MP6: Primitive Disk Device Driver

Po-Chieh Wang
UIN: 933004975
CSCE 611: Operating System

Assigned Tasks

Main: Completed.
Bonus option 1: Completed.
Bonus option 2: Did not attempt.
Bonus option 3: Completed.
Bonus option 4: Completed.

System Design

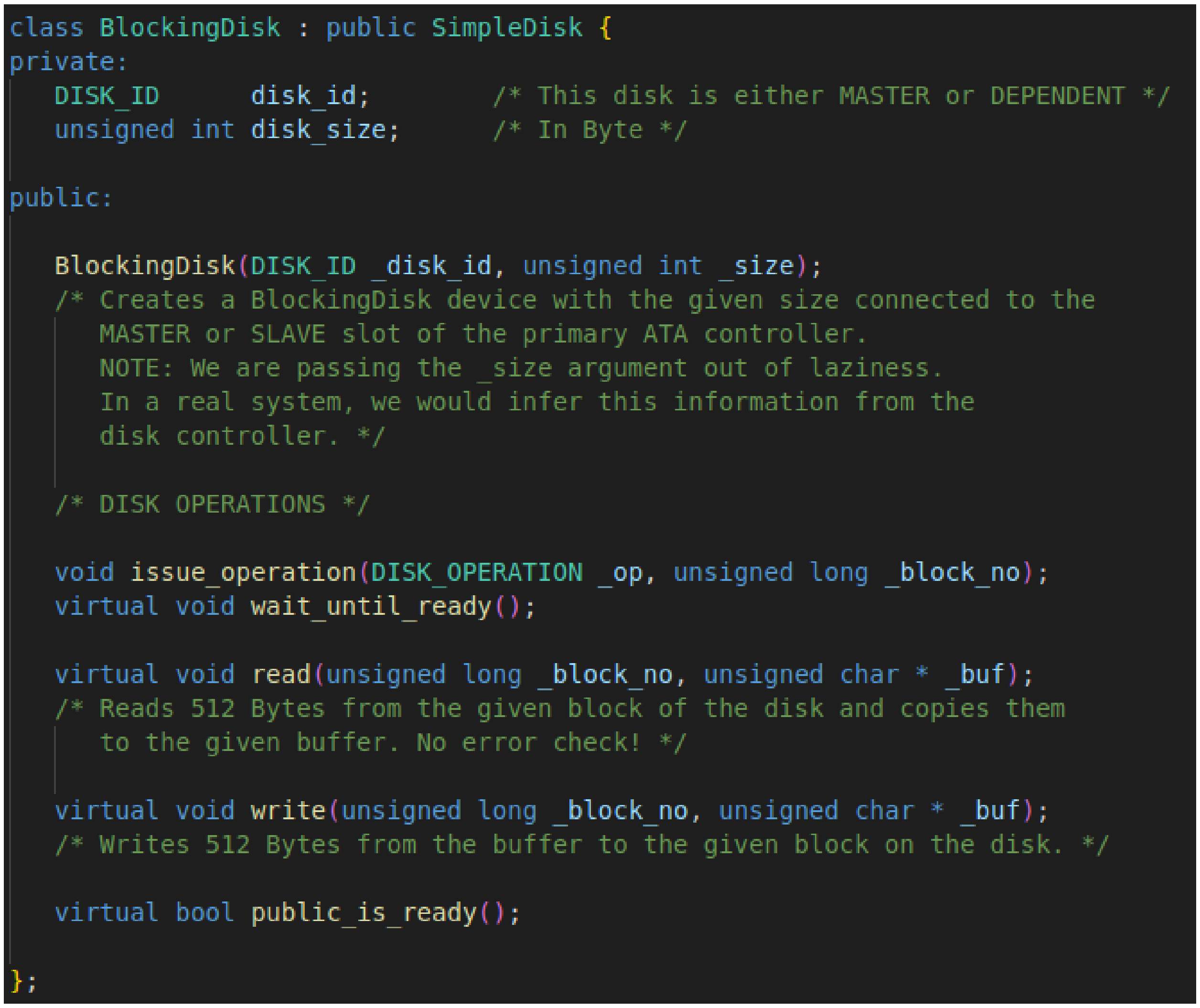
The goal of this machine problem is to implement the blocking disk system. To achieve this goal, I utilized the FIFO scheduler I designed in MP5. When a blocking disk starts to read or write, it will issue the operation and check if the ready signal is up or not. If the ready signal is not up, then this thread will automatically give up cpu and put itself into the end of the queue inside the scheduler. For bonus option 1, I implemented a new class called MirrorDisk which inherits the blocking disk. But the MirrorDisk has two variables of blocking disk, master and dependency, respectively. During the read operation, the mirror disk will issue this operation to two disks and wait until the ready signal to be up. If the ready signal is up, it will start to read the buffer. In the write operation, mirror disk will first write to the master disk until the ready signal is up and then write to the dependency disk until it is ready.

