Design Report

Submitted files

In this assignment, I have completed the basic requirement bonus question 1, bonus question 2, bonus question 3 & 4.

For each submission, I have included all the source files: the changed ones and unchanged ones, but I did not include img files `c.img, d.img, dev_kernel_grub.img`. In order to run all files successfully, these img files have to be included.

1. Basic Requirement

In this basic requirement, I have modified the following files

```
simple_disk.H/C
blocking_disk.H/C
scheduler.H/C
makefile
kernel.C
```

First, I included the `scheduler.H/C` files in this assignment to give CPU to next thread while busy waiting.

To avoid busy waiting in the original simple disk files, I have changed the issue_operation from private to protected mode.

```
protected:
     void issue_operation(DISK_OPERATION _op, unsigned long _block_no);
```

Then, in the blocking disk files, instead of using `wait_until_ready()` in the original read/write files, I changed read and write in blocking disk files.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {

  issue_operation(READ, _block_no);

  if (!is_ready()) {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
   }
    ....
    perform read operatoin
    ....
}
```

And the write files

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {

   issue_operation(WRITE, _block_no);

   if (!is_ready()) {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
   }
    ......
    perform write operation
    ......

}
```

Now busy-wait is avoided and CPU can be transferred to the next thread.

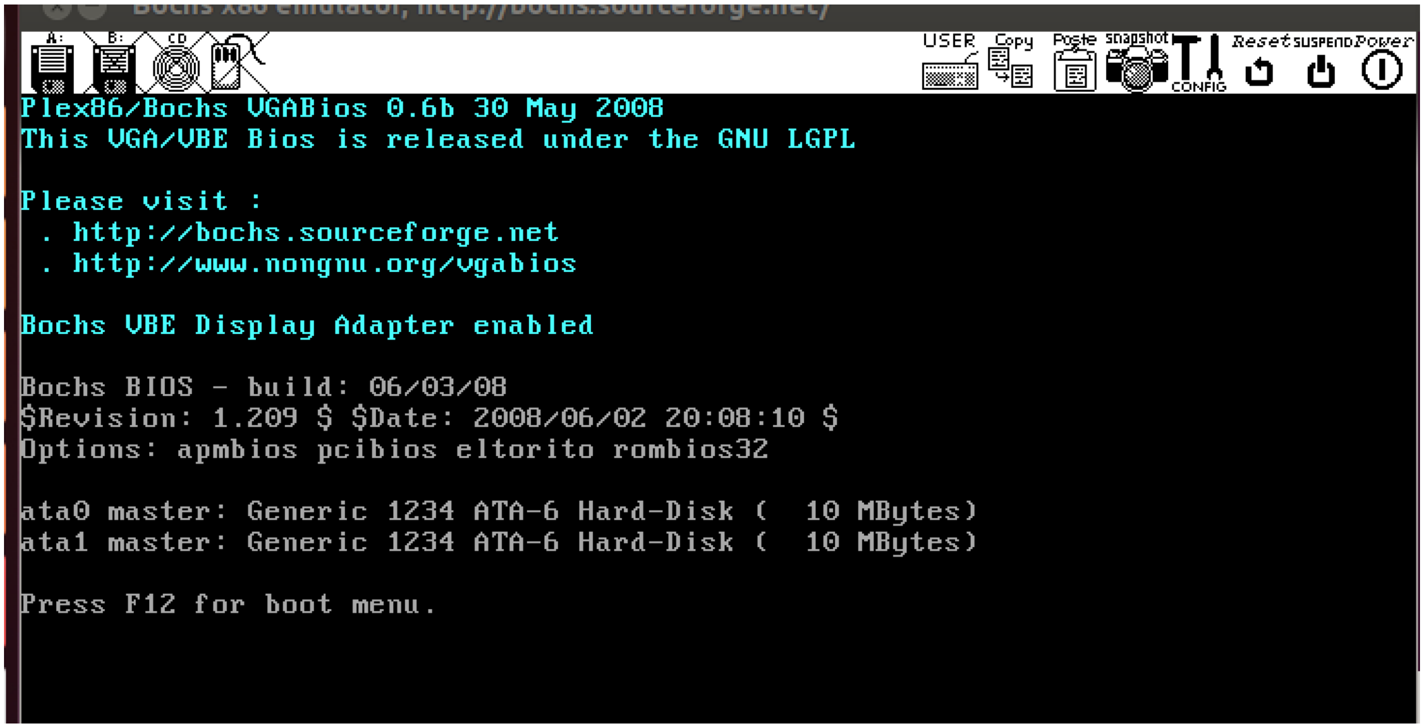
2. Bonus 1

For the fist bonus, we are asked to design the mirrored disk class. I have modified the following files:

```
blocking_disk.C
blocking_disk.H
mirror_disk.C
mirror_disk.H
scheduler.C
scheduler.H
simple_disk.C
simple_disk.H
thread.C
thread.H
kernel.C
makefile
bochsrc.bxrc
```

