MP6: Primitive Device Driver

Joshua Clapp

UIN: 930004089

CSCE410: Operating System

Assigned Tasks

Main: Completed.

Bonus Option 1: Mirror Disk

completed

Bonus Option 2: Interrupts

completed

Bonus Option 3: Thread design

completed

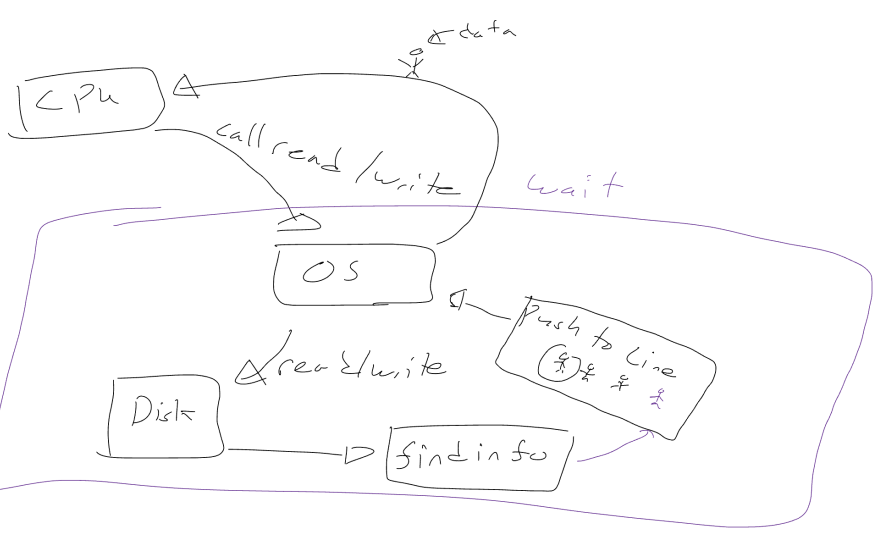
Bonus Option 4: Thread implementation

completed

System Design

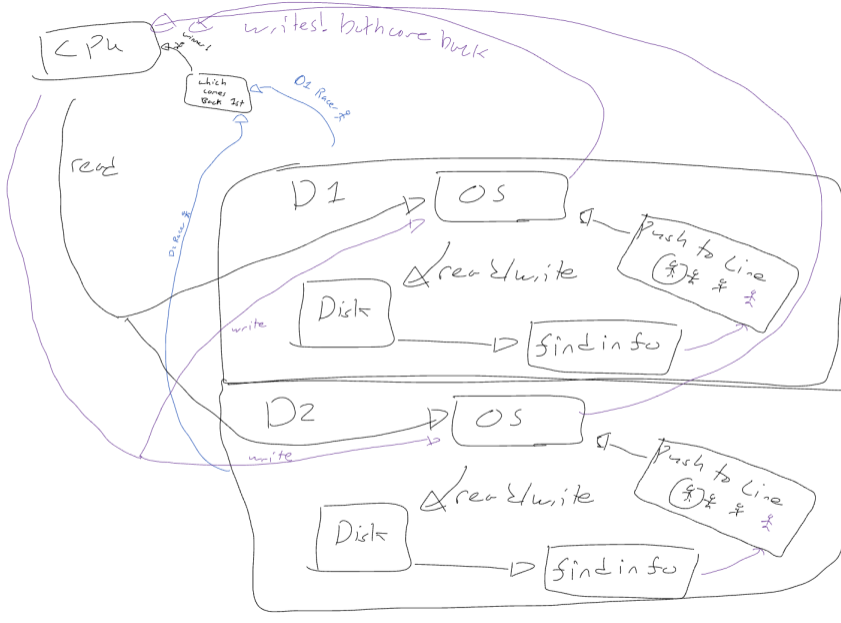
**Main:**

Wait for the read or write call to finish and return to the CPU



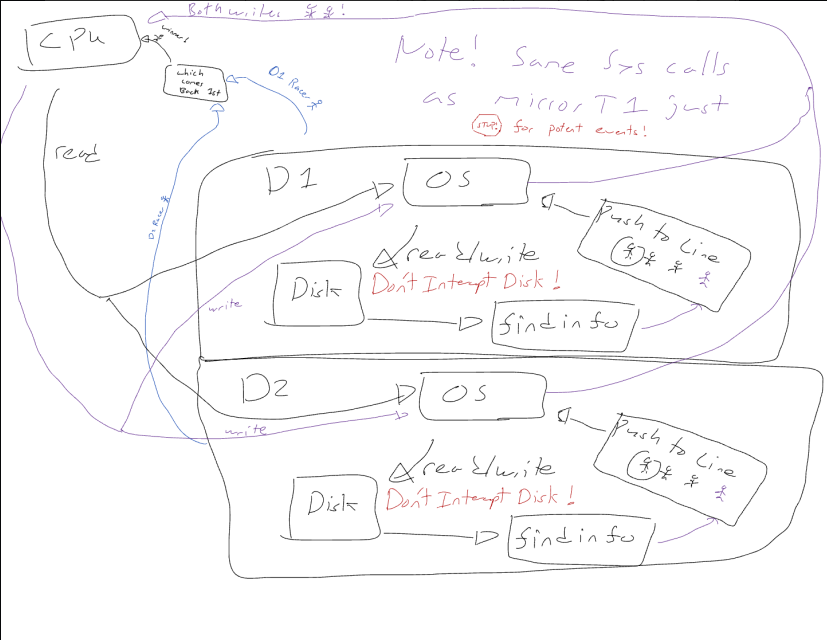
**Bonus Option 1: Mirror Disk**

First disk to finish read gets the prize, both disks must finish write



**Bonus Option 2: Interrupts**

Don’t let anything interrupt while reading or writing, check the disk when something is ready