Design for bonus 3:
Since the scheduler is using FIFO algorithm, we do not need to consider the situation of preemption. Therefore, I implemented a lock inside the mirrordisk. When the thread tries to enter the critical section, it will first check the lock, if the lock is true, which means there are another thread in this critical section, this thread will yield itself and put itself into the end of the queue inside the scheduler. Once it is in the critical section, it will turn on the lock and execute the instruction. Before it leaves the critical section, it will turn off the lock to let others to be able to enter the critical section.

Code Description

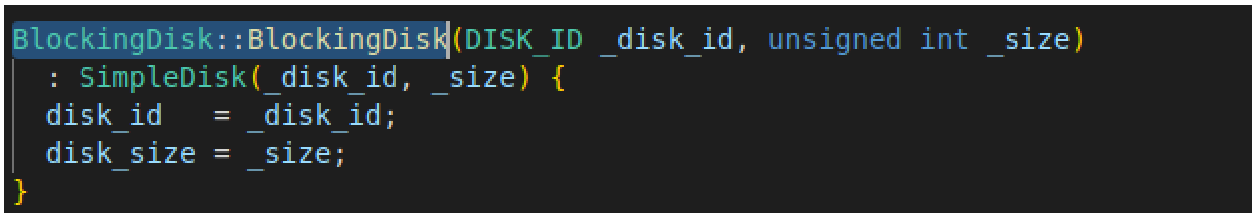
Main: In this section, I implemented blocking_disk.C and blocking_disk.H.

blocking_disk.H: BlockingDisk: This class inherits the class of SimpleDisk, and it overwrites the function of read and write. Also, it provides a function called public_is_ready for the use of bonus 1.



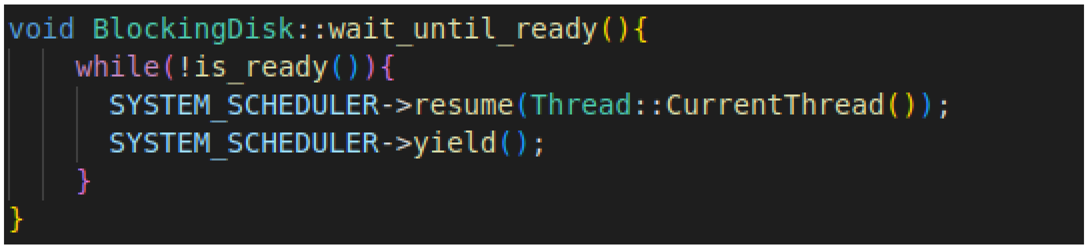
```
class BlockingDisk : public SimpleDisk {
private:
   DISK_ID      disk_id;         /* This disk is either MASTER or DEPENDENT */
   unsigned int disk_size;       /* In Byte */

public:

   BlockingDisk(DISK_ID _disk_id, unsigned int _size);
   /* Creates a BlockingDisk device with the given size connected to the
      MASTER or SLAVE slot of the primary ATA controller.
      NOTE: We are passing the _size argument out of laziness.
      In a real system, we would infer this information from the
      disk controller. */

   /* DISK OPERATIONS */

   void issue_operation(DISK_OPERATION _op, unsigned long _block_no);
   virtual void wait_until_ready();

   virtual void read(unsigned long _block_no, unsigned char * _buf);
   /* Reads 512 Bytes from the given block of the disk and copies them
      to the given buffer. No error check! */

   virtual void write(unsigned long _block_no, unsigned char * _buf);
   /* Writes 512 Bytes from the buffer to the given block on the disk. */

   virtual bool public_is_ready();

};
```

blocking_disk.C: BlockingDisk::BlockingDisk: This function is the same as the constructor in SimpleDisk.