1. The files `blocking_disk.H/C` are not used in the bonus assignment.
2. The files `scheduler.H/C` are included to avoid busy waiting and give up CPU while spinning waiting.
3. The files `simple_disk.H/C`, I have changed the issue_operation from private to protected. `protected: virtual void issue_operation(DISK_OPERATION _op, unsigned long _block_no);`
4. `thread.H/C` are not changed here, but I will change it in the other bonus questions, so I keep them here.
5. In `makefile`, I have included `mirror_disk.H/C` in the make file. In `kernel.C`, I changed `SYSTEM_DISK = new MirroredDisk(MASTER, SYSTEM_DISK_SIZE);`
6. In `bochsrc.bxrc`, I changed the configuration of disk controller and disk. In the original files, both c and d disk are attached to master controller. Here, I linked c disk to master controller and d disk to slave controller.

   ```
   # hard disk
   ata0: enabled=1, ioaddr1=0x1f0, irq=14
   ata1: enabled=1, ioaddr1=0x170, irq=15
   ata0-master: type=disk, path="c.img", cylinders=306, heads=4, spt=17
   ata1-master: type=disk, path="d.img", cylinders=306, heads=4, spt=17
   ```



```
Plex86/Bochs VGABios 0.6b 30 May 2008
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 06/03/08
$Revision: 1.209 $ $Date: 2008/06/02 20:08:10 $
Options: apmbios pcibios eltorito rombios32

ata0 master: Generic 1234 ATA-6 Hard-Disk (  10 MBytes)
ata1 master: Generic 1234 ATA-6 Hard-Disk (  10 MBytes)

Press F12 for boot menu.
```

7. Now, in the mirror_disk class, it becomes easy to check the disk status of c and d. Here, I designed the Mirror Class like this: A is for the disk c and B for the disk d, is_A_ready() and is_B_ready() are to check the status of disk C and disk D.

```
class MirroredDisk : public SimpleDisk {
public:
MirroredDisk(DISK_ID _disk_id, unsigned int _size);
bool is_A_ready();
bool is_B_ready();
virtual void issue_operation(DISK_OPERATION _op, unsigned long _block_no);
virtual void read(unsigned long _block_no, unsigned char * _buf);
virtual void write(unsigned long _block_no, unsigned char * _buf);}
```

8. In the implementation, master controller has status register 0x1F7 and slave controller has status register 0x177. So,

```
bool MirroredDisk::is_A_ready() {
        return ((Machine::inportb(0x1F7) & 0x08) != 0);
}
bool MirroredDisk::is_B_ready() {
        return ((Machine::inportb(0x177) & 0x08) != 0);
}
```

9. Now, in the issue_operation function, operation will be issued to both master controller and slave controller.

```
  //send instructions to controller A
  Machine::outportb(0x1F1, 0x00);
  Machine::outportb(0x1F2, 0x01);
  Machine::outportb(0x1F3, (unsigned char)_block_no);
  Machine::outportb(0x1F4, (unsigned char)(_block_no >> 8));
  Machine::outportb(0x1F5, (unsigned char)(_block_no >> 16));
  Machine::outportb(0x1F6, ((unsigned char)(_block_no >> 24)&0x0F) | 0xE0 | (id() <<
4));
  Machine::outportb(0x1F7, (_op == READ) ? 0x20 : 0x30);
  //send instruction to controller B
  Machine::outportb(0x171, 0x00);
  Machine::outportb(0x172, 0x01);
  Machine::outportb(0x173, (unsigned char)_block_no);
  Machine::outportb(0x174, (unsigned char)(_block_no >> 8));
  Machine::outportb(0x175, (unsigned char)(_block_no >> 16));
  Machine::outportb(0x176, ((unsigned char)(_block_no >> 24)&0x0F) | 0xE0 | (id() <<
4));
  Machine::outportb(0x177, (_op == READ) ? 0x20 : 0x30);
```

