MP6

Clarification: I finish the whole MP6, including the **basic** requirements, **option1**, **option2**, **option3** and **option4**.

Notice:
(1)Basic version, all source code files are in the MP6 directory.
(2)Option1, all source code files are in the MP6/op1 directory
(3)Option2, all source code files are in the MP6/op2 directory
(4)Option3, Describe in this document, '4.(**OPTION3**)Design a thread-safe disk system'.
(5)Option4, all source code files are in the MP6/op4 directory

1.**BlockingDisk**, without busy wait

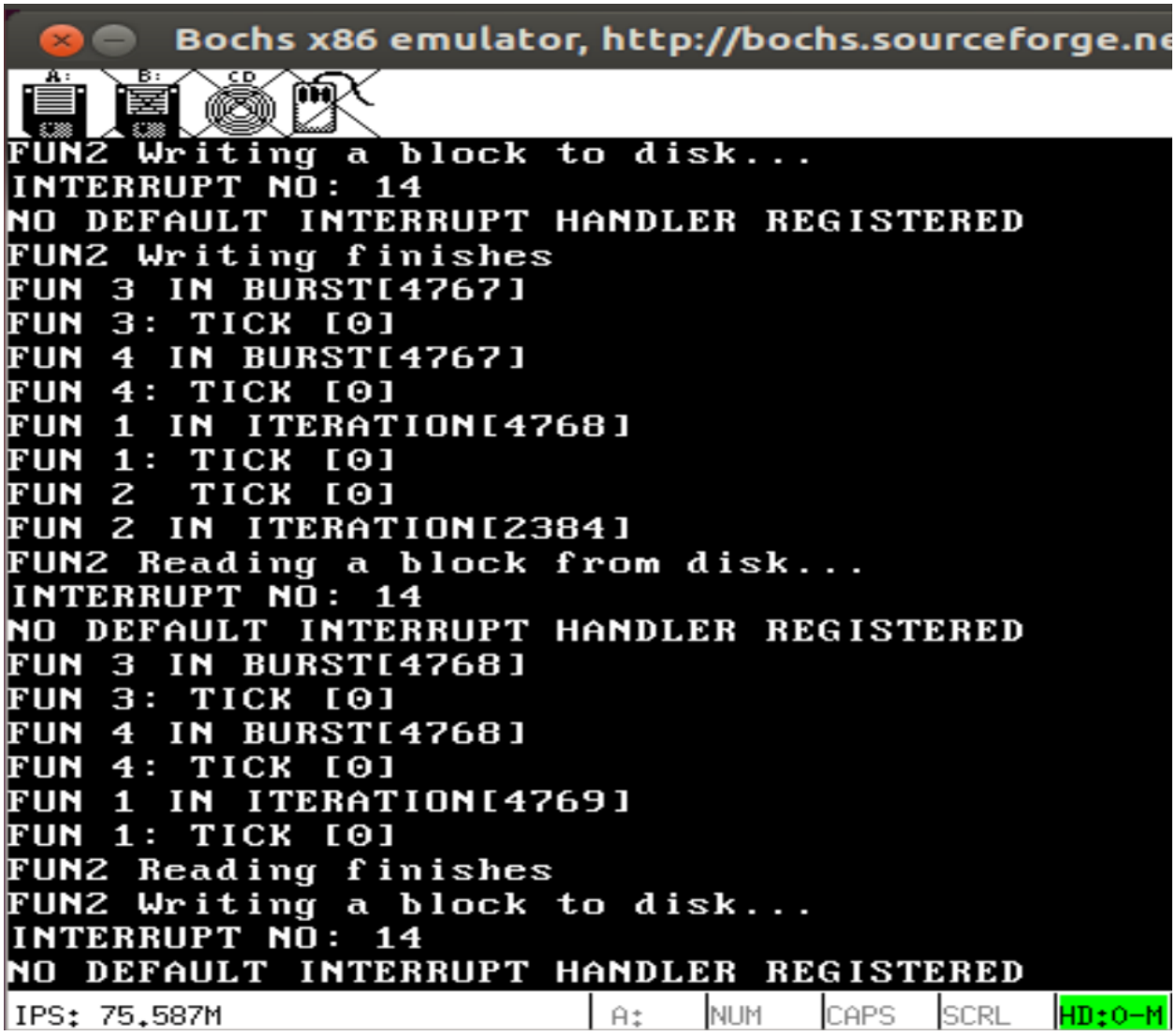
Based on and derived from SimpleDisk, mainly modify the wait_until_ready() from busy loop to give up CPU and wait if necessary. Also I use my Scheduler implemented in MP5.

