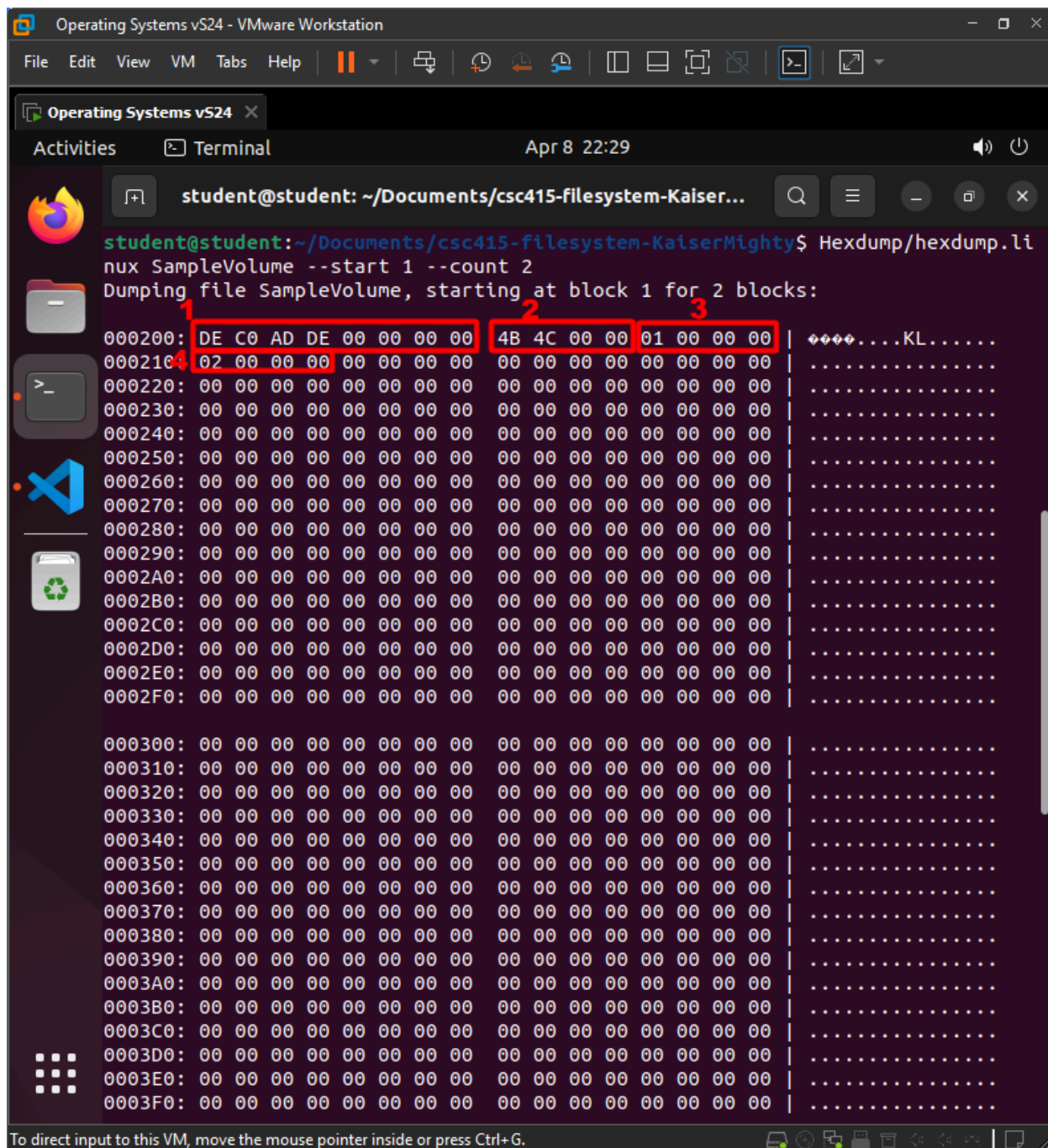


Maximum Effort File System Project

Name	Student ID	GitHub Name
Teammate 1	123456789	GitHubUser1
Teammate 2	123456789	GitHubUser2
Teammate 3	123456789	GitHubUser3
Krishna	123456789	KaiserMighty

Milestone 1

1. Hex Dump



```
Operating Systems vS24 - VMware Workstation
File Edit View VM Tabs Help
Operating Systems vS24
Activities Terminal Apr 8 22:29
student@student: ~/Documents/csc415-filesystem-Kaiser...
student@student:~/Documents/csc415-filesystem-KaiserMighty$ Hexdump/hexdump.linux SampleVolume --start 1 --count 2
Dumping file SampleVolume, starting at block 1 for 2 blocks:
000200: DE C0 AD DE 00 00 00 00 4B 4C 00 00 01 00 00 00 | ****....KL.....
000210: 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000250: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000260: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000270: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0002F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000310: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000320: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000340: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000350: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000360: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000380: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
000390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
0003F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.
```

1. **Signature:** First 4 bytes is 0xDEADC0DE (our signature), last 4 of long are unused.
2. **Total Blocks:** Hex 4C4B translating to 19531 in decimal, the correct value.
3. **Free Space:** 1 is the block number where the free space extents are, correct value.
4. **Root Directory:** 2 is the block number of where the root directory is, correct value.

```
Operating Systems vS24 - VMware Workstation
File Edit View VM Tabs Help
Operating Systems vS24 x
Activities Terminal Apr 8 22:29
student@student: ~/Documents/csc415-filesystem-Kaiser...
0003F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
5 6 7 8
000400: 34 00 00 00 19 4C 00 00 FF FF FF FF FF FF FF FF | 4....L.....
000410: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000420: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000430: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000440: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000450: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000460: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000470: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000480: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000490: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004A0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004B0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004C0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004D0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004E0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0004F0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
9
000500: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000510: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000520: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000530: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000540: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000550: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000560: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000570: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000580: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
000590: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005A0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005B0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005C0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005D0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005E0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
0005F0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | .....
student@student:~/Documents/csc415-filesystem-KaiserMighty$
```

- 5. **Extent Location:** 34 hex translates to 52 decimal, correctly representing where the next free blocks start
- 6. **Extent Count:** 4C19 hex translates to 19481 decimal, representing how many contiguous free blocks are available.
- 7. **Unused Location:** FFFF hex translates to -1 decimal, our sentinel for an unused extent.
- 8. **Unused Count:** FFFF hex translates to -1 decimal, our sentinel for an unused extent.
- 9. **Unused Extents:** Correctly represents all other extents as unused.

```
student@student: ~/Documents/csc415-filesystem-KaiserMighty$ Hexdump/hexdump.linux SampleVolume --start 3 --count 2
Dumping file SampleVolume, starting at block 3 for 2 blocks:

000600: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000610: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000630: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000660: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000670: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000690: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0006F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....

000700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000710: 00 00 00 00 00 00 00 00 02 00 00 00 32 00 00 00 | .....2...
000720: FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000730: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000740: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000750: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000760: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000770: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000790: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0007F0: 00 00 00 00 00 00 00 00 00 64 00 00 01 00 00 00 | .....d....
```

