CSC 415: Milestone 1
FILE SYSTEM

By:
Github Name: cole-d
Cole Douglas - SFSU ID: 922128256
Bhargava Kadiyala - SFSU ID: 920736411
Pablo Partida - SFSU ID: 921514214
Halia Tavares - SFSU ID: 922128256

Group Name: “Team Taro”
Github Link (username: cole-d):
https://github.com/CSC415-2024-Spring/csc415-filesystem-cole-d.git

CSC 415-03
Professor Robert Bierman
May 6, 2024

Table of who worked on what components:

Bhargava Kadiyala	Milestone 1, setBit, clearBit, getBitZeroOrOne, Milestone 1 Writeup
Cole Douglas	Milestone 1, fs_mvdir, findInDir, loadDir, isDEaDir, createDE, findEmptyDEInDir, printDir, printFour, loadDir, isDEEmpty, fs_rmdir, initFreeSpace, growFreeSpace, allocFreeSpace, deallocFreeSpace, deallocBlocks, loadFreeSpace, initFileSystem, exitFileSystem, fs_stat, fs_delete, fs_create, **parsePath**, fs_mkdir, fs_setcwd, fs_getcwd, b_open, b_read, b_write, b_close, Final Writeup, File documentation
Halia Tavares	Milestone 1, fs_mvdir, findInDir, loadDir, isDEaDir, createDE, findEmptyDEInDir, printDir, printFour, loadDir, isDEEmpty, fs_rmdir, initFreeSpace, growFreeSpace, allocFreeSpace, deallocFreeSpace, deallocBlocks, loadFreeSpace, initFileSystem, exitFileSystem, fs_stat, fs_delete, fs_create, **parsePath**, fs_mkdir, fs_setcwd, fs_getcwd, b_open, b_read, b_write, b_close, Final Writeup, File documentation
Pablo Partida	Milestone 1, fs_openDir, fs_readDir, fs_loseDir, Milestone 1 Writeup, Filesystem Writeup, File documentation

The plan for each phase and changes made

Milestone 1

In milestone one we didn't do as much planning which hurt us in the later milestones. We were really confused on the structure of a directory entry and the vcb as well as what a vcb was and what it should look like. Our plan was just to follow the steps given to us when the milestone was assigned and looked up the things we did not know along the way. One of the big things we were confused about was what file metadata was which the file metadata video (3) helped to explain to us. We also watched the lectures to try and get an idea of how the vcb and the freespace should look. We thought we had written the free space correctly as a bit map which we later realized was written wrong since we never tested any of our functions at this stage other than making sure it would compile.

Resources:

1. File System Concept: https://www.youtube.com/watch?si=Q6TY5xLnJeRZPde7&v=mzUyMy7Ihk0&feature=youtu.be

2. Files and Filesystems: Files & File Systems: Crash Course Computer Science #20 - YouTube

3. File Metadata: https://www.youtube.com/watch?si=ugo4viYjv7CRvl7P&v=3QYZqeTnNiM&feature=youtu.be

Milestone 2

During milestone 2, we realized our initialization of the filesystem and the freespace were written incorrectly. We had to go back and fix our root directory initialization and freespace map which we were given guidance on during office hours. We made sure to loop through the freespace and check that the bits we were marking were actually being marked used but realized that they were not due to a simple operational error. After fixing this we then could actually focus on milestone 2. We started with the get and set current working directory functions since we thought they would be easier functions and then realized we needed to first complete writing the parse path function. To write parse path we referenced the lecture that we talked about and added print statements to make sure it was running as we thought it should. During this time we also had to write the helper functions for parse path which were loaddir, findindir, and isdeadir. Although we were able to finish writing parse path we were still confused on how it used the structure ppredata which we then realized is just a structure we fill in order to get multiple pieces of data returned back to us. Once we understood that, writing make directory and get directory were a lot easier. When writing make directory we initially had issues trying to make directories in other directory as well as using the cd function to navigate back to the parent. We realized this was because we were creating the directories but not writing them to disk there for we couldn't find anything we were creating because it wasn't actually there. After we finished that, debugging set was easier as well. From there we finished writing the delete and remove functions which went quickly due to our new understanding of how directories are managed along with the

freespace map. The last part of milestone two we had to finish was opendir, readdir, and closedir. When it came to making open, read , and close our first thought was that it would be like assignment 5, but this time we were working with directories instead. It was important to understand what we use inside of these functions, like our two structs fdDir and fs_diriteminfo. After multiple office hours, it became clearer and clearer to us what needed to be done for these functions. We initially struggled the most with open but using a helper function to make sure we were in a valid path as well as a valid directory. Then loading in our directory, and initializing the directory struct made the rest easier to finish. Once the directory item info was allocated, we knew we would have to loop through directory entries, until reaching the end. Read meant we wanted to populate our directory item info, and once that was accomplished all that was left was to close the directory, and free the correct directories to not cause any memory issues.

```
4/22 10:18am
- ask bierman:
  - We don't understand how ppretdata and DE structs are related and
how to pass them into parsepath and helper functions
  - things work when we remove the sample volume but don't save when we
don't remove...
  - does loaddir work...

Updates:
  - finished writing get/set/make
  - debugged findindir, isdeadir,
  - currently debugging load dir and parse path
    - will be debugging get/set/make next
```

```
4/23 10:00am
Bierman Questions:
  - are we loading freespace right...?
  - seg faults on set

Updates:
  - cleaned up random comments and prints

To-Do:
  - currently debugging set
  - debug get/make
```

```
4/23 1:30pm
Current Bierman Help Mes:
  - are we loading freespace right...?

Updates:
  - seg fault fixed (warnings are bag ig...)
  - debugged makedir and getcwd
  - mostly debugged setcwd still fixing a few things but it kinda works

To-Do:
  - need to be able to cd .
  - need to be able to create dirs inside of other dirs
  - set cleaner function
  - will do remove dir after this
```

```
4/24 10:00am
Current Bierman Help Mes:
  - are we loading freespace right...?
  - setcwd how to go backwards/ fix paths
  - what does fs stat do

Updates:
  - make parent not loading properly
    - make can't make inside another directory and it gets all messed
up
  - working on final things for set

To-Do:
  - fix make to have valid entries in parent
  - set needs to fix weird paths
  - write remove dir and delete
  - fs stat
  - move directory
  - time stuff
```

```
4/25 1:00pm
Current Bierman Help Mes:
  - uhhh... we don't care about freespace anymore probs need help on
remove and delete

Updates:
  - yay parents work now and cd and md work in the shell
  - ^only update cause it took us like 5 hrs to fix >:(

To-Do:
  - set needs to fix weird paths.. (maybe we skip TT)
  - write remove dir and delete
  - fs stat
  - move directory
  - time stuff
```

```
4/26 8:00am
Current Bierman Help Mes:
  - line 79 freespace.c: do we need to write when we alloc fs

Updates:
  - wrote deallocate, isdir, isfile, removedir, delete
  - debugging something wrong w loading the fs map

To-Do:
  - move directory
  - debug stuff to work better
  - set needs to fix weird paths.. (maybe we skip TT)
```

```
4/28 10:00am
Current Bierman Help Mes:
 - remove not checking if empty ... what do we pass in lol?
    - had to hard code... calcs not working
    - weird side but: dot is weird

Updates:
  - found another bug in remove
  - freespace isn't being loaded currently

To-Do:
  - move directory
  - fix freespace
  - fix remove bug
```

```
5/4 11:00am
Current Bierman Help Mes:
    - how to do seek ?
    - why vrun error

Updates:
  - finished read
  - finished write?? maybe
  - added headers to everything
  - added bare bones comments and deleted prints

To-Do:
  - DO WRITE UP
    - go back to update better comments
    - look at memory and free shit
  - testread/write
  - fix move
    -needs to "reassign" directoires intead of deleteing and
creating
  - fix hardcode in remove
  - b_seek, b_close, exit_filesystem
  - clean weird paths
  - fix more vrun errors
  - fix print in startup
  - figure out dates
```

Planning out which functions might be easy and getting an idea of what each is.

~~get~~ - ~~easy~~

easy	maybe	hard
get	Stat	open/read/close
isFile	delete	setcwd
isDir	remove	mkdir

dir -> is

open / read / close exam

Parse path / Setcwd

- fix parse path
  -> finish 3 helper functions
- fix create directory from root dir

Trying to write down what we know about each function and how they might work.

setcwd
getcwd > idk what these do

4/14

- mkdir –
- make a directory
- man instance of directory

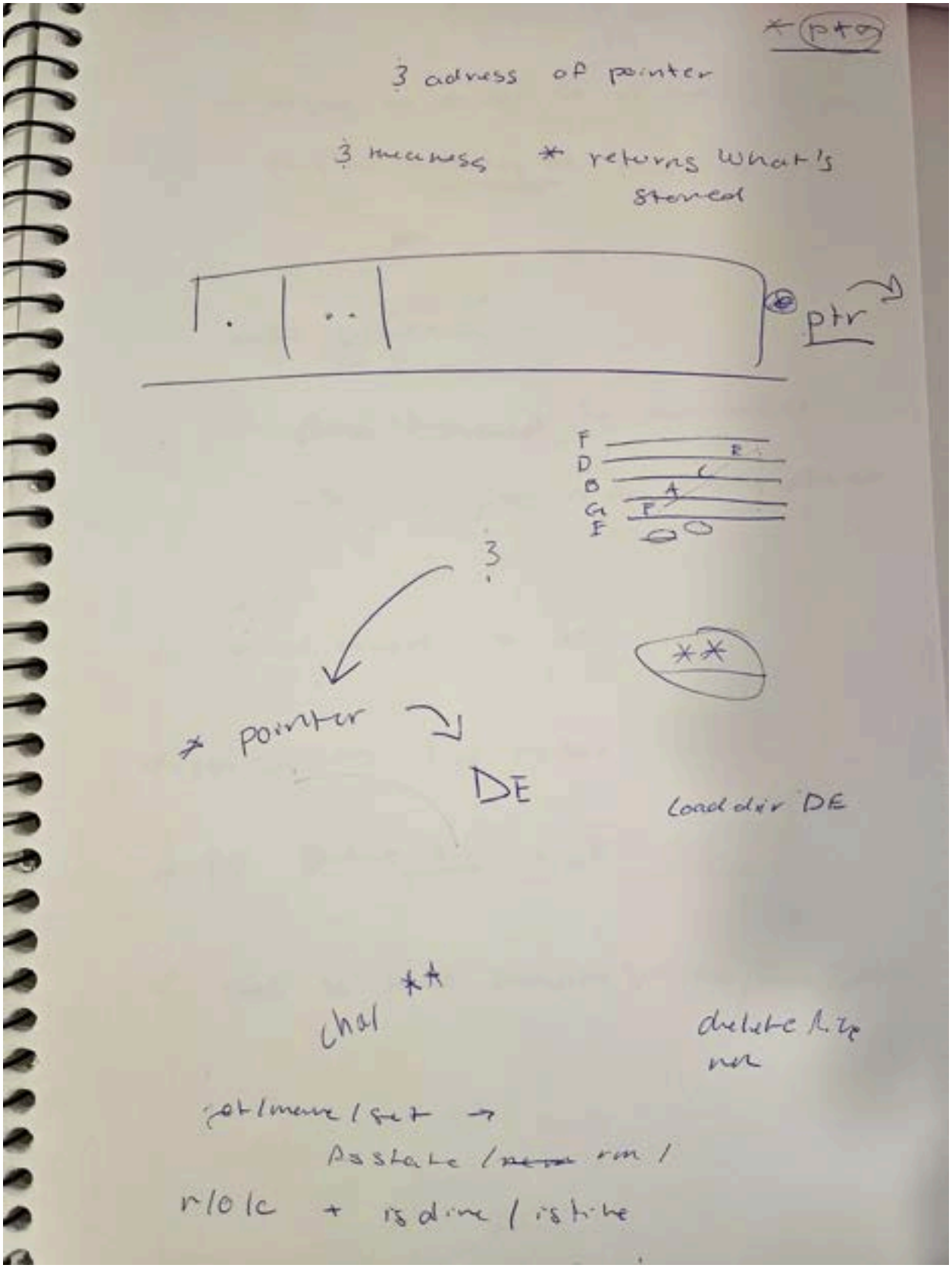
- isFile
  directories[] isFiles 0
  return 0 or 1 if we'll is a file
  same fcns here
- isDir
  return 0 or 1 if it's a directory

- opendir / readdr / closedr

- fs_stat ??

fs_delete → delete file

Piecing together how a directory works.



Understanding how parsepath and ppretdata structure works

135

-> passing in struct so pp can fill it in

pass in address of the structure

malloc (size of pp ned ard)

don't need to set parent

-> just pass in an empty struct

did..

now need to do.

