CSC 415
Operating System Principles

File System Project

Team

CurryOS Connoisseurs

https://github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132

Members

Arjun Singh Gill	github.com/ArjunS132	922170168
Vignesh Guruswami	github.com/elitewhale75	922379195
Mos Kullathon	github.com/mosguinz	921425216
Sid Padmanabhuni	github.com/SidPad03	921652677

Our Approach

FS Init is responsible for setting up everything needed for the file system to function properly. This all begins with checking if formatting is required. We will check the first block (not the block partition) for the signature. Since our signature is going to be a long int, we will only need to read the first 8 bytes of the block. Because these bytes are little endian, we will need to go byte by byte in order to convert the hex value to decimal. We can do this by separating each half of the byte with division and modulo by 16. After this as we move along each half byte we can shift the value by 4 and add it to our overall sum. Once we go through all 8 bytes we can now compare the value and see if we need to format the disk.

In the event that we need to format the disk, we will create the volume control block from scratch. The volume control block will contain the proper signature, the total number of blocks the file system will be allowed to work with, how large our blocks will be, where the first usable block will be located, the location of our free space system and the location of our root directory. Finally we will have to actually write it to disk and load the VCB into memory. The root directory itself will also have to be created and the free space system will have to be initialized.

In the event that we do not need to format the disk we will read it from the first block and load it into memory. We will also load the root directory and load it into the memory as well. Free space is already initialized so we simply have to load it into memory.

Our free space system will be using linked allocation along with a free space allocation table. Normal linked allocation would involve dedicating the first eight bytes to pointing to the location of the next block of the file/freespace. If we've reached the end of the file we can simply denote it with FF FF FF FF.

There are a few issues with this, however. For one, linked allocation is incredibly slow on its own. Since there is no random access when it comes to blocks in a file, in order to reach a block in the middle of the file you would have to access each one sequentially, as you need to read a block in order to find the next one in the chain. The other issue with this is our blocks will be handicapped as we are losing eight bytes in every single block, which can add up the larger our files grow.

The solution to the linked allocation issues comes from the FAT table. Here we will dedicate a certain amount of blocks to store a linked list, where each element in the linked list contains only one field: the location of the next block in the sequence. This way, only one read operation is needed, to read the table from disk and load it into memory. Then following the chain would simply go through the linked list until we reach the block that we need. The free space system itself will take up space in the beginning of the volume that linked allocation wouldn't have needed, but isolating our free space system from our files creates a level of abstraction that makes collaborative work easier to conduct, and separating functionality goes a long way towards making components in our file system more modular. Our files will no longer explode in size from all the extra blocks we need from the eight bytes we lose storing the location of the next block in the beginning of every block.

Before creating our directories we will need to solidify the directory system itself. Our directories will be arrays that contain directory entry structures. These directory entry structures contain many important fields. The first will be the location of the file the directory entry points to on disk. The next will be the size of the file in bytes. Then we will store whether the file is a directory or non-directory. Then we will store the name of the file, which will be unique and not shared by any other file or directory within the parent directory. Lastly we will contain some metadata such as the time the file was created, the time the file was last accessed, and the time the file was last modified. It is important to separate directory entries from the files they describe. Directory entries contain information that will help us logically navigate our file system and manipulate files. Directories and files are physically identical: they are both blobs that take up space. Directories are files that simply contain big arrays with directory entries.

Directories and directory entries will also need to follow certain conventions. The first two directory entries in every directory will contain two directory entries that are filled in by default. The first entry will be the dot entry, which simply contains all the information about the directory itself, not counting its own name. This is because the name of the directory will actually be stored in the parent, as the name of a directory is a logical necessity. This leads us to the parent directory, which is the second entry in the directory as the dot dot entry. The dot dot entry is crucial as the directory needs to be linked to another directory, otherwise it is taking up space but never accessible. Floating directories are a danger we want to avoid no matter what. Because they are no longer accessible it is impossible to free a floating directory with our free space system. This relationship between parent and child needs to be strong and established, so every directory will follow this convention.

We will also mark unused directory entries in every directory with a location of negative two, which is FF FF FF FE in hex (two's complement). This will be very easy to detect and initialize, as no -2 index exists in our FAT table. This will also make removing files/directories very simple, as we simply need to call the free space system's free functions and set the location of the directory entry back to -2 to mark it unused. The other fields can be filled with absolutely anything, which won't matter as long as the directory entry itself is easily identifiable as an unused directory entry.

Create Directory will serve a few functions. Its first use will likely be to create the root directory itself. One of the first things we will need to ascertain is the amount of bytes total that this new directory will occupy. This is important as prior to writing to disk, we will need to be working with this new directory in memory. This isn't as simple as multiplying the number of directory entries by the size of our actual directory entries. This is because we are accessing our volume as blocks. This means we may run into a scenario where the amount of directory entries requested by the user is not the maximum number of directory entries that can be filled by the block. Because we interface with our file system through blocks, we will run into scenarios where the space we allocate for memory differs from the actual space occupied by the directory. Therefore we will need to do a conversion from bytes, to blocks, and back to bytes. This way we can have directories whose size will fill up the blocks they need.

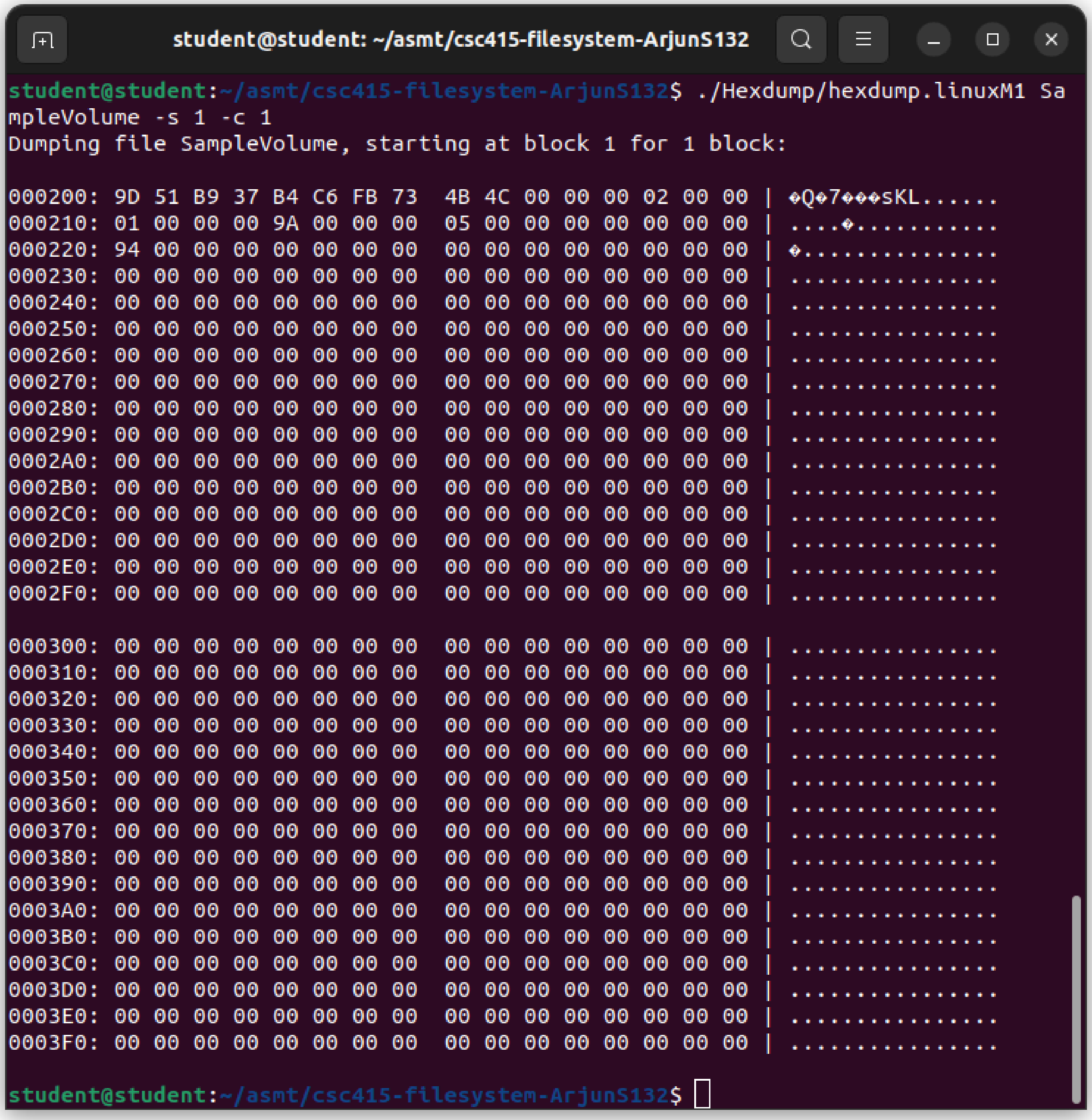
Next we must request blocks from the free space system according to the amount we computed prior. Because our directory entries will fit evenly within this requested space we can later cleanly write the array we create in memory to disk. The first order of business is to initialize the dot and dot dot entries. The dot entry will simply contain the location of the newly requested blocks and the size of the new directory. We also do need to specify that this is a directory. The parent's information can be copied considering we pass the parent directory entry as a parameter to the create directory routine. Next we will set every element in the array's location to -2. With this all entries barring the dot and dot dot's entry will be marked as unused. Finally we have to write our array back into disk.

We will however have to account for cases where we are creating the root directory. A lot of it is very similar to creating a normal directory. We will recognize root directory creation by the user passing in null as an argument to our create directory function. The root directory's distinction is that its dot and dot dot entries are the same barring the name.

File System Description

Volume Control Block (VCB)

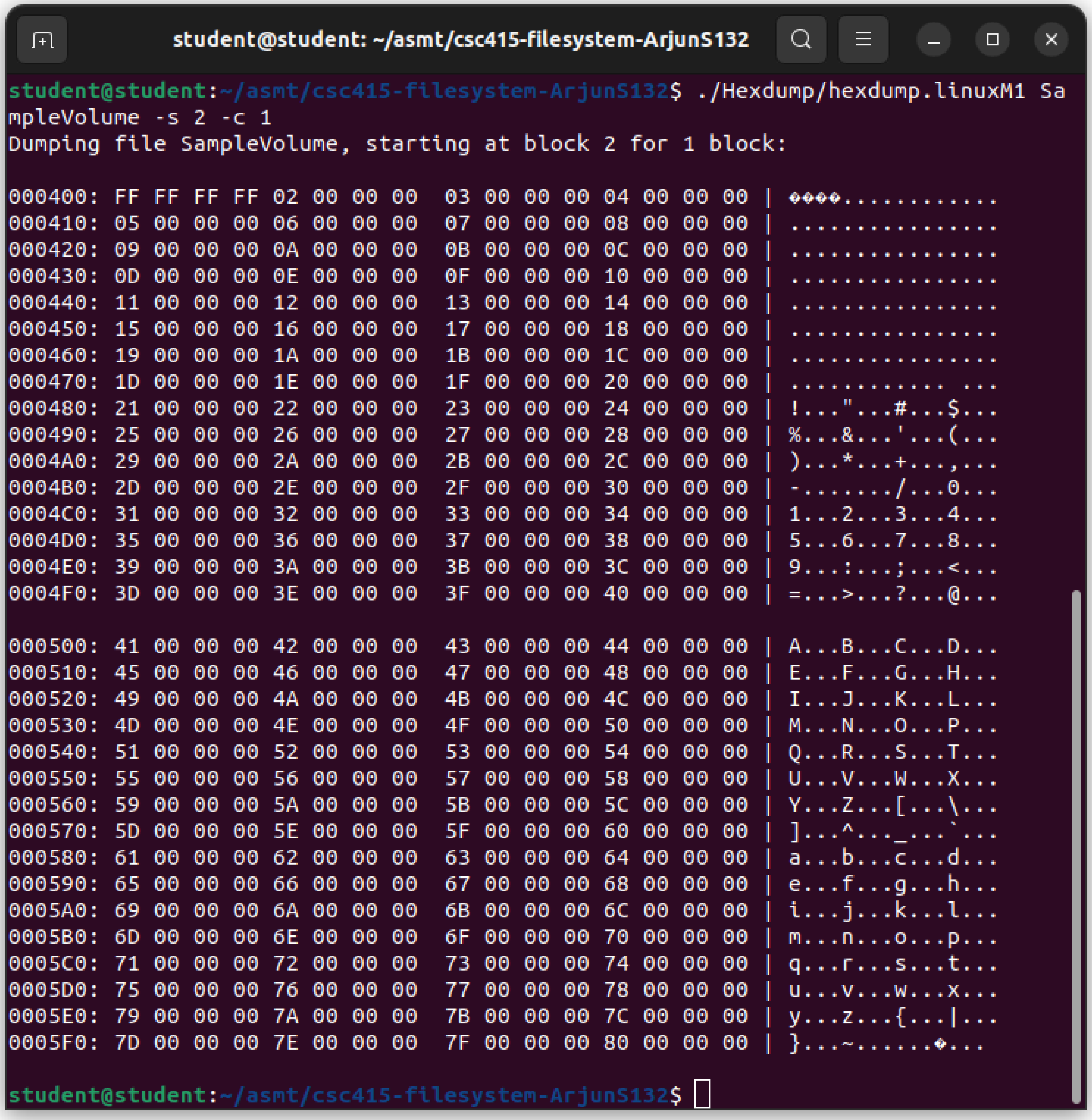
Below is the hex dump of the volume control block (VCB). In our file system, the VCB is stored on block one. Note that the zeroth block contains the partition header.



```
student@student:~/asmt/csc415-filesystem-ArjunS132$ ./Hexdump/hexdump.linuxM1 Sa
mpleVolume -s 1 -c 1
Dumping file SampleVolume, starting at block 1 for 1 block:

000200: 9D 51 B9 37 B4 C6 FB 73  4B 4C 00 00 00 02 00 00 | �Q�7���sKL......
000210: 01 00 00 00 9A 00 00 00  05 00 00 00 00 00 00 00 | ....�...........
000220: 94 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 | ................

student@student:~/asmt/csc415-filesystem-ArjunS132$
```