- 10. **Filename:** 2E in hex translates to “.”, which is the correct filename.
- 11-13. **Timestamps:** Timestamps for creation, modification, and access. Set to 0 since they are currently not implemented.
- 14. **Extent Location:** 2 is the correct block where our directory begins.
- 15. **Extent Count:** 32 in hex is 50 in decimal, the correct number of blocks (1 per DE).
- 16. **Unused Extents:** Begins with FFFF or -1 to indicate end of used extents.
- 17. **Filesize:** 6400 in hex translates to 25600, which is correct since it’s 512*50.
- 18. **isDirectory:** 1 is the correct flag, since this is a directory.

```

student@student: ~/Documents/csc415-filesystem-Kaiser...
0007F0: 00 00 00 00 00 00 00 00 00 64 00 00 01 00 00 00 | .....d.....
19
000800: 2E 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000810: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000820: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000830: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000840: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000860: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000870: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000890: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0008F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
20
000900: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
21
000910: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
22
000920: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
23
000930: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
24
000940: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
25
000950: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000960: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000970: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
000990: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0009A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0009B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0009C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0009D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
0009E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | .....
26
27
0009F0: 00 00 00 00 00 00 00 00 00 64 00 00 01 00 00 00 | .....d.....
student@student: ~/Documents/csc415-filesystem-KaiserMighty$

```

- 19. **Filename:** 2E2E in hex translates to “..”, which is the correct filename.
- 20-22. **Timestamps:** Timestamps for creation, modification, and access. Set to 0 since they are currently not implemented.
- 23. **Extent Location:** Unused since the root directory doesn't have a parent.
- 24. **Extent Count:** Unused since the root directory doesn't have a parent.
- 25. **Unused Extents:** Completely unused since the root directory doesn't need these.
- 26. **Filesize:** 6400 in hex translates to 25600, but in reality won't ever be referenced.
- 27. **isDirectory:** 1 is the directory flag, but in reality won't ever be referenced.

2. Description of VCB Structure

Our volume control block structure has the following fields within it: "totalBlocks" which is an integer that represents the total number of blocks that the volume has; "rootDirectory" which is an integer that represents the starting block number for our root directory; "signature" which is a long that will hold our signature/magic number in hexadecimal. Our signature is 0xDEADC0DE. The volume control block has another field "freeBlocks" which is a 2d array of integers that holds extents. This array of extents will keep track of which blocks are free in the volume. Similarly the volume control block also has another field "allocatedBlocks" which is another 2d array of integers that holds extents. This array of extents will keep track of the used blocks within our volume.

3. Description of Free Space Structure

Our free space structure uses extents. We decided to use extents for our file allocation, and figured that it would be easier to use the same system for free space allocation. The extent struct is just two integers, for the location and count respectively. For practical space allocation, an array of these extents are used. To initialize the freespace, pass the total number of blocks to the initFreeSpace function. The function will malloc free space globally. The VCB includes the block number for where free space is, by default this is 1. Only one extent is initialized at first; it starts at block 2, and has the totalBlocks-2 as it's count (1 for the VCB, and 1 for the freeSpace itself). The freeSpace is then written to disk with LBA write (the free space extents use up an entire block).

There are two functions included to find free extents and return used extents to the free space system. These functions return and take an array of extents respectively. While finding free space, the system iterates through each free space extents until it either finds a contiguous free extent, or an array of discontiguous extents that meets the requested amount of blocks (whichever condition is first met). The return free space function iterates through the user function, comparing the user's extents' to find a corresponding free extent where the $\text{user.location} + \text{user.count} == \text{freespace.location}$. Once located, the user's extent.count is subtracted from the freespace.location to return those blocks to being free.

4. Description of Directory System

Our Directory System is an entry that is comprised of a filename, which is the name of the file/entry. We also have an integer extent which has the location of the free blocks. We also have isDirectory which is a directory flag to tell us if its a directory or not. We also some metadata which are Dates - Modified, created, and accessed. These Just tell us additional information about the entry, when a file/entry was created, changed or opened. We also have a file size which shows us how big each entry is in bytes. In the Directory System, we have a root directory which is the beginning of our directory. The system is an array of directory entries, with . and .. starting at index 0 and 1 respectively. Everything else would be the rest of the entries which could be for example, home, usr, temp, etc.

5. Table of Who Worked On Which Components

Name	Component
Teammate 1	Part 1
Teammate 2	Part 2
Krishna	Part 3
Teammate 3	Part 4

6. Meeting Info and Task Division

Our group had two meetings, however we kept in contact and provided updates on our tasks in our discord server a couple times each week. During our first meeting we decided to divide the tasks of milestone 1 so that each member would be responsible for one of the four parts and then talked about how we wanted to handle file allocation and the free space allocation in our file system. We decided on using an array of extents to track the number of free blocks and their locations as well as using another array of extents to track the number of used blocks and their locations within our volume. Teammate would be responsible for Part 1, Teammate would be responsible for Part 2, Krishna would be responsible for Part 3, and Teammate would be responsible for Part 4.

During our second meeting, we talked about how we should revise our volume control block structure in accordance with the feedback we got on our File System design assignment. We then talked in more detail about how the free space allocation was going to be handled using extents so that the group had more understanding of how part 3 was going to be implemented.

7. Issues and Resolutions

Part 1 Issues and Resolutions

- My first issue was understanding the purpose of a volume control block and why we need one in the first place.
 - I resolved this by rewatching all the canvas file system lectures which helped me review and understand the concepts relating to a volume control block.
- My second issue was figuring out what a magic number or signature was.
 - I resolved this issue by watching the March 7th zoom lecture recording which went over what a magic number was, and why it's important.
- My third issue was figuring out how to check if our volume was already formatted.
 - I resolved this issue by watching the March 7th zoom lecture recording and also the Steps for Milestone 1 pdf which helped me figure out what I was supposed to do for part 1 of the milestone.