- pretest -> find in Dir
- pp parent nil line 135
- need to load Sample Volume / Free Space

Writing some pseudo code for findInDir

```
int findinDir(ppietdata * parent)

for (entry Count)
        strcm
    if (parent[i].name = token)
        return i)

    else
        return -1;
}
```

More notes on how a directory and directory entry might work.

global loadcwd

loaded RD

PP retdata

↗

pointers to array of DEs

array of DE

parent array of DE

from . entry

. is itself .→ [0]

file size how many bytes for DE

Size of Blob file / DE

int entryCount = parent[0].fileSize / sizeof(DE)

20*

Strtok to see if matching names

rootdir [10] DEs

./.. → home

Planning out createDir and the root directory

home has a size
loc
name

DE ~~d~~ references a blob
of data w/ can reference
another directory

-> creating a directer

# of entries * DE malloc ✓

<>

need a parent

root dr pointer

array of 50 entries

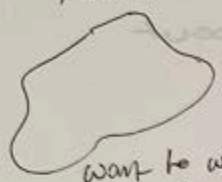
More planning for createdir

addressing # des

19

DEs in the root

malloc



want to write out

size = ~~size~~

~~# of DEs~~

num * sizeof(DE)

.. parent

. current

foo.txt → diff location / diff size

iBAwrite is really wrong

Blob of data to disk

right from rootdir (not + i)

X no loop

node

LoadDir pseudo code

load

is the opposite

-> here's a DE

~~to~~ don't care about

name

-> location

-> whats size

tells how many blocks

used and going to malloc

blocks needed * block size

for buffer load

new cor is directory at

that

load brings ~~DE~~ from disk to

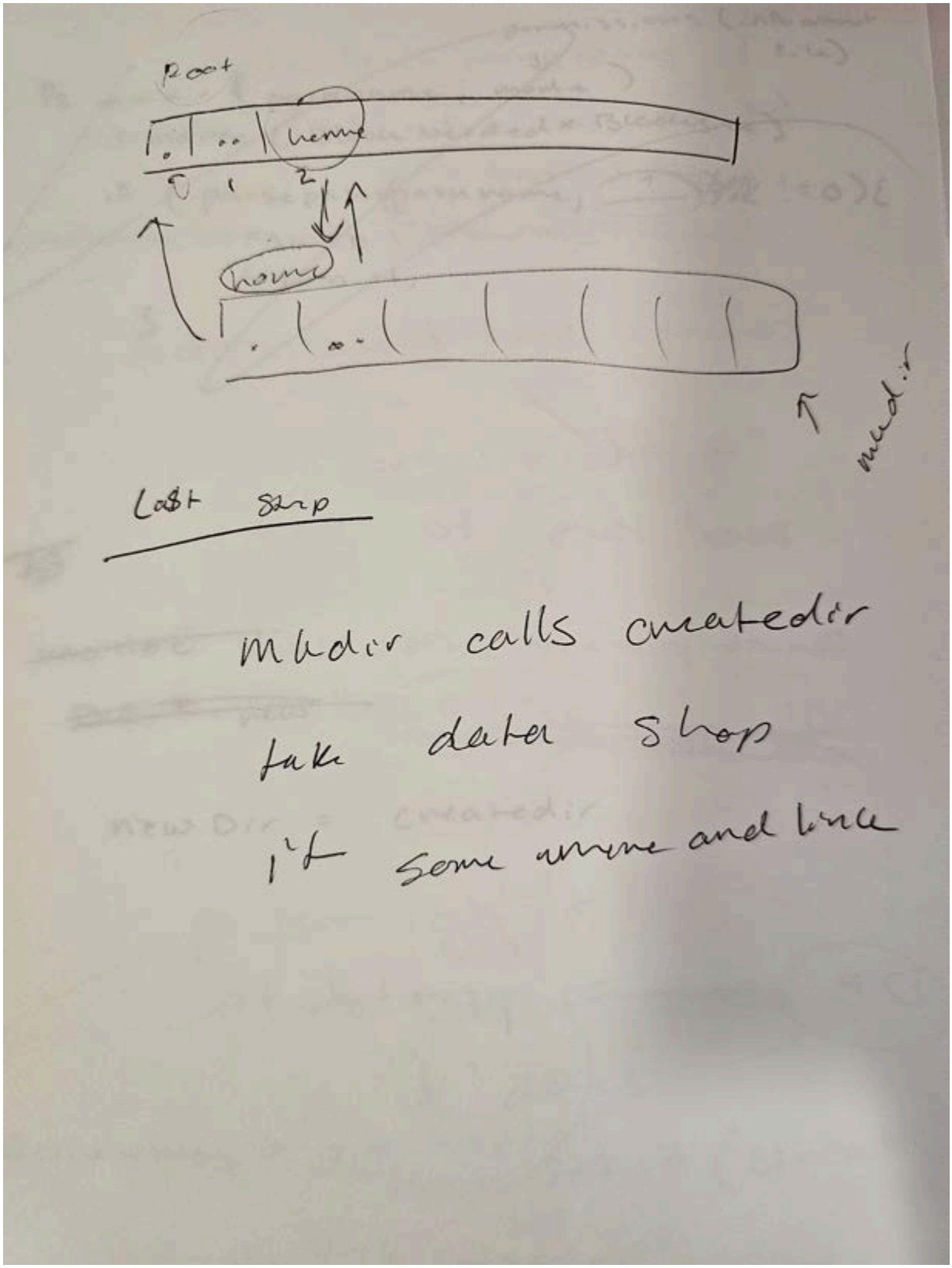
memory

-> WHERE -> HOW BIG

FindInDir pseudo code

```
FindInDir
    -> name of home
        Strcpm compares
    index based on . filesize

    i = 0 to numof entries
    find it return i
    don't find it
        return -1 entry doesn't
        exist

What do helpers do simply

Load -> this DF I want to load
        into memory

is this entry a DE
```

Understanding how navigating a directory works and some mkdir notes



Setcwd notes and parsepath notes

1. ensuring valid path

2. loading cwd

endcpy 3. if their path is relative or absolute

helper func to get rid of dots/dotdot

if (path is absolute)

[Parse path ←] : The key: returns parent pointer and index

Load ← given de (struct) → load that de into ram

name fix up ←

knowing location how many blocks to read

LBA read in to RAM

test by doing load dine

after rd.

once you have load dine

can wor

Incorrect mkdir pseudo code

permissions (info about file)

fs_mkdir (pathname , mode)

? malloc (Block Needed * Blocksize)

if (parsepath(pathname, ?) != 0){

FAILED

return -1;

}

~~malloc~~ +

~~DE * new~~

newDir = createdir

Mkdir pseudo code

1. parseparm -> is valid
2. call createdir(#of DE and parent = cwd
3. ~~attach new dir to parent~~?

parent = last thing in cwd

need this to set

Parent param in new Dir

~~to do the op~~

DE ⟷ ppretdata

{

DE * parent=DE[]

}

Setcwd pseudo code

0 or 1 return

Set

- updating cwd

parse path is it valid?

->

last

get the last thing

go through it

-> see if exists

if it does

then set it (strcat

if it doesn't

return -1

return 0

Planning setcwd how navigation works

user wants to go to

/home/docs/filex

we are at

/home

first we absolute/relative path(pp)

~~if absolute~~

comp : we are at home

- does docs exist in home?

-> ~~yes~~ no

return -1

-> yes

does ~~home~~ filex exist in docs

-> no

return -1

-> yes

~~-> is the last thing?~~

the add to cwd/updat

More directory navigation or perhaps bug fixing

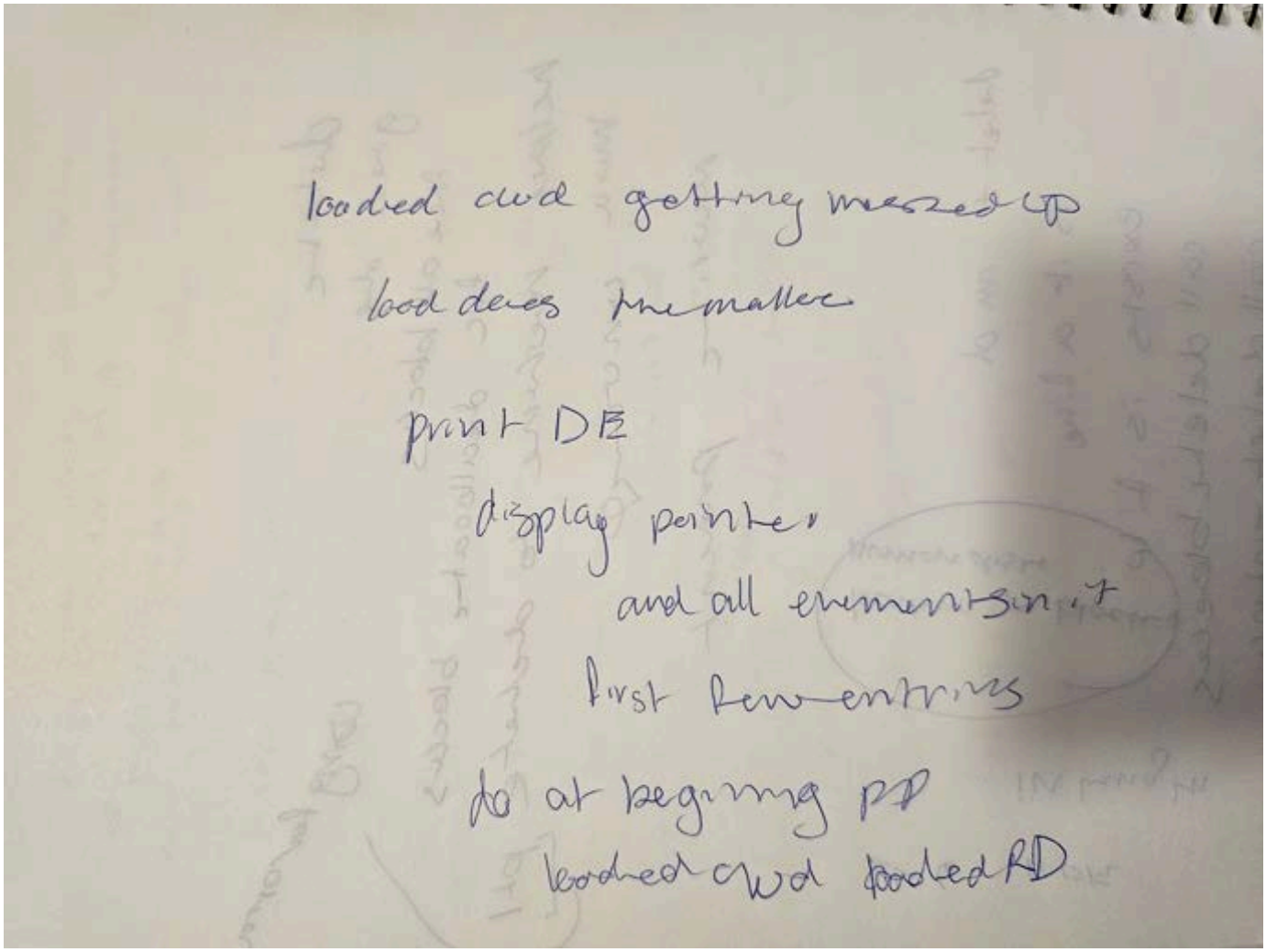
- buffer

/home/Docs/Rilex

/home

check is home here ✓
yes
cont.

Troubleshooting and planning for our loaded cwd in setcwd



loaded cwd getting messed up

load does the matter

print DE

display pointer

and all elements in it

first few entries

do at beginning PP

loaded cwd loaded RD

Adjust path logic and some pseudo code

adjust path -> 0 or 1 success or fail

pprmDo

int adjustPath(↑) {

/ . /

if .

← /x/.. /x/y

array of pointers

delim

strtok(

if (parent lastElementIndex == "..")

loaded CWD = the string w/o last dir

Planning we did after completing parsepath, mkdir, and set/getcwd

need to do

- ~~fix make to have valid entries in parents~~ fix
- set needs to fix weird paths ←
- fs_rmdir / fs_delet
  - ↑ deletes directories
  - ↑ deletes files
- fs_stat ?? fills astruct? w/ info about...
- move directory command
- time stuff

~~Parent size is not being properly updated~~

loadDir?

parent and lastelement

if its a -1

then by

-> check index is not -1

-> last el exists

is it a directory

loaddirectory (&adnesspaventdindex)

directory [directory] = load dire

(i

ectory [directory] = load dir [3 parent [index not found]

Size of (fd dir)

StartLocation = 0;

OpenDir psuedo

get rid of double thing

read keeps giving next name

->open just sets that u

di is this struct

-> need to malloc and

assign it to di

direct

read will strcpy name into file

in read

return Di (pointer to the structure)

free directory that you loaded

Read doesn't need to malloc anything

look at entry

directory->dirp[] [deposition]

↑

is there a name there

in a loop [if location is -1 then it is used

cpy name, set file type, increment

readDir psuedo

how many entries do you have

how many times you loop

know size of DE blob

size listed in de[0] size/sizeof(DE)

is the num of entries you can

calculate. in open instead of

Start location ~~change~~

if i is used

increment Start

and loop

keep loop until ...