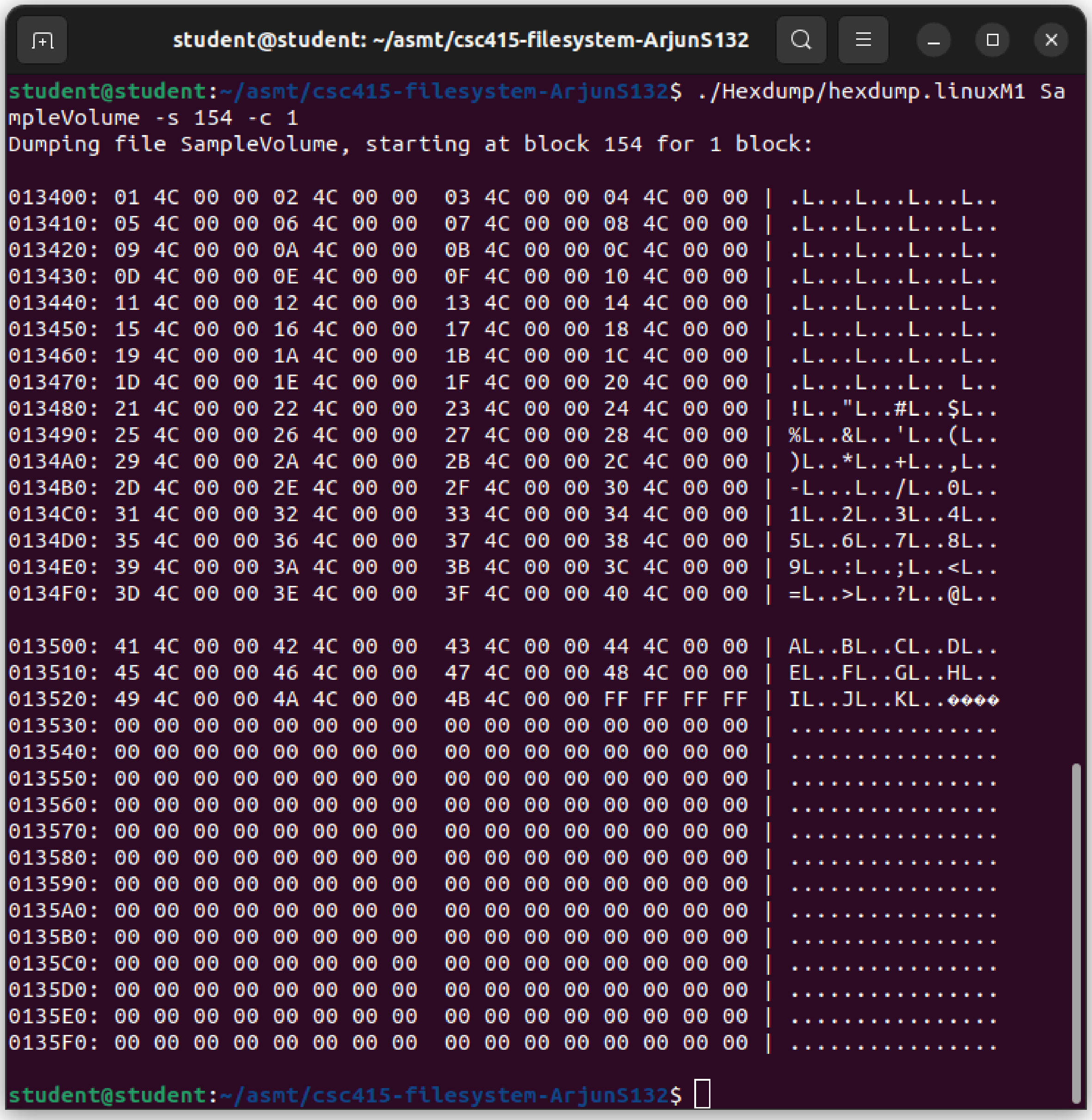
Free-space List

Below is the hex dump for the beginning of the free-space list. For our implementation, it starts at block two — after the VCB. Here, note that the zeroth index is marked with -1 (or `0xFFFFFFFF`) as it is occupied by the VCB. The following indices then point to the indices of the following blocks. For example, the first block points to the second, the second points to the third, and so on.



```
student@student:~/asmt/csc415-filesystem-ArjunS132$ ./Hexdump/hexdump.linuxM1 Sa
mpleVolume -s 2 -c 1
Dumping file SampleVolume, starting at block 2 for 1 block:

000400: FF FF FF FF 02 00 00 00  03 00 00 00 04 00 00 00 | ����............
000410: 05 00 00 00 06 00 00 00  07 00 00 00 08 00 00 00 | ................
000420: 09 00 00 00 0A 00 00 00  0B 00 00 00 0C 00 00 00 | ................
000430: 0D 00 00 00 0E 00 00 00  0F 00 00 00 10 00 00 00 | ................
000440: 11 00 00 00 12 00 00 00  13 00 00 00 14 00 00 00 | ................
000450: 15 00 00 00 16 00 00 00  17 00 00 00 18 00 00 00 | ................
000460: 19 00 00 00 1A 00 00 00  1B 00 00 00 1C 00 00 00 | ................
000470: 1D 00 00 00 1E 00 00 00  1F 00 00 00 20 00 00 00 | ............ ...
000480: 21 00 00 00 22 00 00 00  23 00 00 00 24 00 00 00 | !..."...#...$...
000490: 25 00 00 00 26 00 00 00  27 00 00 00 28 00 00 00 | %...&...'...(...
0004A0: 29 00 00 00 2A 00 00 00  2B 00 00 00 2C 00 00 00 | )...*...+...,...
0004B0: 2D 00 00 00 2E 00 00 00  2F 00 00 00 30 00 00 00 | -......./...0...
0004C0: 31 00 00 00 32 00 00 00  33 00 00 00 34 00 00 00 | 1...2...3...4...
0004D0: 35 00 00 00 36 00 00 00  37 00 00 00 38 00 00 00 | 5...6...7...8...
0004E0: 39 00 00 00 3A 00 00 00  3B 00 00 00 3C 00 00 00 | 9...:...;...<...
0004F0: 3D 00 00 00 3E 00 00 00  3F 00 00 00 40 00 00 00 | =...>...?...@...

000500: 41 00 00 00 42 00 00 00  43 00 00 00 44 00 00 00 | A...B...C...D...
000510: 45 00 00 00 46 00 00 00  47 00 00 00 48 00 00 00 | E...F...G...H...
000520: 49 00 00 00 4A 00 00 00  4B 00 00 00 4C 00 00 00 | I...J...K...L...
000530: 4D 00 00 00 4E 00 00 00  4F 00 00 00 50 00 00 00 | M...N...O...P...
000540: 51 00 00 00 52 00 00 00  53 00 00 00 54 00 00 00 | Q...R...S...T...
000550: 55 00 00 00 56 00 00 00  57 00 00 00 58 00 00 00 | U...V...W...X...
000560: 59 00 00 00 5A 00 00 00  5B 00 00 00 5C 00 00 00 | Y...Z...[...\...
000570: 5D 00 00 00 5E 00 00 00  5F 00 00 00 60 00 00 00 | ]...^..._...`...
000580: 61 00 00 00 62 00 00 00  63 00 00 00 64 00 00 00 | a...b...c...d...
000590: 65 00 00 00 66 00 00 00  67 00 00 00 68 00 00 00 | e...f...g...h...
0005A0: 69 00 00 00 6A 00 00 00  6B 00 00 00 6C 00 00 00 | i...j...k...l...
0005B0: 6D 00 00 00 6E 00 00 00  6F 00 00 00 70 00 00 00 | m...n...o...p...
0005C0: 71 00 00 00 72 00 00 00  73 00 00 00 74 00 00 00 | q...r...s...t...
0005D0: 75 00 00 00 76 00 00 00  77 00 00 00 78 00 00 00 | u...v...w...x...
0005E0: 79 00 00 00 7A 00 00 00  7B 00 00 00 7C 00 00 00 | y...z...{...|...
0005F0: 7D 00 00 00 7E 00 00 00  7F 00 00 00 80 00 00 00 | }...~.......�...

student@student:~/asmt/csc415-filesystem-ArjunS132$ 
```

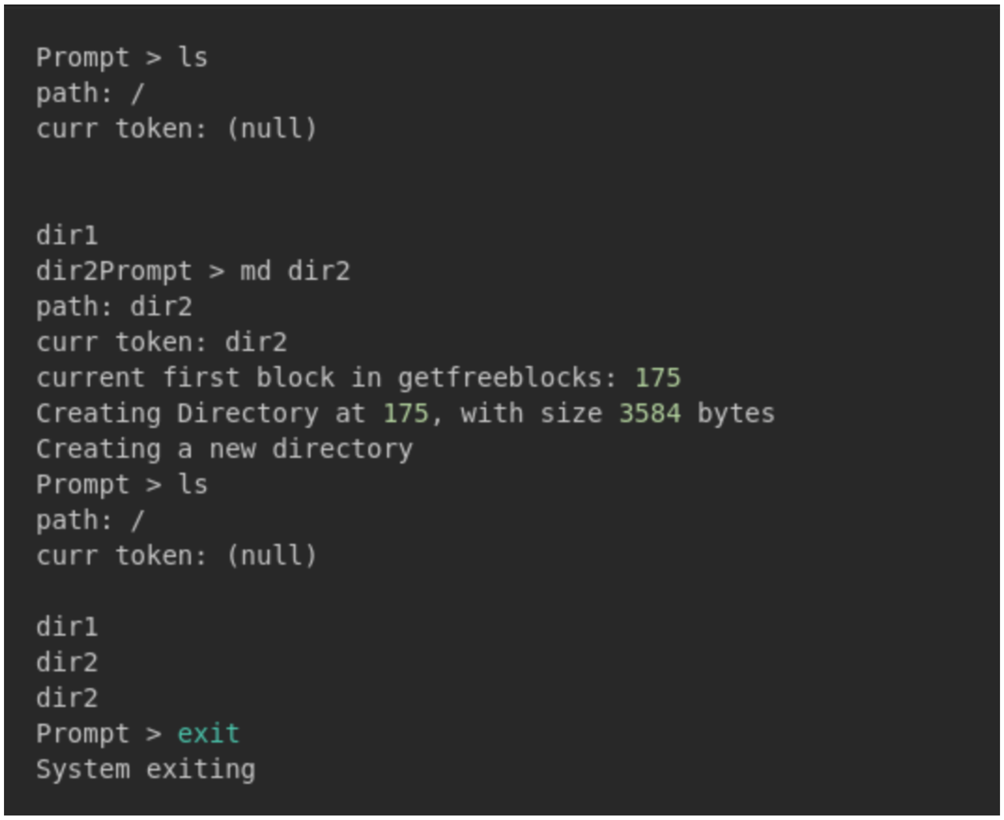
Here, at block 154, the value of the last index is marked with `0xFFFFFF` to indicate that this is the end of the free-space list. Note that the value of the index before it is `0x00004C4B`, which is 19531 in decimal, which is the size of our file system in blocks.


student@student: ~/asmt/csc415-filesystem-ArjunS132

```
student@student:~/asmt/csc415-filesystem-ArjunS132$ ./Hexdump/hexdump.linuxM1 Sa
mpleVolume -s 154 -c 1
Dumping file SampleVolume, starting at block 154 for 1 block:

013400: 01 4C 00 00 02 4C 00 00  03 4C 00 00 04 4C 00 00 | .L...L...L...L..
013410: 05 4C 00 00 06 4C 00 00  07 4C 00 00 08 4C 00 00 | .L...L...L...L..
013420: 09 4C 00 00 0A 4C 00 00  0B 4C 00 00 0C 4C 00 00 | .L...L...L...L..
013430: 0D 4C 00 00 0E 4C 00 00  0F 4C 00 00 10 4C 00 00 | .L...L...L...L..
013440: 11 4C 00 00 12 4C 00 00  13 4C 00 00 14 4C 00 00 | .L...L...L...L..
013450: 15 4C 00 00 16 4C 00 00  17 4C 00 00 18 4C 00 00 | .L...L...L...L..
013460: 19 4C 00 00 1A 4C 00 00  1B 4C 00 00 1C 4C 00 00 | .L...L...L...L..
013470: 1D 4C 00 00 1E 4C 00 00  1F 4C 00 00 20 4C 00 00 | .L...L...L.. L..
013480: 21 4C 00 00 22 4C 00 00  23 4C 00 00 24 4C 00 00 | !L.."L..#L..$L..
013490: 25 4C 00 00 26 4C 00 00  27 4C 00 00 28 4C 00 00 | %L..&L..'L..(L..
0134A0: 29 4C 00 00 2A 4C 00 00  2B 4C 00 00 2C 4C 00 00 | )L..*L..+L..,L..
0134B0: 2D 4C 00 00 2E 4C 00 00  2F 4C 00 00 30 4C 00 00 | -L...L../L..0L..
0134C0: 31 4C 00 00 32 4C 00 00  33 4C 00 00 34 4C 00 00 | 1L..2L..3L..4L..
0134D0: 35 4C 00 00 36 4C 00 00  37 4C 00 00 38 4C 00 00 | 5L..6L..7L..8L..
0134E0: 39 4C 00 00 3A 4C 00 00  3B 4C 00 00 3C 4C 00 00 | 9L..:L..;L..<L..
0134F0: 3D 4C 00 00 3E 4C 00 00  3F 4C 00 00 40 4C 00 00 | =L..>L..?L..@L..

013500: 41 4C 00 00 42 4C 00 00  43 4C 00 00 44 4C 00 00 | AL..BL..CL..DL..
013510: 45 4C 00 00 46 4C 00 00  47 4C 00 00 48 4C 00 00 | EL..FL..GL..HL..
013520: 49 4C 00 00 4A 4C 00 00  4B 4C 00 00 FF FF FF FF | IL..JL..KL..����
013530: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013540: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013550: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013560: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013570: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013580: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
013590: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................
0135F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 | ................

student@student:~/asmt/csc415-filesystem-ArjunS132$
```

Issues

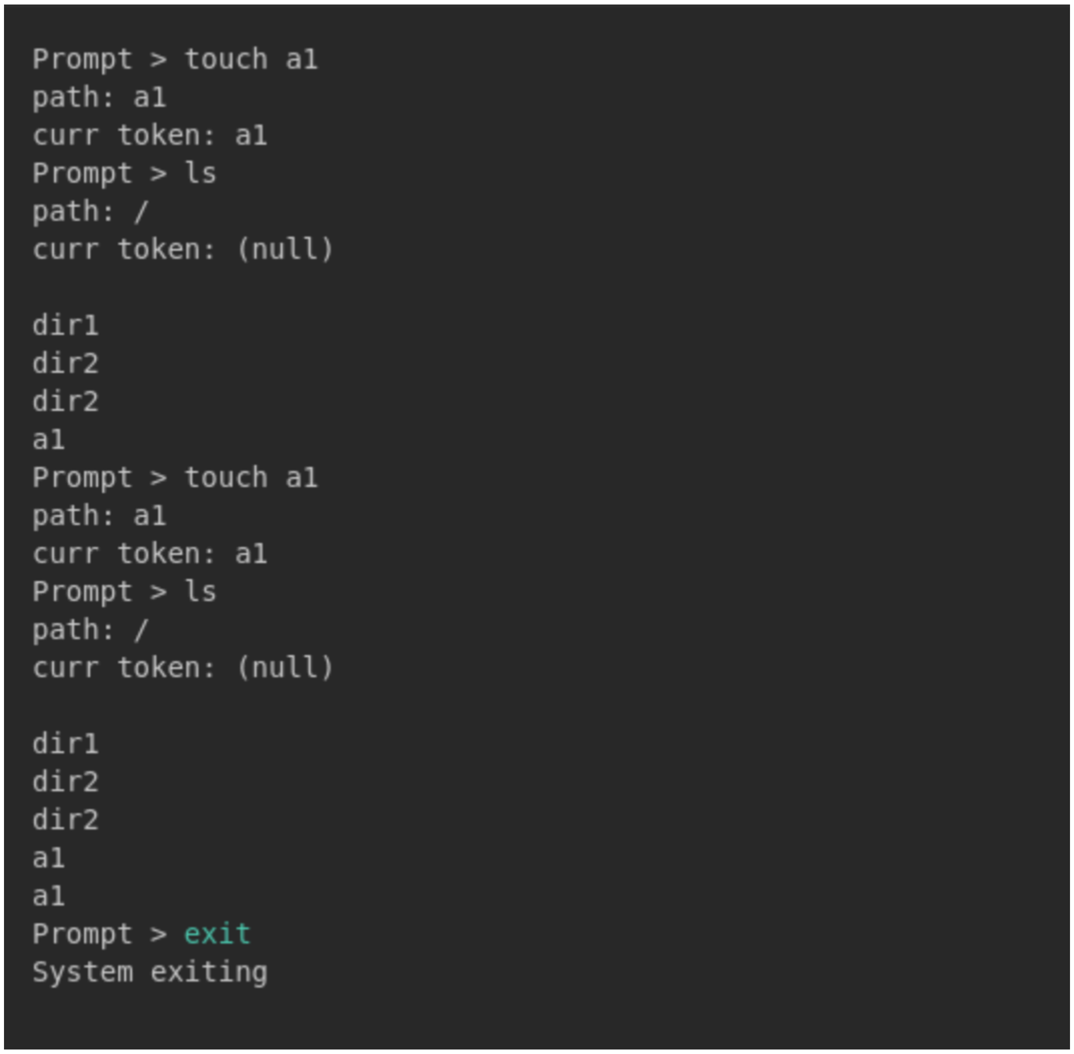
`md` command lets you create multiple directories with the same name at the same location.



```
Prompt > ls
path: /
curr token: (null)


dir1
dir2Prompt > md dir2
path: dir2
curr token: dir2
current first block in getfreeblocks: 175
Creating Directory at 175, with size 3584 bytes
Creating a new directory
Prompt > ls
path: /
curr token: (null)

dir1
dir2
dir2
Prompt > exit
System exiting
```

Initially overlooked, this was simply fixed by checking if the given directory name already exists in the current parent folder. Previously, it would allocate space to the folder.