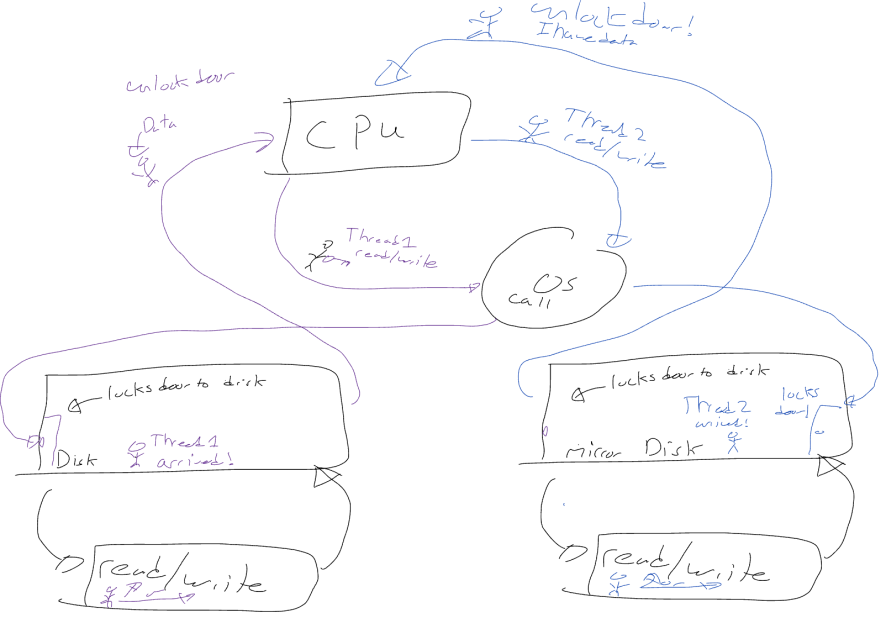


**Bonus Option 3: Thread design**

To make this thread safe, I will use a lock based system to handle the requests for the IO operations serially. This allows a relatively simplistic implementation as it is the normal read and write calls but with a change making the operation atomic.

**Bonus Option 4: Thread implementation**

This is the implementation of bonus 3 using a lock based mux system.



Code Description

**Main:**

I changed blocking\_disk.C, blocking\_disk.H, MirroredDisk.C, MirroredDisk.H, and Kernel.C. For the main assignment, I modified blocking\_disk.C and modified the wait read command.

**To run this logic**

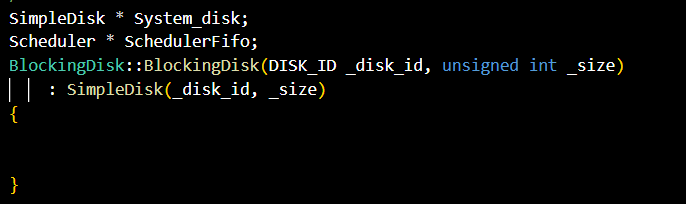
Enable #define BLOCKING\_DISK in kernel.C

This can be control F with Main Task

Make and rerun it

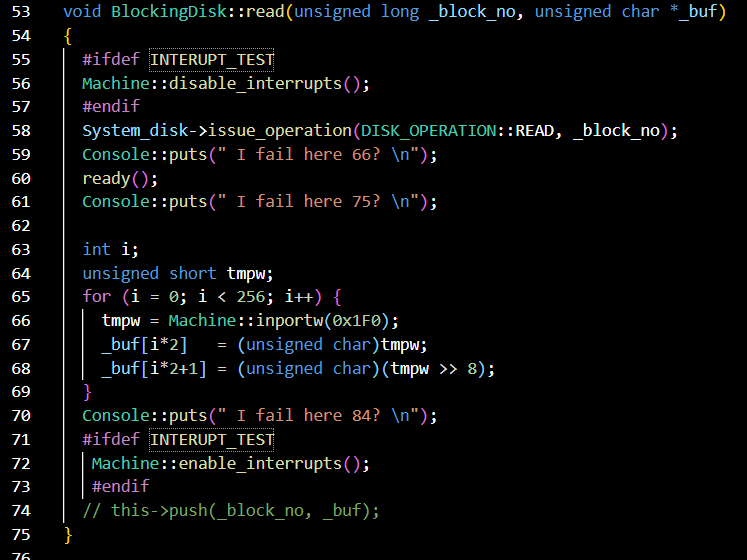
blocking\_disk.C:BlockingDisk(DISK\_ID \_disk\_id, unsigned int \_size):

This initializes our blocking disk calling the SimpleDisk constructor.



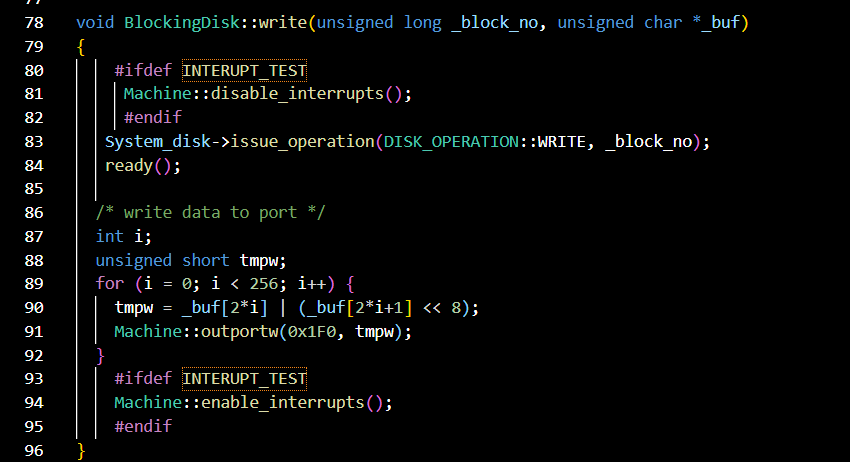
blocking\_disk.C:read(unsigned long \_block\_no , unsigned char \* \_buf):

The only change is to check using the ready() that is a FIFO scheduling operation. The ready() operation calls the SimpleDisk::is\_ready() operation and pushes into the scheduler when finished.



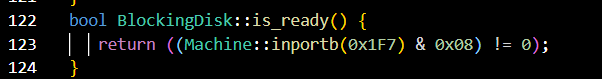
blocking\_disk.C:write(unsigned long \_block\_no , unsigned char \* \_buf):

The main change from Simple Disk is when the OS is ready() calls the is\_ready() operation from the simple disk class. This then pushes the thread into the FIFO queue and pops when it is finished.



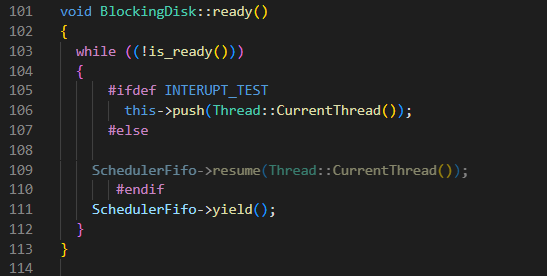
blocking\_disk.C:is\_ready():

This is a copy of the simple disk class as calling is\_ready from a simpleDisk pointer gave me a headache. It checks if the OS has finished its operation.



blocking\_disk.C:ready():

This calls the is\_ready() operation and then pushes the thread into the scheduler, yielding when it is complete.



Bonus 1: Mirror Disk

I wrote a new file MirroredDisk.C, and called the read operation twice on the first blocking disc, returning when the first disk returns. The write operation calls both disks, master and slave.