eventuall i > #of entry fall out

and return null

Fs_delete and rmDir psuedos and differences

delete

given dir

free all blocks

fuc deallocate blocks

using for loop

helper routine to delete ptl

mark unused

rewrite parent

delet + rmd

is it a file

exists is file

call delete blocks

call delet entry

location is block number disk

ab 6 11 0/5

in length

Milestone 3

During milestone 3, we had a difficult time since most people in our group had not been able to finish assignment 5. Therefore we went to office hours for help to explain the five functions in this milestone and how we should start completing them. We started with b_open which opens and prepares a file for b_read or b_write. We referenced the code from the lectures as well as the lecture which reviewed assignment 5 and utilized similar logic in order to complete the function. We ran into difficulties because of the amount of variables and calculations we had to keep track of so we had to run through the code multiple times to ensure our math was correct in order to debug the read function. To write the b_write function we were able to apply the same logic as read but with some adjustments. We needed to create a function which could preallocate blocks to the files we wanted to write, which is similar to how we wrote our function for allocating freespace. However, this function needed to be able to relocate used blocks as well. We had to work the logic a couple times as you can see in the image of drawings below. We had issues trying to get the data actually into the new file utilizing the cp2fs function. We worked through the logic multiple times. Initially we thought that it was our grow blocks function which wasn't working properly. We realized that the function wasn't working properly and was skipping a block which was causing a bug, however, this did not fix our original issue. So we moved onward through the logic and realized we were not writing properly to our buffer which was the source of our issue. Once we fixed that we were able to complete writing. Afterwards, we were able to finish b_close as well.

```
isDirEmpty(&parent[listelement])
                 ↓
dir = loadDir(indirEntry)
cnt = dir[0].size / sizeof(DE)
for(i=2; i < cnt; i++)
    if(dir[i].loc != -1)
        free(dir);
        return False;
free(dir);
Return True
```

```
4/30 2:00pm
Current Bierman Help Mes:
    - where do we make an instance of the struct or do we use the array
instance
    - how does read/write work ...?

Updates:
  - fixed remove
  - fixed free space
  - touch command working :D
  - almost kinda done with b_open
    - understand the flags now and have them being checked in open
    - just need to finish storing data or something
  - fixed isdir and isfile

To-Do:
  - trunc flag in open
  - fix hardcode in remove
  - b_seek, b_read, b_write, b_close
  - clean weird paths
```

```
5/1 10:00am
Current Bierman Help Mes:
    - where do we make an instance of the struct or do we use the array
instance
    - how does read/write work ...?

Updates:
  -

To-Do:
  - trunc flag in open
  - fix hardcode in remove
  - b_seek, b_read, b_write, b_close
  - clean weird paths
```

```
5/1 1:00pm
Current Bierman Help Mes:
    - read: should part variables be global ??
    - smashing error
    - how does read/write work ...?

Updates:
  - kinda fixed up open idk beirman said it looked ok
  - currently working on b_read

To-Do:
  - fix hardcode in remove
  - b_seek, b_write, b_close
  - clean weird paths
```

```
5/2 10:00am
Current Bierman Help Mes:
    - help us fix read/write
      - why plus file size
      - why do we change buflength?
    - how to do seek ?

Updates:
  - read and write written
  - debugging read and write
  - ls is working
  - some vrun errors

To-Do:
  - fix hardcode in remove
  - b_seek, b_close
  - clean weird paths
  - move
  - fix more vrun errors
```

```
5/3 8:00am
Current Bierman Help Mes:
    - how do we test read?
      - how does a file work where is the body?
      - but a file is a dir so it can't have a body?
    - help us fix read/write
      - why plus file size
      - why do we change buflength?
    - how to do seek ?

Updates:
  - finished move

To-Do:
  - test and fix read/write
  - fix hardcode in remove
  - b_seek, b_close
  - clean weird paths
  - fix more vrun errors
  - fix print in startup
```

FCB -> preallocc

if(remainingBytes < FCB->

fcB->bufferSize = Re

if(FP + count > fileSize) fcb->BufPos = 0

// Growing

neededBlocks = (FP + count + BlkSize - 1) / BlkSize

(needed > FCB->preAlloc) preAlloc = needed * 2

LOC = GrowBlocks(preAlloc + 500);

// ocupiedcnt, newcnt

FCB->preAlloc + preAlloc

fileSize = FP + count

B_write planning

fp + what line I want to write

new fp

old file size + amount my write

new fp

- prealloc

2 byte pre 1

2 → 5

if go past extending file

do in open → if need more

than prealloc x num of blocks

B_open planning with the trunc flag and append flag

~~index of element of last name = last~~

update after create bc now it does exist

now

else return -1
(doesn't exist and didn't spec create)

trunc flag

-> take index and delete blocks from it

if (flags & trunc == trunc){

deallocate de (& ppinfo -> parent[ppinfo LEI])
* deallocate blocks is needed
-> should be called in dealloc = (bordes)

}

now ppinfo -> parent[class element].filesize
= 0
and Location = 0 //not freeing

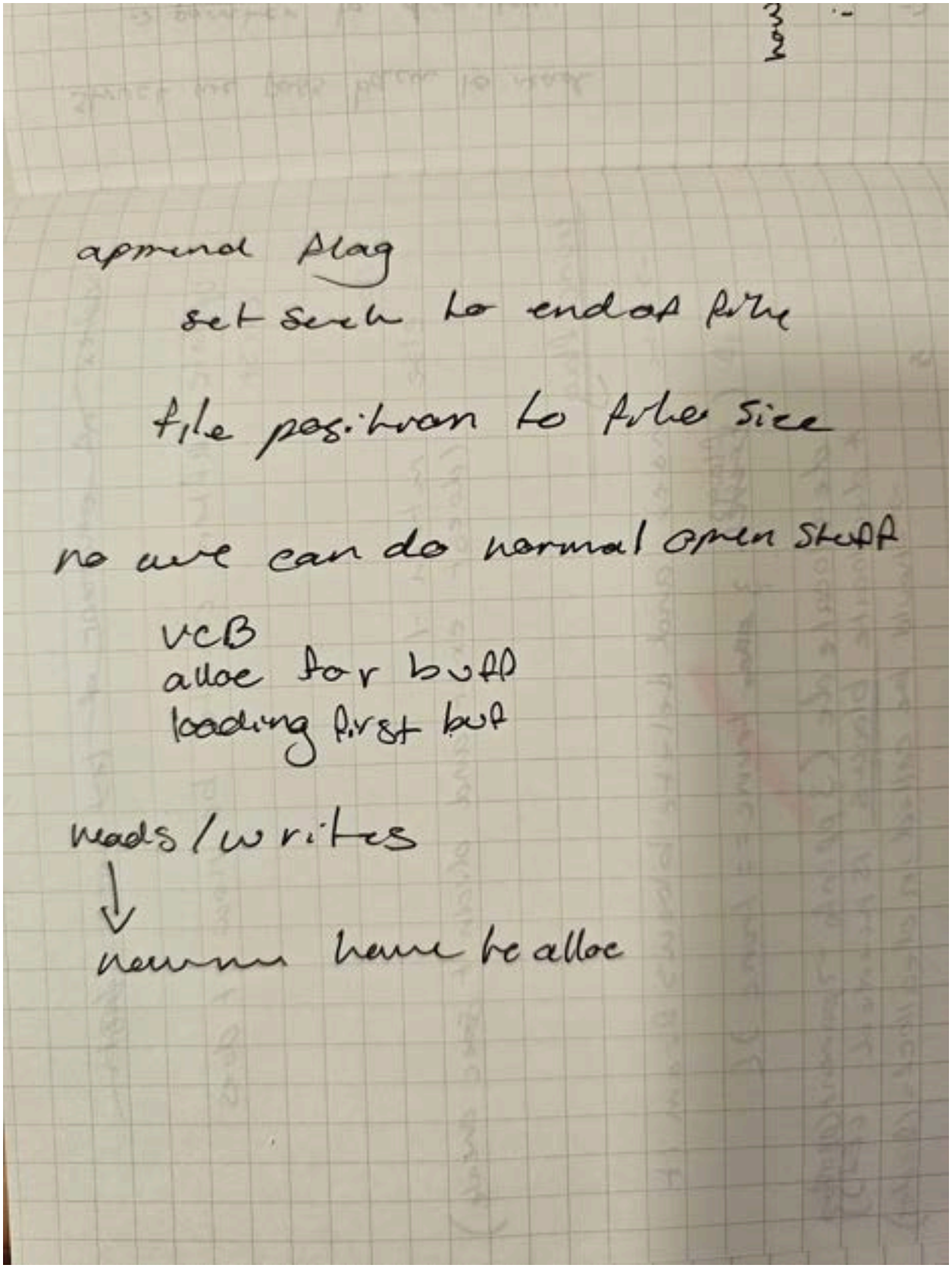
}

~~Read~~

Append

-> sets current file position to end of file
need to load last block into buffer
Set index of block filepositon % blocksize
tells you where in the block you are

B_open planning append flag and the rest of open's duties



append flag

set seek to end of file

file position to file size

now we can do normal open stuff

VCB
alloc for buff
loading first buf

reads/writes

↓

[illegible] have be alloc

More b_open planning trying to understand which flags are which and how to even check a flag

Use linux flags

-> stored ? defined

? Sets

if -1 -> file does not exist

unless create flag is spec.

if (pp inf -> lastelement index = -1) {

if (create flag

how to check flag

if (flags & Create == create){

(true) flag is set

-> find a free de

-> set loc to 0 set size to

> helper funct to create file

(block s 0 and filesize is 0

no location (not -1)

if not

get out

3

create flag done

we set last element index to one you found in find a free de

Some notes and planning for opendir/readdir and also b_open

struct we pass back to read

-> pointer to directory

assign the di to malloc

- call fdd instead of directory

fdd of directory = current directory

if not empty

fill di struct
it l
and loop

fall out of loop it used all entries

-> free everything in the close

delete > get rid of :
call dealloc
set 0 to/0e

-> Start w/ open

1. parse path

2. does it exist

if yes

is it a file?

if not

go away

create flage specifi

Put in info

name set len to

Set date document
date

-> create flag

-> truncate flag
if spec.

Learning about read and write flags for b_open

Load it -> LBA Read (loading the blocks)

Read / write flags

need to go into fcb

00 Read
01 write
10 R/W

mask w/ 11

-> create read/write mask
bits you're going to mask w/

int RWMask = (0 - ROnly or 0 - WriteOnly

int RW Flag = Flags & RWMask

now RWFlag either equal to Read/W/RW

Store this in the fcb
(add it to the fcb)

now to check flag if RW == write only
or RW == 0 - RW

now we know write flag is set

can see if read.

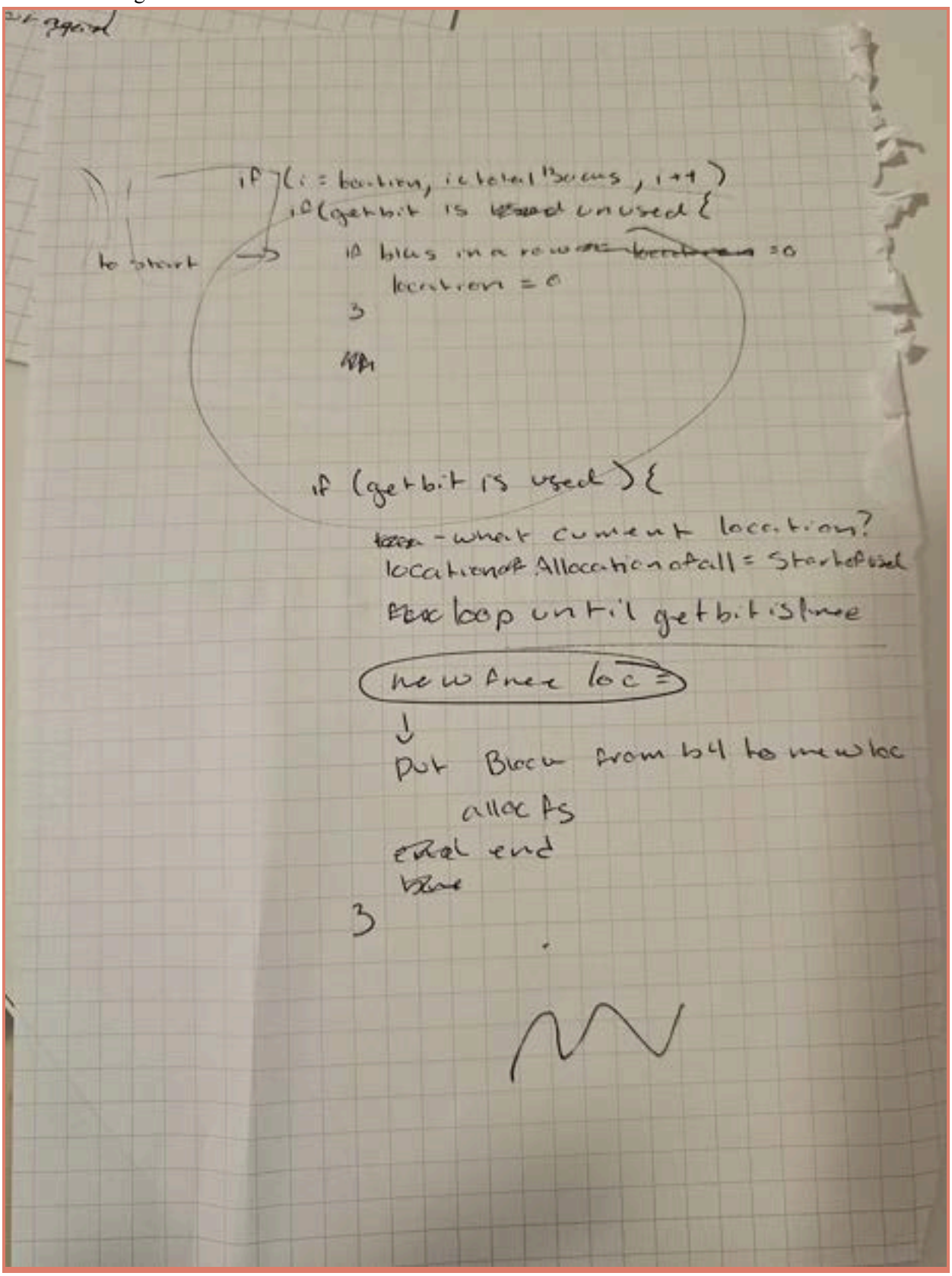
know the parent of (lastelementindex]

loop parent

DE * parent in fcb
Store index of entry
known

3

Growblocks logic



Learning how to create a testfile for b_read

Set size to strlen of string

create our poem

= some text

more than a block

(not more than 2)