`touch` command allows for the creation of multiple files with the same name at the same location.



```
Prompt > touch a1
path: a1
curr token: a1
Prompt > ls
path: /
curr token: (null)

dir1
dir2
dir2
a1
Prompt > touch a1
path: a1
curr token: a1
Prompt > ls
path: /
curr token: (null)

dir1
dir2
dir2
a1
a1
Prompt > exit
System exiting
```

`rm` command does not work as expected, and a file/directory can't be removed from the volume

This was because the `isDirectory` variable for the VCB was not correctly set (was flipped the other way around). This caused the handling for removing a file using `rm` to not function correctly.

Reading the block byte by byte to check for the signature fails.

We misunderstood how the VCB worked initially. Once we loaded in the VCB using LBARead, we were able to correctly verify the signature and format the volume if needed.

We end up overwriting the VCB when trying to create a new directory after the root directory.

The primary issue here was that the freespace map was not being marked correctly when in use.

When calling using `..` in root, `setcwd` would have the incorrect `cwd` pathname

A helper function was needed in order to “clean” the path when in root in order to handle dot and dot-dot

`ls` would not list the entirety of the directory after files are moved or removed

Fixed by iterating through the entire directory through maximum DE count instead of just checking if the entry’s location was valid.

`parsePath` would segfault when the last element in the path is a file

An additional check was needed to check if every single element in the path was the file before loading the path.

`b_write` did not write entirety of the last block

Fixed by calling `fileWrite` when part one of the block is populated.

Functions

Command Line Functions

ls

The `ls` function is responsible for listing the files and directories within the current working directory. We start by initializing the variable and structures to handle the command-line options (which are ‘long’, ‘all’, and ‘help’) using `getopt_long`. The function enters a loop to parse these options, and sets the flags (‘fllong’ and ‘flall’) based on whether the ‘long’ or ‘all’ options are given. If the help command is detected or an unknown command is entered, it prints a usage message and exits. If there are non-option arguments, it treats them as file or directory paths and lists their contents, indicating if the file is found or not. If no paths are provided, it defaults to listing just the contents of the current working directory.

cp

The `cp` command copies one file to another directory in the file system.

- If two arguments are given, the first is the source, and the second is the destination

- If one argument is given, then the source is set to the destination, essentially overwriting the file in the directory with the same file. This prevents the file from being removed or lost.

The function opens the source file in read-only mode and the destination file in write-only mode, creating the file if it already does not exist. It then enters a loop, where it reads chunks of data from the source file and writes them to the destination file until the entire content of the source file is copied. Each read chunk is written before the next read chunk can be read, continuing until a read operation reads less bytes than what the buffer can hold. This indicates the end of the file in the source file. Both the source and destination files are finally closed.

mv

The `mv` command moves a file from one directory to another. It takes two arguments, the source and destination paths. When the correct number of arguments are given, `fs_mv` handles moving the source file to the destination path.

fs_mv

This function moves a directory within the file system and handles the change of file location and name adjustments. It first parses the source and destination paths using parsePath, and populates the PPRETDATA structure with information about the parent directory and the last element of the path. If issues such as invalid path or if the destination is not an actual directory, the function exits and returns an error. The destination directory is searched for a vacant space to put the source file in. If it is found, the function copies the directory entry from the source to the vacant space.

We also made sure to handle scenarios where the source path is a path to a directory rather than a directory entry. The original directory at the source path is set to vacant, once moved over to the destination path. The function also makes sure to write the updated directory structures back to the file system and frees memory that was allocated.

md

The `md` command creates a directory in the current working directory. The function takes one argument, which is the path to the new directory that is to be created. Given a correct argument, the function calls `fs_mkdir` with default permissions, which handles the creation of the directory and linking it back to the parent.

fs_mkdir

This function is responsible for creating a new directory. We first allocate memory to hold directory information and the new directory entry. Then we can parse the path to extract details such as the parent directory and the name of the new directory. The function checks for available space in the parent directory to accommodate the new directory entry. After a directory is created, a new directory is initialized to zero. This new directory entry is linked to the parent directory's entries array. Then the parent directory is written back to the file system (this includes the addition of the new directory). We also made sure to handle errors that may arise during parse path, finding vacant space, and during read/write.

rm

The `rm` command removes a directory or files, based on the provided path. The function accepts one argument, the path of the directory/directory entry to be removed. The function determines whether the path is a directory or file. Depending on the type, it calls `fs_rmdir` to remove a directory or `fs_delete` to remove a file. If the path does not correspond to a specific file or directory, an error is thrown.

fs_rmdir

This function is responsible for removing a directory from the file system. We first allocate memory for path parsing related data, followed by parsing the path name to identify and validate the target directory entry. This structure can then be loaded and checked if it is empty, since directories must be empty before they can be removed. If the directory is not empty, then the operation is aborted and returns an error. The directory entry is marked as deleted by setting its location to an invalid value (-2). The function writes these changes back to the file system, updating the parent directory's content to reflect the changes that were made.

fs_delete

This function is responsible for the removal of a file within the file system. We first allocate memory to store path parsing related data in order to locate the file within the file system. If path parsing fails and it can't be found, then the operation is aborted and returns an error. Once we locate the file, we check if it is of a non-zero size. If it is, then we release allocated storage blocks back to the file system using `returnFreeBlocks`. Once they are released, or the file size is zero, the function marks the directory entry as deleted by setting its location field to an invalid state (-2). These changes can be written back to the file system, to ensure that the removal of the file reflects on the file system structure.

touch

The `touch` command is responsible for creating a file within the current directory. The responsible function first checks if the correct number of arguments were provided (a file name). After parsing the filename, the function opens/creates the file with write-only access. If this fails, then it returns an error. Then the file is opened, and then closed so that the timestamp of the file can be updated.

cat

The cat command is responsible for displaying the contents of a file. It's important to note that our implementation of cat only handles one file at a time. The responsible function takes one argument (the name of the file). The file is then opened in read-only mode. If this doesn't work, an error is thrown to the user followed by an error code from the file operation. Once the file is opened, the function reads the file in chunks into a buffer, subsequently printed to standard output. This loop continues until a read operation reads less than the buffer size, which indicates EOF.

cp2l

The cp2l command is responsible for copying a file from our file system to the regular linux filesystem. It accepts either two command-line arguments:

- The source file path in the custom file system
- The destination file path in the Linux file system

Similar to cp if only one command line argument is provided, the function interprets it as being both the source and destination path. This lets us overwrite the file in the same location or copy a file within the same directory.

We can first open the source file in read-only mode using `b_open` and the destination file in write-only mode with creation and truncate options enabled, using the standard Linux `open` call. It enters a loop to read data from the source file to a buffer, then write this directly to the destination file in Linux. This loop iterates until all of the content is copied. This process repeats until a read operation fetches fewer bytes than the buffer's capacity (EOF has been reached). Then we close both the source file and the destination files using `b_close`.

cp2fs

The `cp2fs` command enables a user to copy files from the Linux filesystem to our file system. This command essentially performs the opposite functionality of the `cp2l` command. The only differences here are that the function opens the source file for reading in Linux and the destination source file in our file system.

cd

The `cd` command lets the user switch between directories. The function requires only one argument, which is the path to switch to. If the required arguments are not provided, it outputs a usage message and exits with an error. The function handles paths enclosed in double quotes, by stripping these quotes if necessary. The function then changes the current working directory to the specified path, using `fs_setcwd`.

fs_setcwd

This function first allocates memory to store the information returned by the parse path function. If the path is at the root (indicated by '-2', it loads the root directory and sets the current working directory path to the root. If the path is not at the root, and parsing results in an error, then we return -1. For a valid path, we dynamically allocate memory for a directory entry and load this from the file system based on the parse path data. If the loaded directory is not a valid directory, our function is designed to throw an error.

Once the valid directory entry is loaded, the function updates `cwd` to reflect the newly loaded directory, and adjusts the `cw PathName` to the full path, formatting it using `cleanPath`. Before returning, we make sure to write the changes back to disk to make sure that the file system is up to date.

pwd

The `pwd` command is responsible for retrieving and displaying the current working directory to the user. The function first allocates memory for a buffer that is large enough to hold the maximum directory path length. It calls the function `fs_getcwd`, which can retrieve the path into a buffer. If it returns null, then we return an error message to the user. Otherwise, it prints the current working directory path stored in a pointer. It returns after freeing the allocated memory.

fs_getcwd

This function just copies the current working directory path which is stored in `cwdPathName` into the `pathname` buffer.

Helper Functions

NMOverM

This function calculates how many blocks are needed to store a given amount of data when the data does not perfectly fit into blocks.

findInDir

The function searches for a directory entry within a given directory `searchDirectory` that matches a specified `name`. It iterates over each directory entry and if it finds a matching name that is not marked as deleted (when location !- -2), it returns the index of that entry. If there is no match found, then it returns -1.

findVacantSpace

This function looks for an empty space in a directory to add a new directory entry. It iterates through the directory entries and checks two conditions:

1. If there's an entry with the same name (dup)
2. If an entry is marked as vacant (location == -2)

If a vacant entry is found, it returns an index, otherwise, it prints that the directory is full and returns -1.

loadDir

The `loadDir` function is responsible for loading a directory from the file system into memory based on a specified index within a search directory array. The function begins by calculating the number of blocks needed to store the directory entries, using the `NMOverM` function. It then retrieves the location of the directory from the search directory using the provided index. Memory for the directory entries is allocated dynamically, and the `fileRead` function is called to load the directory data from the disk into this memory. If the read operation fails -1 is returned. The function frees the allocated memory and returns null to indicate an error. Otherwise, it returns a pointer to the loaded directory entries.

fs_isDir

This function loads a directory entry from a specified location in the filesystem. It calculates the size of the directory entry in blocks, retrieves the location from the search directory at the given index, and then dynamically allocates memory for these directory entries. The directory content is read from the disk into the allocated space using `fileRead`. It frees the allocated memory and returns null. Otherwise, it returns a pointer to the loaded directory entries.

fs_isFile

The `fs_IsFile` function is needed for determining whether a given filename refers to a file within the current working directory. This function begins by utilizing the `findInDir` function to search for a directory entry matching the specified filename within `cwd`. If `findInDir` returns an index value of -1, this indicates that no such entry was found. If an entry is found, the function then checks the `isDirectory` attribute of the located directory entry. If this is set to 0, it means that the entry is not a directory, and the entry is a file. Similarly, if `isDirectory` is set to 1, this means that the entry is a directory and not a file, and the function returns 0.

cleanPath

The `cleanPath` function makes it more consistent and easier to interpret by removing redundant and unnecessary elements such as "." (current directory) and ".." (parent directory). This function first tokenizes the input path using the delimiter "/", and stores each token (directory name) in an array called `pathTable`. As it processes each token, it uses a second array indices to keep track of the indices of valid directory names, skipping over "." and properly handling ".." by stepping back in the path structure unless at the root. After processing all tokens, it constructs a new, cleaned path by concatenating valid directory names from `pathTable` based on the indices stored in `indices`, making sure that each is correctly separated by a "/" delimiter.

fs_readdir

The `fs_readdir` function reads directory entries sequentially from a specified directory stream represented by the `fdDir` pointer. This function calculates the number of blocks needed to store all directory entries using `NMOverM`, based on the size of the directory entry and the block size from `volumeControlBlock`. Then the function allocates memory and attempts to read the directory's contents from disk into this memory. If successful, the function iterates through the entries, skipping any marked as deleted. For each valid entry encountered, it updates the `fs_diriteminfo` structure (`di`) in dirp* with details such as the entry's record length, file type, and name, before incrementing the directory's index to the next entry. If no more entries are found or if the initial read fails, it frees the allocated memory and returns null (the end of the directory contents or an error).

fs_stat

The `fs_stat` function retrieves comprehensive file or directory statistics for a specified `pathname` and populates these details into a buffer. Initially, it allocates memory for path processing and employs the `parsePath` function to resolve the given path into its constituent elements. Upon successful resolution, it uses `findInDir` to identify the directory entry for the last element in the path within the parsed parent directory. If the directory entry is successfully located, the function then extracts and assigns various attributes such as file size, block size, number of blocks (calculated based on file size and block size), and access, modification, and creation times from the directory entry to the buffer structure. It then returns this structure.

fs_closedir

The `fs_closedir` function closes a directory stream that has been previously opened and managed via a `fdDir` pointer. The function starts by checking if the dirp* is null, which would indicate that no valid directory stream was provided. In our case, it logs an error message and returns 0 to signal an unsuccessful closure attempt. If `dirp*` is not null, the function proceeds to free the memory allocated for the directory stream, effectively closing it and releasing system resources associated with it. It then returns 1 to indicate that the directory has been successfully closed.

fs_opendir

The `fs_opendir` function opens a directory specified by `pathname` for reading, setting up a directory stream (`fdDir`). Initially, it allocates memory for parsing the path and attempts to resolve `pathname` into directory components using `parsePath`. If the path resolution fails (`parsePath` returning -1), the function immediately frees the allocated resources and returns null, to show that the directory could not be opened. Upon successful resolution, it uses `findInDir` to locate the directory entry corresponding to the last element of the path. If the entry is found and confirmed to be a directory (not a file), the function then allocates and initializes a new `fdDir` structure, setting fields such as record length (`d_reclen`), directory entry position, and location based on the directory's attributes and the computed block size. It also initializes `fs_diriteminfo` within `fdDir` to store individual directory item information. If any condition fails, it prints an error and returns null.

isEmpty

The function iterates through the directory entries starting from the third one (ignoring "." and ".."). During each iteration, it checks if the `location` attribute of the directory entry is greater than 0, indicating that the entry is actively used and thus corresponds to either a file or a subdirectory. If any entry meets this condition, the function immediately returns 0, signifying that the directory is not empty. If the loop completes without finding any active entries, it returns 1, indicating that the directory is indeed empty apart from the standard "." and ".." entries.

parsePath

The `parsePath` function in the provided C code meticulously resolves a given filesystem path into its constituent directory entries, and updates the provided PPREDATA structure with details about the final directory entry in the path. This function starts by checking the validity of its inputs, returning -1 if either `pathName` or `ppinfo` is null. Depending on whether the path is absolute (starting with '/') or relative, it loads the root or the current working directory respectively using the `loadDir` function.