Part 2 Issues and Resolutions

- My first issue was understanding how to initialize the file system properly if it had not been already
 - I resolved this by consulting my teammates on which methods to use and what values to set
- My second issue was trying to see what proper values to set and how to make sure that we got the desired result
 - I resolved this by looking through my teammates code and understanding how each part of the file system would be initialized

Part 3 Issues and Resolutions

- My first issue was linking my external object to fslnit.c.
 - I resolved this by adding the object to the makefile, creating a corresponding header for my external file, and including that within fslnit.
- My second issue was potentially repetitive code outside of my file.
 - I resolved this issue by creating helper functions inside my file for copying extents from one array to another and writing the contents of extents to disk.

Part 4 Issues and Resolutions

- My first issue was understanding the reason why we needed to calculate the actual number of entries we needed.
 - My issue was resolved by looking at the March 19th lecture and understanding that sometimes calculating blocks needed will leave us with a certain amount of bytes but those bytes may have bytes that are wasted which we can salvage by calculating the actual entries we need so we waste less bytes.
- My second issue was trying to initialize the name of the "." and ".." directory entries. I was getting that the expression must be a modifiable lvalue
 - I resolved this issue by watching the lecture and reading the error messages to figure out that I cant just assign the name directly to "." and "..". I have to strcpy them to be able to set it equal to the first and second directory entries, respectively
- My third issue was trying to initialize the rest of the directory entries in a loop.
 - I resolved the issue by watching the March 19th lecture and finding out that one of the ways is to set each entries first byte of the name to null, which meant it would be an unused entry
- My fourth issue was transitioning over to extents. So for our system, we needed a way for files to support discontiguous. So one way to do that was for us to switch to extents. That meant revising parts of my code to be able to work with extents.
 - I think a good part of me learning was to go back to the March 14th lecture and read up on extents and how useful they are for entries and then ask some groupmates for further guidance.

Full Project

Work Table (Files and Functions)

File Name	Function Name	Implementation
fsInit	createDirectory	Teammate 2
fsInit	initFileSystem	Teammate 1
fsInit	exitFileSystem	Krishna
fsFreeSpace	All Functions	Krishna
fsPath	All Functions	Krishna
fsshell	cmd_mv	Krishna
mfs	fs_mkdir	Teammate 2
mfs	fs_rmdir	Krishna
mfs	fs_mvitem	Krishna
mfs	fs_opendir	Teammate 1
mfs	fs_readdir	Teammate 1
mfs	fs_closedir	Teammate 1
mfs	getCwd	Krishna
mfs	setCwd	Krishna
mfs	setCwdString	Krishna
mfs	freeDir	Krishna
mfs	fs_getcwd	Teammate 2
mfs	fs_setcwd	Teammate 2
mfs	fs_isFile	Teammate 2
mfs	fs_isDir	Teammate 2
mfs	fs_delete	Krishna
mfs	fs_stat	Teammate 1
b_io	b_open	Teammate 1
b_io	b_read	Krishna
b_io	b_write	Krishna
b_io	b_seek	Teammate 2
b_io	b_close	Krishna

mfs.c functions

Fs_mkdir (Teammate 2)

The function's purpose is to create a directory given a specified path. This function takes in a path name as an argument which is the passed by the user. We first take the result from parsing the pathname and see if we get a valid path to use. If we do, then we proceed to create a directory, in our case, we chose 50 entries. Then we find an unused entry to set to be the index and use that index to update the variables of our directory.

Issues/Resolutions

- My first issue was how to be able to get the last index for our directory.
 - I resolved this issue by realizing that we need to retrieve the index via a function which allows us to get the index of an unused directory.

fs_rmdir (Krishna)

The function's purpose is to remove a directory from the file system. It takes a path as an argument. We parse the path to check if the result is an existing directory. If yes, we simply set its name to null and return its extents to free space. We also iterate through the directory and recursively delete/remove all existing items (both files and directories) within the directory and return their blocks to free space as well.

Issues/Resolutions

- I had issues with removed directories still showing up when I did ls.
 - I discovered that this was being caused by a duplicate of the root directory being loaded by parsePath, which is what I would change and write to disk. Meaning that on disk, I correctly removed the file (which could be tested by exiting and re-entering the file system). However, since the "duplicate" root I wrote to wasn't the one loaded in memory, it would never update that one.
 - I resolved this by setting a sentinel for the root directory, I chose to set the filesize of root to -1. Then I simply checked if the parsePath result had a -1 filesize, if it did, I simply got the already loaded into memory root and discarded the parsePath result.
- I realized that I could delete the . and .. directories, included root.
 - To prevent this, I hard coded some limitations to make sure the returned index is at least 2 (skipping 0 and 1 for . and .. respectively).

fs_mvdir (Krishna)

The function's purpose is to move an item (directory or file) from where it currently is to a user specified destination. We call `parsePath` to check if the destination is a directory with an available slot. If yes, we simply set the filename to null at the source and copy all of its data (filename, filesize, extents, etc) to the destination slot. We then just write it all to disk.

Issues/Resolutions

- I struggled to figure out what to write within `fsshell`.
 - I decided to keep it simple. I copied the code for "`cmd_cp`" to retain the argument parsing, but changed the second half to simply call an `mfs` function. I ended up putting all the `mv` logic in it's own function in `mfs`, where it made most sense to place.

fs_isFile (Teammate 2)

The purpose of this function is to determine whether the current path specified is a directory or if its a file. The function takes in a filename as a parameter and uses that for parsing the path. If the name of the file is valid, then we proceed to check if the `isDirectory` flag in our directory entry struct is not 0. If its not 0 we return 1 indicating it is a file, otherwise return 0, indicating it is not, i.e. a directory.

Issues/Resolutions

- I didnt really have much of an issue with this because all you do is check for directory or file and return the 1 or 0 respectively
 - In the beginning i was confused about the argument being passed called a filename and not pathname and after talking with some groupmates, I realized its used in the same manner, just means 2 different things so we can just parse the path.
 - We did however change the definition of our `isDirectory` flag so its no longer 0 or 1, rather it is either 0 or not. This is to make it work with discontiguous file systems.

fs_isDir (Teammate 2)

The purpose is similar to that of `isFile`. Basically you take in a pathname and parse it. If that parsed path is valid, then we check if it is > 0 , and if it is, we return 1, indicating it is a directory, and 0, indicating it isnt, i.e. a file.

