In this machine problem, I chose to accomplish **option 1**, **option 3**, and **option 4** bonus attempted.

First of all, I just updated the scheduler class to be able to handle the blocking disk queue and ready queue. The yield function would check the disk object and make sure that the disk has the operation, then it would schedule the thread. For the blocking disk class, which was very straightforward like the simple disk class. In the read and write function, just issue the operation first, then check if the disk is ready, if it is not ready, we kind of resumed the system scheduler and then implemented the yield function, this would be the process like “wait until ready”. The thread would be added to the blocked queue when the disk returns ready. Other than that, the thread would wait the disk I/O. In the kernel.C, I also updated the simple disk to blocking disk.

Here are the changes I made in blocking disk class.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  // SimpleDisk::read(_block_no, _buf);
  // make issue_operation to public
  issue_operation(READ, _block_no);
  if (!SimpleDisk::is_ready()) {
    Thread* cur = Thread::CurrentThread();
    SYSTEM_SCHEDULER->add_block(cur);
    SYSTEM_SCHEDULER->yield();
  }
  // from simple_disk
  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);
  }
}


void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  // SimpleDisk::write(_block_no, _buf);
  // make issue_operation to public
  issue_operation(WRITE, _block_no);
  if (!SimpleDisk::is_ready()) {
    Thread* cur = Thread::CurrentThread();
    SYSTEM_SCHEDULER->add_block(cur);
    SYSTEM_SCHEDULER->yield();
  }
  // from simple_disk
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
    Machine::outportw(0x1F0, tmpw);
  }
}
```

Then, for the design option 1, I just mainly wrote the almost same mirrored disk class as the blocking disk, and in the kernel.C just added the mirrored disk class and assign dependent as the disk id instead of the master. The read and write function in the mirrored disk class were mostly the same. Just issue the operation as read or write, and then check if it is ready and execute the read and write operations.

```
void MirroredDisk::read(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  // SimpleDisk::read(_block_no, _buf);
  // make issue_operation to public
  issue_operation(READ, _block_no);
  if (!SimpleDisk::is_ready()) {
    Thread* cur = Thread::CurrentThread();
    SYSTEM_SCHEDULER->add_block(cur);
    SYSTEM_SCHEDULER->yield();
  }
  // from simple_disk
  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);
  }
}


void MirroredDisk::write(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  // SimpleDisk::write(_block_no, _buf);
  // make issue_operation to public
  issue_operation(WRITE, _block_no);
  if (!SimpleDisk::is_ready()) {
    Thread* cur = Thread::CurrentThread();
    SYSTEM_SCHEDULER->add_block(cur);
    SYSTEM_SCHEDULER->yield();
  }
  // from simple_disk
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
    Machine::outportw(0x1F0, tmpw);
  }
}
```

Design for option 3 and 4: this should ensure that the device driver would run multiple thread in a single processor safely. When several threads issued the operations as read or write, we can put the read or write operations into the block queue and we can make the thread abandon the CPU at this moment, which a single read or write operation request would be handled at one time. We compiled the program, and there were two threads issuing the read operations, the design made the block queue threads, and the actual operation is in FIFO manner sequentially, which the thread safe was implemented well.