Strlen of that blksize-1 / blksize

malloc

Strcpy int but we malloced

LBAwrite to block 500

create a DE * testfile

in rootdir

Find open file

Setname

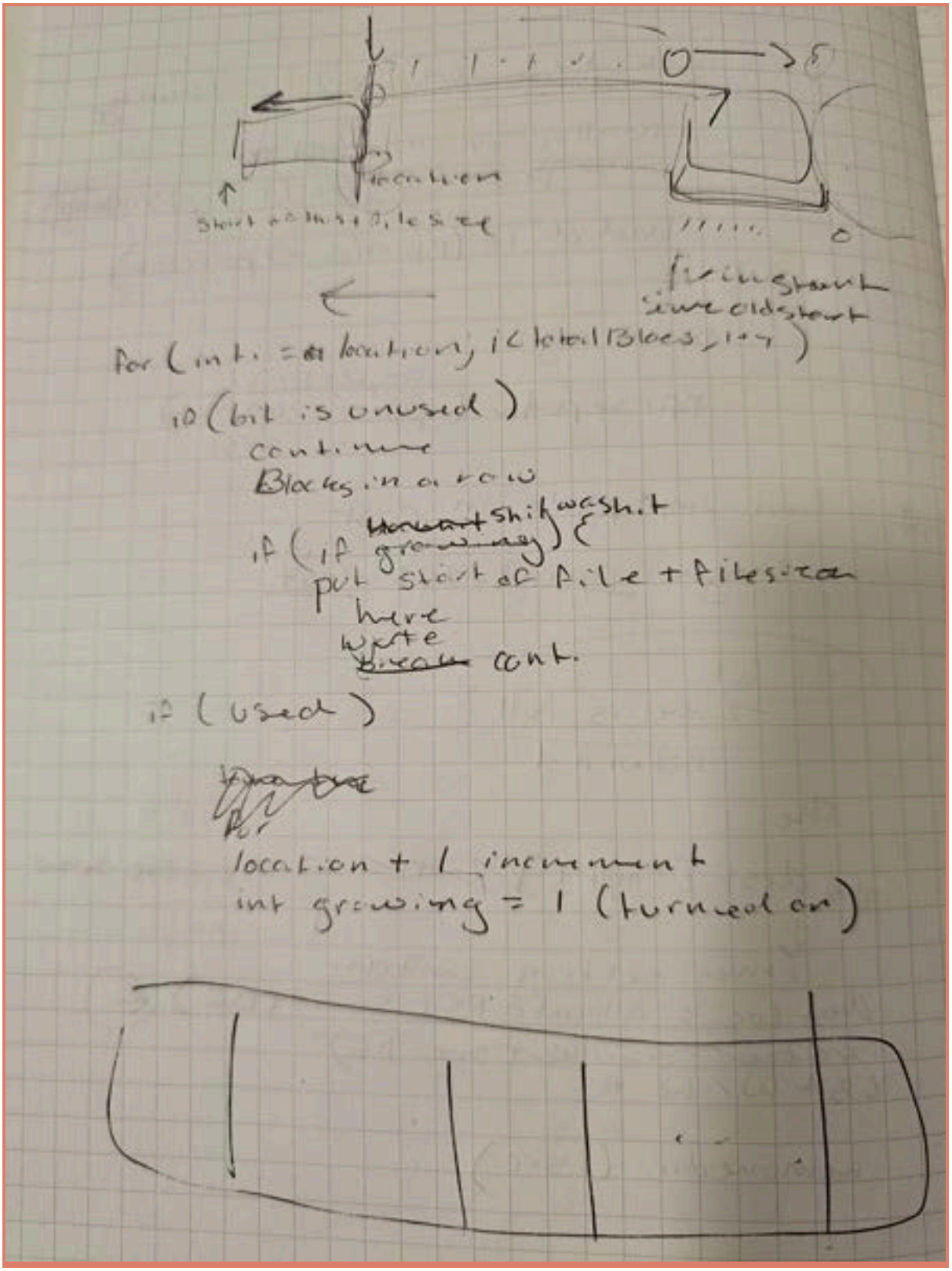
loc = 500

size = len of string

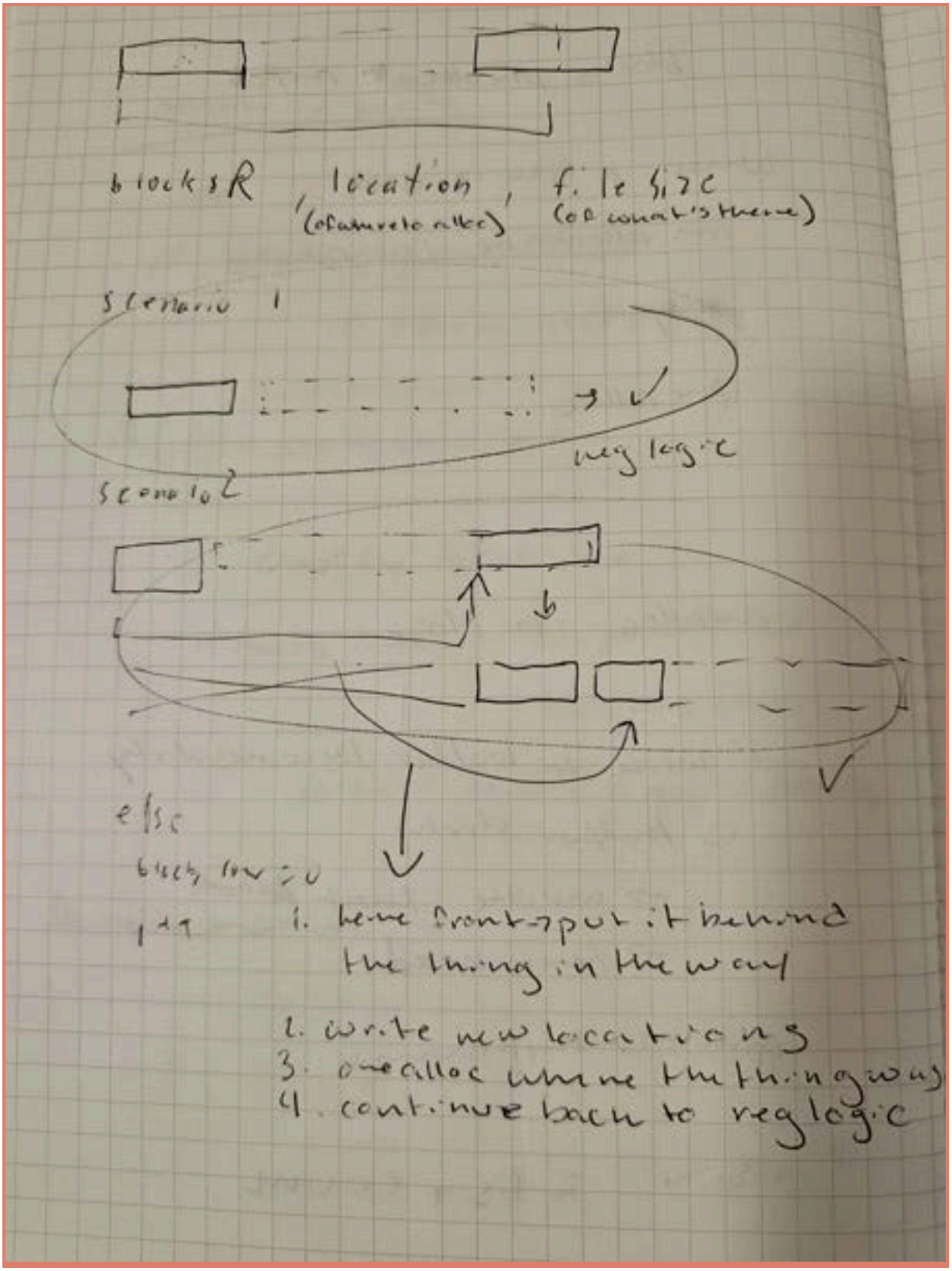
write the root again

type = file

More growblocks logic trying to understand how the function needs to move the file's info in the free space



Even more logic with drawings for grow blocks deciding what variables we need



Description of your file system

To best break down our file system, I wanted to recall the 2 milestones as well as the guidelines given to us in the README. A big help in our project was making sure to always plan out what our priorities were, what to finish first to better grasp the concepts and overall goal of our fully functioning file system. In the first milestone we began by initializing our volume control block (VCB), which contains information about our file system. Our VCB keeps track of the signature, location of the freespace, number of blocks in the volume, the location of the root directory, the size of the root directory, and block size. The free space management system uses a bitmap to keep track of available blocks on the disk. The root directory is initialized to represent a file or subdirectory and is sorted in blocks allocated from free space. We then can move on to milestone 2 which was a big bulk of the work, where we want to create functions to allow us to perform file and directory operations. These consisted of creating directories (fs_mkdir), removing directories(fs_readdir), opening directories (fs_opendir), reading directory entries (fs_readdir) and closing directories(fs_closedir). We also needed to be able to get our current working directory (fs_getcwd) and also set the current directory(fs_setcwd), allowing us the user to navigate through the directory structure. Here is also where we create the entry structure (struct fs_diriteminfo), where we will include the information about each directory entry. After completing milestone 2, we moved onto the I/O operations, which includes the function b_open, which we use to open files using a filename. We have the b_read function to be able to read the data from the opened file. Next we did the b_write function that lets us write data from a buffer to an open file. Our b_seek would be used to move the file pointer associated with an opened file to another location, and finally we would use b_close to close our opened file.

Finally, our file system project successfully manages files and directories within our virtual disk.

Issues you had

Milestone1:

A lot of problems occurred because of how we did not understand how to read the hexdump. We were putting our rootdirectory at block -1 and couldn't figure out where it was in the hexdump. I think we were also writing a lot of things in the wrong places because we weren't properly converting from bytes to blocks. The concept of the bitmap was also a bit foreign how the bits were representative of blocks in our volume.

Milestone2:

Some issues we ran into when we started milestone was that we didn't understand what the ppretdata struct was or how parsepath was even supposed to work. Initially we understood we had to parse the path variable passed in but we didn't get the idea of a struct being passed in to be filled with information. The struct was confusing to us and even after watching the lecture and talking to Professor Bierman in office hours we needed a good day or two to really understand exactly what was going on.

Another problem we faced was that we didn't understand why things worked when we removed the sample volume but didnt after the initial make run. Of course this was because our loads from the volume into memory weren't set up and then weren't working correctly.

We had a large amount of warnings and our setCWD was segmentation faulting and we didn't know why. We had assumed they were unrelated to the current function we were working on and when we asked Professor Bierman he insisted that warnings should be treated as errors and to handle those before continuing. Once we fixed the warnings the segmentation fault went away.

We had to learn how to load free space because we didn't really need to deal with loading the free space until we needed to remove directories and access the map to do so.

We misunderstood how move works initially called remove dir on the source directory and tried to create a new one in the destination directory instead of simply reassigning the source directory.

We had to some issues with *directory, because we weren't sure how to use dirp and di, but simplified the variables

We were not sure how fileType worked, as it would come up as an error, but when ran would run with no issues.

We were iterating through the loop incorrectly, and would not increment.