Issues/Resolutions

- For this one, I didnt really find an error, as u just write a simple if function to test if its a file or directory and return 0 or 1 respectively.

getCwd (Krishna)

This is a simple getter function that just returns the global and loaded into memory directory. It's used in parsePath for getting the cwd.

setCwd (Krishna)

This is a simple setter function that sets the caller's directory as the cwd. This function is used for loading root into the cwd when we start the file system in fsInit.

setCwdString (Krishna)

This is a simple setter function that sets the caller's directory as the cwd. This function is used for setting the cwd string to be "/" when we start the file system in fsInit.

freeDir (Krishna)

This is a handy function I made because I found myself writing the same lines of code repeatedly. All it does it check if the caller's directory is root, cwd, or null; if it isn't, it will free it.

fs_delete (Krishna)

This functions purpose is to delete non-directory directory entries. It calls parse path and checks the result to make sure it exists and isn't a directory. If those conditions pass, it just sets the filename to be null and returns it's allotted blocks to the free space system.

fs_getcwd (Teammate 2)

This functions purpose is to retrieve the current working directory. How it works is that it takes in 2 parameters, one for pathname, and one to check the size of our current working directory. First we check the length. If the length of our absolute path name exceeds the size bytes, then we have to return NULL, indicating there is an error. If there isnt a excess size taken, then we copy our current working directory into our argument passed in the function and return that pathname variable.

Issues/Resolutions

- My first issue was trying to figure out what to return. I thought at first that maybe we needed to return the cwd and not the pathname
 - I realized that doesnt really make sense because our cwd contains the current directory, so we would need to copy that into the pathname specified and that would let the user know what directory is the current one.

fs_setcwd (Teammate 2)

I think this was one of the hardest ones to understand. The purpose of this function is to take the pathname and change the cwd based on the specified argument. Basically we need to first check if the path is valid or not by parsing it, check if the entry is valid, and also if it is a directory or not. If all of it is valid, then we proceed to load the directory into memory. That is going to be the directory to change. Then we just set that directory to the cwd variable and we are done, at least with the systems end of it. Now we need to update the users end of it. To do that we need to identify if its an absolute path or relative path, and update the string accordingly. That is technically the function finished but we can take it a step further and handle the case where the use passes in a path with all . and .. - all we need to do is loop through and collapse all the . and .. and copy the resultant path. Now the function is complete.

Issues/Resolutions

- The first issue was loading the directory. I had no idea how to do that and what function to call
 - I resolved this by contacting my groupmate and he helped create a helper function to load the cwd.
- My second issue didnt come until after realizing we needed to deal with the user passing in either an absolute path or relative path. If he passed in an absolute path, its simple, just copy what he put in. if its a relative path, we have to do all that work to concatonate the path, and also to check for . and .. to collapse the path for readability.
 - I resolved the issue by first looking at the man page for a more descriptive approach on how to deal with certain cases and then looking for the announcement on how to finish this function which also helped me understand how relative and absolute paths work.

fs_opendir (Teammate 1)

This function's purpose is to open a directory. It takes in a path name as an argument which is passed in by the user. It starts with declaring a pathInfoptr variable which is a pointer to a pathInfo struct. We then pass this pointer as an argument to the parsePath function along with the pathname. If parsePath returns a value of -1, fs_opendir returns NULL indicating an invalid path. If the path is valid, the function then declares a pointer to a fdDir struct which we use to set the fields of the struct using the pathInfoptr. This function then returns the fdDirptr.

Issues/Resolutions

- My first issue with this function was figuring out how the pathInfo struct was involved in this function and how I would access it.
 - I resolved this issue by reviewing the lecture recording that went over this and also talking it over with my partner which made it more clear on what I had to do.

fs_readdir (Teammate 1)

This function's purpose is to read the directory that was opened by the `fs_opendir` function. This function takes in a `fdDir` struct pointer that is returned by the `fs_opendir` function. This function iterates through the directory opened by the `fs_opendir` function and returns a filled `fs_direntinfo` struct for each item in the directory as it iterates through them. At the beginning of the function, we check to see if the current directory entry position is greater than the number of elements in the directory, in which case the function would return null because there is no directory entry to read. Otherwise, we set the fields of the `fs_direntinfo` struct for the current directory entry and iterate to the next directory entry. After the function iterates through all the directory items in the current directory, the function returns a pointer to the filled `fs_direntinfo`.

Issues/Resolutions

- My first issue with this function was figuring out why there were two separate variables named "d_reclen" in the `fs_direntinfo` struct, and the `fdDir` struct and what they represented
 - I resolved this issue by asking in Slack and finding out that they represented the size of the respective structures they are apart of and that they indeed were separate/different variables.
- My second issue with this function was figuring out how to iterate through the items of the directory
 - I resolved this issue by realizing that I could use the `dirEntryPosition` field of the `fdDir` struct to keep track of where I was in the directory array, and then I could check to see if the position of the current directory entry was bigger than the amount of directory entries inside the directory. In which case, the function would return NULL. Otherwise it would fill out the corresponding `fs_direntinfo` for the current directory entry.

fs_closedir (Teammate 1)

This function's purpose is to close the directory that was opened by `fs_opendir`. The function does this by freeing the memory that was used for the `fdDir` and `fs_direntinfo` structure pointers used in `fs_readdir`. This function returns a 0 when executed successfully and a -1 otherwise.

fs_stat (Teammate 1)

This function's purpose is to fill the fields of the `fs_stat` structure which represent statistics for each directory entry. This function takes in a path, and a buffer variable which is a pointer to the `fs_stat` structure. This uses the buffer variable and fills each of the fields of the `fs_stat` structure for the given directory entry (regardless if the directory entry is a directory or a file). This function uses a pointer to the `pathInfo` structure to access information about the directory entry that will be used for the statistics. This function begins getting the number of blocks that the directory entry is by looping through the amount of extents that directory entry has. It then just uses the `pathInfo` structure pointer to fill the other fields of the `fs_stat` structure for the given directory entry. Function returns a 0 if no error occurred, and a -1 otherwise.

