MP6: Primitive Device Driver

Joshua Clapp
UIN: 930004089
CSCE410: Operating System

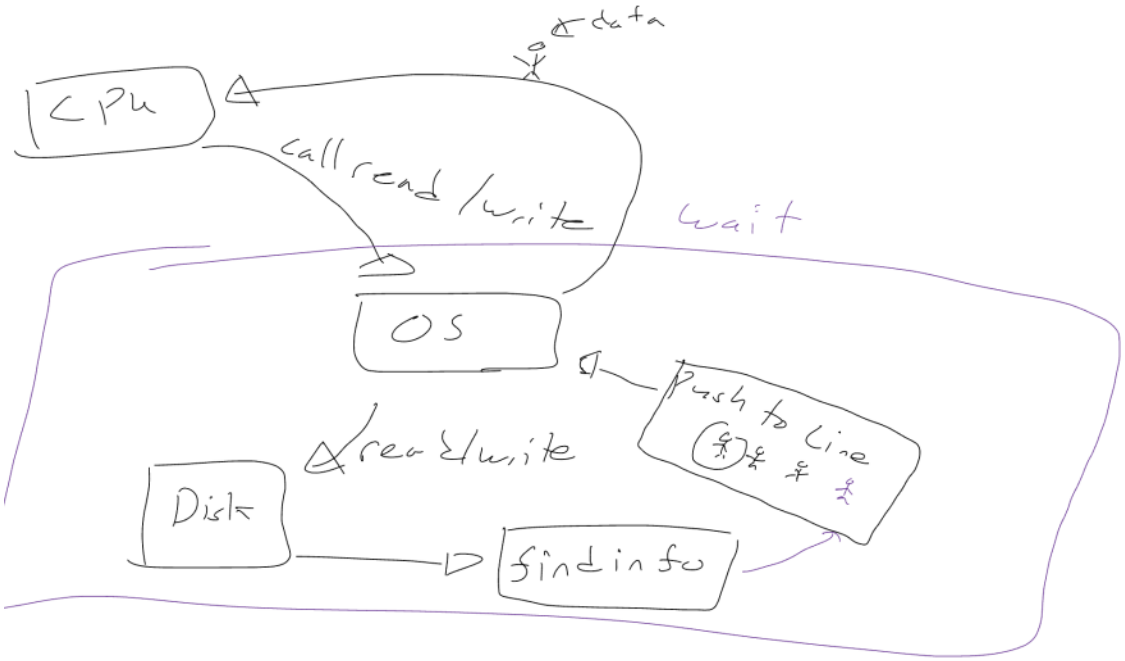
Assigned Tasks

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

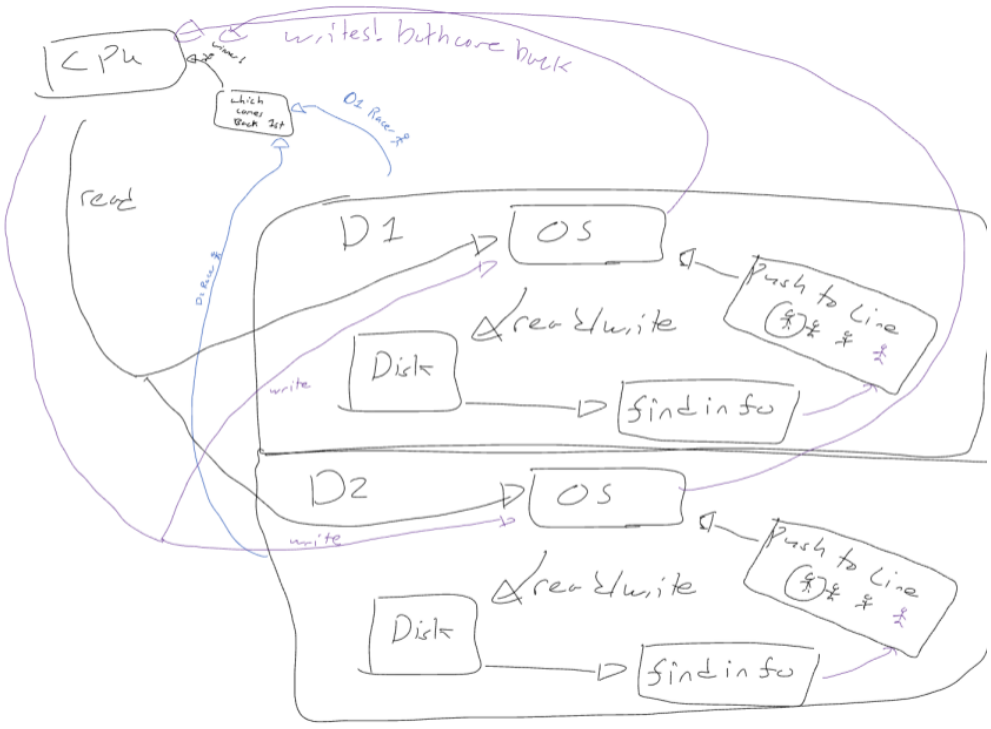
Main:

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



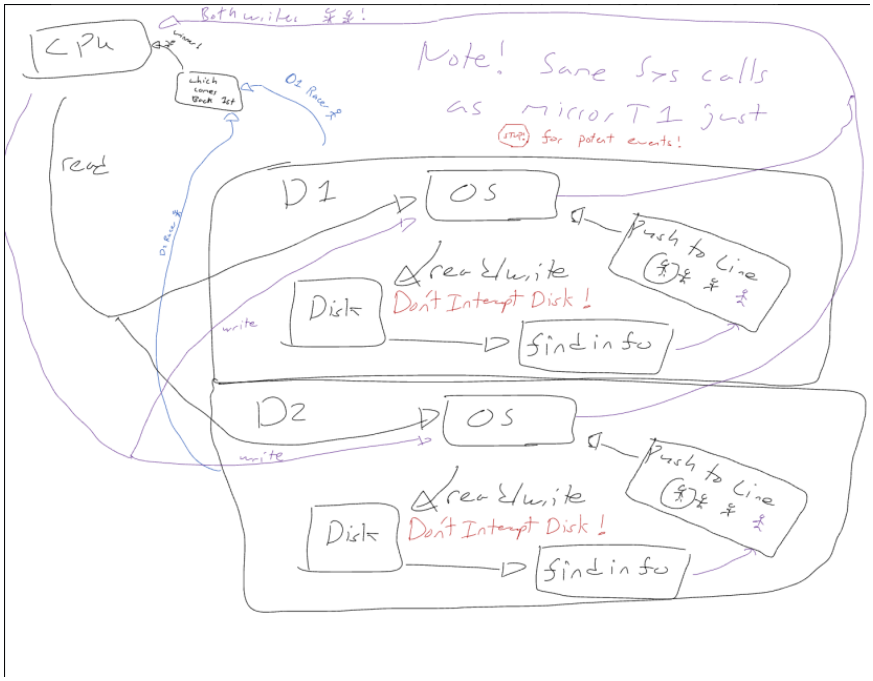
Bonus Option 1: Mirror Disk

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

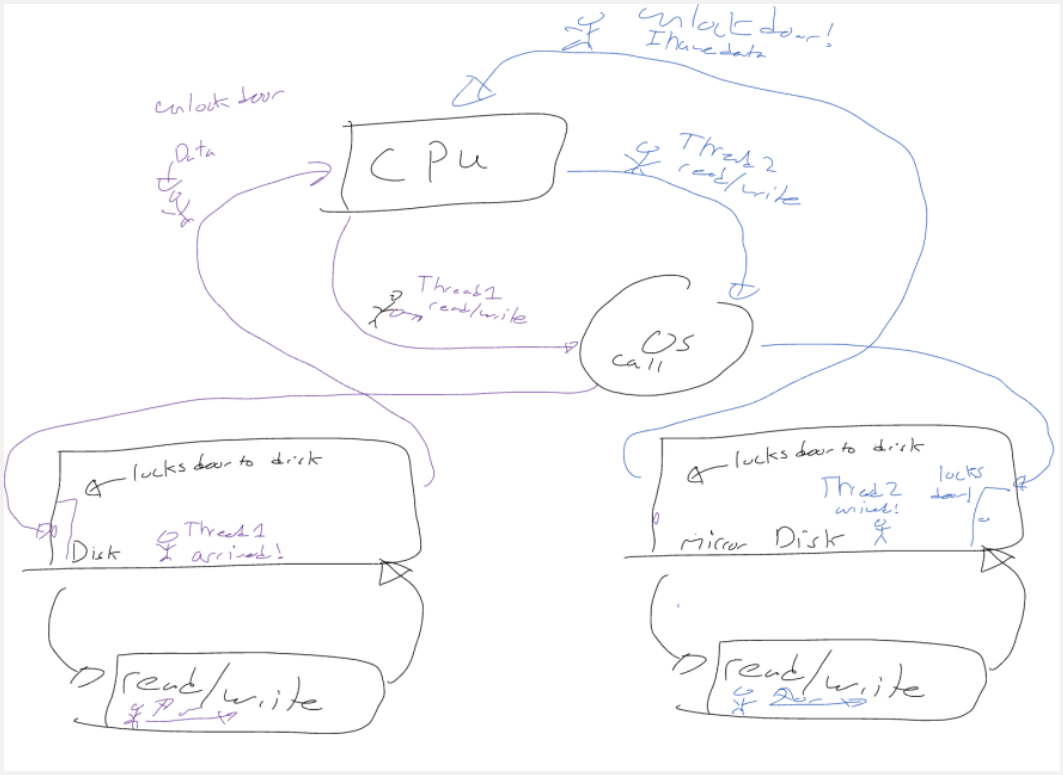


Bonus Option 3: Thread design

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.

```
SimpleDisk * System_disk;
Scheduler * SchedulerFifo;
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
    : SimpleDisk(_disk_id, _size)
{


}
```

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.

```
53  void BlockingDisk::read(unsigned long _block_no, unsigned char *_buf)
54  {
55      #ifdef INTERUPT_TEST
56      Machine::disable_interrupts();
57      #endif
58      System_disk->issue_operation(DISK_OPERATION::READ, _block_no);
59      Console::puts(" I fail here 66? \n");
60      ready();
61      Console::puts(" I fail here 75? \n");
62
63      int i;
64      unsigned short tmpw;
65      for (i = 0; i < 256; i++) {
66          tmpw = Machine::inportw(0x1F0);
67          _buf[i*2]   = (unsigned char)tmpw;
68          _buf[i*2+1] = (unsigned char)(tmpw >> 8);
69      }
70      Console::puts(" I fail here 84? \n");
71      #ifdef INTERUPT_TEST
72      Machine::enable_interrupts();
73      #endif
74      // this->push(_block_no, _buf);
75  }
76
```

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.

```
78  void BlockingDisk::write(unsigned long _block_no, unsigned char *_buf)