The function duplicates the input path to avoid modifying the original, then uses strtok_r to tokenize the path by the '/' delimiter, processing one directory level at a time. If the first token is null, it directly updates `ppinfo` based on whether the path was absolute or relative, then cleans up and returns. Otherwise, it iteratively processes each token (directory name) in the path by finding the directory entry in the current directory matching the token using `findInDir`. If the entry is a directory, it loads this directory to move deeper into the path hierarchy. If a token cannot be matched to a directory entry, or if it matches a non-directory when further tokens are expected (indicating a broken path), the function aborts and returns -1.

Once the end of the path is reached, the function updates `ppinfo` with the details of the last successfully processed directory, including its index and name, and performs cleanup as needed.

createDirectory

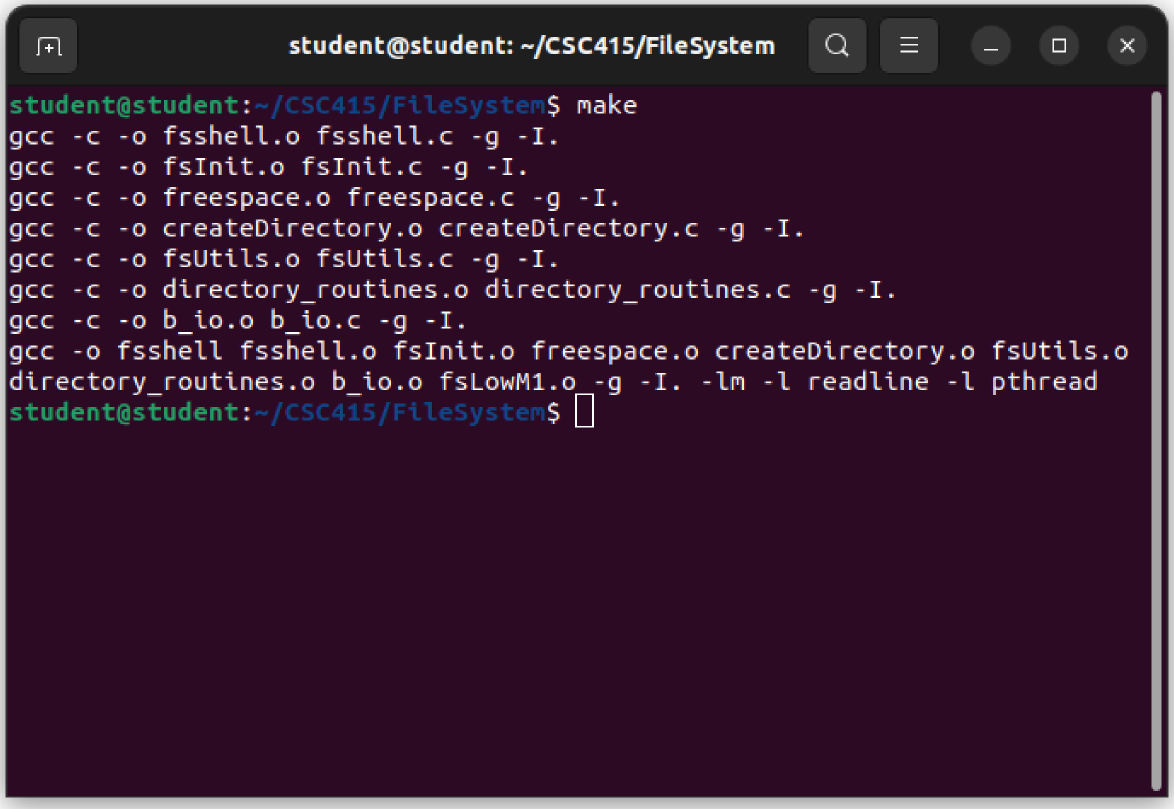
The createDirectory function is used to create a new directory with a specified number of entries. It starts by calculating the storage requirements based on the number of directory entries `numberOfEntries`, using the size of each directory entry to compute the total bytes needed. It then determines how many blocks are necessary to accommodate these bytes, accounting for the minimum block size `MINBLOCKSIZE`, and calculates the maximum number of entries that can fit within these blocks.

Memory for the directory is dynamically allocated, initializing all bytes to zero to denote an unused state. Each directory entry within this buffer is then marked as unused by setting its location to -2. The function requests the necessary number of blocks from the free space management system of the filesystem, storing the location of the first allocated block in blocksRequested.

The first two entries of the directory are specially initialized to represent the current directory (".") and its parent (".."). The first entry (".") points to the directory itself, storing its size and marking it as a directory. The second entry ("..") is configured based on the provided parent directory—if parent is NULL, indicating the creation of a root directory, it duplicates the settings from the first entry. Otherwise, it copies the attributes from the parent. Both entries are timestamped with the current time.

The newly created directory, now fully initialized, is written to disk using the fileWrite function. The memory allocated for the directory buffer is then freed, and the function returns the location of the first block of the new directory. This return value can be used to reference the directory in subsequent operations.

Screenshot of Compilation

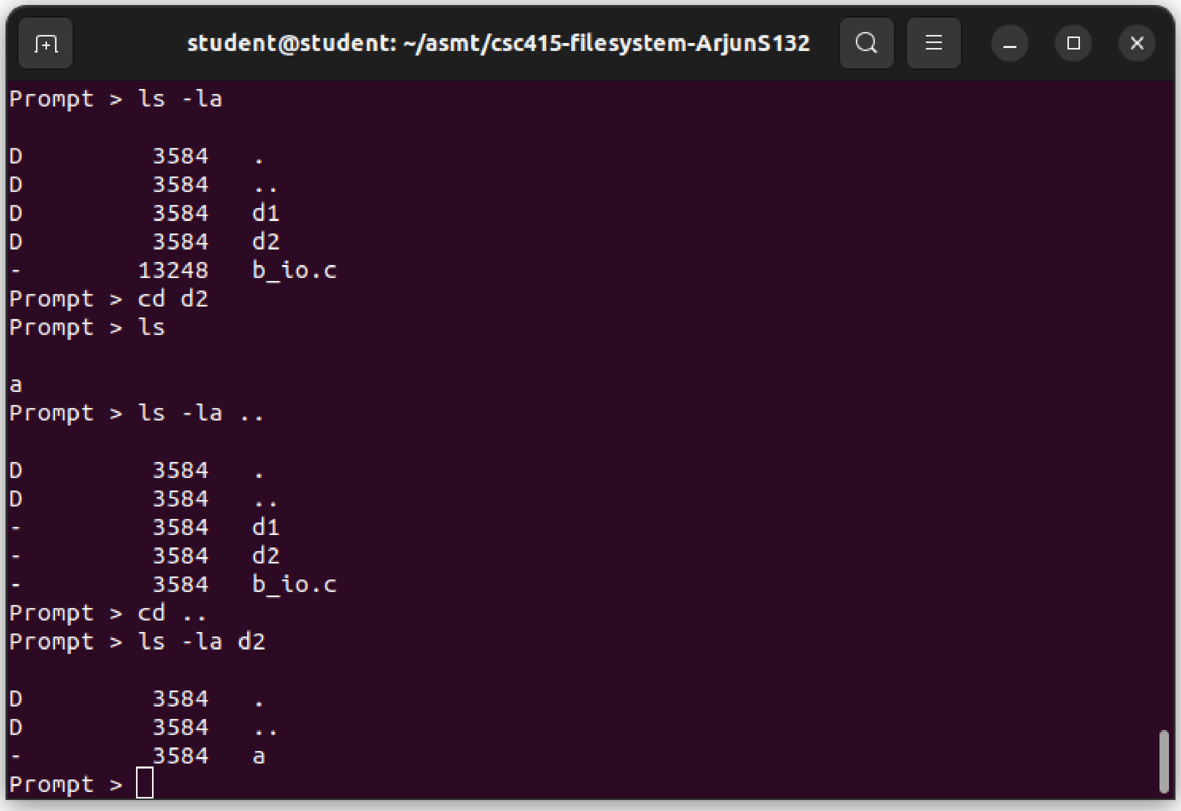


```
student@student:~/CSC415/FileSystem$ make
gcc -c -o fsshell.o fsshell.c -g -I.
gcc -c -o fsInit.o fsInit.c -g -I.
gcc -c -o freespace.o freespace.c -g -I.
gcc -c -o createDirectory.o createDirectory.c -g -I.
gcc -c -o fsUtils.o fsUtils.c -g -I.
gcc -c -o directory_routines.o directory_routines.c -g -I.
gcc -c -o b_io.o b_io.c -g -I.
gcc -o fsshell fsshell.o fsInit.o freespace.o createDirectory.o fsUtils.o
directory_routines.o b_io.o fsLowM1.o -g -I. -lm -l readline -l pthread
student@student:~/CSC415/FileSystem$
```

Screenshots of Command Execution(s)

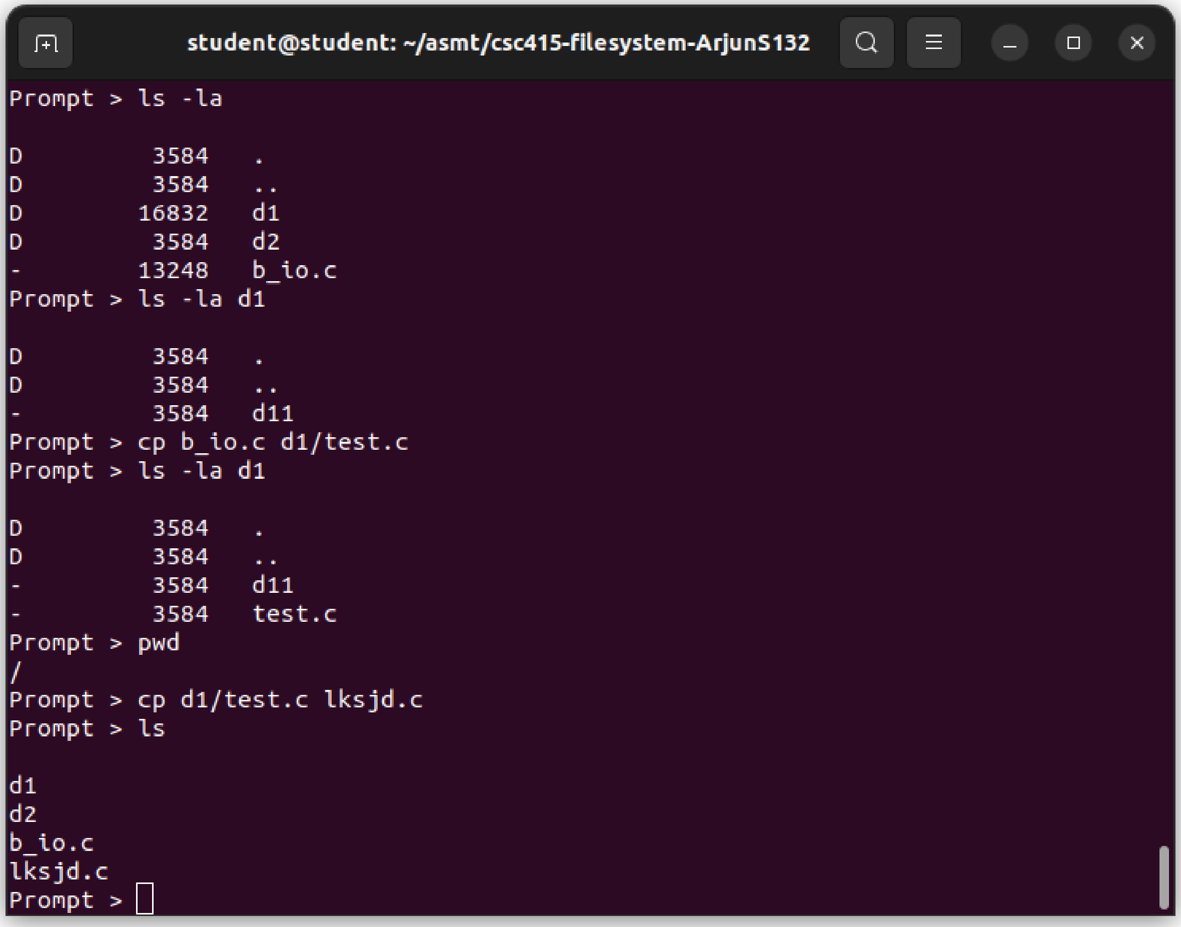
Below is a sample for the execution of each command in the shell.