Issues/Resolution

- My first issue with this function was figuring out if the statistics were for our file system overall, or if they were for a directory entry
 - I resolved this issue by asking in slack and finding out that the statistics were for a directory entry, regardless if it is a directory or a file
- My second issue with this function was figuring out what to do with the passed in path and buffer arguments
 - I resolved this issue by realizing I could make another `pathInfo` structure pointer and then use the `parsePath` function to fill that pointer using the passed in path. Then I could use the filled `pathInfo` structure pointer to then set the statistics for the given directory entry, using the buffer argument.

b_io.c functions

b_open (Teammate 1)

This function's purpose is to open a file with a given file name with the given flags. This function is passed in the file name and the flags by the user. This function begins by checking to see if the file name argument is valid by passing it as an argument to the parsePath function along with a pointer to a pathInfo structure for the parsePath function to fill in if the filename is valid. If the filename is not valid, parsePath returns a -1, and then b_open returns a -1 indicating an error. If the filename is valid, the function then gets a file control block from the fcbArray and then uses that to fill the fields of the b_fcb using the pathInfo pointer to access the information about the file. The function sets the fields of the b_fcb struct, differently based on what flags were passed by the user. The flags passed in by the user are stored in the flags field of the b_fcb struct for the given file so that other functions in b_io.c can check them. If the O_CREAT flag was passed, then the function makes an empty file and sets the corresponding attributes of that empty file for the fcb struct. If the O_APPEND flag was passed, we set the location fields of the fcb structure, to the end of the file such as filePosition to the file size, the blockIndex to the last block of the file, the bytes read to the size of the file, and the index (which represents the current position in the file) to the end of the file. If the O_TRUNC flag was passed as an argument, the function makes the file empty by setting the file attributes to 0. If no flag was passed and the file exists, we set the fi field of the fcb struct to the file pointer obtained from the parsePath function, then the index, bytesRead, and filePosition to 0 and the blockIndex field to the first block of the file so that the fcb pointer starts at the beginning of the file. This function then returns the file descriptor for the passed in file name.

Issues/ Resolutions

- My first issue was figuring out what each of the flags did and how I should check them
 - I resolved this issue by looking at the man page for the linux open function which described the flags and their functions and I figured out how to check the flags by utilizing the approach that the Professor mentioned in slack that uses a bitmask.
- My second issue was figuring out how to utilize the fcb structure and file descriptor
 - I resolved this issue by reviewing assignment 5 and realizing that the process was very similar.

b_read (Krishna)

I was able to get this working by simply copying over my function from assignment 5 and tweaking it a little. I was already reading one block at a time in assignment 5, which worked conveniently for extents since we can't know for sure if we have contiguous files or not. First I do the standard error handling, then I check if the file is open in write only mode, in which case I just return. I then check if the last thing the file did was seek. If it was, I reset the blockIndex based on our formula $[(bytes + blockSize - 1) / blockSize]$. The rest of the logic is identical to my assignment 5. I check if the request goes outside the remaining bytes. If so, I set the request to the remaining bytes. If the buffer is not empty, I read from there first before LBaread. If the buffer has enough bytes to satisfy the request, I copy it to the user buffer and check if we're reached the end of the buffer, resetting it if yes. If the buffer does not have enough bytes for the request, I copy it all to the user buffer and reset it. Then I check if the remaining bytes to read are more than chunkSize. If yes, I read directly into the user buffer. If not, I read into our buffer and memcpy the requested bytes into the user buffer.

b_write (Krishna)

I started off by copying my b_read code into here. I check if the file is open in read only mode, in which case I just return. I then check if the last thing the file did was seek. If it was, I check if the current filePosition is past the fileSize; if it is, I set the fileSize to the filePosition. I check if the block we're about to write to is beyond the existing extents, if it is, I append the extents to get more blocks. Then, I go into the actual writing. If the write request is larger than chunkSize and the buffer is empty, I write directly to disk. If the request is smaller than chunkSize, then I check if the request asks for more bytes than our buffer can currently support. If yes, I fill the buffer and write it to disk, resetting the buffer in the process. If not, I just fill the buffer, and then check if it's full. If it is, I write it to disk and reset the buffer. Lastly, I check if the block we just wrote to is the "largest" block we've written to use (I use this to check where we need to trim extents from), and set the modified time to now. I also add the amount of bytes written to the fileSize.

Issues/Resolutions

- I realized that I had one extra character than I was supposed to.
 - I thought this is because I was not adding the null terminator, so I manually added that in close. But then realized that I can't make assumptions about the data, so I rechecked my logic and realized I forgot to subtract 1 to account for the differences between indexes and sizes.

b_seek (Teammate 2)

This function's purpose is to reposition the file offset of the open file description associated with the file descriptor `fd` to the argument offset according to the directive *whence* using `SEEK_SET`, which updates the file to the offset passed in as the argument, `SEEK_CUR`, which updates the offset to the current location of the file plus the offset passed in, and `SEEK_END`, which sets the offset to be at the end of the file plus offset bytes passed in. I think what makes this hard is that seek seems easy to write, but it changes with the function calls to read and write. So you have to update the offsets based on that.

Issues/Resolutions

- I think the biggest grasp for me was just trying to understand how the function operates and how the offset works. At the moment I am not sure if it has been implemented correctly but an issue following that is to see if seek works with read and write because I know that with write, you can seek out of bounds it's just that you need write to fill in those gaps which the offset is set too. Those changes are something I need to think about.
 - I haven't come up with a solution yet as I am not sure if it's working properly or not

b_seek (Krishna)

I picked up where Teammate 2 left off, debugged it, and extended it to check what the last action was. If the last action was write, then I checked the buffer, if it wasn't empty, I wrote it to file and reset it.

b_close (Teammate 3)

The function's purpose is to close the file, free the buffer associated with the file descriptor, and reset the variables that belong in the VCB. The function takes the '`fd`' variable as a parameter and first checks if the variable is within range. If not, then the function will send out a flag of 'Invalid file descriptor'. The function was fairly easy to test since we had finished all the other functions, and was able to prevent any other time-wasting errors.

Issues/Resolutions

- I had no major issues with this function as it was very self-explanatory, and I had other functions in the project to refer to if I had any doubts. I had an issue with closing the buffer since I accidentally created too many NULL pointers.

b_close (Krishna)

I added a check to see if the last thing we did before closing was a write. If yes, and the buffer is not empty, then I write the remainder of the buffer to the file before closing. I then check if we need to trim the extents, in case the user doesn't use all the allocated extents, and then call my trimming function if needed. Then I write the parent directory to disk to write the changed file control block to disk.

fsshell.c functions

cmd_mv (Krishna)

This is a simple function that just checks if the user passed two parameters, and passes those to the fs_mvitem function in mfs.

Issues/Resolutions

- I realized that I could pass the same directory as the source and destination, which had some weird results.
 - I just hard coded a string comparison that exited if the source and destination were the same.

fsshell.c functions

parsePath (Krishna)

This is an essential function that bridges the gap between the user's string based pathnames and the file systems directoryEntry objects. I implemented the function as shown in class.