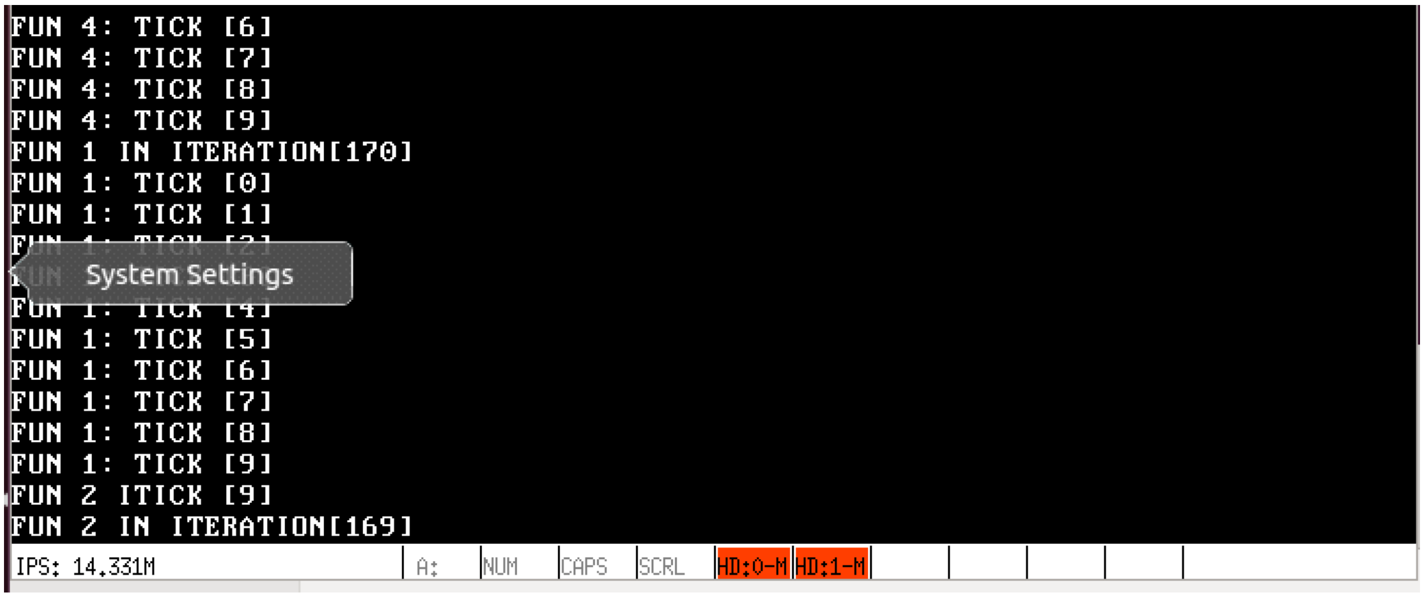
10. In the read operation, we can check the status of A and B, if either disk A and disk B is ready, the read operation can be performed, but if neither A or B is ready, the scheduler will give CPU to the other thread. So,

```
void MirroredDisk::read(unsigned long _block_no, unsigned char * _buf) {
  issue_operation(READ, _block_no);
  if (!(is_A_ready()||is_B_ready())) {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
   }
   if (is_B_ready()) { read B }
   else {read A}
   .....
```

11. In the write operation, we have to make sure both A and B is ready, so the condition is

```
if (!(is_A_ready() && is_B_ready())) {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
 }
int i;
unsigned short tmpw;
for (i = 0; i < 256; i++) {
tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
Machine::outportw(0x170, tmpw);
Machine::outportw(0x1F0, tmpw);
}
```

12. Now the read/write operation can be performed:



```
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[170]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN [illegible]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 ITICK [9]
FUN 2 IN ITERATION[169]
IPS: 14.331M    A:  NUM  CAPS  SCRL  HD:0-M  HD:1-M
```

3. Bonus 2

In this bonus assignment, we have to use interrupt to handle the disk read and write rather than scheduling activities explicitly. To do this, I firstly define the block_disk to be the inherited class of interrupt class. Here, I have made modification to the following files:

```
blocking_disk.C
blocking_disk.H
bochsrc.bxrc
kernel.C
makefile
mirror_disk.C
mirror_disk.H
scheduler.C
scheduler.H
simple_disk.C
simple_disk.H
thread.C
thread.H
```

1. In the threading files, I made the change to enable interrupt once the thread starts.

```
static void thread_start(){
     Machine::enable_interrupts();
     }
```