ls



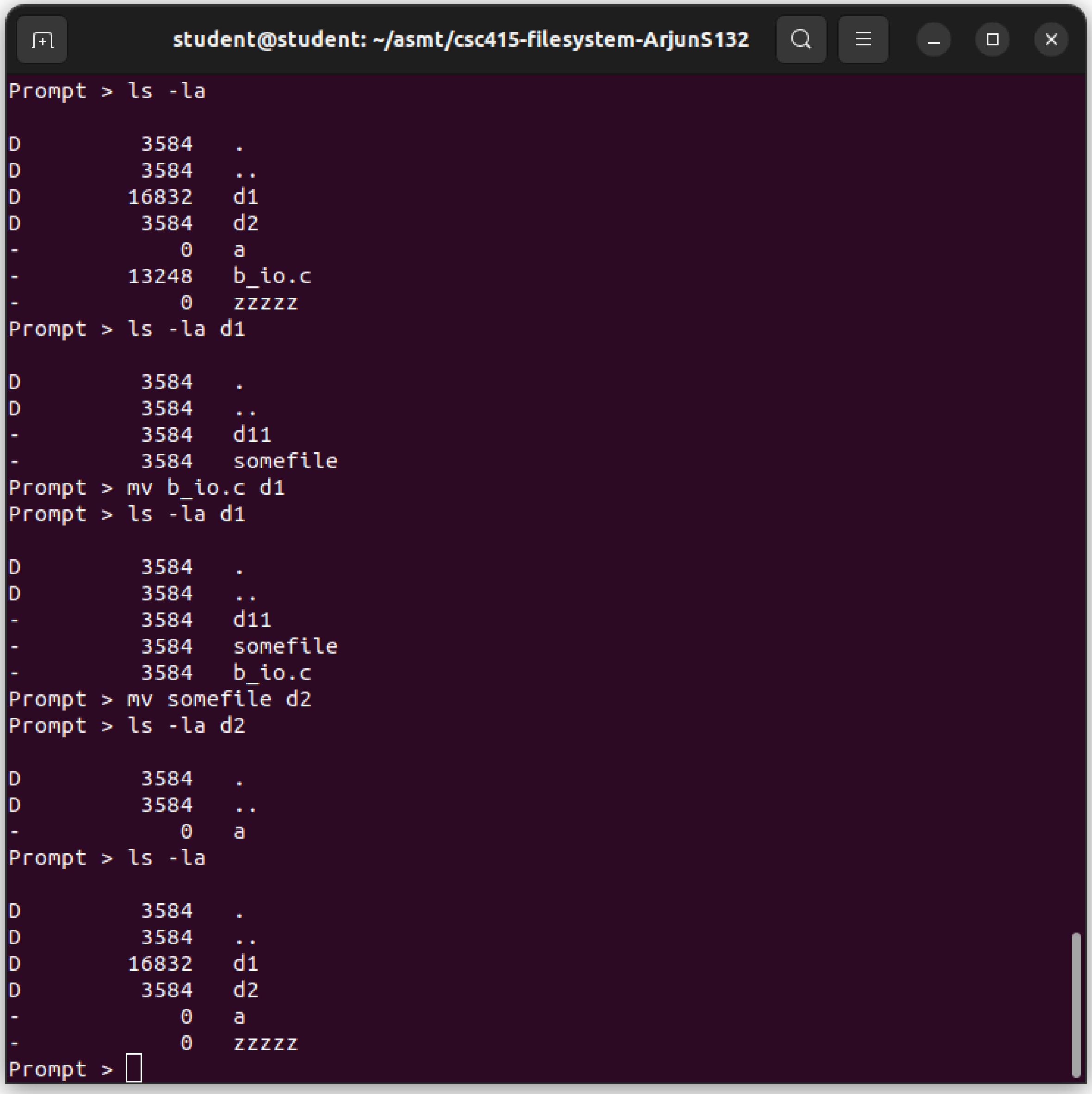
```
Prompt > ls -la

D          3584   .
D          3584   ..
D          3584   d1
D          3584   d2
-         13248   b_io.c
Prompt > cd d2
Prompt > ls

a
Prompt > ls -la ..

D          3584   .
D          3584   ..
-          3584   d1
-          3584   d2
-          3584   b_io.c
Prompt > cd ..
Prompt > ls -la d2

D          3584   .
D          3584   ..
-          3584   a
Prompt >
```

cp



```
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-         13248   b_io.c
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
Prompt > cp b_io.c d1/test.c
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
-          3584   test.c
Prompt > pwd
/
Prompt > cp d1/test.c lksjd.c
Prompt > ls

d1
d2
b_io.c
lksjd.c
Prompt > 
```

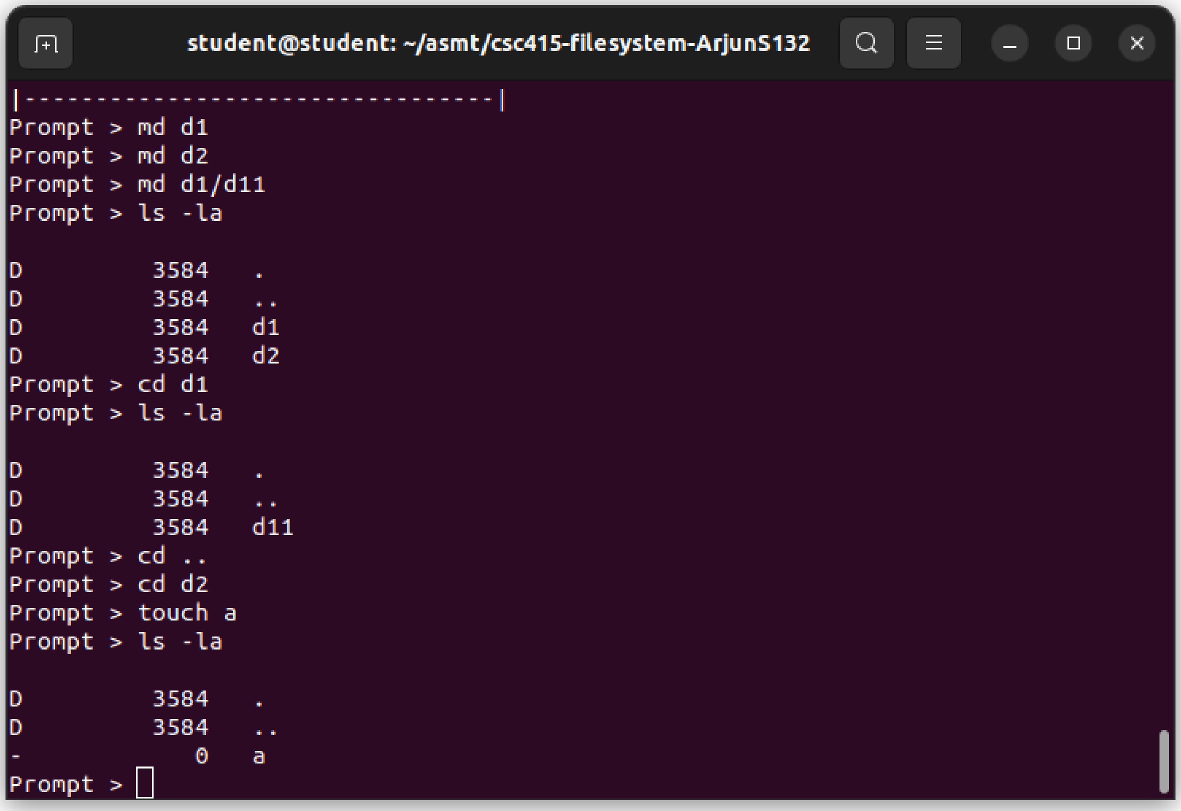
mv



student@student: ~/asmt/csc415-filesystem-ArjunS132

```
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   b_io.c
-             0   zzzzz
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
-          3584   somefile
Prompt > mv b_io.c d1
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
-          3584   somefile
-          3584   b_io.c
Prompt > mv somefile d2
Prompt > ls -la d2

D          3584   .
D          3584   ..
-             0   a
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-             0   zzzzz
Prompt > 
```

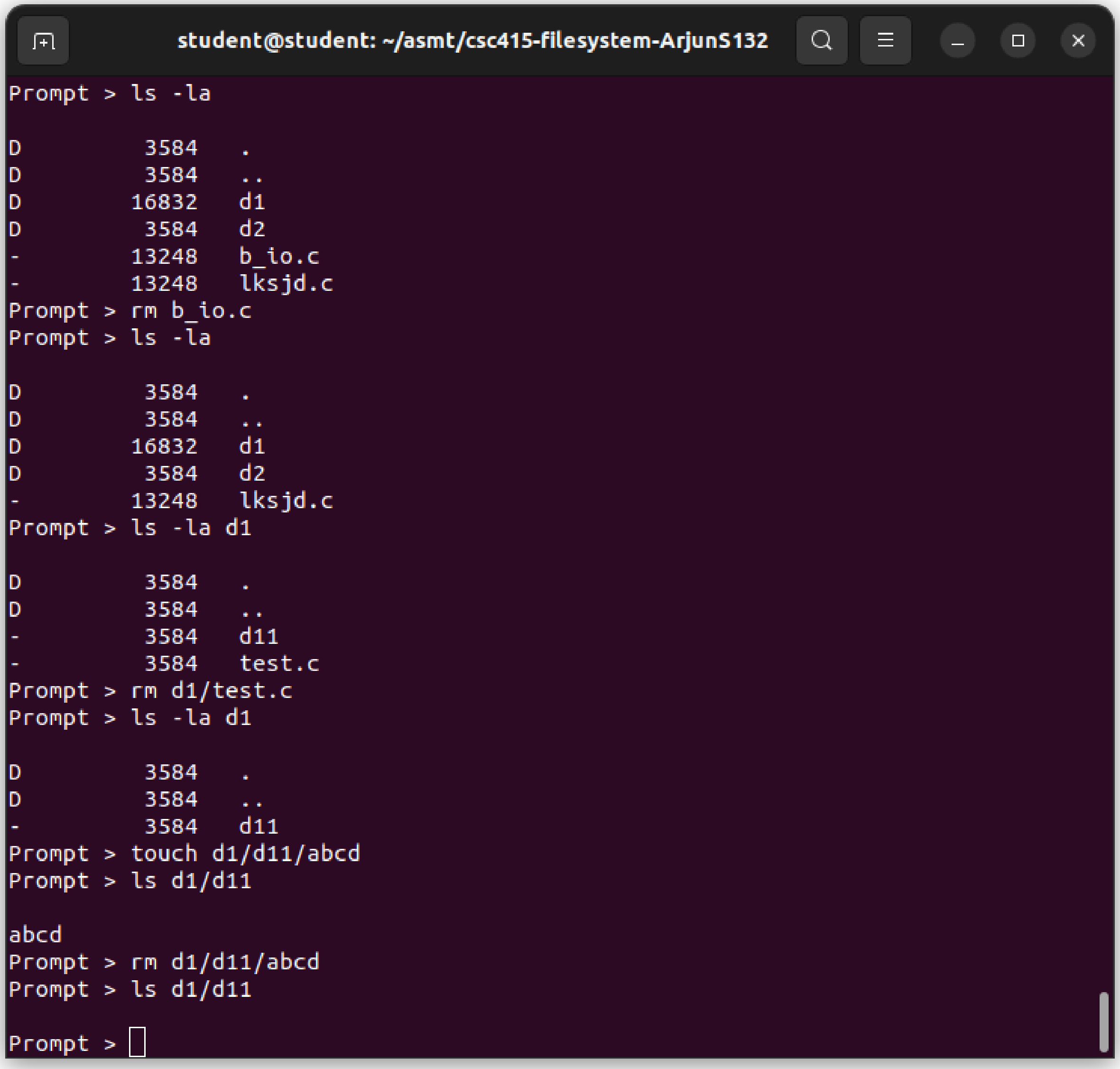
md



student@student: ~/asmt/csc415-filesystem-ArjunS132

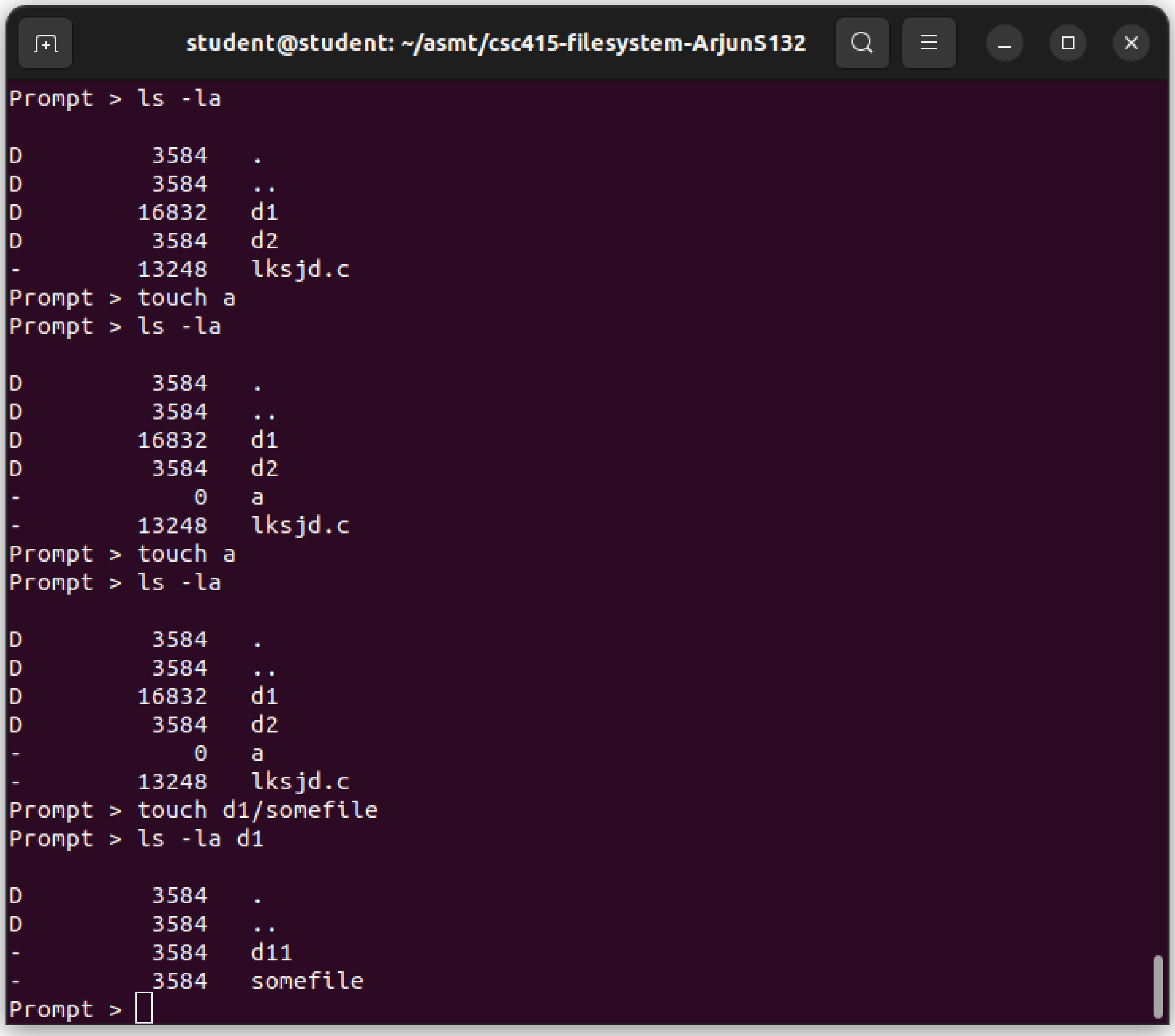
```
|--------------------------------|
Prompt > md d1
Prompt > md d2
Prompt > md d1/d11
Prompt > ls -la

D         3584   .
D         3584   ..
D         3584   d1
D         3584   d2
Prompt > cd d1
Prompt > ls -la

D         3584   .
D         3584   ..
D         3584   d11
Prompt > cd ..
Prompt > cd d2
Prompt > touch a
Prompt > ls -la

D         3584   .
D         3584   ..
-            0   a
Prompt > 
```