Issues/Resolutions

- I saw some weird results when working with root, where parsePath would load a duplicate of root and return that. Resulting in situations where changes in the duplicate would not transfer over to the global root (since it never reloaded after fsInit).
 - I resolved this by calling my getRoot helper function inside parsePath before returning if the parent was root. This prevented me from creating diverging duplicates.

loadDir (Krishna)

The parent directory only holds the control blocks of each directory, or the . entries. Given a . entry, this function will LBAread the rest of the directory with the extents helper function and return it.

findInDir (Krishna)

This function iterates through a provided array of directory entries, checking to see if the provided target string exists in one of the entries filenames. If it does, it returns the index of that entry. If it doesn't, it returns -1.

findUnusedEntry (Krishna)

This function iterates through a provided array of directory entries, checking to see if any of the entries has a null filename. If it does, it returns the index of that entry. If it doesn't, it returns -1.

clearDir (Krishna)

Given a directory, this function will iterate through each entry (except . and ..) and return its extents to free space. If it finds a directory, it will load it and recursively call itself to clear it.

fsFreeSpace.c functions

appendExtents (Krishna)

A helper function I created for b_io operations. If we need more blocks in a file than we call the find free space function and pass the result into this function with the file's existing extents. This function will create a new array of extents and copy all of the existing extents into it. When out of extents, it will copy all of the newly found free space extents. Once complete, it will return the array of extents to the caller, who should then call copyExtents to replace the item's extents with the returned result.

trimExtents (Krishna)

A helper function I created for b_io operations. When appending a file, the system will allocate double the already allocated blocks. If the user doesn't use all of it. Then they call this function with the extents and the index of the last block they used. The function will move through the extents until it reaches the index. Once reached, it will set the count to the index and create a new array of extents with all the remaining blocks. This new array is then passed to returnFreeSpace.

createSecondaryExtents (Krishna)

A helper function that will create secondary extents, which are a full block (512 bytes) of just extents. The function will automatically place the secondaryExtents to the last extent of the caller's extents and set the sentinel. It will then move the caller's last extent to the first element of the secondaryExtent and returns the array to the caller.

createTertiaryExtents (Krishna)

A helper function that will create tertiary extents, which are a full block (512 bytes) of just ints. The function will automatically place the tertiaryExtents to the last extent of the caller's (primary) extents and set the sentinel. It will then move the caller's last extent (which should be a secondaryExtent) to the first element of the tertiaryExtent and returns the array to the caller.

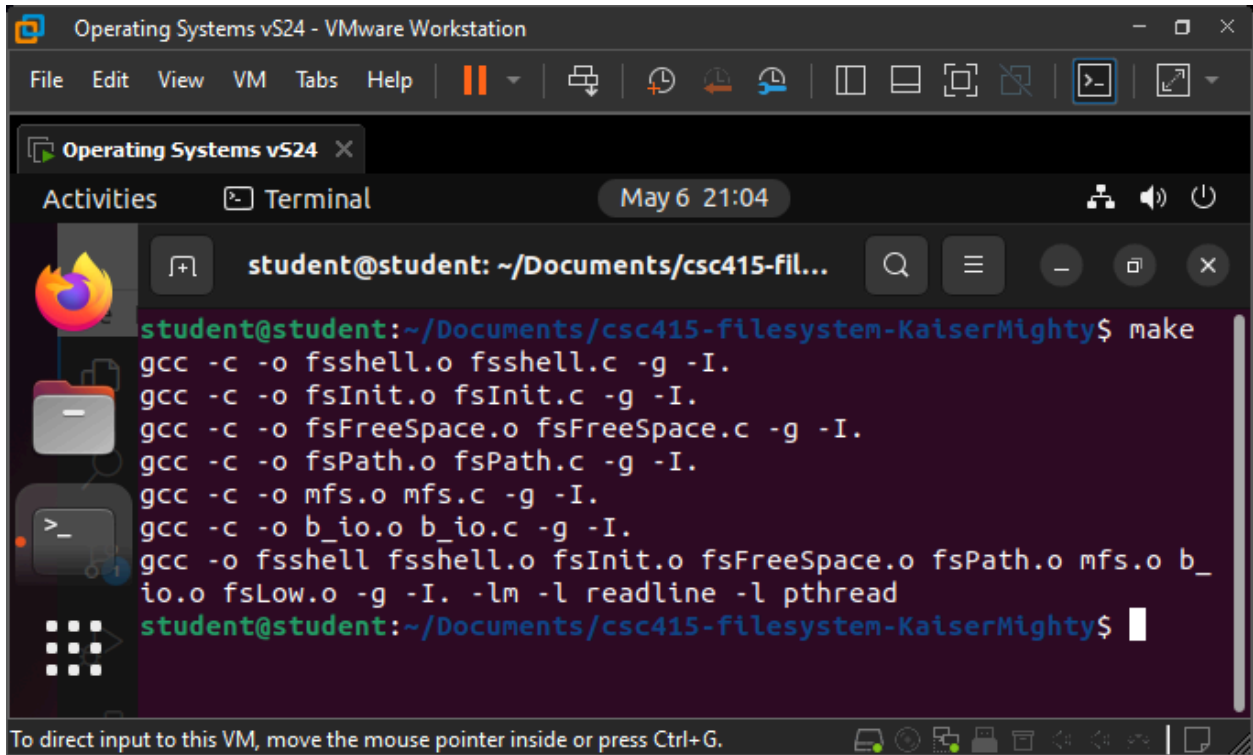
getBlock (Krishna)

A helper function for b_io operations. Given an index, this function will iterate through the extents until it finds the block that corresponds to that index. It does this by counting the "count" portion of each extent until it reaches the target.

getLastBlock (Krishna)

A helper function made primarily for b_io operations, but can also be used to find the total number of blocks allocated to an item. It simply iterates through the extents and returns a sum of all "counts" portions of the extents.