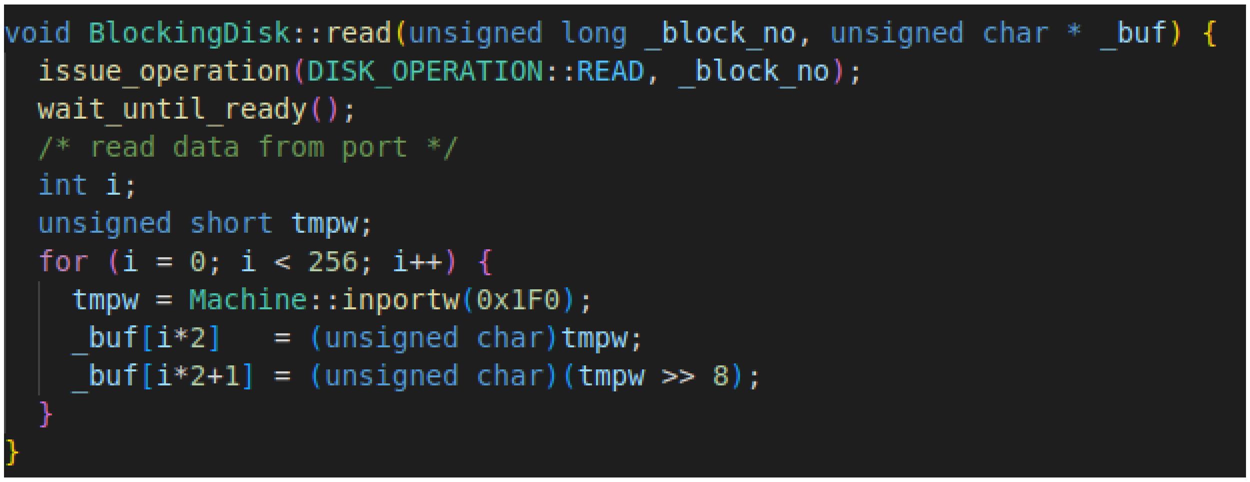
```
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size) {
  disk_id   = _disk_id;
  disk_size = _size;
}
```

blocking_disk.C:BlockingDisk::wait_until_ready: This function checks the ready signal, if the signal is not up, the thread will put itself into the end of the queue in the scheduler and give up cpu.



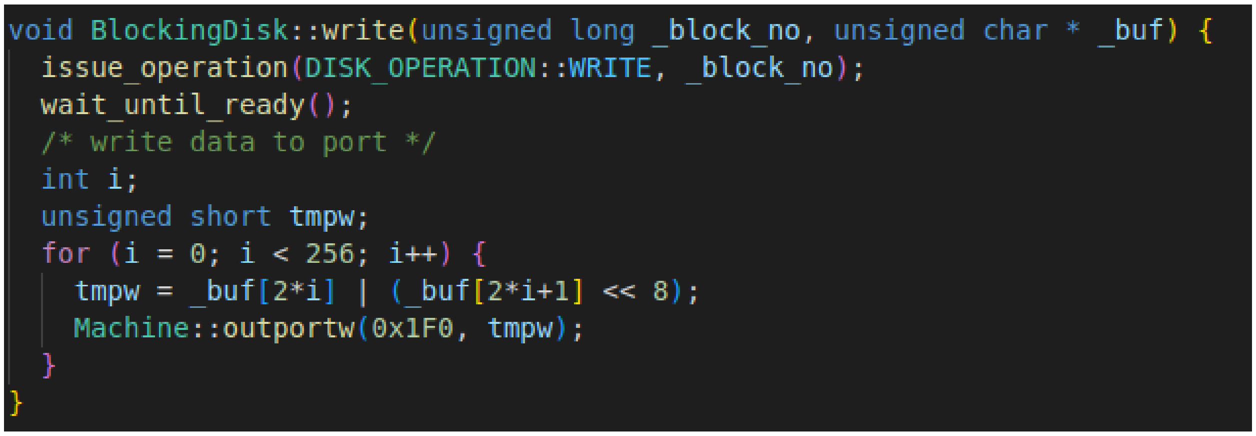
```
void BlockingDisk::wait_until_ready(){
    while(!is_ready()){
      SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
      SYSTEM_SCHEDULER->yield();
    }
}
```