**To run the logic:**

Disable #define BLOCKING\_DISK in kernel.C

This can be control F with Main Task

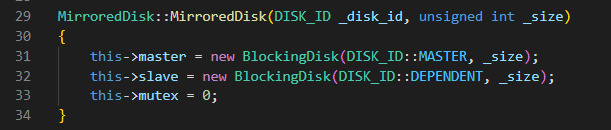
Enable #define MIRROR\_TEST in kernel.C

This can be control F with Task 1

Make and rerun it

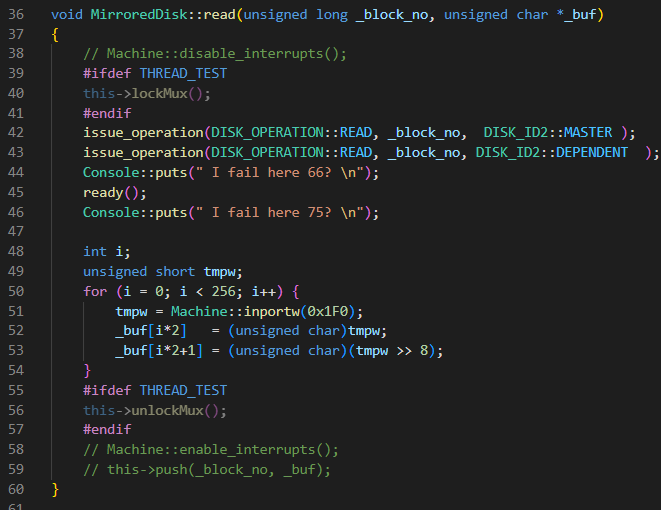
MirroredDisk.C::MirroredDisk(DISK\_ID \_disk\_id, unsigned int \_size):

This creates a master and slave to call disk IO operations.



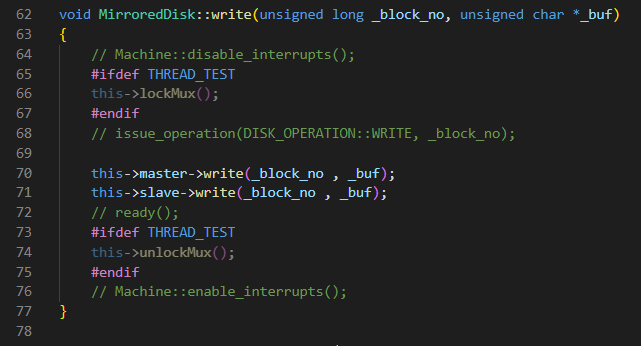
MirroredDisk.C::read(unsigned long \_block\_no, unsigned char \* \_buf):

This is derived from the blocking\_disk::read(unsigned long \_block\_no, unsigned char \* \_buf). It issues a read call to both master and slave, then checks ready(), whichever one finishes first gets the next thread and then runs the read IO call using the machine class.



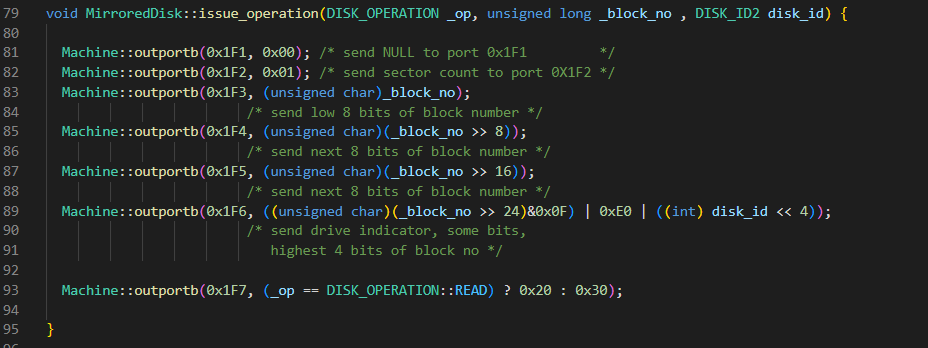
MirroredDisk.C::write(unsigned long \_block\_no, unsigned char \* \_buf):

This is derived from the blocking\_disk::write(unsigned long \_block\_no, unsigned char \* \_buf). It calls the write operation on both master and slave, they both must write to complete.



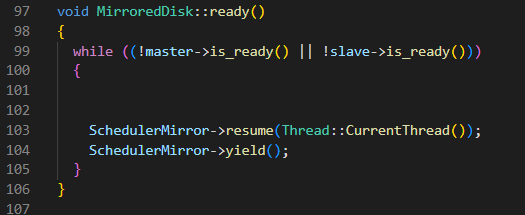
MirroredDisk.C::issue\_operation(DISK\_OPERATION \_op, unsigned long \_block\_no , DISK\_ID2 disk\_id)

This is almost a copy from the simple disk class, the only change is passing in a hard copy of the 0 for master, or 1 for dependent/ slave. This allows us to have a race for which disk finishes the read operation first.



MirroredDisk.C::ready():

This is derived from blocking\_disk.C::ready(). The only change is this function returns when either blocking disk, master or slave, finishes running.



Bonus 2: Interrupts

I added disabled interrupts and enabled interrupts in blocking\_disk.C and in kernel.C added an interrupt register handler.

**To run the logic:**

Disable #define BLOCKING\_DISK in kernel.C

This can be control F with Main Task

Enable #define MIRROR\_TEST in kernel.C

This can be control F with Task 1

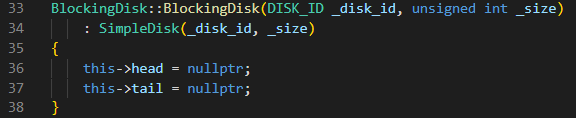
Enable #define INTERUPT\_TEST in blocking\_disk.h

This can be control F with Task 2

Make and rerun it

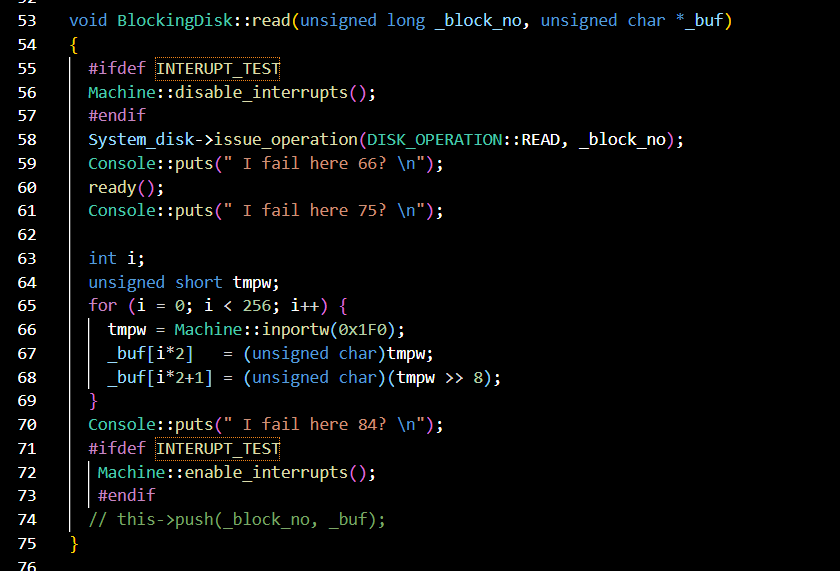
blocking\_disk.C:BlockingDisk(DISK\_ID \_disk\_id, unsigned int \_size):

This initializes our blocking disk calling the SimpleDisk constructor, and creates the head and tail for the scheduling system.