79  {
80      #ifdef INTERUPT_TEST
81      Machine::disable_interrupts();
82      #endif
83      System_disk->issue_operation(DISK_OPERATION::WRITE, _block_no);
84      ready();
85
86      /* write data to port */
87      int i;
88      unsigned short tmpw;
89      for (i = 0; i < 256; i++) {
90          tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
91          Machine::outportw(0x1F0, tmpw);
92      }
93      #ifdef INTERUPT_TEST
94      Machine::enable_interrupts();
95      #endif
96  }
```

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.

```
122 bool BlockingDisk::is_ready() {
123     return ((Machine::inportb(0x1F7) & 0x08) != 0);
124 }
```

blocking_disk.C:ready():

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

```
101  void BlockingDisk::ready()
102  {
103    while ((!is_ready()))
104    {
105        #ifdef INTERUPT_TEST
106          this->push(Thread::CurrentThread());
107        #else
108
109      SchedulerFifo->resume(Thread::CurrentThread());
110          #endif
111      SchedulerFifo->yield();
112    }
113  }
114
```

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.

```
29  MirroredDisk::MirroredDisk(DISK_ID _disk_id, unsigned int _size)
30  {
31      this->master = new BlockingDisk(DISK_ID::MASTER, _size);
32      this->slave = new BlockingDisk(DISK_ID::DEPENDENT, _size);
33      this->mutex = 0;
34  }
```