blocking_disk.C:BlockingDisk::read: During the read operation, this function will issue operation until it is ready. If it is ready, it starts to put the data inside the port to the buffer.



```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
  issue_operation(DISK_OPERATION::READ, _block_no);
  wait_until_ready();
  /* read data from port */
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = Machine::inportw(0x1F0);
    _buf[i*2]   = (unsigned char)tmpw;
    _buf[i*2+1] = (unsigned char)(tmpw >> 8);
  }
}
```

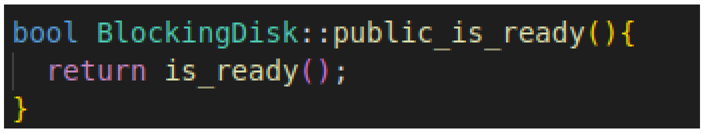
blocking_disk.C:BlockingDisk::write: During the write operation, this function will issue operation until it is ready. If it is ready, it starts to put the data inside the buffer to the port.



```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
  issue_operation(DISK_OPERATION::WRITE, _block_no);
  wait_until_ready();
  /* write data to port */
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
    Machine::outportw(0x1F0, tmpw);
  }
}
```

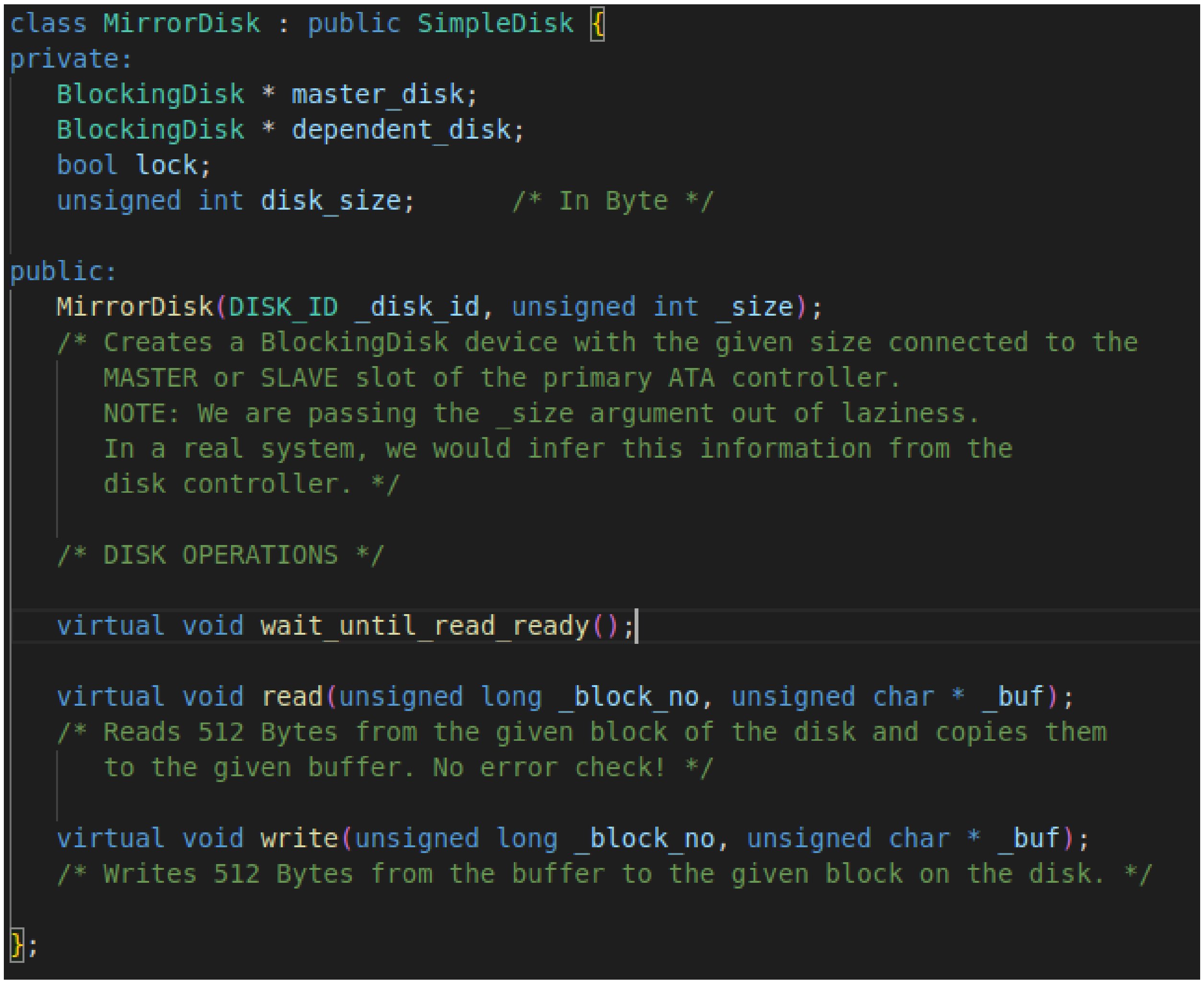
Bonus 1, 3 and 4: In these parts, I implemented blocking_disk.C and blocking_disk.H. I created a class MirrorDisk to implement RAID 1 and mutual exclusion.