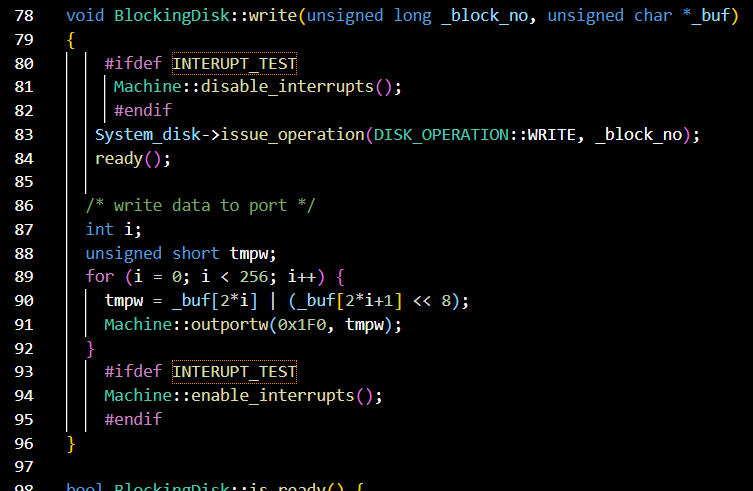
blocking\_disk.C:read(unsigned long \_block\_no , unsigned char \* \_buf):

This is the same as the main task defined before, and for the bonus I added disabling interrupts and enabling at the end.



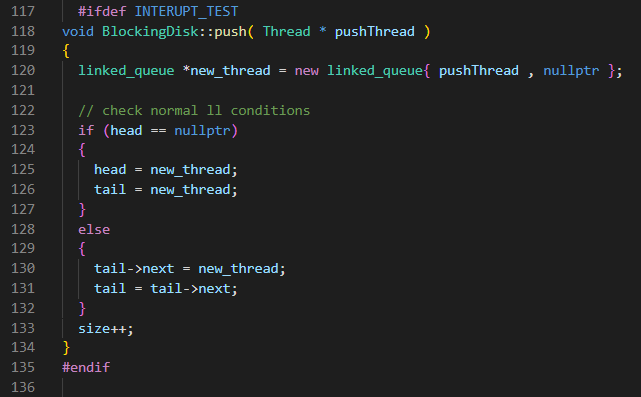
blocking\_disk.C:write(unsigned long \_block\_no , unsigned char \* \_buf):

This is the same as the main task defined above. The only difference is disabling interrupts, performing the operation, and enabling them at the end.



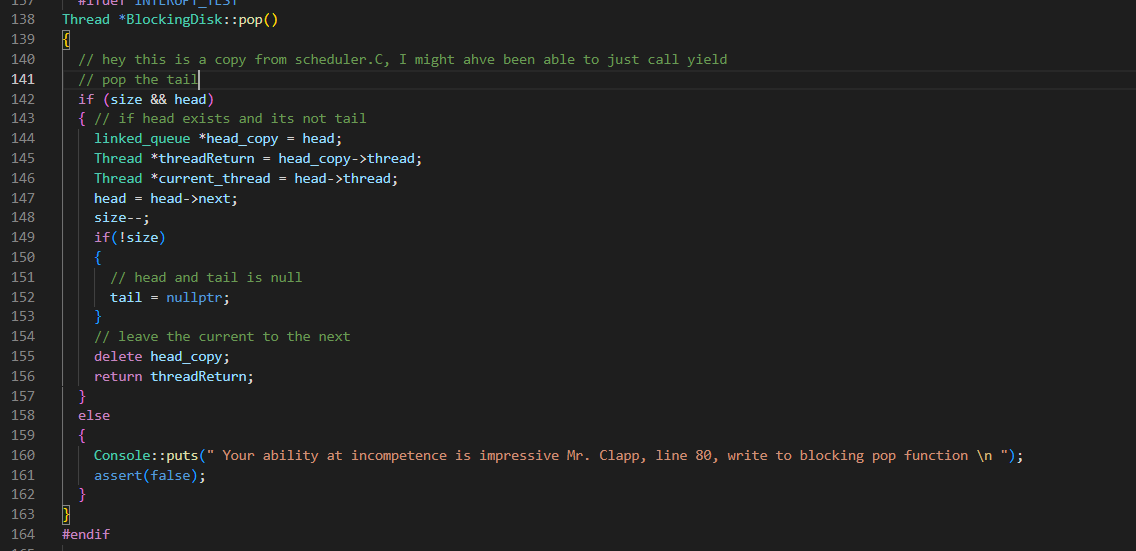
blocking\_disk.C:push(Thread \* pushThread):

This operation creates a new thread in my linked queue class and pushes it into the ready queue. This function was derived from my Scheduler class



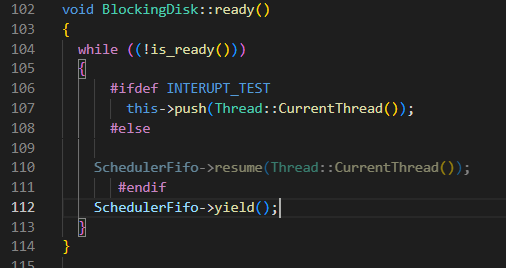
blocking\_disk.C:pop( ):

This operation pops a thread from the ready queue and shifts to the next thread. It was derived from my Scheduler class.



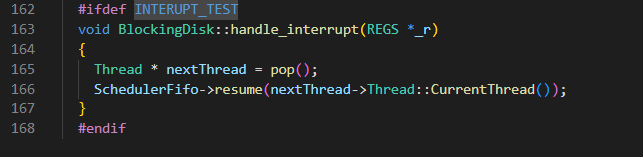
blocking\_disk.C:ready():

This uses the SimpleDisk::is\_ready() that checks the machine code. When interrupts are enabled, we poll for less time and push into our own ready queue, only popping when an interrupt occurs.



blocking\_disk.C:handle\_interrupt(REGS \* \_r):

This takes and handles the interrupt shifting to the next thread.