rm



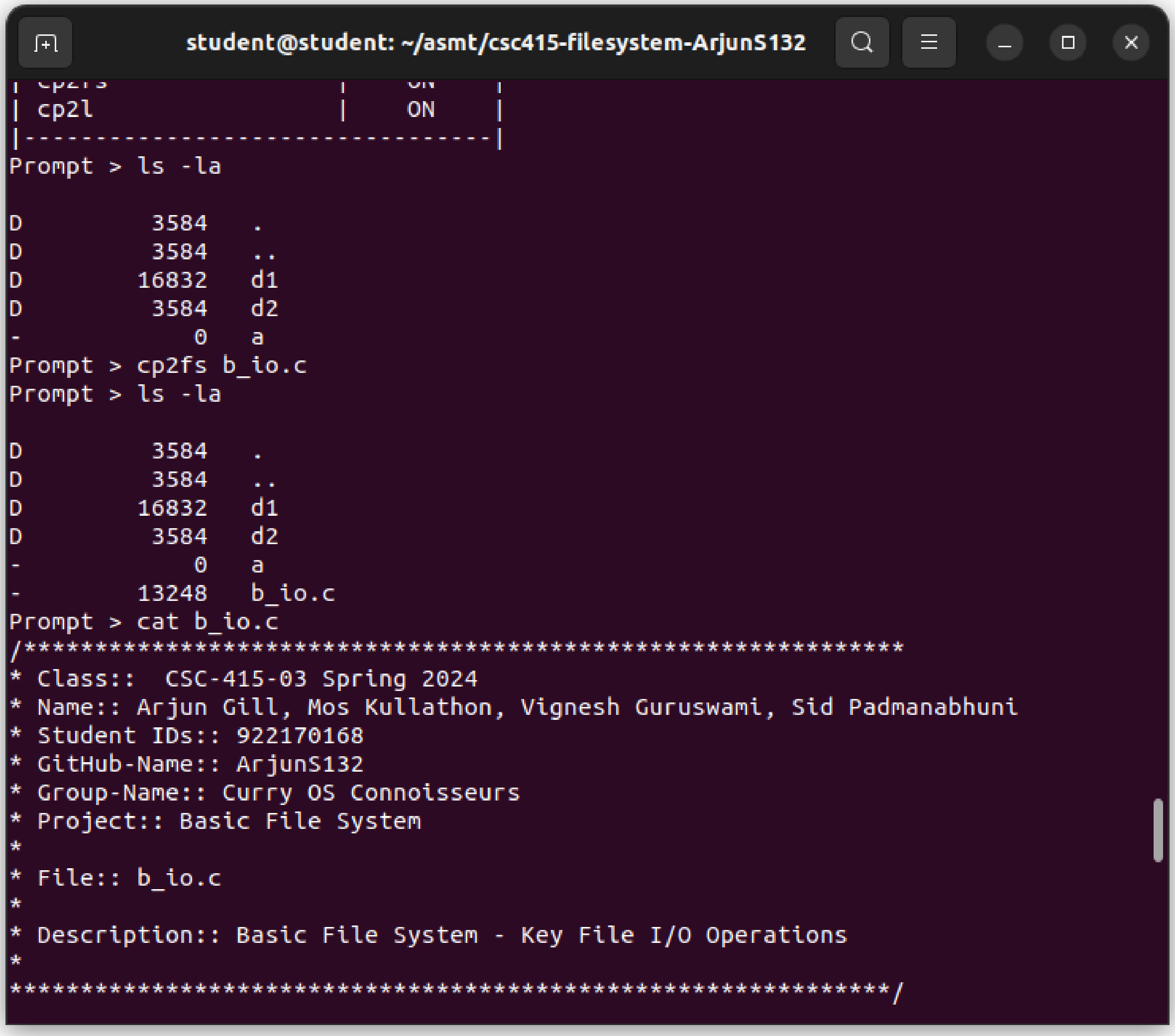
```
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-         13248   b_io.c
-         13248   lksjd.c
Prompt > rm b_io.c
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-         13248   lksjd.c
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
-          3584   test.c
Prompt > rm d1/test.c
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
Prompt > touch d1/d11/abcd
Prompt > ls d1/d11

abcd
Prompt > rm d1/d11/abcd
Prompt > ls d1/d11

Prompt > 
```

touch



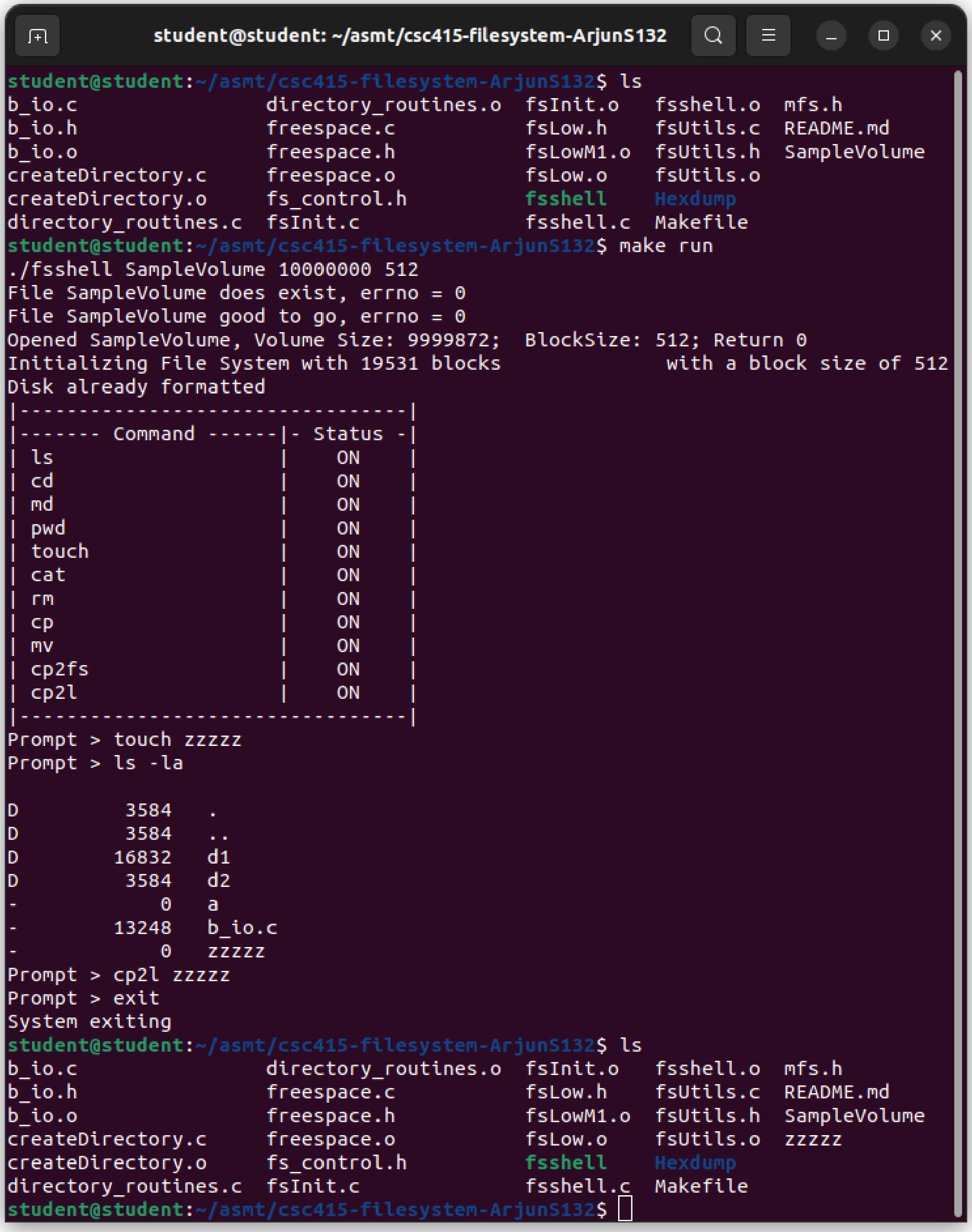
```
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-         13248   lksjd.c
Prompt > touch a
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   lksjd.c
Prompt > touch a
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   lksjd.c
Prompt > touch d1/somefile
Prompt > ls -la d1

D          3584   .
D          3584   ..
-          3584   d11
-          3584   somefile
Prompt > 
```

cat

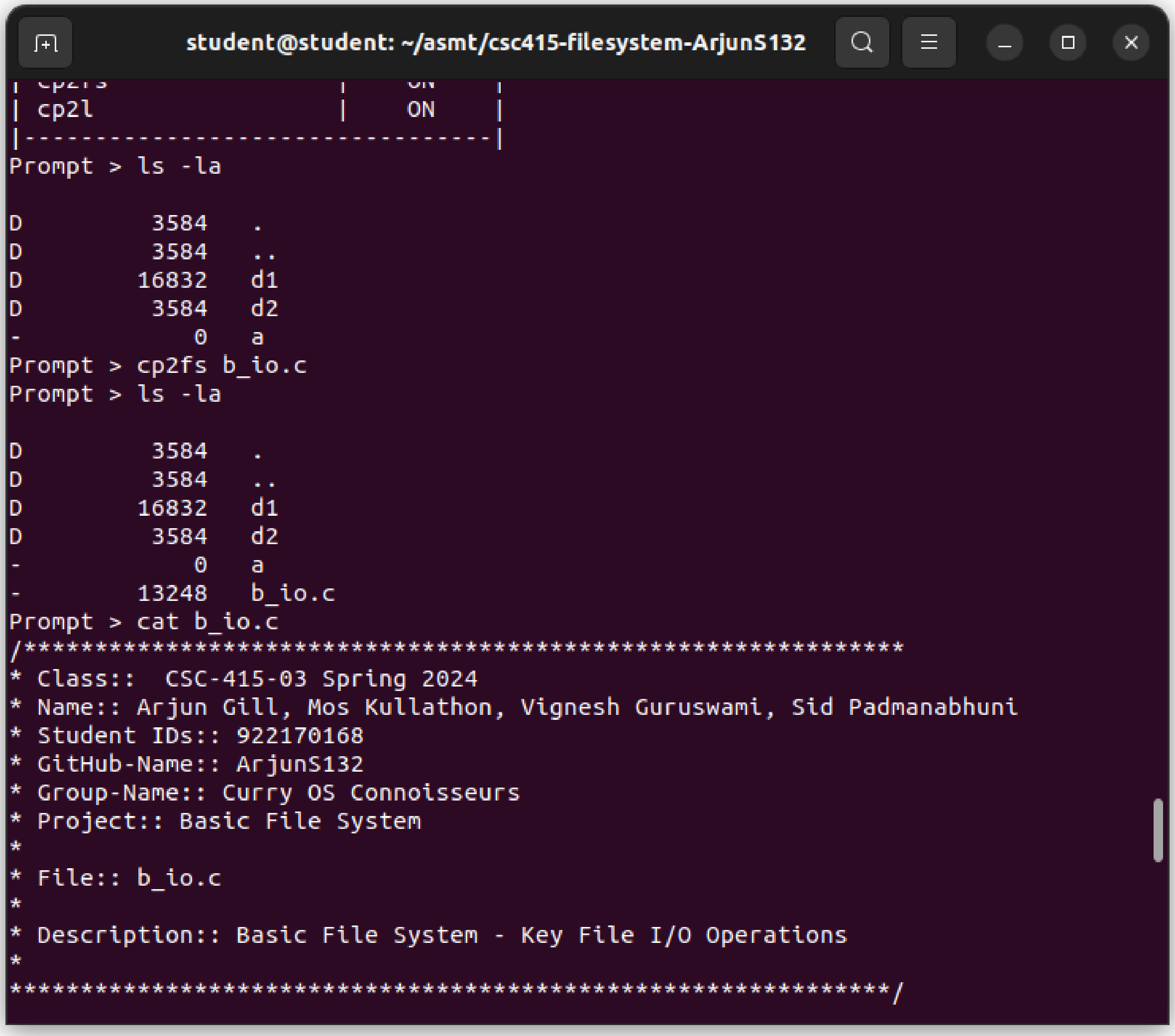


```
| cp2l                   |     ON     |
|---------------------------------|
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
Prompt > cp2fs b_io.c
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   b_io.c
Prompt > cat b_io.c
/**************************************************************
* Class::  CSC-415-03 Spring 2024
* Name:: Arjun Gill, Mos Kullathon, Vignesh Guruswami, Sid Padmanabhuni
* Student IDs:: 922170168
* GitHub-Name:: ArjunS132
* Group-Name:: Curry OS Connoisseurs
* Project:: Basic File System
*
* File:: b_io.c
*
* Description:: Basic File System - Key File I/O Operations
*
**************************************************************/
```

cp2l

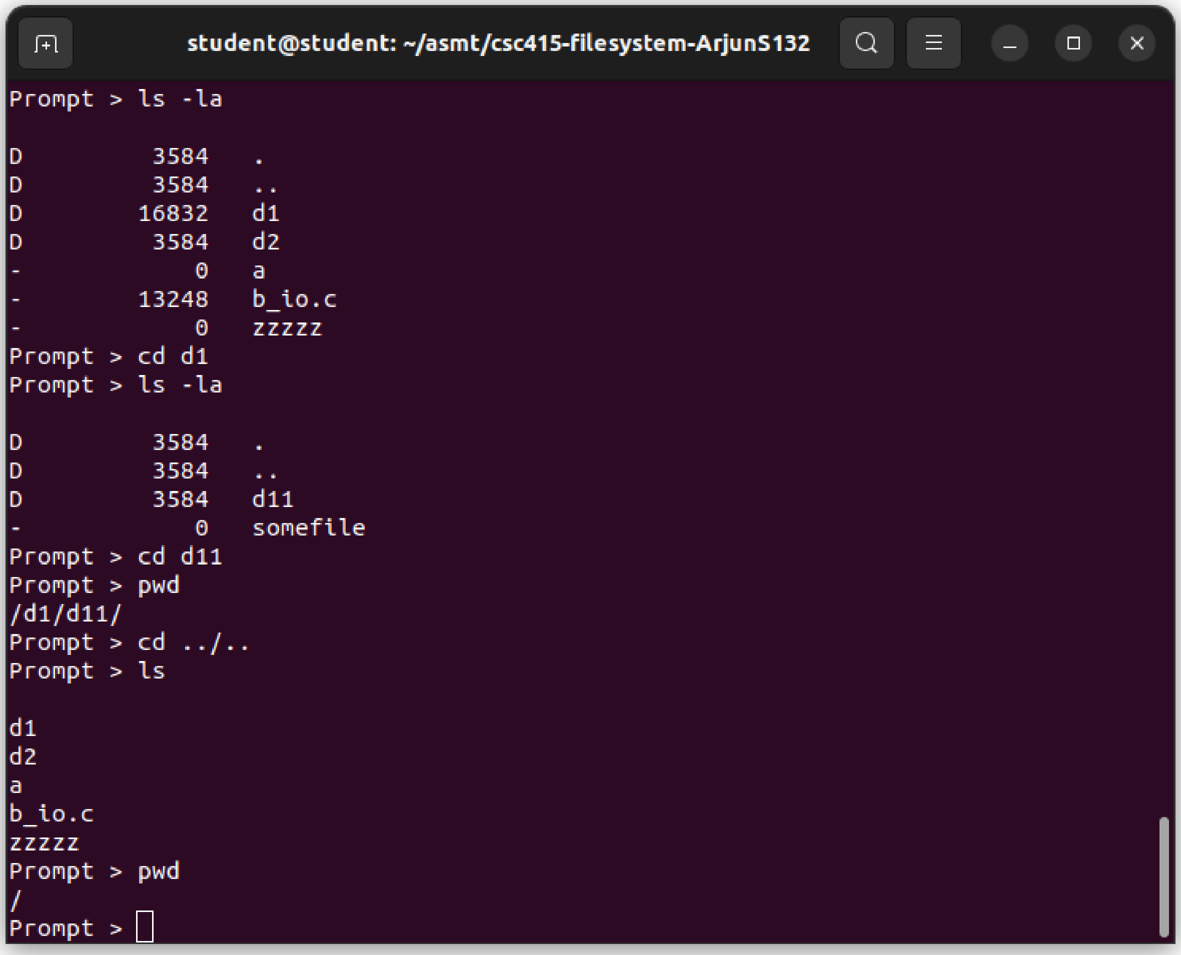

```
student@student:~/asmt/csc415-filesystem-ArjunS132$ ls
b_io.c                directory_routines.o  fsInit.o   fsshell.o  mfs.h
b_io.h                freespace.c           fsLow.h    fsUtils.c  README.md
b_io.o                freespace.h           fsLowM1.o  fsUtils.h  SampleVolume
createDirectory.c     freespace.o           fsLow.o    fsUtils.o
createDirectory.o     fs_control.h          fsshell    Hexdump
directory_routines.c  fsInit.c              fsshell.c  Makefile
student@student:~/asmt/csc415-filesystem-ArjunS132$ make run
./fsshell SampleVolume 10000000 512
File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872;  BlockSize: 512; Return 0
Initializing File System with 19531 blocks              with a block size of 512
Disk already formatted
|-------------------------------|
|------- Command ------|- Status -|
| ls                   |    ON    |
| cd                   |    ON    |
| md                   |    ON    |
| pwd                  |    ON    |
| touch                |    ON    |
| cat                  |    ON    |
| rm                   |    ON    |
| cp                   |    ON    |
| mv                   |    ON    |
| cp2fs                |    ON    |
| cp2l                 |    ON    |
|-------------------------------|
Prompt > touch zzzzz
Prompt > ls -la

D         3584   .
D         3584   ..
D        16832   d1
D         3584   d2
-            0   a
-        13248   b_io.c
-            0   zzzzz
Prompt > cp2l zzzzz
Prompt > exit
System exiting
student@student:~/asmt/csc415-filesystem-ArjunS132$ ls
b_io.c                directory_routines.o  fsInit.o   fsshell.o  mfs.h
b_io.h                freespace.c           fsLow.h    fsUtils.c  README.md
b_io.o                freespace.h           fsLowM1.o  fsUtils.h  SampleVolume
createDirectory.c     freespace.o           fsLow.o    fsUtils.o  zzzzz
createDirectory.o     fs_control.h          fsshell    Hexdump
directory_routines.c  fsInit.c              fsshell.c  Makefile
student@student:~/asmt/csc415-filesystem-ArjunS132$
```