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.

```
36  void MirroredDisk::read(unsigned long _block_no, unsigned char *_buf)
37  {
38      // Machine::disable_interrupts();
39      #ifdef THREAD_TEST
40      this->lockMux();
41      #endif
42      issue_operation(DISK_OPERATION::READ, _block_no,  DISK_ID2::MASTER );
43      issue_operation(DISK_OPERATION::READ, _block_no, DISK_ID2::DEPENDENT  );
44      Console::puts(" I fail here 66? \n");
45      ready();
46      Console::puts(" I fail here 75? \n");
47
48      int i;
49      unsigned short tmpw;
50      for (i = 0; i < 256; i++) {
51          tmpw = Machine::inportw(0x1F0);
52          _buf[i*2]   = (unsigned char)tmpw;
53          _buf[i*2+1] = (unsigned char)(tmpw >> 8);
54      }
55      #ifdef THREAD_TEST
56      this->unlockMux();
57      #endif
58      // Machine::enable_interrupts();
59      // this->push(_block_no, _buf);
60  }
```

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.

```
62  void MirroredDisk::write(unsigned long _block_no, unsigned char *_buf)
63  {
64      // Machine::disable_interrupts();
65      #ifdef THREAD_TEST
66      this->lockMux();
67      #endif
68      // issue_operation(DISK_OPERATION::WRITE, _block_no);
69
70      this->master->write(_block_no , _buf);
71      this->slave->write(_block_no , _buf);
72      // ready();
73      #ifdef THREAD_TEST
74      this->unlockMux();
75      #endif
76      // Machine::enable_interrupts();
77  }
78
```

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.

```
79  void MirroredDisk::issue_operation(DISK_OPERATION _op, unsigned long _block_no , DISK_ID2 disk_id) {
80
81    Machine::outportb(0x1F1, 0x00); /* send NULL to port 0x1F1         */
82    Machine::outportb(0x1F2, 0x01); /* send sector count to port 0X1F2 */
83    Machine::outportb(0x1F3, (unsigned char)_block_no);
84                          /* send low 8 bits of block number */
85    Machine::outportb(0x1F4, (unsigned char)(_block_no >> 8));
86                          /* send next 8 bits of block number */
87    Machine::outportb(0x1F5, (unsigned char)(_block_no >> 16));
88                          /* send next 8 bits of block number */
89    Machine::outportb(0x1F6, ((unsigned char)(_block_no >> 24)&0x0F) | 0xE0 | ((int) disk_id << 4));
90                          /* send drive indicator, some bits,
91                             highest 4 bits of block no */
92
93    Machine::outportb(0x1F7, (_op == DISK_OPERATION::READ) ? 0x20 : 0x30);
94
95  }
```

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.

```
97   void MirroredDisk::ready()
98   {
99     while ((!master->is_ready() || !slave->is_ready()))
100    {
101
102
103      SchedulerMirror->resume(Thread::CurrentThread());
104      SchedulerMirror->yield();
105    }
106  }
107
```

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.

```
33  BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
34      : SimpleDisk(_disk_id, _size)
35  {
36      this->head = nullptr;
37      this->tail = nullptr;
38  }
```

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.

```
53  void BlockingDisk::read(unsigned long _block_no, unsigned char *_buf)
54  {
55    #ifdef INTERUPT_TEST
56    Machine::disable_interrupts();
57    #endif
58    System_disk->issue_operation(DISK_OPERATION::READ, _block_no);
59    Console::puts(" I fail here 66? \n");
60    ready();
61    Console::puts(" I fail here 75? \n");
62
63    int i;
64    unsigned short tmpw;
65    for (i = 0; i < 256; i++) {
66      tmpw = Machine::inportw(0x1F0);
67      _buf[i*2]   = (unsigned char)tmpw;
68      _buf[i*2+1] = (unsigned char)(tmpw >> 8);
69    }
70    Console::puts(" I fail here 84? \n");
71    #ifdef INTERUPT_TEST
72    Machine::enable_interrupts();
73    #endif
74    // this->push(_block_no, _buf);
75  }
76
```

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.

```
78  void BlockingDisk::write(unsigned long _block_no, unsigned char *_buf)
79  {
80      #ifdef INTERUPT_TEST
81      Machine::disable_interrupts();
82      #endif
83     System_disk->issue_operation(DISK_OPERATION::WRITE, _block_no);
84     ready();
85
86     /* write data to port */
87     int i;
88     unsigned short tmpw;
89     for (i = 0; i < 256; i++) {
90      tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
91      Machine::outportw(0x1F0, tmpw);
92     }
93      #ifdef INTERUPT_TEST
94      Machine::enable_interrupts();
95      #endif
96  }
97
```