Bonus 3 & 4: Thread Implementation

I created a lock in mirrordisk.C and locked while reading and writing, unlocking after.

Credit to: <https://courses.engr.illinois.edu/cs241/sp2012/lectures/23-inside_sem.pdf> for implementation.

**To run the logic:**

Disable #define BLOCKING\_DISK in kernel.C

This can be control F with Main Task

Disable #define INTERUPT\_TEST in blocking\_disk.H

This can be control F with Task 2

Enable #define MIRROR\_TEST in kernel.C

This can be control F with Task 1

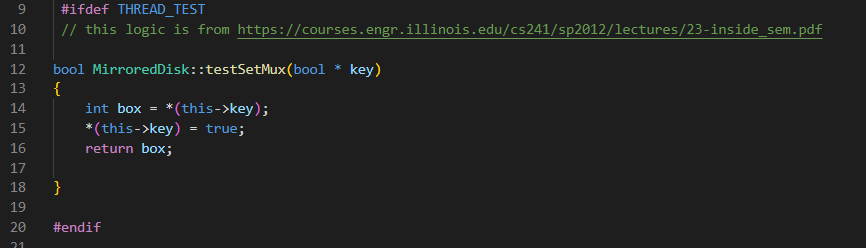
Enable #define THREAD\_TEST in MirrorDisk.C

This can be control F with Task 4

Make and rerun it

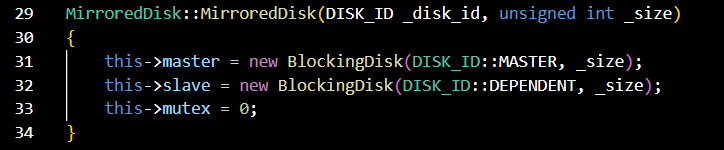
MirroredDisk.C:testSetMux():

Return the old value and update the mux value, this is used in a busy wait to protect the critical sections.



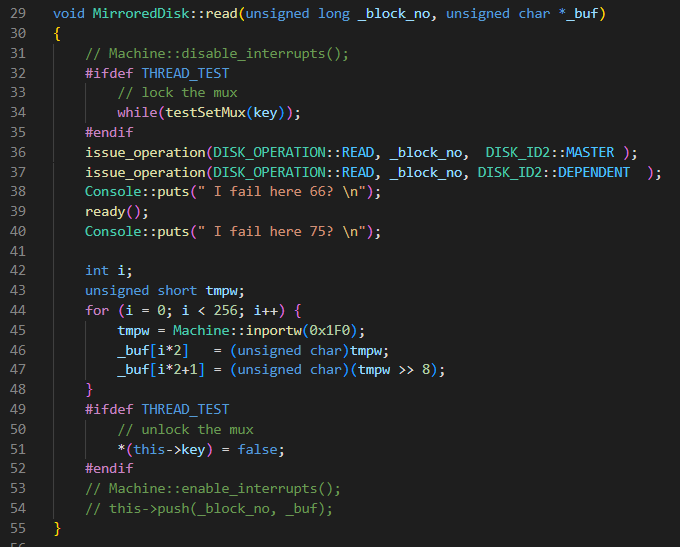
MirroredDisk.C:MirroredDisk(DISK\_ID \_disk\_id, unsigned int \_size):

Create the master and slave threads, set the mux to unlock.



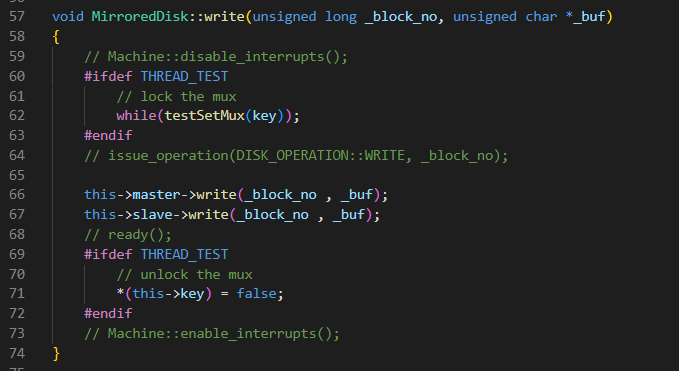
MirroredDisk.C:read(unsigned long \_block\_no, unsigned char \* \_buf):

Lock a thread in the critical section using a busy wait, and perform the read operation described before in Task 1. This is where both master and slave disks are trying to read at the same time and the first to return ends the operation. Then unlock the mux and proceed as normal.



MirroredDisk.C:write(unsigned long \_block\_no, unsigned char \* \_buf):

Lock a thread in the critical section with a busy wait, and perform the write operation to both disks and unlock when the critical section is over.