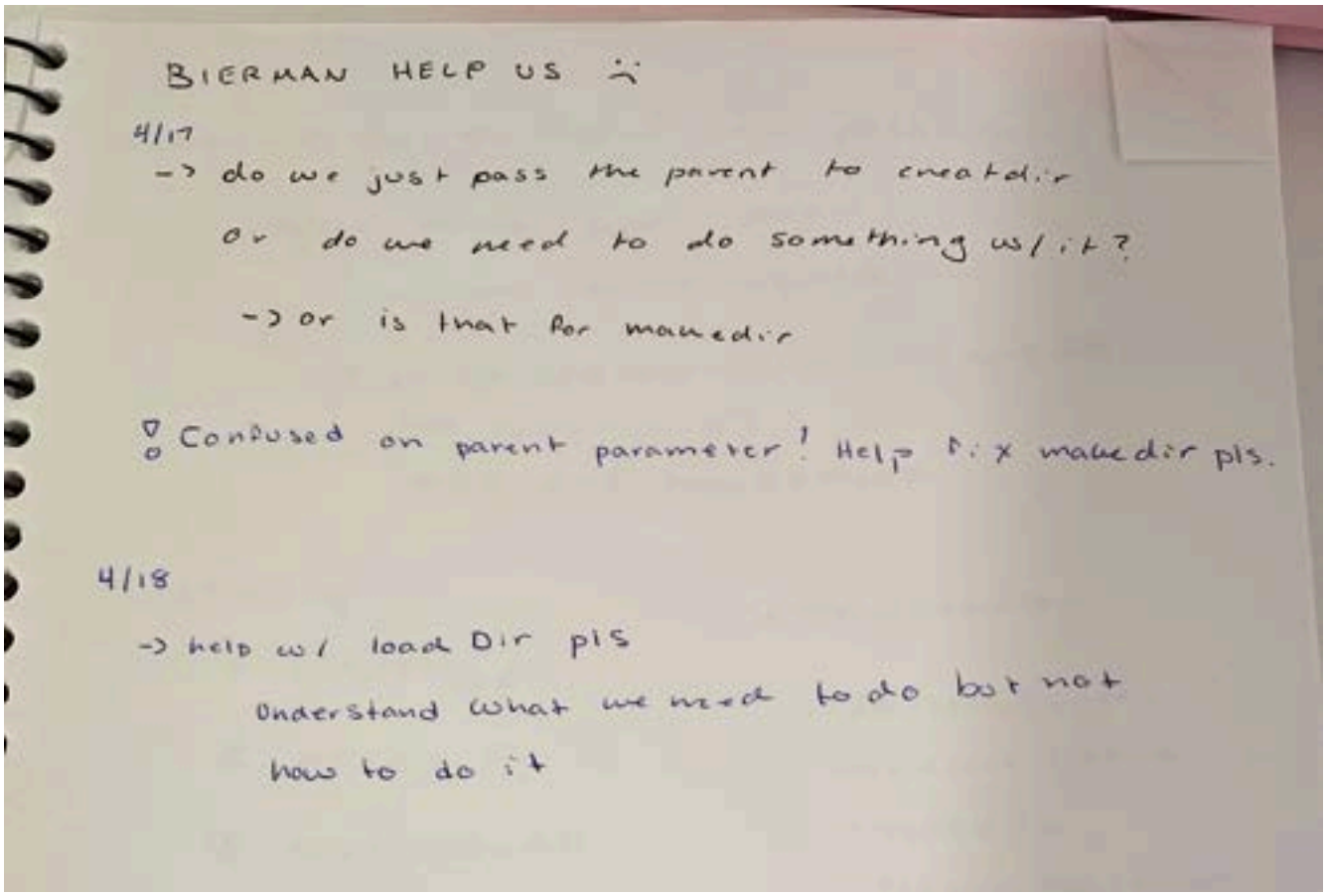
We had a segmentation fault, multiple times, after adding printfs throughout the functions we realized it was the close, and that we were freeing incorrectly.

Milestone3:

A problem we had initially was how flags worked. We had to do logical operations to check if they were there or not. We had a lot of problems while learning how to use buffers. Considering we did not completely understand assignment 5 we had to be very careful changing the read code to the write code. Even though we thought we were careful we still didn't fully

understand how write was different from read so we ended up writing to the users buffer instead of writing from their buffer. We also had trouble testing read and write while not knowing if our errors were from one or the other. So we had to create a dummy file to test read. Then we could printf our way through the read to figure out where it was going wrong. We also weren't checking if our buffers were dirty to be able to check if the last block needed to be written. There are also errors when we copy too much as if the machine runs out of memory after too many copies. We aren't sure why that happens.

Questions we had for bierman



Details of how each of your functions work

b_open

The function used when trying to set a file up for reading, writing and seeking. Can be called with different flags. Will get fileinfo from parsepath then check flags including create which will create the file specified if it does not exist. For Trunc and Append we need to check write and read flags respectively. Truncate will empty the file by deallocating the space in the map and changing the parent info. Append will start at the end of the file and set all the variables tracking location with that knowledge. After checking all the flags this function needs to assign all the information in the fcb for this file and then it returns the file descriptor which will be its index in the fcbArray.

b_seek

Intended to move the index for write or read depending on which flags are specified. Unfortunately, we could not complete this function.

b_read

Will use the file descriptor passed by open to access the file control block for the specific file we are reading. So it's passed in the file descriptor, the users buffer to read to and the amount to read. We then need to account for three different scenarios when reading. We want to fill from our buffer first then if there is still more to fill and it's greater than a block we can lbaRead straight into their buffer in full blocks. Lastly there will be a portion left that is less than a block and that also needs to be put into the buffer. Once these three parts have been taken care of we just add them together to return the amount of bytes read into their buffer.

b_write

Also uses the same file descriptor created by open to access the file. Is passed in a buffer to write from and the amount to write. First we have to make sure there is enough space to write if not make sure to allocate more blocks for the file. Then we write in three parts like when we read. What's in the buffer then if there is enough to write in full blocks then what's less than a block at the end. Returns the bytes read which are all three parts together.

b_close

Function that frees what has not been freed, trims the extra space we preallocated for our write, as well as writes the last block if the buffer is dirty.

fs_mvdir

The move directory function moves one directory into another. It works by being passed in a source pathname and a destination name path name. We call parsepath on both to get a ppretinfo struct for each and then add the source directory to the destination directory in an empty location.

After we set the source directory's location in the parent directory to -1 which is unused. We make sure to write all three directories and return 0 for success.

findInDir

The find in dir function finds a directory inside of it's parent. The parent directory as well as a token which is the name of the directory we are looking for is passed into the function. We calculate the entry count of the parent directory to then loop through the parent directory to find the one which matches the token. Once found, the function returns the index at which it was found. If it is not found, it does not exist and the function will return -1.

loadDir

The load dir function loads a directory from disk. The parent directory of what we want to load is passed into the function and we calculate the size of the directory in blocks as well as malloc some memory for us to load into so that we can perform an LBAread and return the directory.

isDEaDir

The function isDEaDir checks to see if a directory entry is a directory or file. A directory entry is passed in and the boolean parameter of the directory entry is checked to see if the directory entry is a directory or file. If it is a directory it will return 0, otherwise it will return 1.

createDE

The createDE function creates a directory entry. The number of directory entries, block size and parent of the new directory entry is passed into the function. We malloc memory for the new de and initialize all the directory entries inside of it. Next we call allocate free space function which finds free space in our free space map which we can store and write the new entry. The allocate free space returns the location at which freespace was found. We use that to set the location for the new directory we created then initialize the "." and ".." entries in index 0 and 1. Finally we write the directory to disk and return the new directory.

findEmptyDEInDir

The function find empty directory entry in dir finds an empty directory entry inside of a directory. The directory in which we want to find an empty space is passed into the function. We then loop through the entries and check if it is used or unused. When an unused directory entry is found the function will return the index at which it was found. If there are no empty directory entries then the function will return -1.

printDir

This function was used for testing. It works similar to ls but only displays one directory and what is stored in it's parameters.

printFour

This function was used for testing. It works similar to ls but displays only the first four entries in a directory as well as the information stored in their parameters.

isDEEmpty

The function is Directory entry empty check to see if a directory entry is empty. An entry is passed into the function and we load it in order to loop through the directory's entries. We check the location parameter in each index, if it is set to -1 then it is unused. Therefore if we find an entry that is not set to -1, then the directory is not empty and we return -1. Otherwise we return 0, indicating that the directory is empty.

fs_rmdir

The remove directory function removes a directory entry. First we call parse path to get information of the parent directory as well as do checks to make sure what the user wants to remove is a directory and it is empty. In addition, we check to make sure the directory the user wants to use actually exists as well. Next we deallocate the free space that was dedicated to the directory in the free space map by calling deallocFreeSpace. We then set the parameters in the directory that is being removed back to the default and set the location to -1 which indicates it's unused. We then write this update to disk.

initFreeSpace

Init freespace does all the things needed to initialize our free space map. So it mallocs the map and sets the bits used by the map itself and the vcb to used and everything else to unused and then writes the map to the volume.

growFreeSpace

The growFreeSpace function is called only in write when needing to allocate more space for an already allocated file or directory. Since the location is already known we pass in the location of the file and the end location of the file as well as the amount of blocks requested to allocate. We initially operate as if its normal free space allocation starting from the end of the file we are allocating space for but if there is not enough contiguous space after the location of the file we must then find a new location for the file plus the blocks requested and also clear the bits of the previous location associated with the file. This will return the location of the file which will be the same if it didn't need to move or different if it did need to move.

allocFreeSpace

This is the function to call when you need want to allocate free space in the map you pass in the amount of blocks requested and it will iterate through the free space bitmap until it finds enough

contiguous empty blocks then it will shift the bits it found to used and return the location of beginning of the allocated blocks the bitmap \

deallocFreeSpace

This function is passed in a directory entry and go to the location in the entry and will iterate through the bitmap shifts the bits to unused

deallocBlocks

While deallocFreeSpace is passed in a directory entry this function is passed in a location and amount and just iterates through the bitmap and clears those bits associated.

loadFreeSpace

The function to load the free space map which is a global variable from our volume into memory just reads from the location which is stored in the vcb and returns a map pointer.

setBit

A function which is passed in a block number in the volume and set the respective bit in the bitmap to used.

clearBit

A function which is passed in a block number in the volume and set the respective bit in the bitmap to free.

getBitZeroOrOne

A function which is passed in a block number in the volume and return the bit as zero for unused and one for used.

initFileSystem

This function starts by mallocing a VCB pointer then read into that pointer from the first block in volume to initialize the VCB in memory and checks for signature. If it's not there, it initializes VCB info and initializes the rootDir and FS map then updates volume with a write. We store the number of blocks, the signature, the location of the rootDir and the location of FreeSpace as well as the blockSize all in the VCB. If the signature is already there we just load the fs map and root dir and initialize the CWD to root.

exitFileSystem

This functions frees anything that was initialized and not freed in init and frees anything else that needs to be freed as well as writes what hasn't been written.

fs_stat

A function that is passed in a path and structure to fill that holds extra file/directory information about the directory passed in. Calls parsepath which allows us to access all the information about the file to fill the structure. Returns 0 on success.

fs_delete

Takes in a path and deletes the file by reassigning the values in the structure to those of an unused one. Uses parsepath to access the file information. Returns 0 on success

fs_create

Passed in a filename and will create a file. The difference between a file and a dir is that files are empty initially with no location or size and isDir is false. Returns a pointer to the file.

fs_setcwd

For our setcwd, we want to set the current directory, based on our pathname and be able to handle absolute and relative paths. We do this by first allocating memory for our ppretdata struct ppinfo, we then will use parsePath and isDEaDir to check that we are working with a directory. Then we load in the directory linked to the last element in parsePath and set it a temp variable. Now we move on to set our current working directory to our new temp, after we make sure that we are not in the loadedRD, and if we are then free the memory allocated for loadedCWD. Finally, we check if the pathname is an absolute path or a relative one, if absolute will strcpy the pathname to loadedCWDString, if relative will strcat the pathname to loadedCWDString. We would want to return a 0, to show that it has successfully set the current working directory.

adjustPath

This function will trim down paths if there are . or .. in the path. For . is simply deletes it from the path and for .. it removes .. and the entry before. But we have not yet implemented it.

parsePath

Parsepath is a function that we use multiple times, and really makes things easier for us because its purpose is to parse through the pathname and retrieve parent and last element info . We do this by tokinizing the pathname using ‘/’ as a delimiter, and start off by calling for the first token using strtok. Now we want to check for the delimiter, if start with ‘/’ then we set our parent to start parent. We then loop through the rest of the tokens, using the index of the current token behaving in a similar logic to token 1 we now do the same for token 2, if the directory is not the last element, then we update to point to the loaded directory. After looking at all the tokens it will return a 0.

fs_getcwd

This function is small and to the point, all we want to do is grab the current working directory path that is stored in loadedCWDString and copy it into pathname. We also include size to make sure the copied string is no bigger than the size limit to prevent any errors.

fs_mkdir

The purpose of this function is to create new directories within our file system. We want to make sure to properly allocate memory, use parsePath, check for existing directories, then create new directory entries. Then be able to update the parent directory, write these changes to disk, and clean up to avoid any memory issues. Some important helper functions we use are parsePath, createDE, and finEmptyDEinDir, then after we can update the directory entry information, name, location, size, isDir flag, and the date. When writing we also have to make sure to calculate the correct size of the parent directory in terms of blocks, and use LBAwrite to do so.

fs_opendir

The purpose of this function is to open a directory, parse through it, validate the path, allocate memory , and then finally return a file descriptor struct for our opened directory. We accomplish this by first duplicating our pathname and use strdup to make sure we can modify it. Then allocating memory for ppretdata which we will use to hold our information for pathInfo. We also make sure to check that we are on a valid path, using our helper function isDEaDir. Then using loadDir, we load the directory entry to the last element of the path. After we verify these things we can then allocate memory for our fdDir struct, which is our file descriptor for our directory. Initialize the struct, setting the required information, and then returning the fdDir struct.