cp2fs



```
| cp2l                    |     ON     |
|---------------------------------|
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
Prompt > cp2fs b_io.c
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   b_io.c
Prompt > cat b_io.c
/**************************************************************
* Class::  CSC-415-03 Spring 2024
* Name:: Arjun Gill, Mos Kullathon, Vignesh Guruswami, Sid Padmanabhuni
* Student IDs:: 922170168
* GitHub-Name:: ArjunS132
* Group-Name:: Curry OS Connoisseurs
* Project:: Basic File System
*
* File:: b_io.c
*
* Description:: Basic File System - Key File I/O Operations
*
**************************************************************/
```

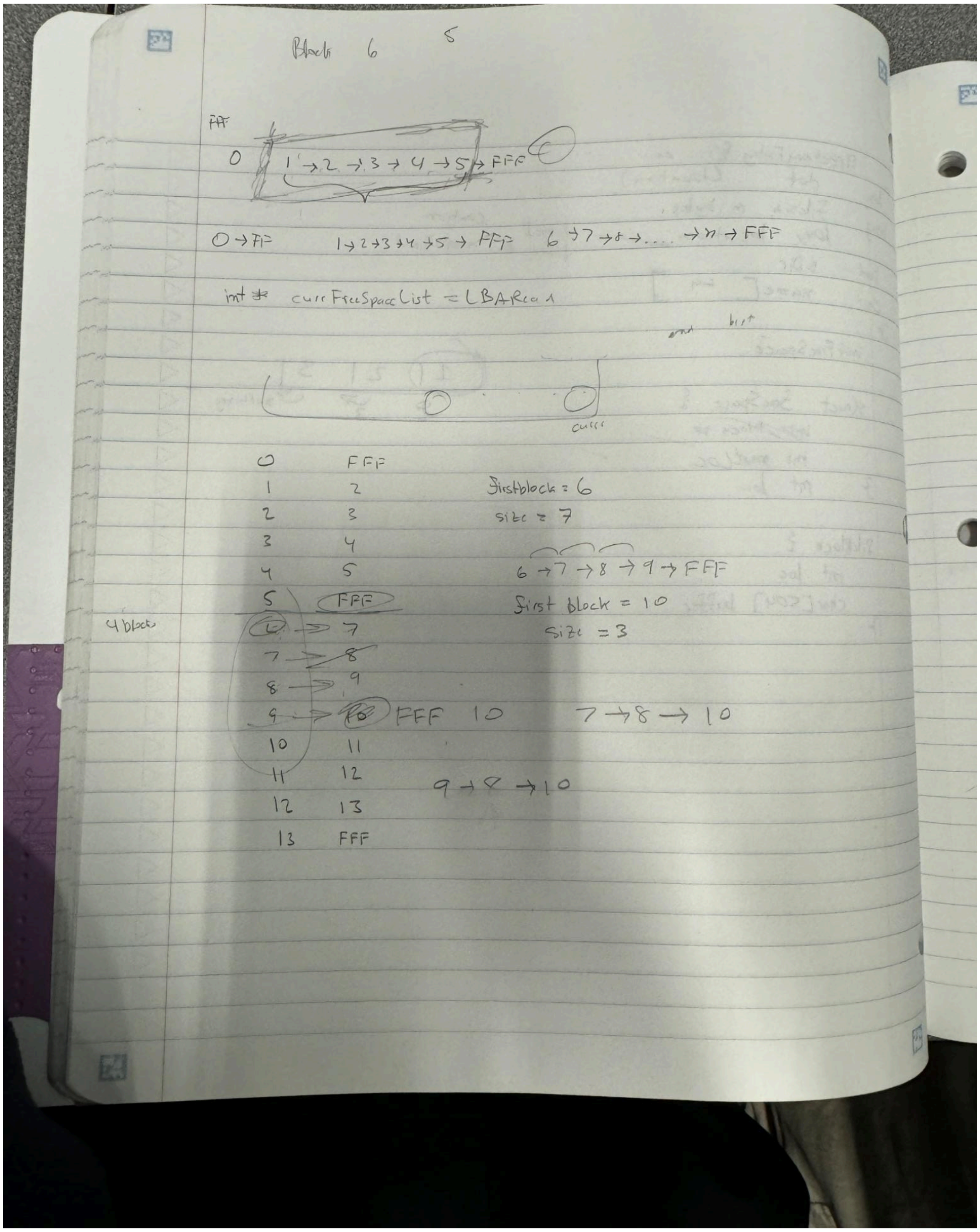
cd and pwd



```
Prompt > ls -la

D          3584   .
D          3584   ..
D         16832   d1
D          3584   d2
-             0   a
-         13248   b_io.c
-             0   zzzzz
Prompt > cd d1
Prompt > ls -la

D          3584   .
D          3584   ..
D          3584   d11
-             0   somefile
Prompt > cd d11
Prompt > pwd
/d1/d11/
Prompt > cd ../..
Prompt > ls

d1
d2
a
b_io.c
zzzzz
Prompt > pwd
/
Prompt > 
```