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

```
117     #ifdef INTERUPT_TEST
118   void BlockingDisk::push( Thread * pushThread )
119   {
120     linked_queue *new_thread = new linked_queue{ pushThread , nullptr };
121
122     // check normal ll conditions
123     if (head == nullptr)
124     {
125       head = new_thread;
126       tail = new_thread;
127     }
128     else
129     {
130       tail->next = new_thread;
131       tail = tail->next;
132     }
133     size++;
134   }
135   #endif
136
```

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.

```
138 Thread *BlockingDisk::pop()
139 {
140   // hey this is a copy from scheduler.C, I might ahve been able to just call yield
141   // pop the tail
142   if (size && head)
143   { // if head exists and its not tail
144     linked_queue *head_copy = head;
145     Thread *threadReturn = head_copy->thread;
146     Thread *current_thread = head->thread;
147     head = head->next;
148     size--;
149     if(!size)
150     {
151       // head and tail is null
152       tail = nullptr;
153     }
154     // leave the current to the next
155     delete head_copy;
156     return threadReturn;
157   }
158   else
159   {
160     Console::puts(" Your ability at incompetence is impressive Mr. Clapp, line 80, write to blocking pop function \n ");
161     assert(false);
162   }
163 }
164 #endif
```

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.

```
102   void BlockingDisk::ready()
103   {
104     while ((!is_ready()))
105     {
106       #ifdef INTERUPT_TEST
107         this->push(Thread::CurrentThread());
108       #else
109
110       SchedulerFifo->resume(Thread::CurrentThread());
111         #endif
112       SchedulerFifo->yield();
113     }
114   }
115
```

blocking_disk.C:handle_interrupt(REGS * _r):

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

```
162  #ifdef INTERUPT_TEST
163  void BlockingDisk::handle_interrupt(REGS *_r)
164  {
165    Thread * nextThread = pop();
166    SchedulerFifo->resume(nextThread->Thread::CurrentThread());
167  }
168  #endif
```

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.

```
9   #ifdef THREAD_TEST
10  // this logic is from https://courses.engr.illinois.edu/cs241/sp2012/lectures/23-inside_sem.pdf
11
12  bool MirroredDisk::testSetMux(bool * key)
13  {
14      int box = *(this->key);
15      *(this->key) = true;
16      return box;
17
18  }
19
20  #endif
```

MirroredDisk.C:MirroredDisk(DISK_ID _disk_id, unsigned int _size):

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

```
29  MirroredDisk::MirroredDisk(DISK_ID _disk_id, unsigned int _size)
30  {
31      this->master = new BlockingDisk(DISK_ID::MASTER, _size);
32      this->slave = new BlockingDisk(DISK_ID::DEPENDENT, _size);
33      this->mutex = 0;
34  }
```

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.

```
29  void MirroredDisk::read(unsigned long _block_no, unsigned char *_buf)
30  {
31      // Machine::disable_interrupts();
32      #ifdef THREAD_TEST
33          // lock the mux
34          while(testSetMux(key));
35      #endif
36      issue_operation(DISK_OPERATION::READ, _block_no,  DISK_ID2::MASTER );
37      issue_operation(DISK_OPERATION::READ, _block_no, DISK_ID2::DEPENDENT  );
38      Console::puts(" I fail here 66? \n");
39      ready();
40      Console::puts(" I fail here 75? \n");
41
42      int i;
43      unsigned short tmpw;
44      for (i = 0; i < 256; i++) {
45          tmpw = Machine::inportw(0x1F0);
46          _buf[i*2]   = (unsigned char)tmpw;
47          _buf[i*2+1] = (unsigned char)(tmpw >> 8);
48      }
49      #ifdef THREAD_TEST
50          // unlock the mux
51          *(this->key) = false;
52      #endif
53      // Machine::enable_interrupts();
54      // this->push(_block_no, _buf);
55  }
```

MirroredDisk.C:write(unsigned long _block_no, unsigned char * _buf):

