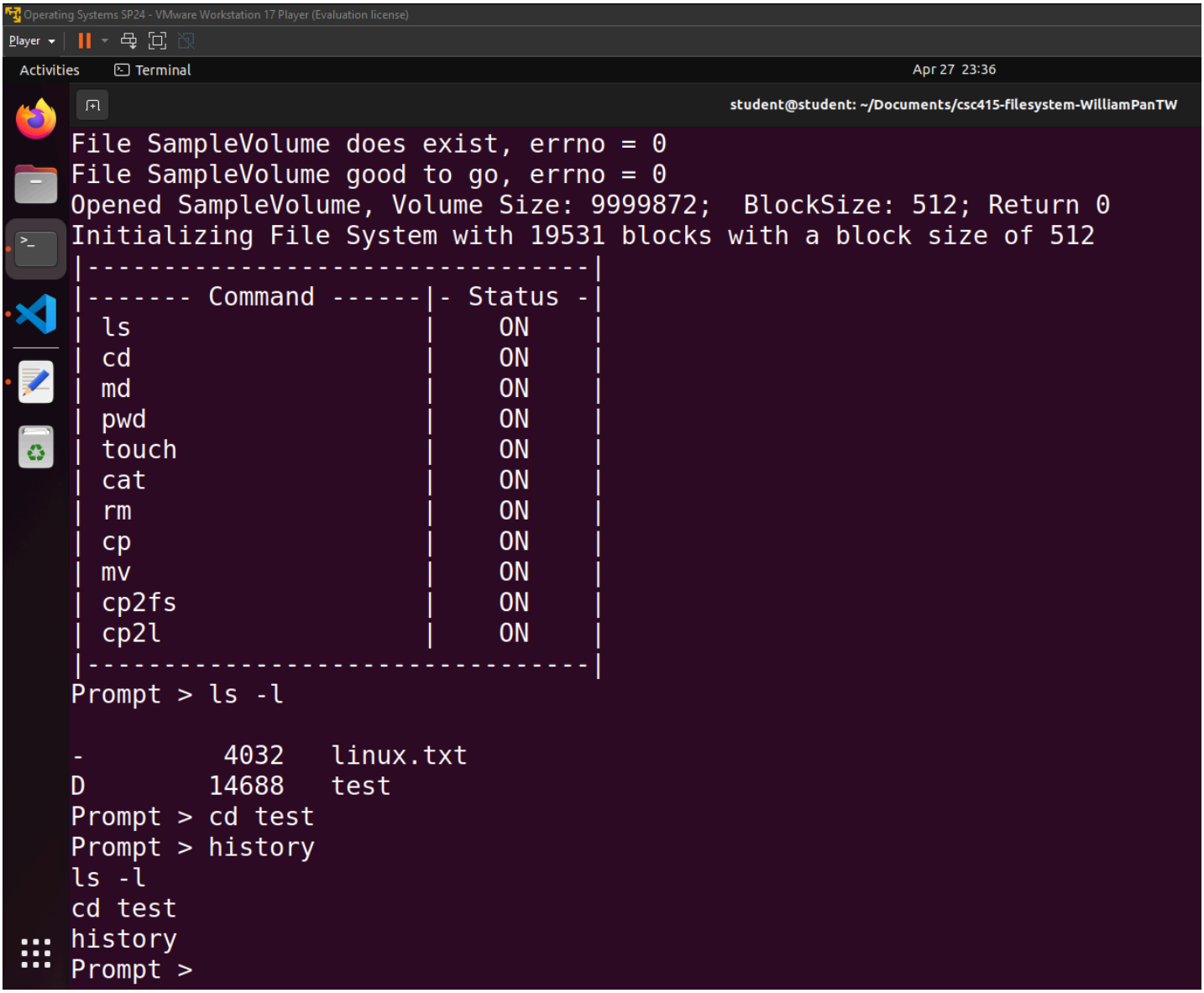
5.12. cd - Changes directory



5.13. pwd - Prints the working directory



5.14. history - Prints out the history



5.15. help - Prints out help