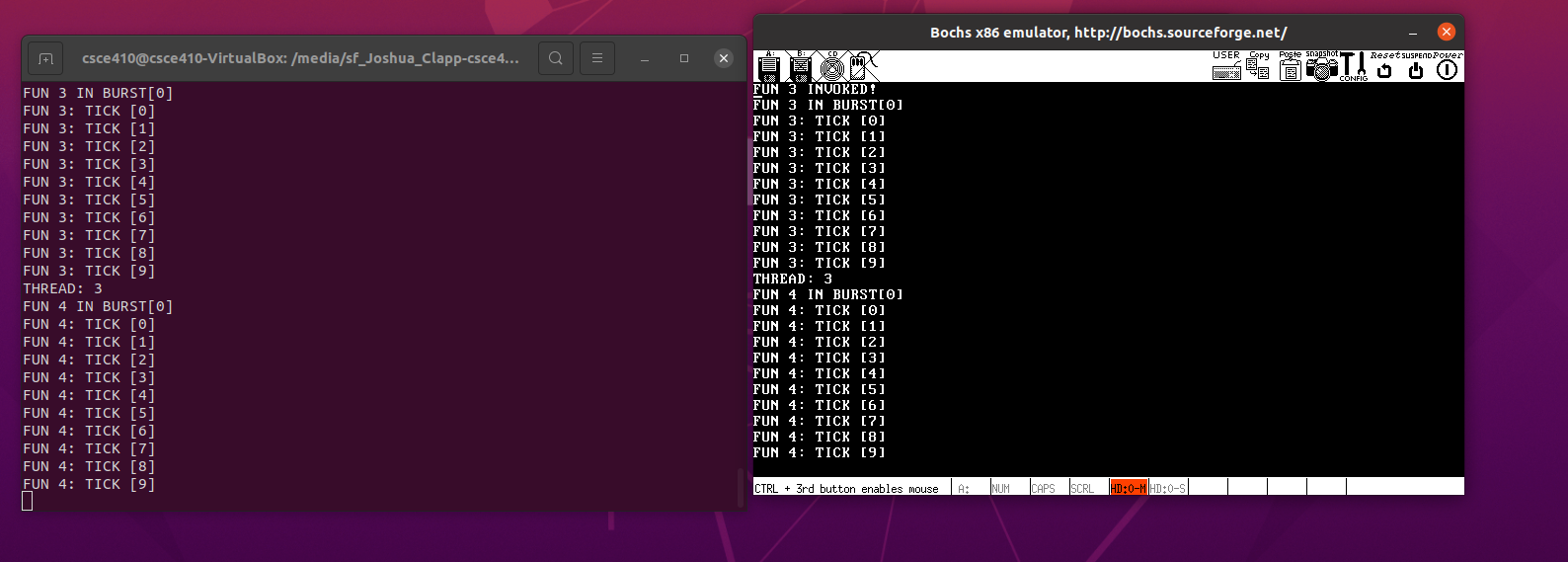


Testing

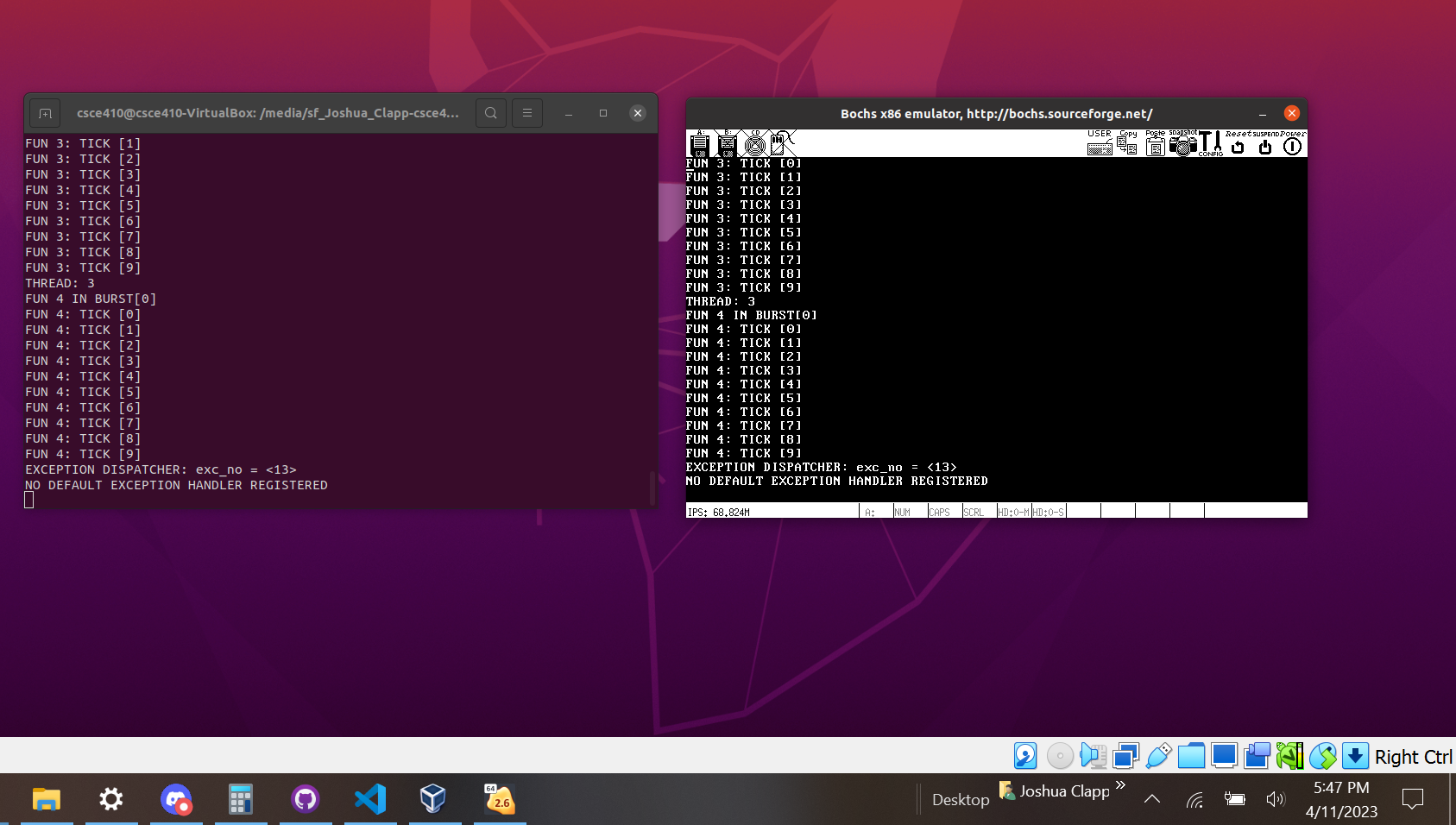
I relied on the test cases provided. I added nothing, I am ignoring optimizing by using busy wait or what happens if both disks interrupt and fail to write or read. My coverage using the testing given is rather simplistic and limited with no specific edge case targeting.

There is a bug in my base code that causes a panic message to appear at the end of running, however, talking with the TA’s it appears my logic works correctly otherwise.