blocking_disk.C:BlockingDisk::public_is_ready(): This function provides a public version of is_ready.



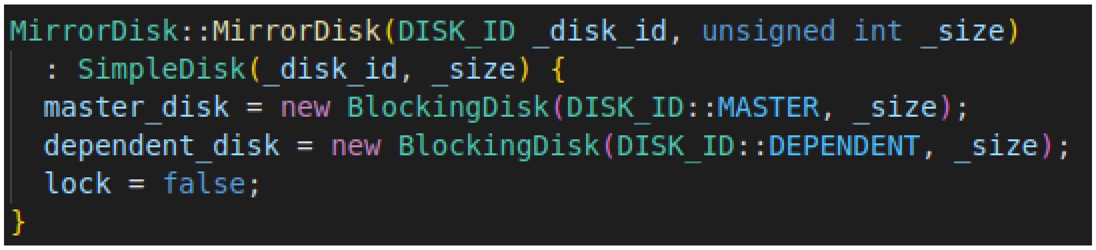
```
bool BlockingDisk::public_is_ready(){
  return is_ready();
}
```

blocking_disk.H:MirrorDisk: This class inherits from the SimpleDisk and with two private variables, master_disk and dependent_disk, respectively. These two pointers point to two BlockingDisk. Also, it has a lock variable to implement mutual exclusion.



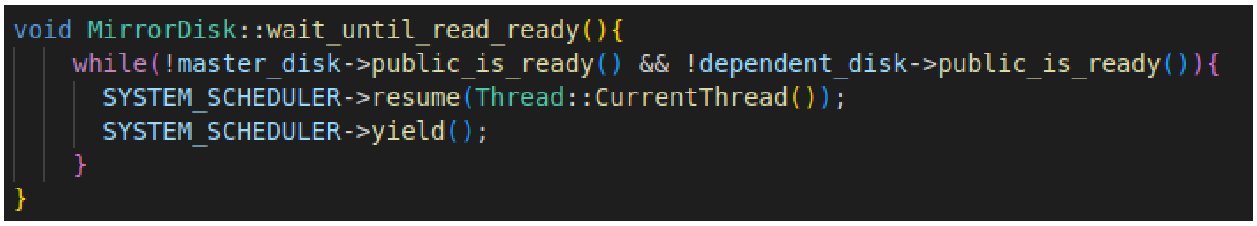
```
class MirrorDisk : public SimpleDisk {
private:
   BlockingDisk * master_disk;
   BlockingDisk * dependent_disk;
   bool lock;
   unsigned int disk_size;      /* In Byte */

public:
   MirrorDisk(DISK_ID _disk_id, unsigned int _size);
   /* Creates a BlockingDisk device with the given size connected to the
      MASTER or SLAVE slot of the primary ATA controller.
      NOTE: We are passing the _size argument out of laziness.
      In a real system, we would infer this information from the
      disk controller. */

   /* DISK OPERATIONS */

   virtual void wait_until_read_ready();

   virtual void read(unsigned long _block_no, unsigned char * _buf);
   /* Reads 512 Bytes from the given block of the disk and copies them
      to the given buffer. No error check! */

   virtual void write(unsigned long _block_no, unsigned char * _buf);
   /* Writes 512 Bytes from the buffer to the given block on the disk. */

};
```

blocking_disk.C:MirrorDisk::MirrorDisk: This constructor creates two blockingdisk, one is the master disk and the other is the dependent disk. Also, it set the lock to false in the beginning.