2. In `simple_disk.H/C` files, I added interrupt class as the parent class of disk class, so the interrupt handling method will be inserted into simple disk class.

```
class SimpleDisk: public InterruptHandler  {
....
virtual void handle_interrupt(REGS *_r){}
....
}
```

3. The "blocking_disk.H/C" inherited from simple_disk.H/C needs to have interrupt handler also. I used a private variable `thread` in the blocking disk class, and this thread is used to memorize the thread accessing the read/write operation. When interrupt occurs, the saved thread will be activated to handle the request.

```
class BlockingDisk : public SimpleDisk {
private:
    Thread * thread;
public:
   BlockingDisk(DISK_ID _disk_id, unsigned int _size);
   virtual void handle_interrupt(REGS *_r);
   virtual void read(unsigned long _block_no, unsigned char * _buf);
   virtual void write(unsigned long _block_no, unsigned char * _buf);
};
```

4. In the read/write operation, the blocking disk class will first issue operation to disks, then transfer control to the next thread and save the current thread to the blocking disk class.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
    issue_operation(READ, _block_no);
    if (!is_ready()) {
        thread = Thread::CurrentThread();
        SYSTEM_SCHEDULER->yield();
    }
    ....
}
```

5. In the interrupt handler, the handler will first save the current thread to the scheduler if it is not already in the current scheduler. In the meantime, since the previous read/write thread is already saved in the blocking class, now we can remove the thread from scheduler because we do not need the scheduler to schedule the thread now. And I also disabled the interrupt to avoid unnecessary unexpected error in the result, then dispatch to the read/write thread.

```
void BlockingDisk::handle_interrupt(REGS *_r) {

  if (thread == NULL) return;
      if (!SYSTEM_SCHEDULER->contains(Thread::CurrentThread()))
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
  SYSTEM_SCHEDULER->terminate(thread);
  if (Machine::interrupts_enabled()) Machine::disable_interrupts();
  Thread::dispatch_to(thread);
}
```

6. In the read/write thread, once the interrupt is handled, and thread is transferred back here, the read/write operation is done and thread variable is assigned to NULL.

```
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);
}
thread = NULL;
```

7. In kernel.C file, the interrupt handler needs to be registered with interrupt controller, so the interrupt can be handled properly.

```
SYSTEM_DISK = new BlockingDisk(MASTER, SYSTEM_DISK_SIZE);
InterruptHandler::register_handler(14, SYSTEM_DISK);
```

4. Bonus 3 & 4

In this bonus assignment, we have to use lock for read and write operation. In the beginning, I was thinking to use pthread_mutex_t and pthread_mutex_lock, to lock operations for read and write. Later on, I find these two operation can not be used properly, so I decided to use one simple variable in the blocking disk class to lock and unlock before read and write operation performed.

In my design, I used two boolean variable locked_write and locked_read to monitor the read or write operation performed on blocking_disk class. In the meantime, I also need to disable the interrupt during lock/unlock operation.

```
class BlockingDisk : public SimpleDisk {
private:
    bool locked_write;
    bool locked_read;
public:
   BlockingDisk(DISK_ID _disk_id, unsigned int _size);
   virtual void read(unsigned long _block_no, unsigned char * _buf);
   virtual void write(unsigned long _block_no, unsigned char * _buf);
   void lock_write();
   void lock_read();
   void unlock_write();
   void unlock_read();
};
```

Now, in the read or write operation, we need to check if the read or write operation is locked or not. If it is locked, we will give CPU to the other thread.

```
//add lock to read
if (locked_read) {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
}
```

And then, if the locked_read is not being locked, then the lock can be locked to prevent other read operation to continue. In the meantime, we have to avoid the interrupt during locking and read operation, so here I disabled the interrupt.

```
bool interupt_state_changed = false;
if (Machine::interrupts_enabled()) {
    Machine::disable_interrupts();
    interupt_state_changed = true;
}
lock_read();
....
perform read operation
....
unlock_read();
if (interupt_state_changed) {
    Machine::enable_interrupts();
interupt_state_changed = false;
```

After the read/write operation, the interrupts can be enabled again, and lock need to be unlocked.

The other helper methods I have added in this class is to lock or unlock :

```
void BlockingDisk::lock_read(){
    locked_read = true;
}
void BlockingDisk::unlock_read(){
    locked_read = false;
}
void BlockingDisk::lock_write(){
    locked_write = true;
}
void BlockingDisk::unlock_write(){
    locked_write = false;
}
```