fs_readdir

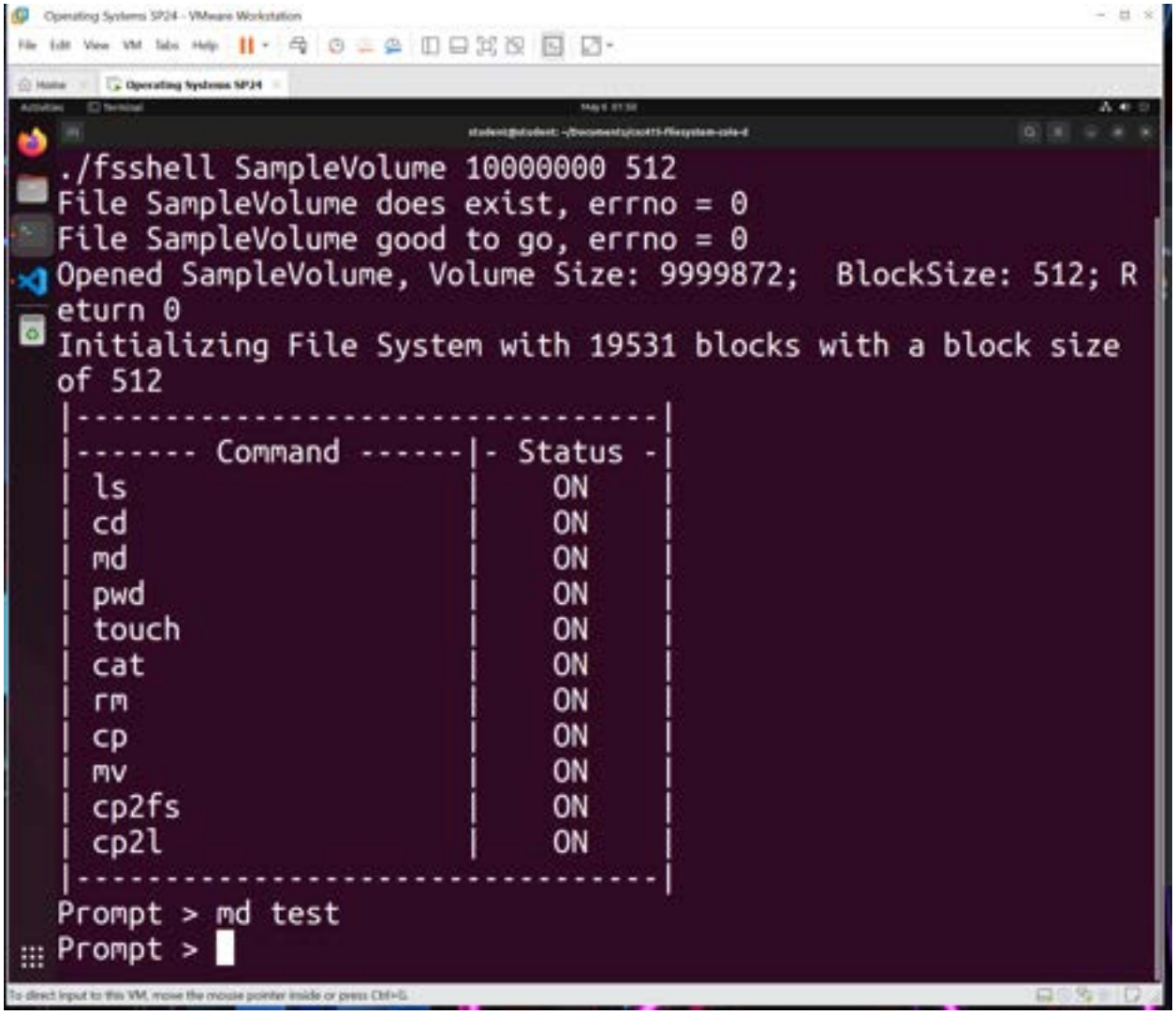
The main function is to be able to read the directory entry by entry, and populate it with information about each entry, until we reach the end. We do this by first checking the number of entries, and then allocating memory to our struct fs_diriteminfo. Then by doing so we can store information about the directory item. Next we want to be able to iterate through the directory starting from dirEntryPosition, checking if the directory entry is not empty, then once it reaches one that has info it will grab that information and populate our diriteminfo struct. We finally want to check for when we reach the end of the directory.

fs_closedir

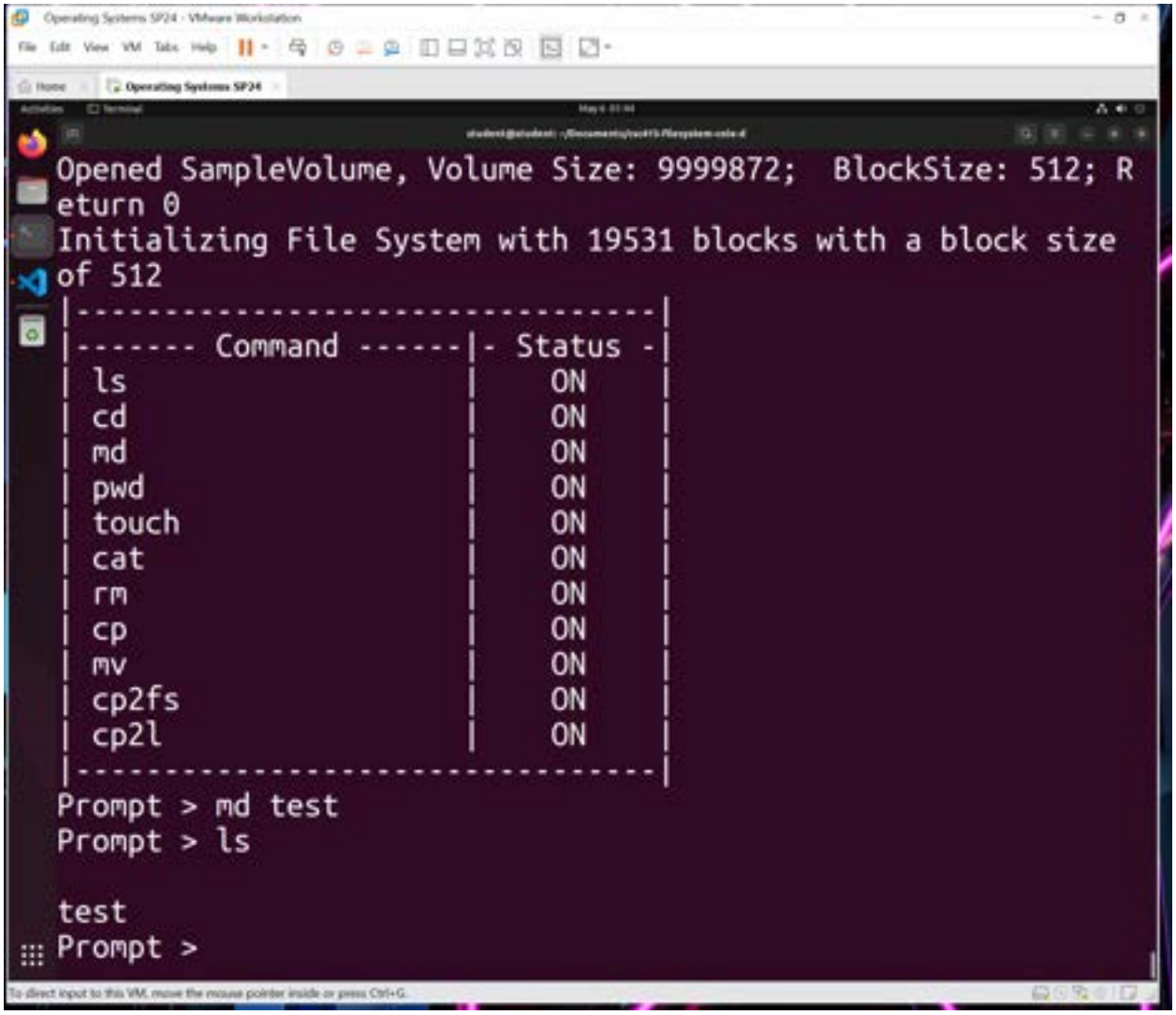
For our close function the goal is to free up the needed memory used in our directory structure, specifically dirp. We do this by checking the di pointer and freeing it, as well as our fdd. We also want to make sure to check that it is not our root directory or current working directory. This ensures that the memory associated with our directory structure and its information is properly deallocated.