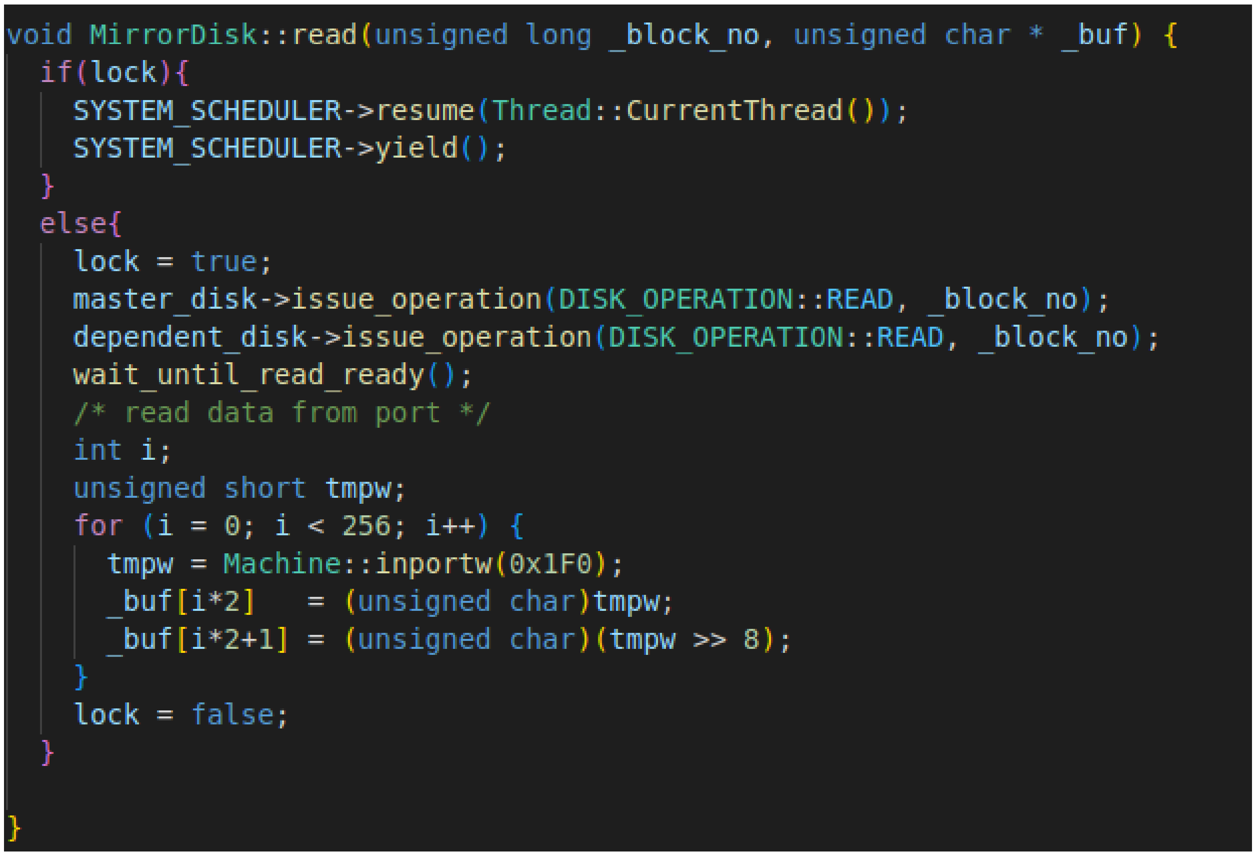
```
MirrorDisk::MirrorDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size) {
  master_disk = new BlockingDisk(DISK_ID::MASTER, _size);
  dependent_disk = new BlockingDisk(DISK_ID::DEPENDENT, _size);
  lock = false;
}
```

blocking_disk.C:MirrorDisk::wait_until_read_ready(): This function is used in the read operation. When one of the disks is ready, the ready signal will be up. Then the read operation could continue. If not, it will put this thread to the end of the queue inside the scheduler.