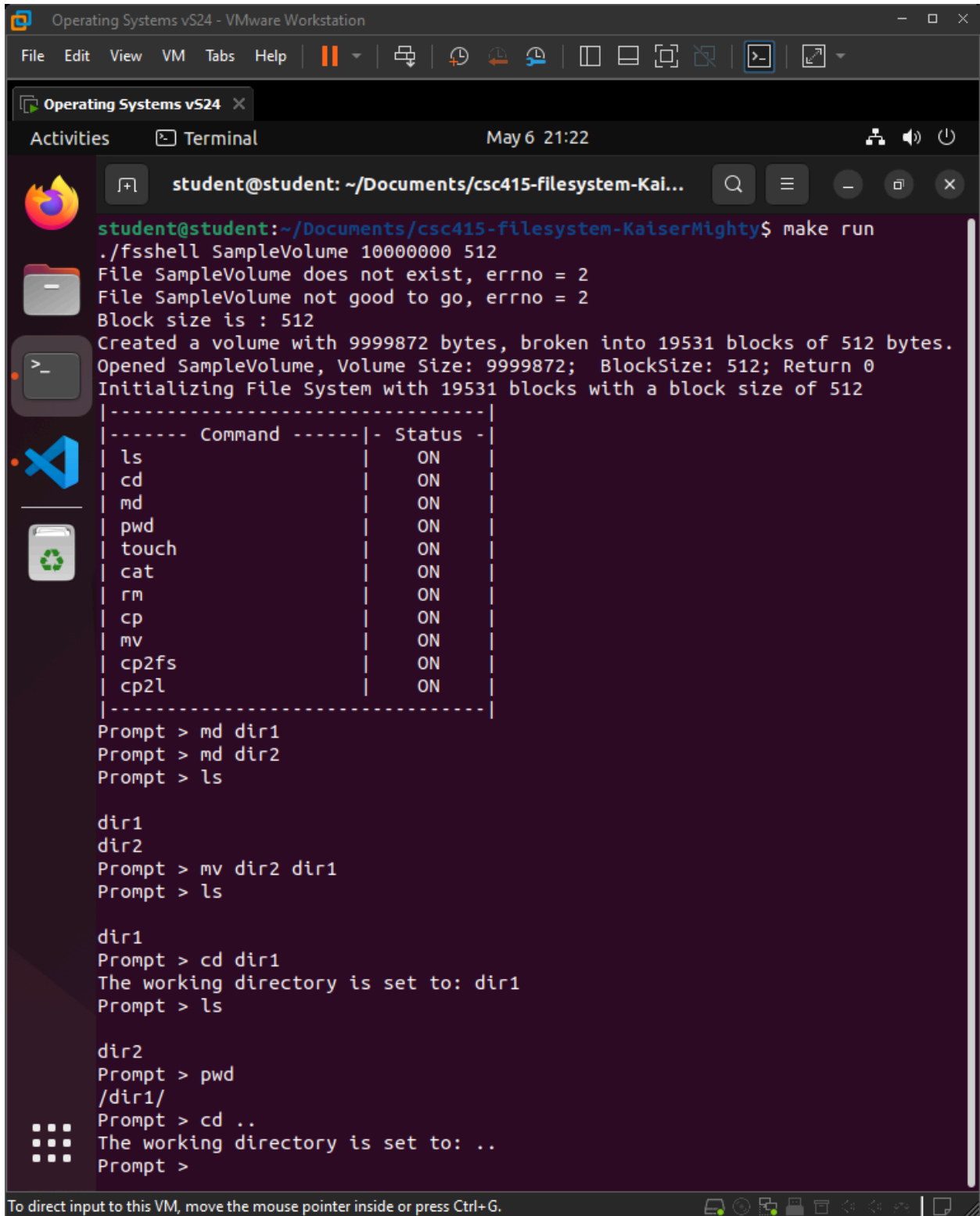
Screenshots (Make)



```
Operating Systems vS24 - VMware Workstation
File Edit View VM Tabs Help
Operating Systems vS24
Activities Terminal May 6 21:04
student@student: ~/Documents/csc415-fil...
student@student:~/Documents/csc415-filesystem-KaiserMighty$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o fsFreeSpace.o fsFreeSpace.c -g -I.
gcc -c -o fsPath.o fsPath.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 fsFreeSpace.o fsPath.o mfs.o b_
io.o fsLow.o -g -I. -lm -l readline -l pthread
student@student:~/Documents/csc415-filesystem-KaiserMighty$
```

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

Screenshots (Make Run)



```
Operating Systems vS24 - VMware Workstation
File Edit View VM Tabs Help
Operating Systems vS24 x
Activities Terminal May 6 21:22
student@student: ~/Documents/csc415-filesystem-Kai...
student@student:~/Documents/csc415-filesystem-KaiserMighty$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does not exist, errno = 2
File SampleVolume not good to go, errno = 2
Block size is : 512
Created a volume with 9999872 bytes, broken into 19531 blocks of 512 bytes.
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 > md dir1
Prompt > md dir2
Prompt > ls

dir1
dir2
Prompt > mv dir2 dir1
Prompt > ls

dir1
Prompt > cd dir1
The working directory is set to: dir1
Prompt > ls

dir2
Prompt > pwd
/dir1/
Prompt > cd ..
The working directory is set to: ..
Prompt >
```

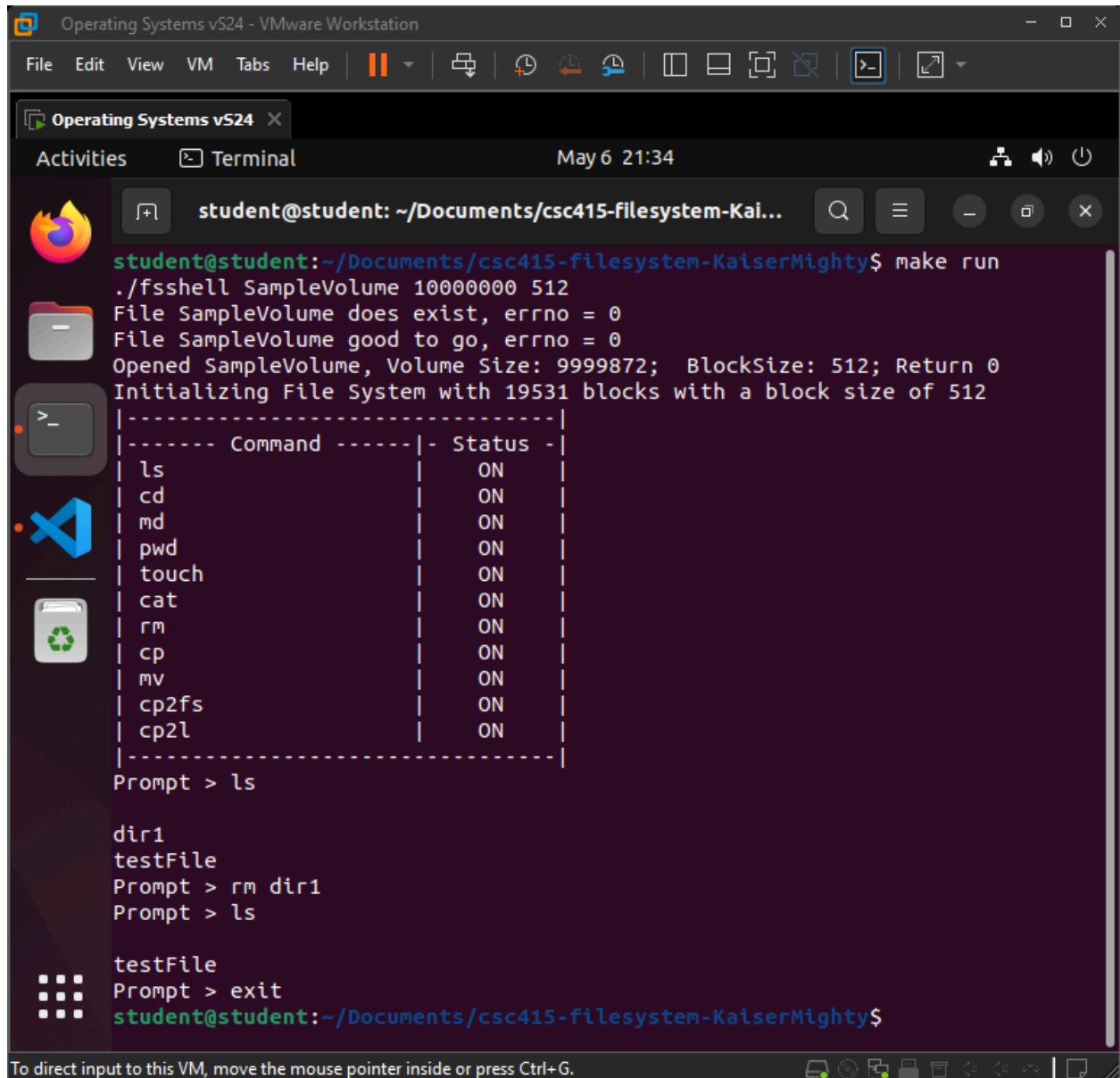
md - make directory
mv - move directory
cd - change directory