Screen shots showing each of the commands listed in the readme

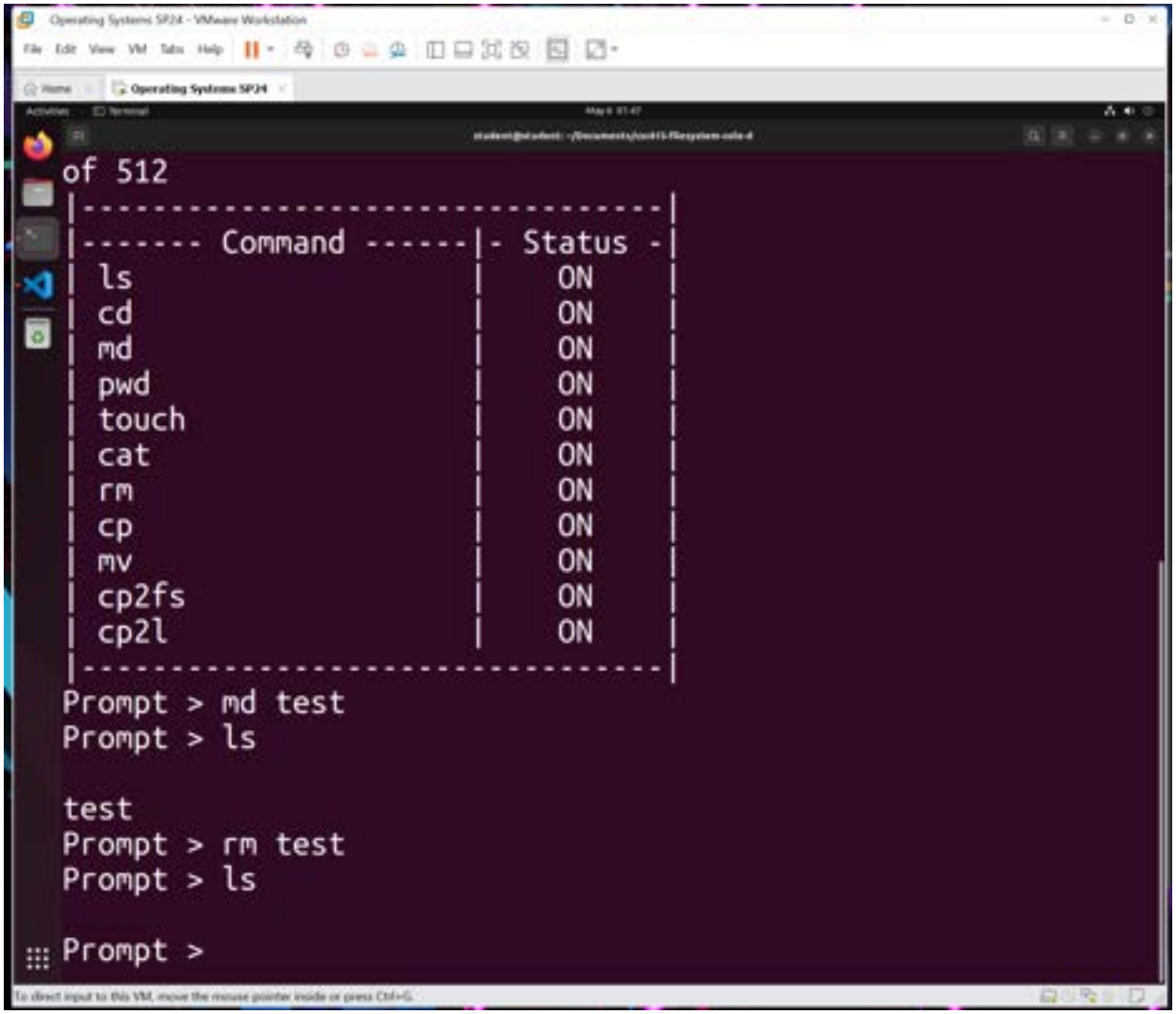
mkdir or md



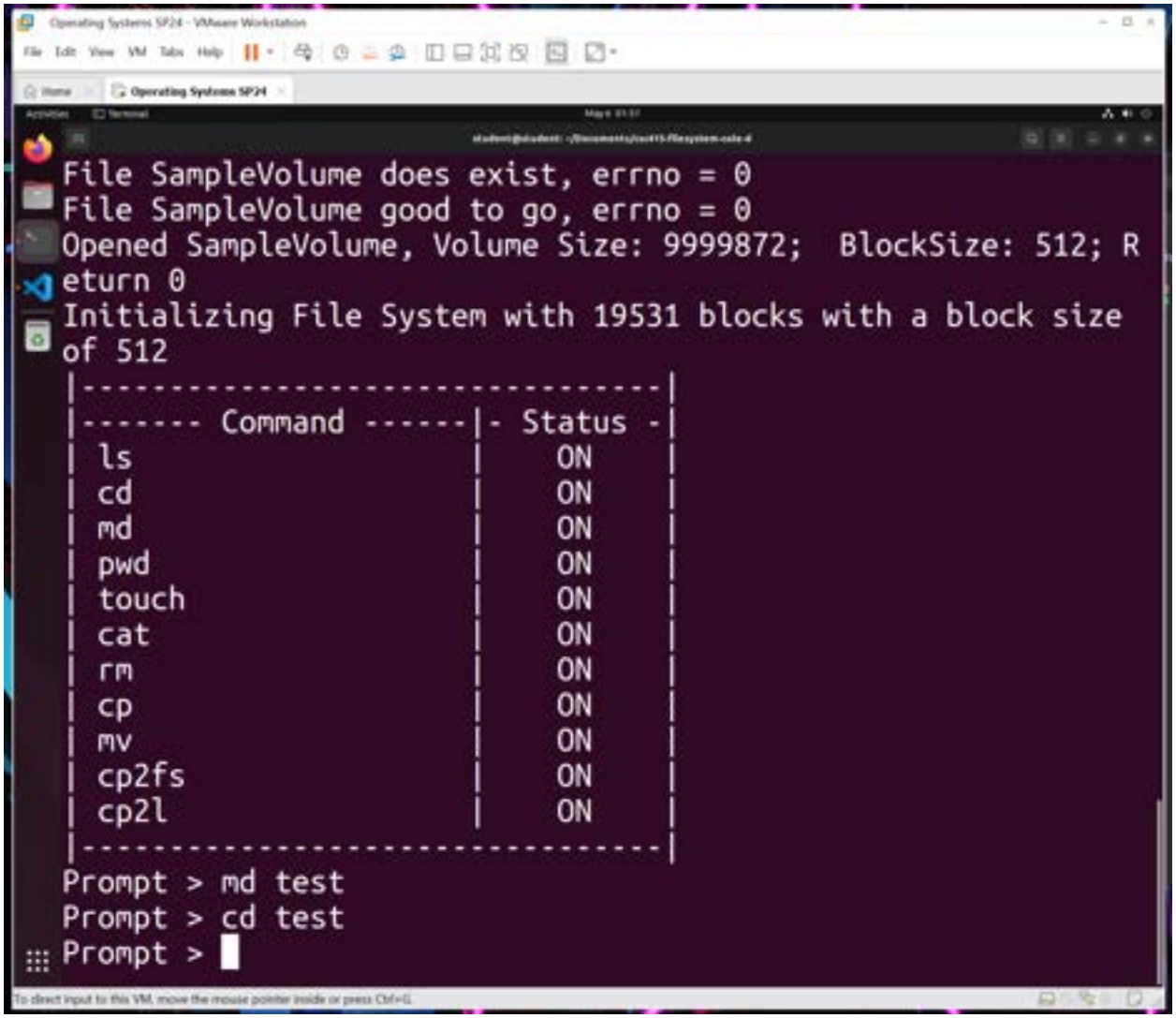
opendir, readir, closedir or ls



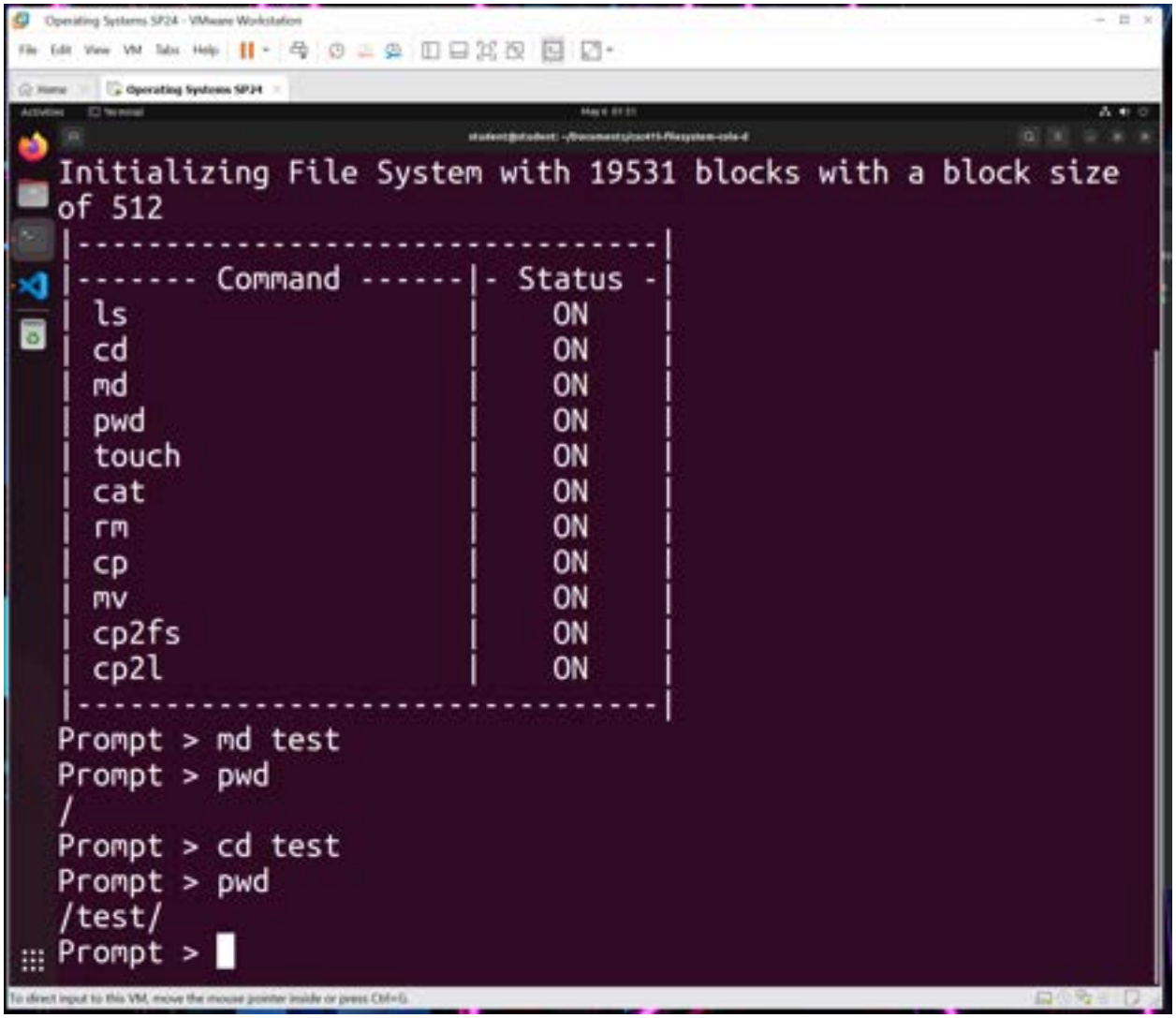
rmdir or rm


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 test
Prompt > ls
test
Prompt > rm test
Prompt > ls
Prompt >

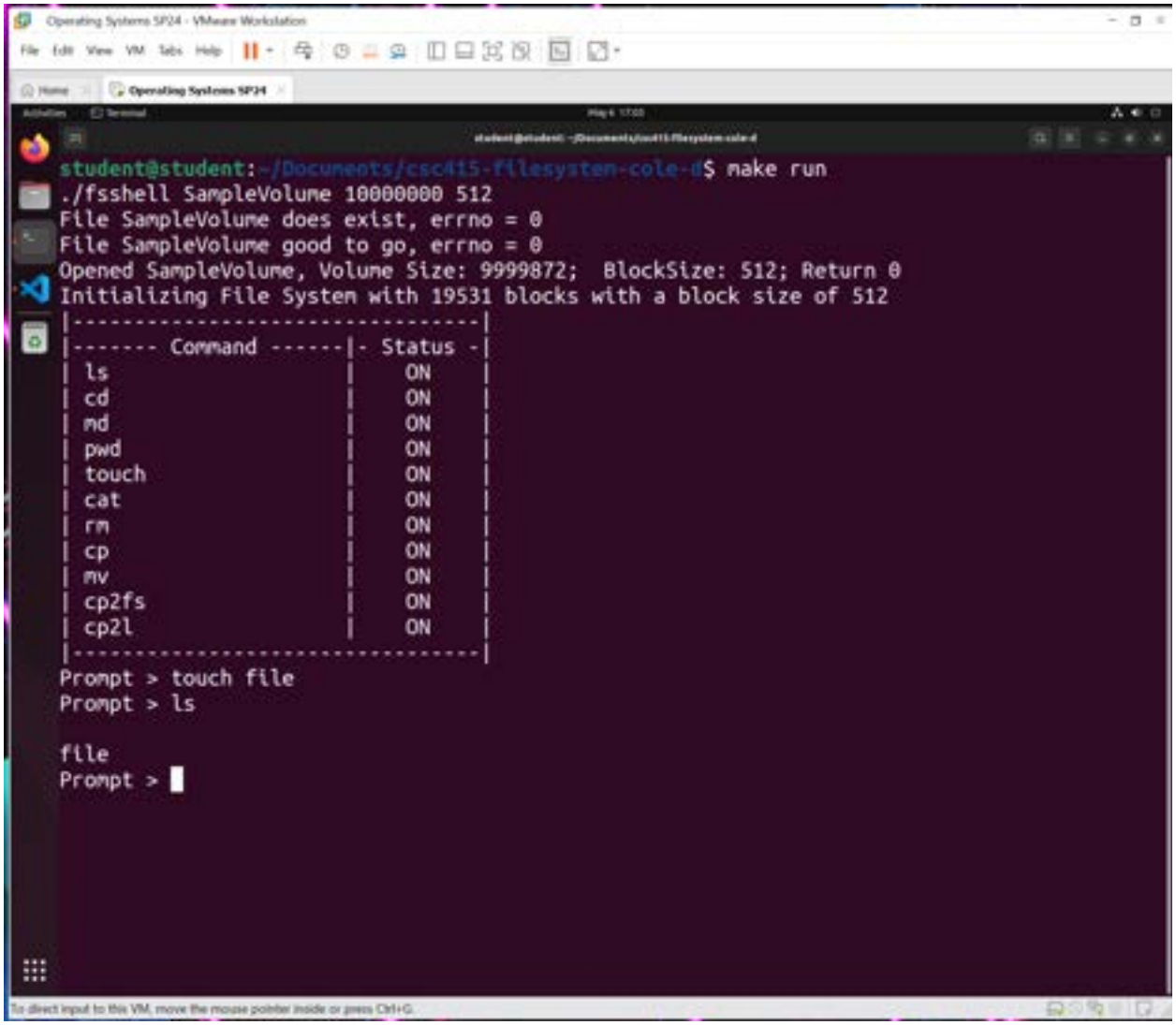
setCWD or cd


File SampleVolume does exist, errno = 0
File SampleVolume good to go, errno = 0
Opened SampleVolume, Volume Size: 9999872; BlockSize: 512; R
eturn 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 test
Prompt > cd test
Prompt >