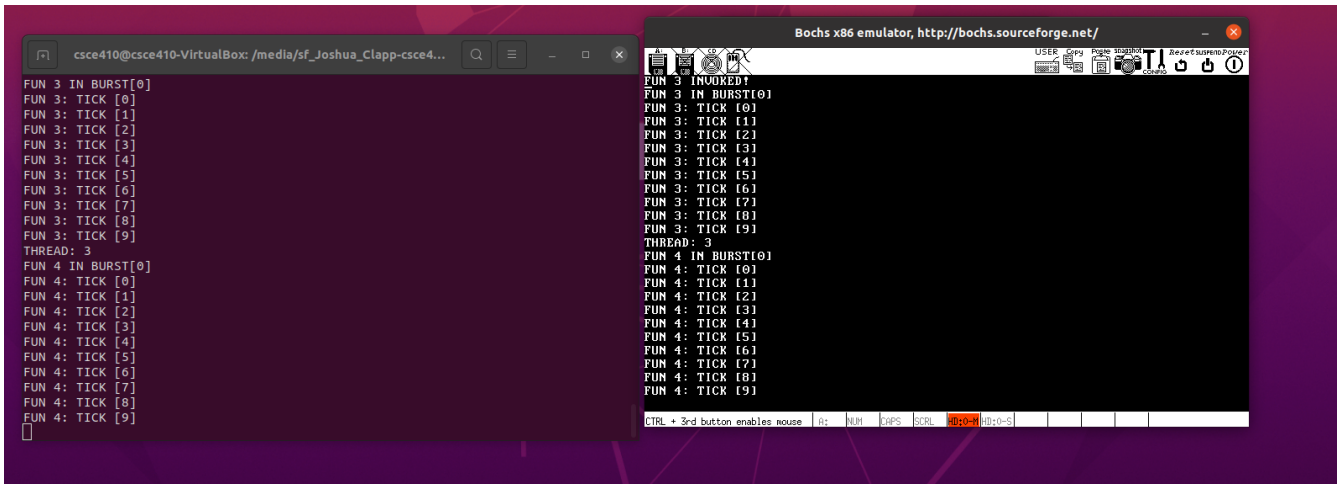
Lock a thread in the critical section with a busy wait, and perform the write operation to both disks and unlock when the critical section is over.

```
57  void MirroredDisk::write(unsigned long _block_no, unsigned char *_buf)
58  {
59      // Machine::disable_interrupts();
60      #ifdef THREAD_TEST
61          // lock the mux
62          while(testSetMux(key));
63      #endif
64      // issue_operation(DISK_OPERATION::WRITE, _block_no);
65
66      this->master->write(_block_no , _buf);
67      this->slave->write(_block_no , _buf);
68      // ready();
69      #ifdef THREAD_TEST
70          // unlock the mux
71          *(this->key) = false;
72      #endif
73      // Machine::enable_interrupts();
74  }
```