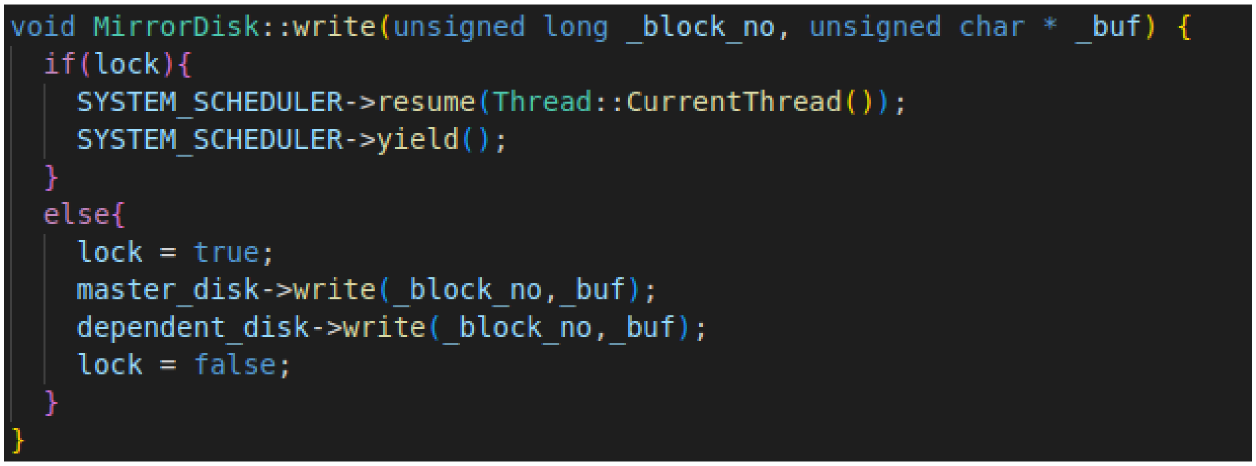
```
void MirrorDisk::wait_until_read_ready(){
    while(!master_disk->public_is_ready() && !dependent_disk->public_is_ready()){
      SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
      SYSTEM_SCHEDULER->yield();
    }
}
```

blocking_disk.C:MirrorDisk::read: This function will first check the lock. If the lock is true, which means there is another thread inside the critical section, this function will yield itself. When it is in the critical section, it will first change the lock to true and start to do a read operation. Once one of the disks is ready, it starts to read data from the port and put it into the buffer. Before it leaves the critical section, it will turn off the lock.



```
void MirrorDisk::read(unsigned long _block_no, unsigned char * _buf) {
  if(lock){
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
  }
  else{
    lock = true;
    master_disk->issue_operation(DISK_OPERATION::READ, _block_no);
    dependent_disk->issue_operation(DISK_OPERATION::READ, _block_no);
    wait_until_read_ready();
    /* read data from port */
    int i;
    unsigned short tmpw;
    for (i = 0; i < 256; i++) {
      tmpw = Machine::inportw(0x1F0);
      _buf[i*2]   = (unsigned char)tmpw;
      _buf[i*2+1] = (unsigned char)(tmpw >> 8);
    }
    lock = false;
  }

}
```