github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132

Brainstorm Ideas



Block 6 5

FF

0 1 → 2 → 3 → 4 → 5 → FFF

0 → FF 1 → 2 → 3 → 4 → 5 → FFF 6 → 7 → 8 → ... → n → FFF

int * currFreeSpaceList = LBARead

curr

0	FFF
1	2
2	3
3	4
4	5
5	FFF
6	7
7	~~8~~
8	9
9	~~10~~ FFF 10
10	11
11	12
12	13
13	FFF

4 blocks

firstblock = 6
size = 7

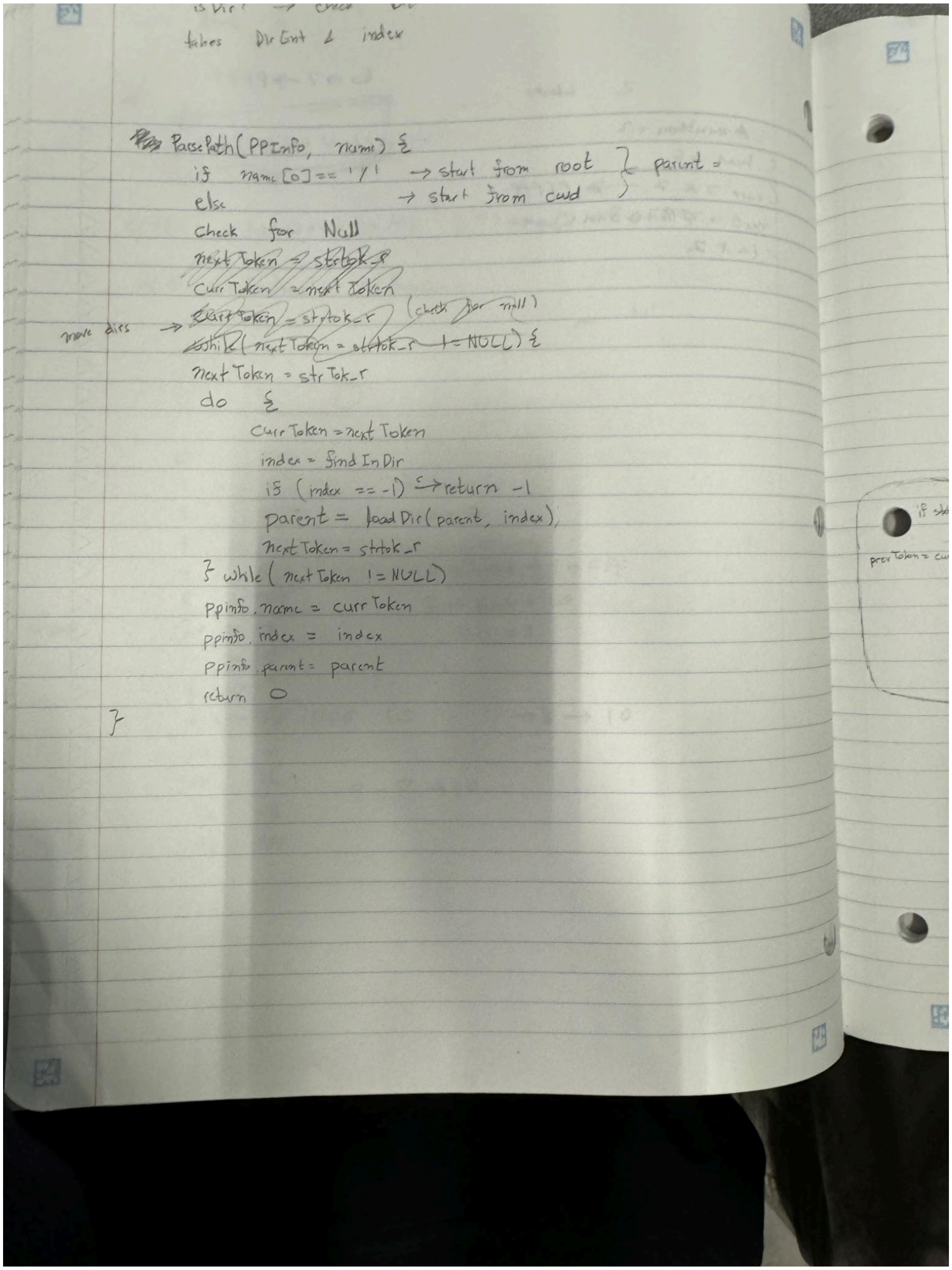
6 → 7 → 8 → 9 → FFF

first block = 10
size = 3

7 → 8 → 10

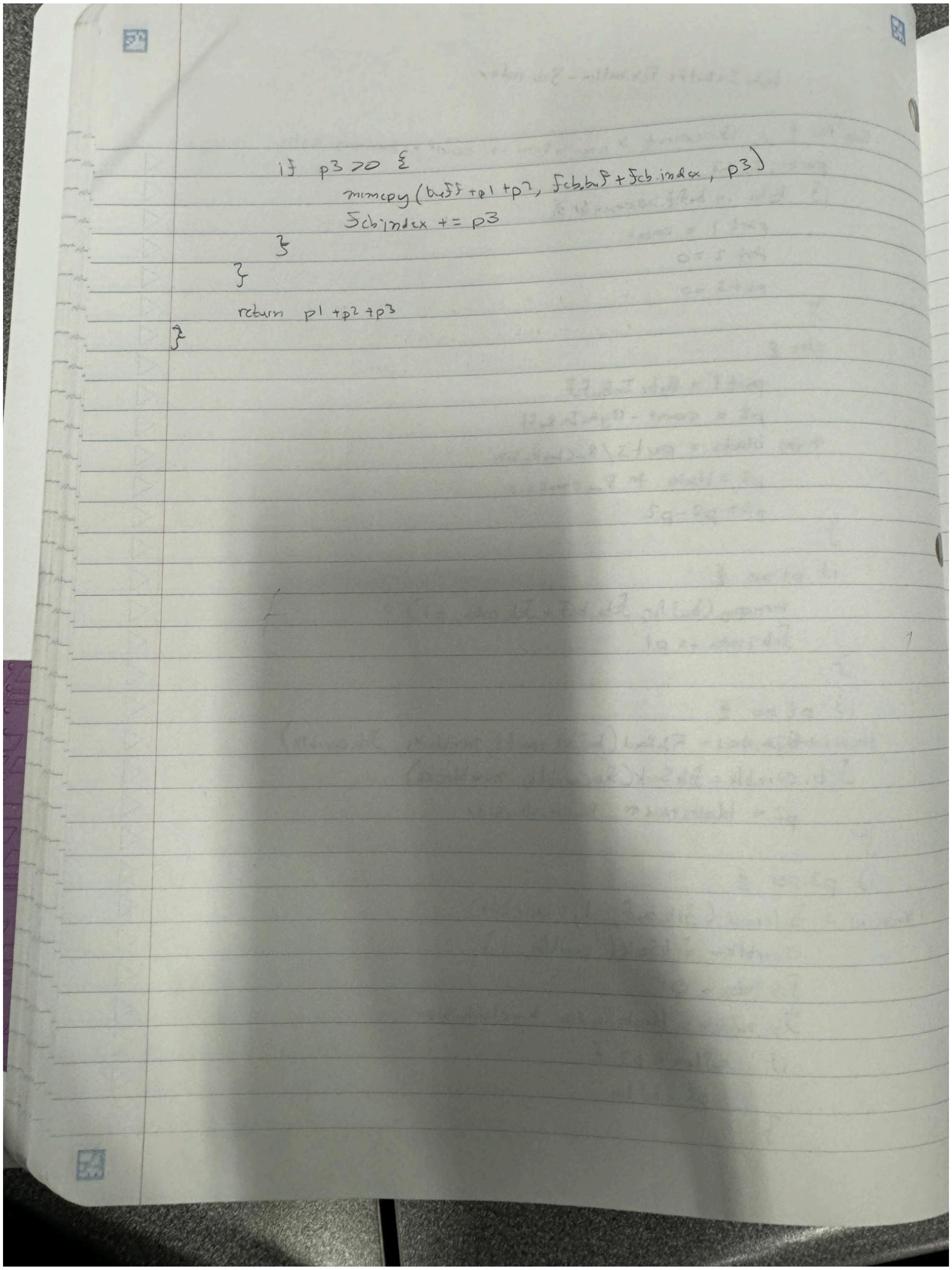
9 → 8 → 10

github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132

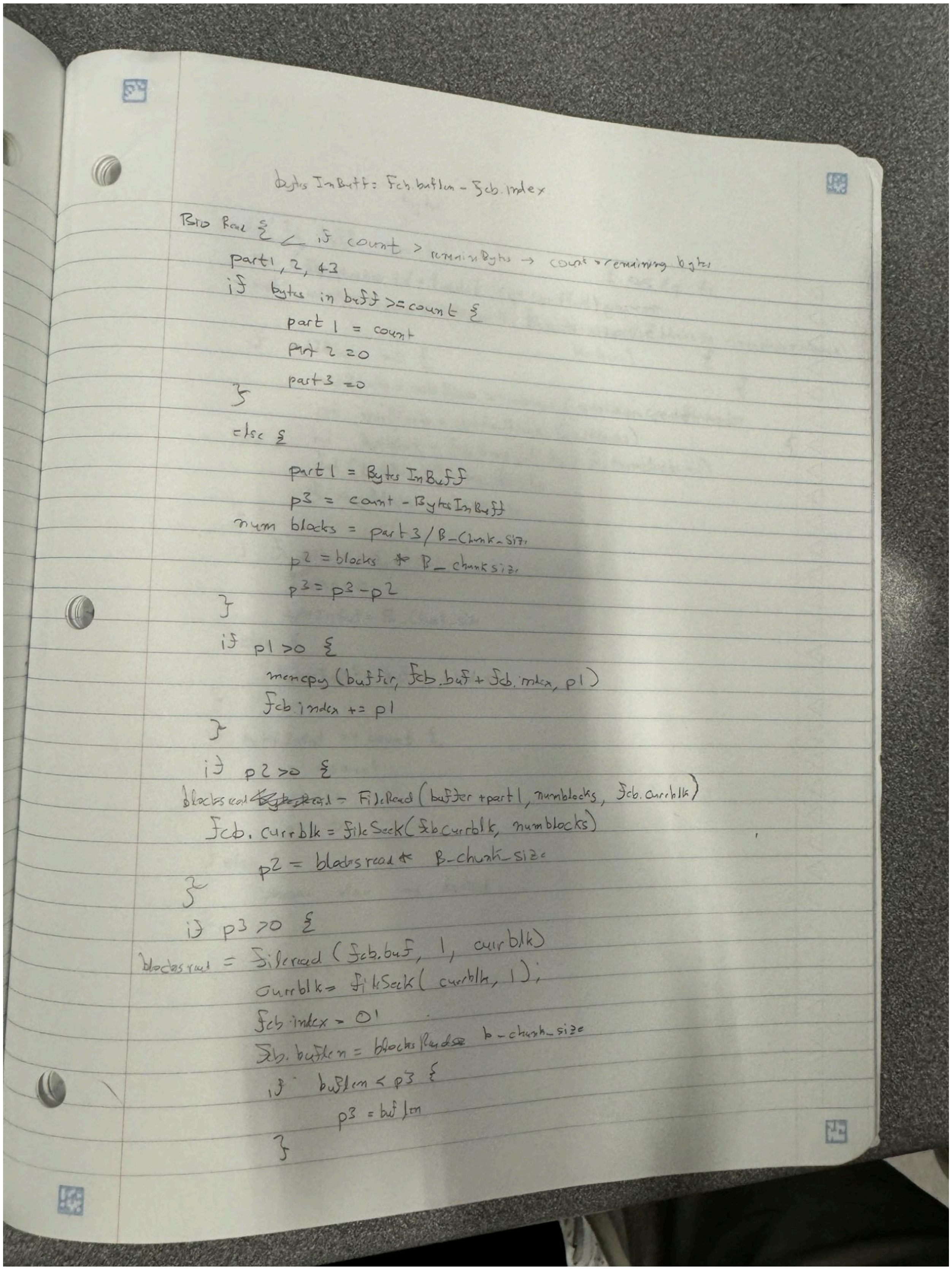


```
takes  DirEnt  &  index

ParsePath(PPInfo,  name) {
    if  name[0] == '/'   → start from root  } parent =
    else                 → start from cwd   }
    check for Null
    ~~nextToken = strtok_r~~
    ~~currToken = nextToken~~
    ~~currToken = strtok_r~~   ~~(check for null)~~
move dirs →  ~~while( nextToken = strtok_r != NULL) {~~
    nextToken = strTok_r
    do {
        currToken = nextToken
        index = findInDir
        if (index == -1) { → return -1
        parent = loadDir(parent, index);
        nextToken = strtok_r
    } while( nextToken != NULL)
    ppinfo.name = currToken
    ppinfo.index = index
    ppinfo.parent = parent
    return 0
}
```

github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132

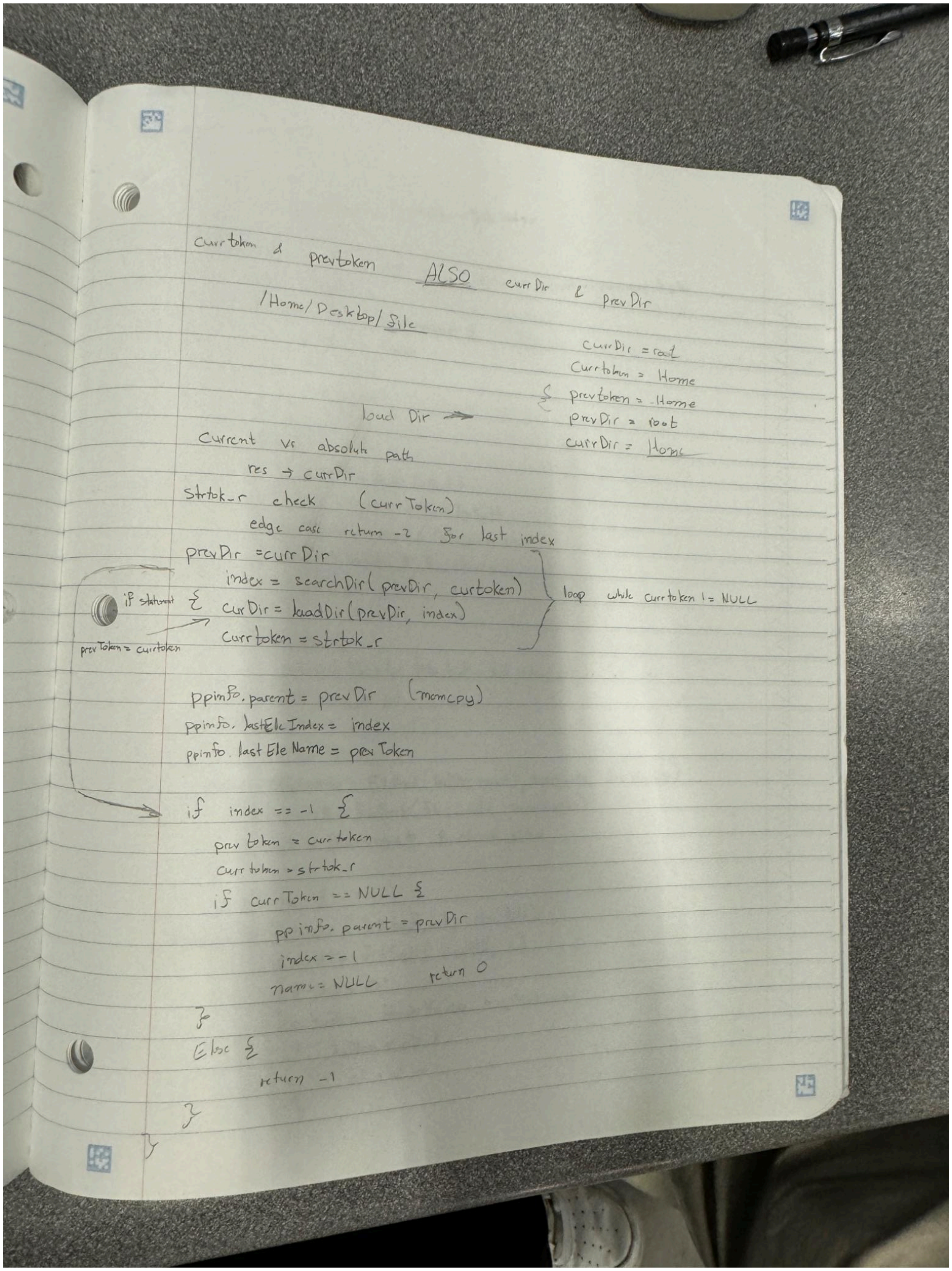


```
        if p3 >0 {
            memcpy(buff +p1 +p2, fcb.buf + fcb.index , p3)
            fcb.index += p3
        }
    }
    return p1 +p2 +p3
}
```



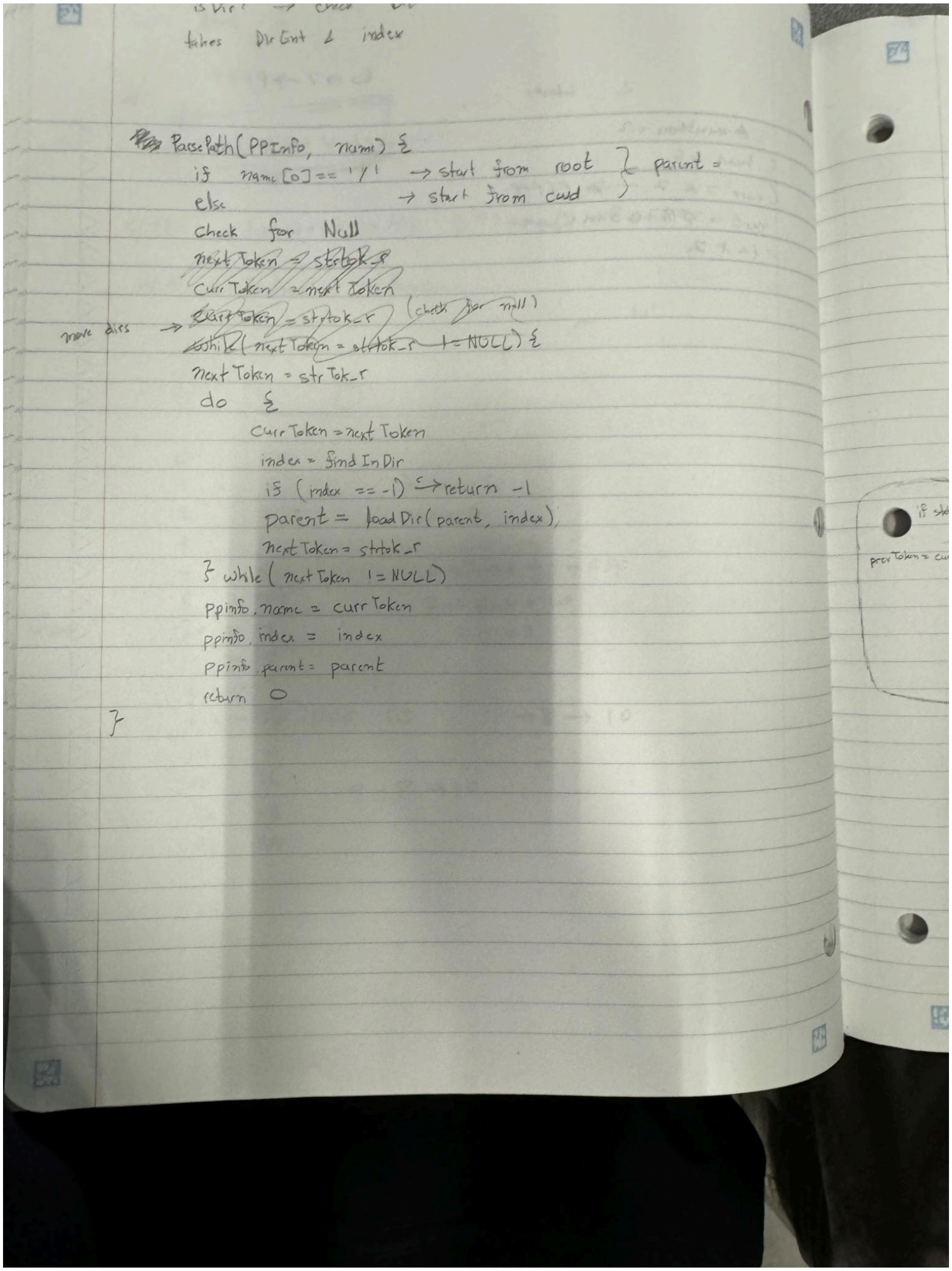
```
            bytes InBuff = Fcb.buflen - fcb.index

Bio Read {  ← if count > remainingBytes → count = remaining bytes
   part1, 2, & 3
   if bytes in buff >= count {
        part 1 = count
        part 2 = 0
        part3 = 0
   }
   else {
        part1 = Bytes InBuff
        p3 = count - BytesInBuff
   num blocks = part3 / B_Chunk_Size
        p2 = blocks * B_chunksize
        p3 = p3 - p2
   }
   if p1 > 0 {
       memcpy (buffer, fcb.buf + fcb.index, p1)
       fcb.index += p1
   }
  if p2 > 0 {
blocksread ~~BytesRead~~ = FileRead (buffer + part1, numblocks, fcb.currblk)
  fcb. currblk = fileSeek( fcb.currblk, numblocks)
       p2 = blocksread * B_chunk_size
  }
  if p3 > 0 {
blocksread = fileread ( fcb.buf, 1, currblk)
      currblk = fileSeek( currblk, 1);
      fcb.index = 0;
      fcb. buflen = blocksRead * b_chunk_size
       if  buflen < p3 {
             p3 = buflen
       }
```

github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132



```
Curr token & prevtoken   ALSO   curr Dir & prev Dir
        /Home/Desktop/file

                                        currDir = root
                                        Currtoken = Home
                                        prevtoken = Home
                  load Dir →            prevDir = root
Current vs absolute path                currDir = Home
     res → currDir
Strtok_r check (curr Token)
     edge case return -2 for last index
prevDir = curr Dir
     index = searchDir(prevDir, curtoken)
if statement { curDir = loadDir(prevDir, index)      loop while currtoken != NULL
     Currtoken = strtok_r
prev Token = currtoken

ppinfo.parent = prev Dir   (memcpy)
ppinfo.lastEleIndex = index
ppinfo.last Ele Name = prev Token

if index == -1 {
   prev token = curr token
   curr token = strtok_r
   if curr Token == NULL {
         pp info.parent = prev Dir
         index = -1
         name = NULL          return 0
}
Else {
      return -1
}
}
```

github.com/CSC415-2024-Spring/csc415-filesystem-ArjunS132



```
isDir? -> check Dir
takes DirEnt & index

ParsePath(PPInfo, name) {
    if name[0] == '/'   -> start from root  } parent =
    else                -> start from cwd   }
    Check for Null
    ~~next Token = strtok_r~~
    ~~Curr Token = next Token~~
more dirs -> ~~Curr Token = strtok_r~~ ~~(check for null)~~
    ~~while( next Token = strtok_r != NULL) {~~
    next Token = strTok_r
    do {
        Curr Token = next Token
        index = find In Dir
        if (index == -1) -> return -1
        parent = load Dir( parent, index);
        next Token = strtok_r
    } while ( next Token != NULL)
    Ppinfo.name = curr Token
    ppinfo.index = index
    Ppinfo.parent = parent
    return 0
}
```

if st

prev Token = cu