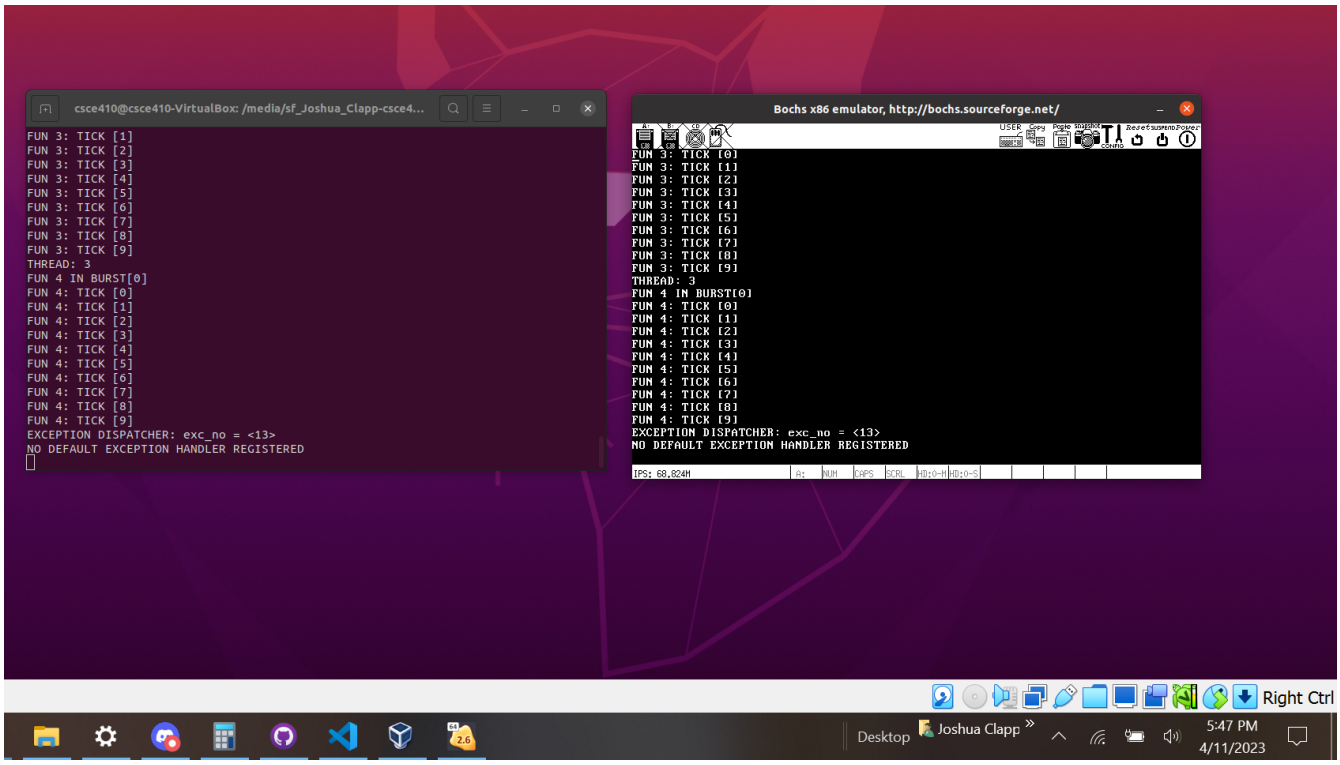
Testing

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 with using the testing given is rather simplistic and limited with no specific edge case targeting.

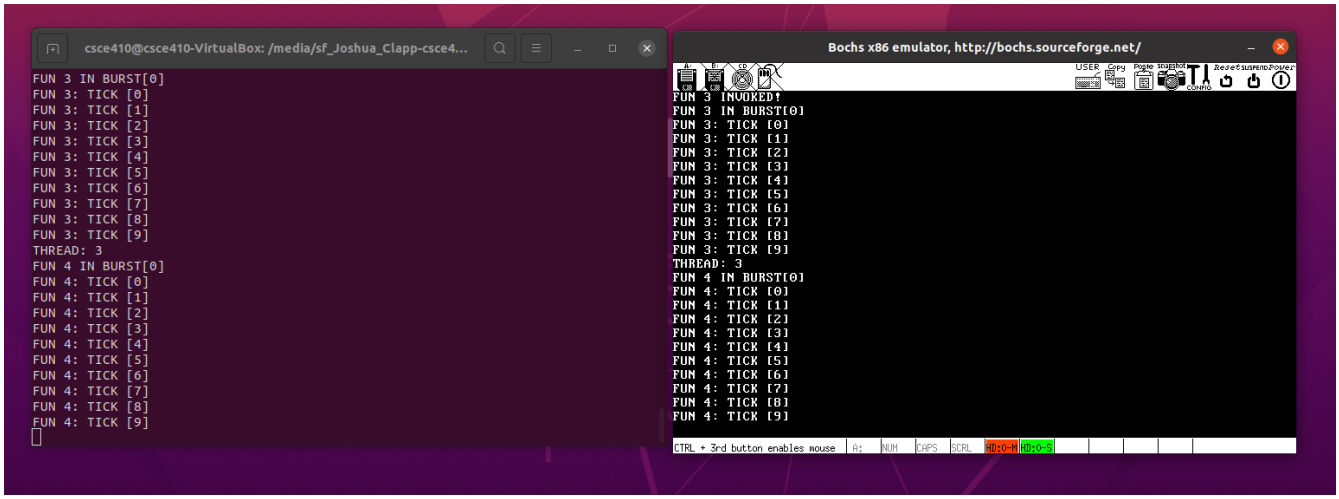
Main Task: Disk scheduling with FIFO no interrupts



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