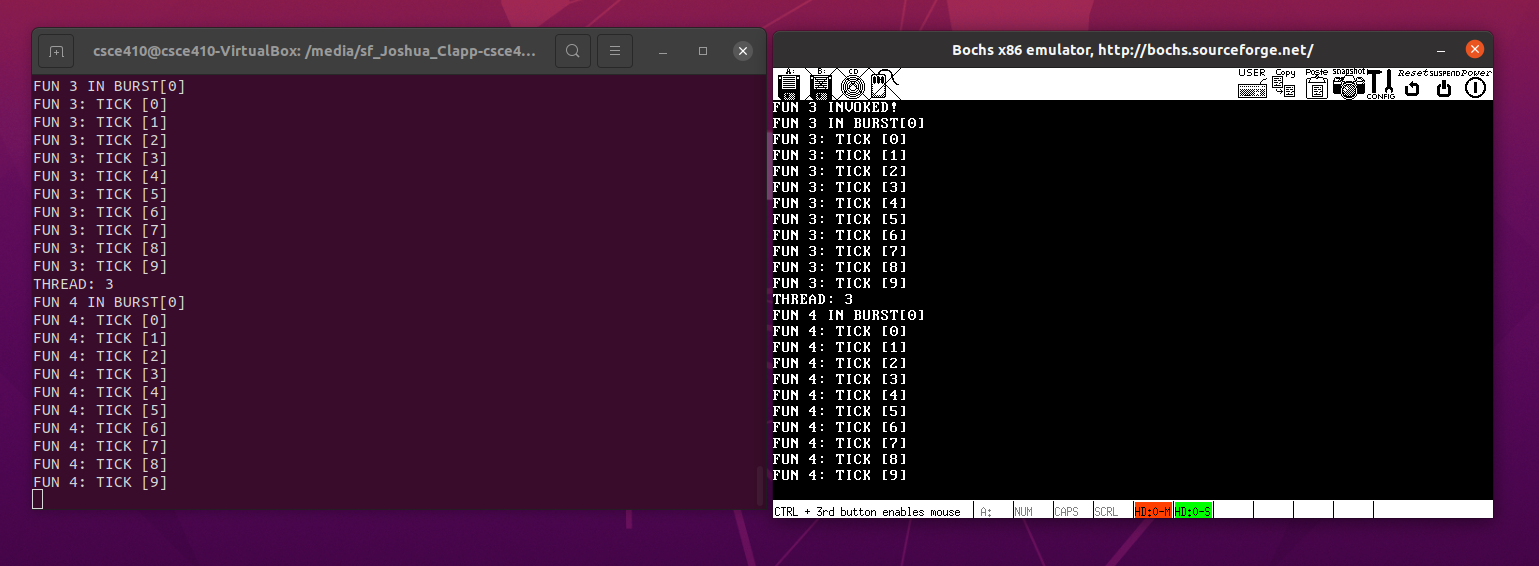
I have tried getting new mp6 source files, re transferring from mp5, and just getting a new c.img file. At the bottom will be our email conversation screenshots!

**Main Task:** Disk scheduling with FIFO no interrupts  


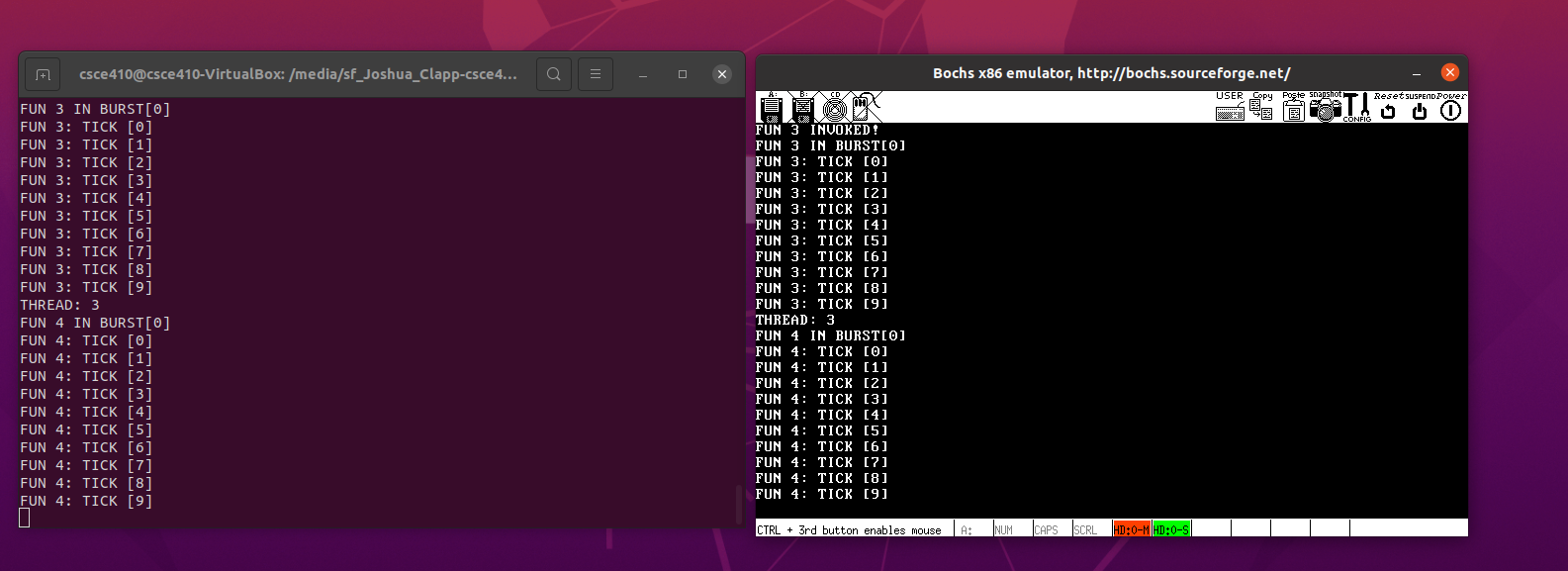
**Bonus Task 1:** Mirror disk scheduling with FIFO no interrupts



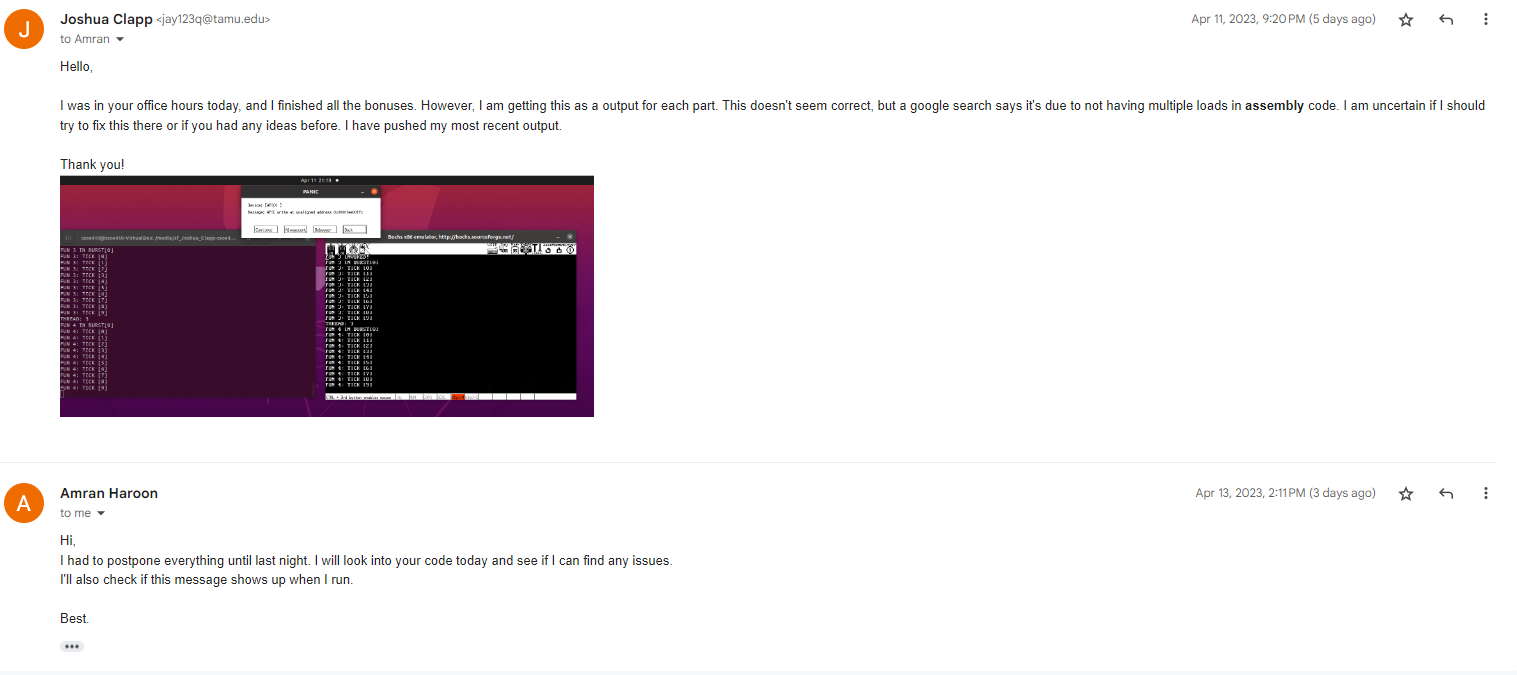
**Bonus Task 2:** Mirror disk scheduling with FIFO interrupts