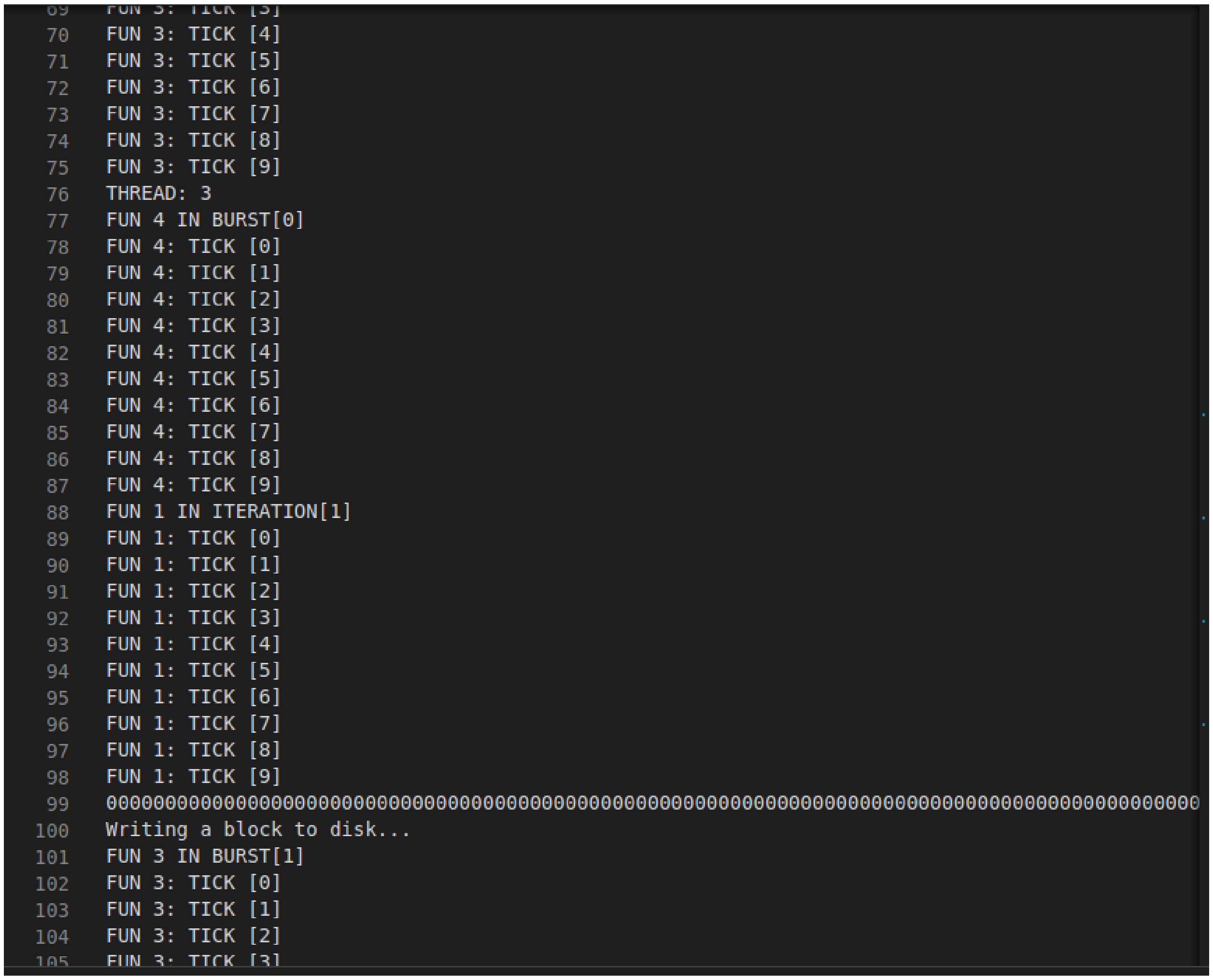
blocking_disk.C:MirrorDisk::write:This function will first check the lock. If the lock is true, which means there is another thread inside the critical section, this function will yield itself. When it is in the critical section, it will first change the lock to true and start to do a write operation. Once the master disk is ready, the dependent disk can start to do the write operation. Before it leaves the critical section, it will turn off the lock.



```
void MirrorDisk::write(unsigned long _block_no, unsigned char * _buf) {
  if(lock){
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
  }
  else{
    lock = true;
    master_disk->write(_block_no,_buf);
    dependent_disk->write(_block_no,_buf);
    lock = false;
  }
}
```

Testing

Since the disks are emulated, the IO requests always come back immediately. Therefore, even if we implement another thread to do IO operation, we still cannot test the mutual exclusion. For testing the main function and bonus 1, I added a #ifdef MIRROR_DISK_ to change the system disk from simple disk to my mirror disk class. The other modification I did is to change putch to puti to let the function correctly print out what we read. The result shows that my implementation of main and bonus 1 are correct.



```
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
THREAD: 3
FUN 4 IN BURST[0]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[1]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
Writing a block to disk...
FUN 3 IN BURST[1]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
```