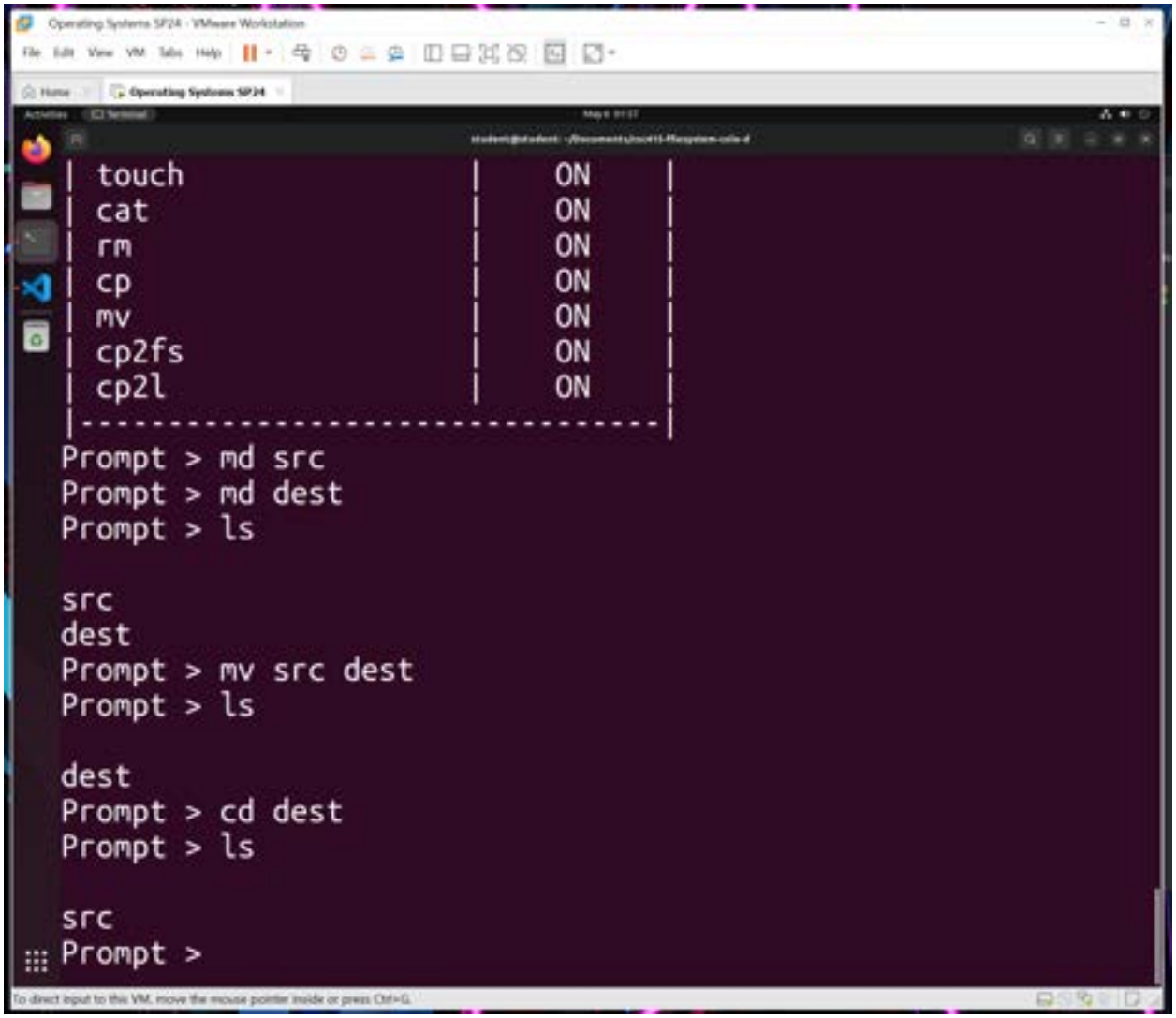
getCWD or pwd



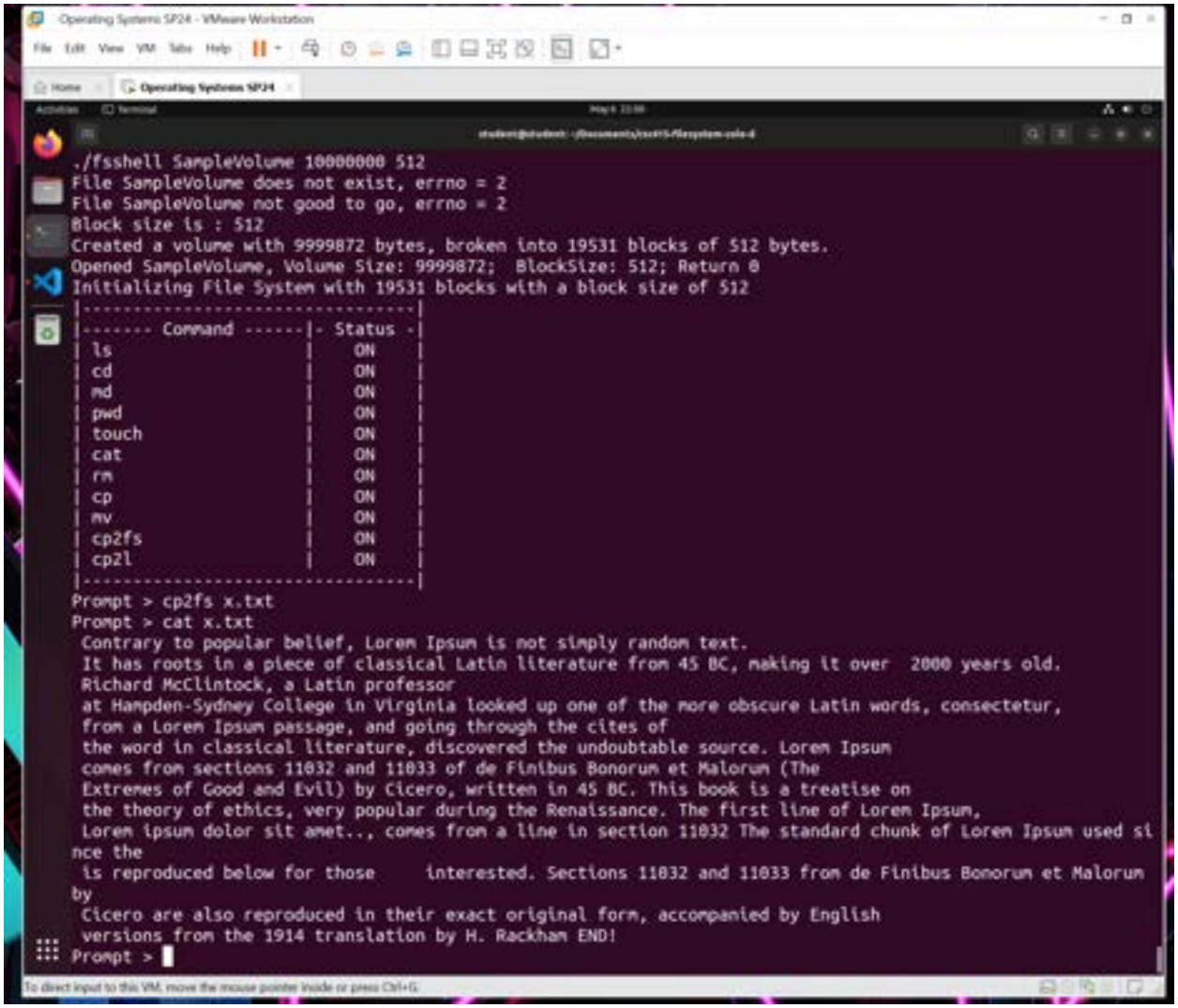
touch


student@student:~/Documents/csc415-filesystem-cole-d$ 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
-------- 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 file
Prompt > ls
file
Prompt >

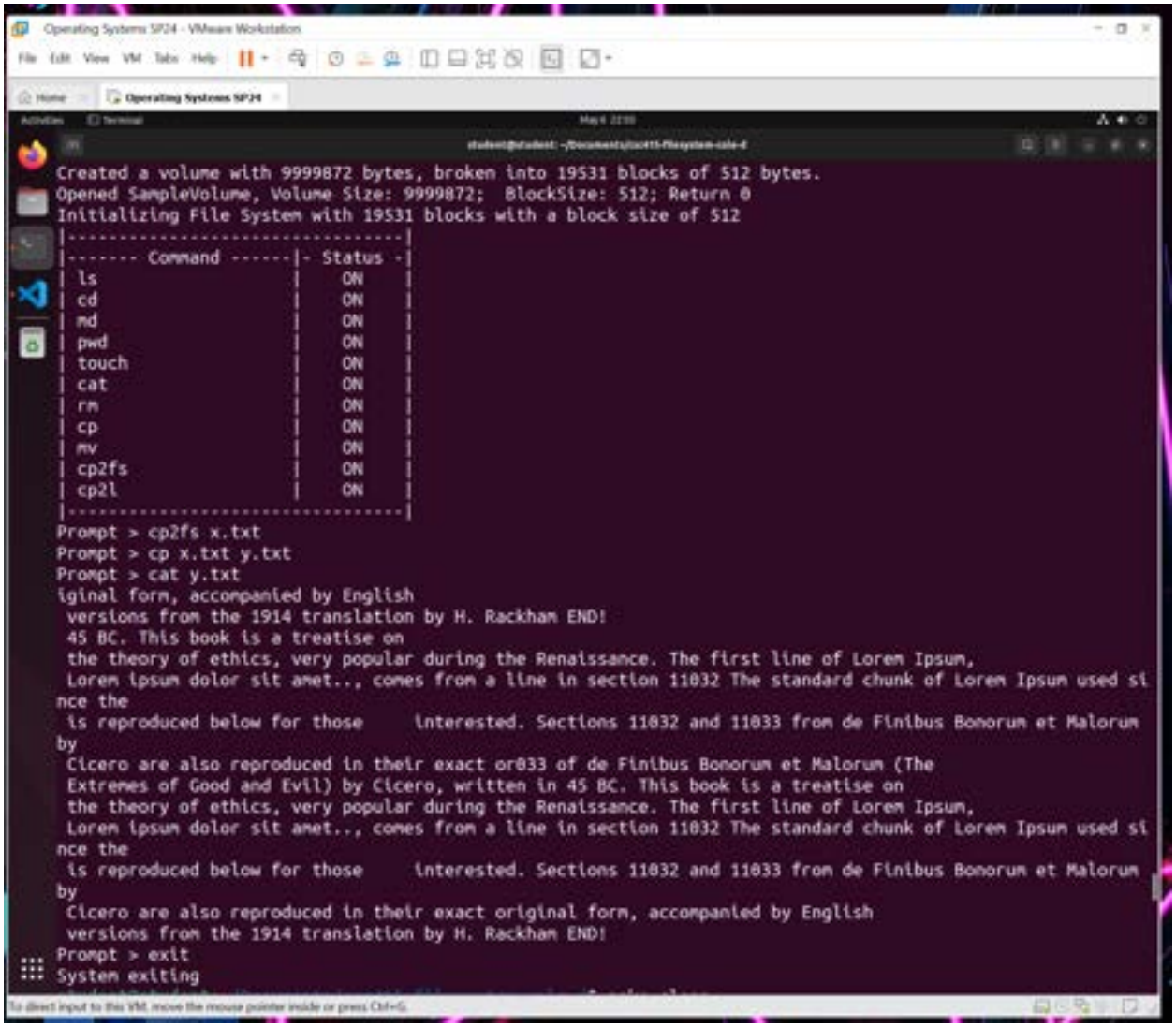
mv


touch	ON
cat	ON
rm	ON
cp	ON
mv	ON
cp2fs	ON
cp2l	ON
Prompt > md src
Prompt > md dest
Prompt > ls
src
dest
Prompt > mv src dest
Prompt > ls
dest
Prompt > cd dest
Prompt > ls
src
Prompt >

cat


./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 > cp2fs x.txt
Prompt > cat x.txt
Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old.
Richard McClintock, a Latin professor
at Hampden-Sydney College in Virginia looked up one of the more obscure Latin words, consectetur,
from a Lorem Ipsum passage, and going through the cites of
the word in classical literature, discovered the undoubtable source. Lorem Ipsum
comes from sections 11032 and 11033 of de Finibus Bonorum et Malorum (The
Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on
the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum,
Lorem ipsum dolor sit amet.., comes from a line in section 11032 The standard chunk of Lorem Ipsum used since the
is reproduced below for those interested. Sections 11032 and 11033 from de Finibus Bonorum et Malorum by
Cicero are also reproduced in their exact original form, accompanied by English
versions from the 1914 translation by H. Rackham END!
Prompt >

cp



Doesn’t work for small files part3 isn’t writing correctly

For larger files though it does seem to have an issue

```
| cp2l                  |    ON    |
|-------------------------------------|
Prompt > cp2fs fsshell.c
Prompt > cp fsshell.c test.c
Prompt > cat test.c
int argcnt, char *argvec[])
        {
#if (CMDCP2FS_ON == 1)
        int testfs_fd;
        int linux_fd;
        char * src;
        char * dest;
        int readcnt;
        char buf[BUFFERLEN];

        switch (argcnt)
                {
                case 2: //only one name provided
                        src = argvec[1];
                        dest = src;
                        break;

                case 3:
                        src = argvec[1];
                        dest = argvec[2];
                        break;

                default:
                        printf("Usage: cp2fs Linuxsrcfile [destfile]\n");
                        return (-1);
                }


        testfs_fd = b_open (dest, O_WRONLY | O_CREAT | O_TRUNC);
        linux_fd = open (src, O_RDONLY);
        do
                {
                readcnt = read (linux_fd, buf, BUFFERLEN);
                b_write (testfs_fd, buf, readcnt);
```

No problems at the bottom of the copied file

```
        while (1)
                {
                cmdin = readline("Prompt > ");
#ifdef COMMAND_DEBUG
                printf ("%s\n", cmdin);
#endif

                cmd = malloc (strlen(cmdin) + 30);
                strcpy (cmd, cmdin);
                free (cmdin);
                cmdin = NULL;

                if (strcmp (cmd, "exit") == 0)
                        {
                        free (cmd);
                        cmd = NULL;
                        exitFileSystem();
                        closePartitionSystem();
                        // exit while loop and terminate shell
                        break;
                        }

                if ((cmd != NULL) && (strlen(cmd) > 0))
                        {
                        he = history_get(history_length);
                        if (!((he != NULL) && (strcmp(he->line, cmd)==0)))
                                {
                                add_history(cmd);
                                }
                        processcommand (cmd);
                        }

                free (cmd);
                cmd = NULL;
                } // end while
        }
Prompt >
```

cp2fs

```
./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 > cp2fs x.txt
Prompt > cat x.txt
 Contrary to popular belief, Lorem Ipsum is not simply random text.
 It has roots in a piece of classical Latin literature from 45 BC, making it over  2000 years old.
 Richard McClintock, a Latin professor
 at Hampden-Sydney College in Virginia looked up one of the more obscure Latin words, consectetur,
 from a Lorem Ipsum passage, and going through the cites of
 the word in classical literature, discovered the undoubtable source. Lorem Ipsum
 comes from sections 11032 and 11033 of de Finibus Bonorum et Malorum (The
 Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on
 the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum,
 Lorem ipsum dolor sit amet.., comes from a line in section 11032 The standard chunk of Lorem Ipsum used si
nce the
 is reproduced below for those     interested. Sections 11032 and 11033 from de Finibus Bonorum et Malorum
by
 Cicero are also reproduced in their exact original form, accompanied by English
 versions from the 1914 translation by H. Rackham END!
Prompt >
```

cp2l

```
gcc -o fsshell fsshell.o fsInit.o directory.o freeSpace.o mfsCWD.o mfsORC.o mfs.o b_io.o
fsLow.o -g -I. -lm -l readline -l pthread
./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 > cp2fs x.txt
Prompt > cp2l x.txt x.txt
Prompt > ls -l

-         1121   x.txt
Prompt > exit
System exiting
student@student:~/Documents/csc415-filesystem-cole-d$ ls -l x.txt
-rw-rw-r-- 1 student student 1121 May  6 22:09 x.txt
student@student:~/Documents/csc415-filesystem-cole-d$
```