ls - list
pwd - print working directory


```
Operating Systems vS24 - VMware Workstation
File Edit View VM Tabs Help
Operating Systems vS24 x
Activities Terminal May 6 21:30
student@student: ~/Documents/csc415-filesystem-Kai...
The working directory is set to: ..
Prompt > pwd
/
Prompt > cp2fs linuxFile testFile
Prompt > ls
dir1
testFile
Prompt > cat testFile
Test file to check the cp2fs and cp2l commands for our filesystem.
cp2fs linuxFile testfile
cp2l testfile fsFile
Prompt > cp testFile dir1 copiedFile
Usage: cp srcfile [destfile]
Prompt > cp testFile dir1/copiedFile
Prompt > ls
dir1
testFile
Prompt > ls dir1
dir2
copiedFile
Prompt > touch /dir1/copiedFile
Prompt > touch testFile
Prompt > cat /dir1/copiedFile
Test file to check the cp2fs and cp2l commands for our filesystem.
cp2fs linuxFile testfile
cp2l testfile fsFile
Prompt > ls
dir1
testFile
Prompt > cp2l testFile fsFile
Prompt > ls -a -l
D          -1  .
D          -1  ..
D       25600  dir1
-         112  testFile
Prompt > exit
student@student:~/Documents/csc415-filesystem-KaiserMighty$
```

cp2fs - copy from linux to file system
touch - open and close file

cp2l - copy from file system to linux
cat - print file contents to shell

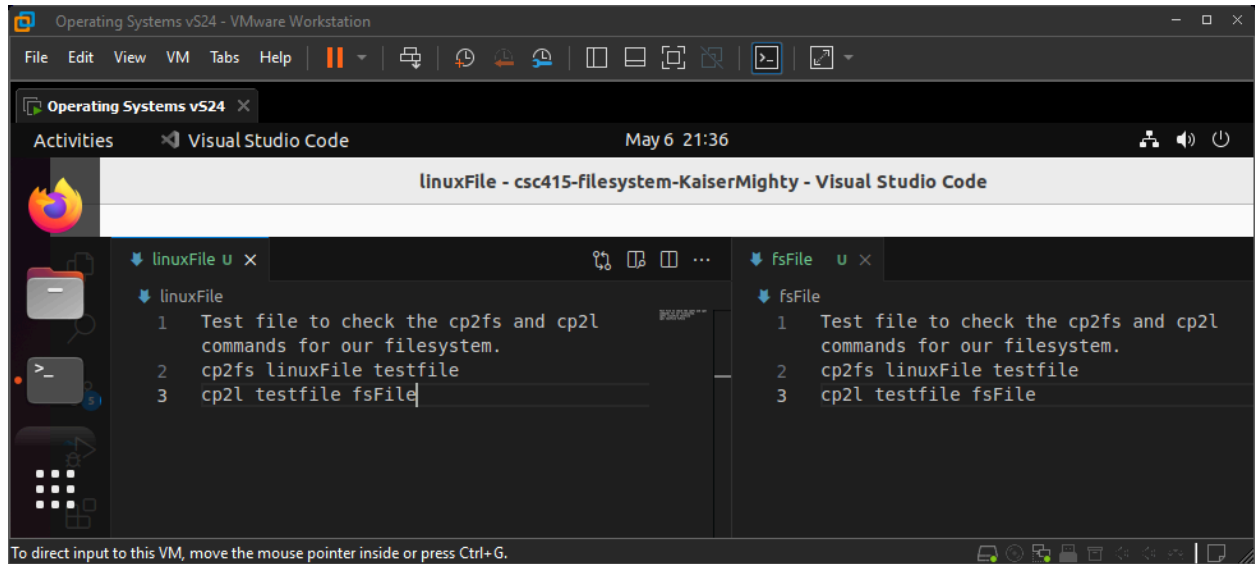


The screenshot shows a VMware Workstation window titled "Operating Systems vS24 - VMware Workstation". Inside, a terminal window is open with the title bar "Operating Systems vS24" and "Terminal". The terminal shows a user named "student" at a prompt "student@student: ~/Documents/csc415-filessystem-KaiserMighty\$". The user runs the command "make run". The program output includes: ".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", and "Initializing File System with 19531 blocks with a block size of 512". A table follows, listing commands and their status:

Command	Status
ls	ON
cd	ON
md	ON
pwd	ON
touch	ON
cat	ON
rm	ON
cp	ON
mv	ON
cp2fs	ON
cp2l	ON

After the table, the prompt "Prompt > ls" is shown, followed by the output "dir1" and "testFile". Then, the prompt "Prompt > rm dir1" is shown, followed by "Prompt > ls", which outputs "testFile". Finally, the prompt "Prompt > exit" is shown, and the terminal returns to the user prompt "student@student: ~/Documents/csc415-filessystem-KaiserMighty\$".

rm - remove from file system



linuxFile - file created in vscode

fsFile - file after cp2fs and cp2l