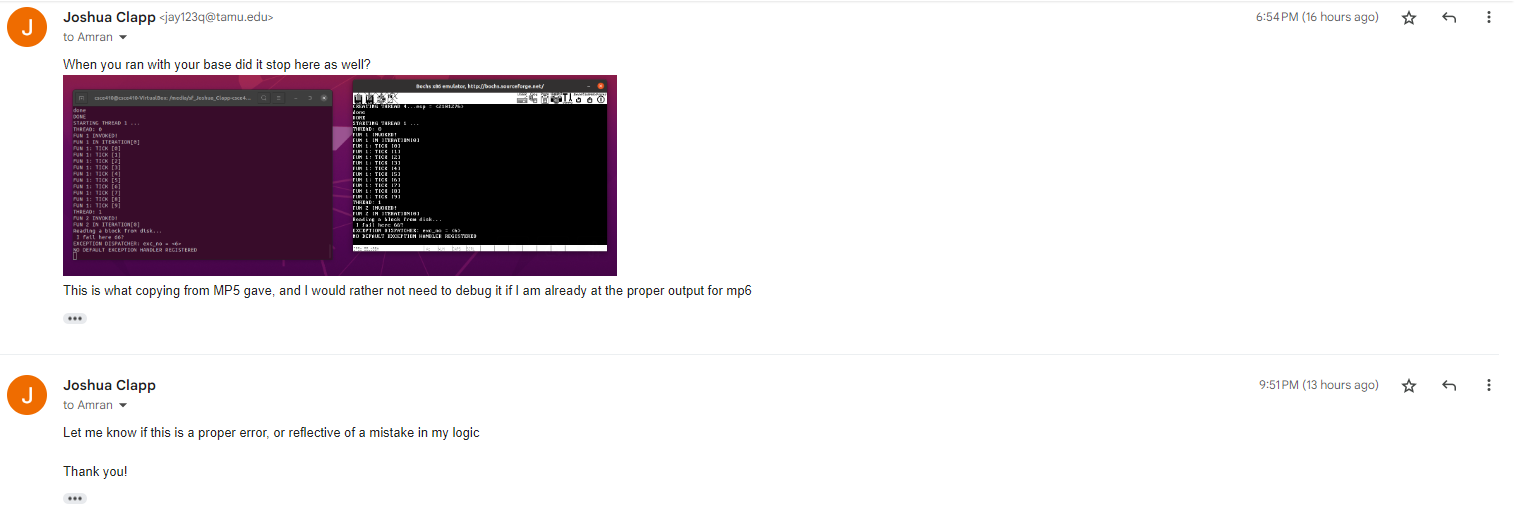
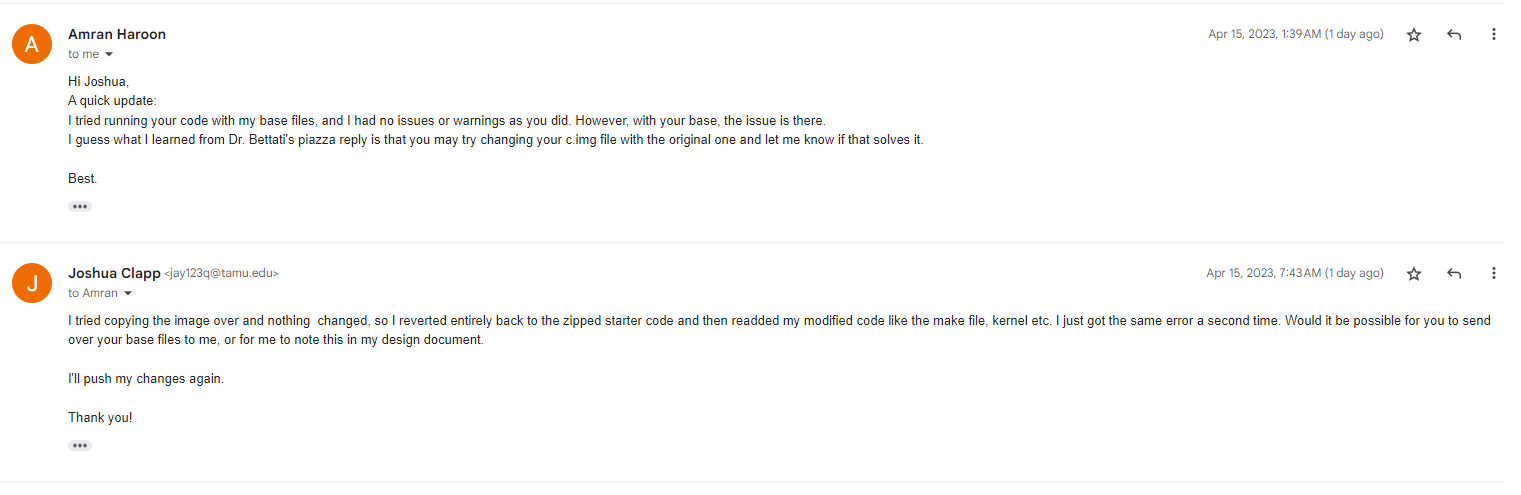


**Bonus Task 3:** Mirror disk scheduling with threads FIFO



**Email proof!**

****



Quick note, the error the last email mentions was just a failure on my end to properly transfer the files