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

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

Testing result:
As shown below, other threads will be running before the disk is ready to read/write, which means the blocking disk is not busy waiting for read/write.


Bochs x86 emulator, http://bochs.sourceforge.ne
A:
B:
CD
FUN2 Writing a block to disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN2 Writing finishes
FUN 3 IN BURST[4767]
FUN 3: TICK [0]
FUN 4 IN BURST[4767]
FUN 4: TICK [0]
FUN 1 IN ITERATION[4768]
FUN 1: TICK [0]
FUN 2 TICK [0]
FUN 2 IN ITERATION[2384]
FUN2 Reading a block from disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN 3 IN BURST[4768]
FUN 3: TICK [0]
FUN 4 IN BURST[4768]
FUN 4: TICK [0]
FUN 1 IN ITERATION[4769]
FUN 1: TICK [0]
FUN2 Reading finishes
FUN2 Writing a block to disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
IPS: 75.587M
A:
NUM
CAPS
SCRL
HD:0-M

2.(**OPTION1**)Support for **Disk Mirroring**

Class MirrorDisk

```
class MirrorDisk : public SimpleDisk {

private:
    unsigned int disk_size; // in byte
    DISK_ID disk_id;

public:
    MirrorDisk(DISK_ID _disk_id, unsigned int _size);
    MirrorDisk(unsigned int _size);

    void issue_read(unsigned long _block_no);

    void issue_write(DISK_ID _disk_id, unsigned long _block_no);

    virtual void read(unsigned long _block_no, unsigned char * _buf);

    virtual void write(unsigned long _block_no, unsigned char * _buf);

    virtual void wait_until_ready();

    virtual bool is_ready();

};
```

(2.1)**Read Process**

Function read will issue read to both MASTER and SLAVE disk, when any of this 2 disks is ready, we can read the data into buffer.

```
void MirrorDisk::read(unsigned long _block_no, unsigned char * _buf)
{
    issue_read(_block_no);
    wait_until_ready();

    unsigned short tmpw;
    for(int i=0;i<256;++i) {
        tmpw = Machine::inportw(0x1F0);
        _buf[i<<1] = (unsigned char)tmpw;
        _buf[i<<1 + 1] = (unsigned char)(tmpw>>8);
    }
}
```

```
void MirrorDisk::issue_read(unsigned long _block_no)
{
    // 1. issue read to MASTER
    Machine::outportb(0x1F1, 0x00); // send NULL to port 0x1F1
    Machine::outportb(0x1F2, 0x01); // send sector count to port 0x1F2
    Machine::outportb(0x1F3, (unsigned char)_block_no);  // low 8 bits
    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 | (
MASTER<<4));
    Machine::outportb(0x1F7, 0x20);

    // 2. issue read to SLAVE
    Machine::outportb(0x1F1, 0x00); // send NULL to port 0x1F1
    Machine::outportb(0x1F2, 0x01); // send sector count to port 0x1F2
    Machine::outportb(0x1F3, (unsigned char)_block_no);  // low 8 bits
    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 | (
SLAVE<<4));
    Machine::outportb(0x1F7, 0x20);
}
```

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

(2.2)**Write process**

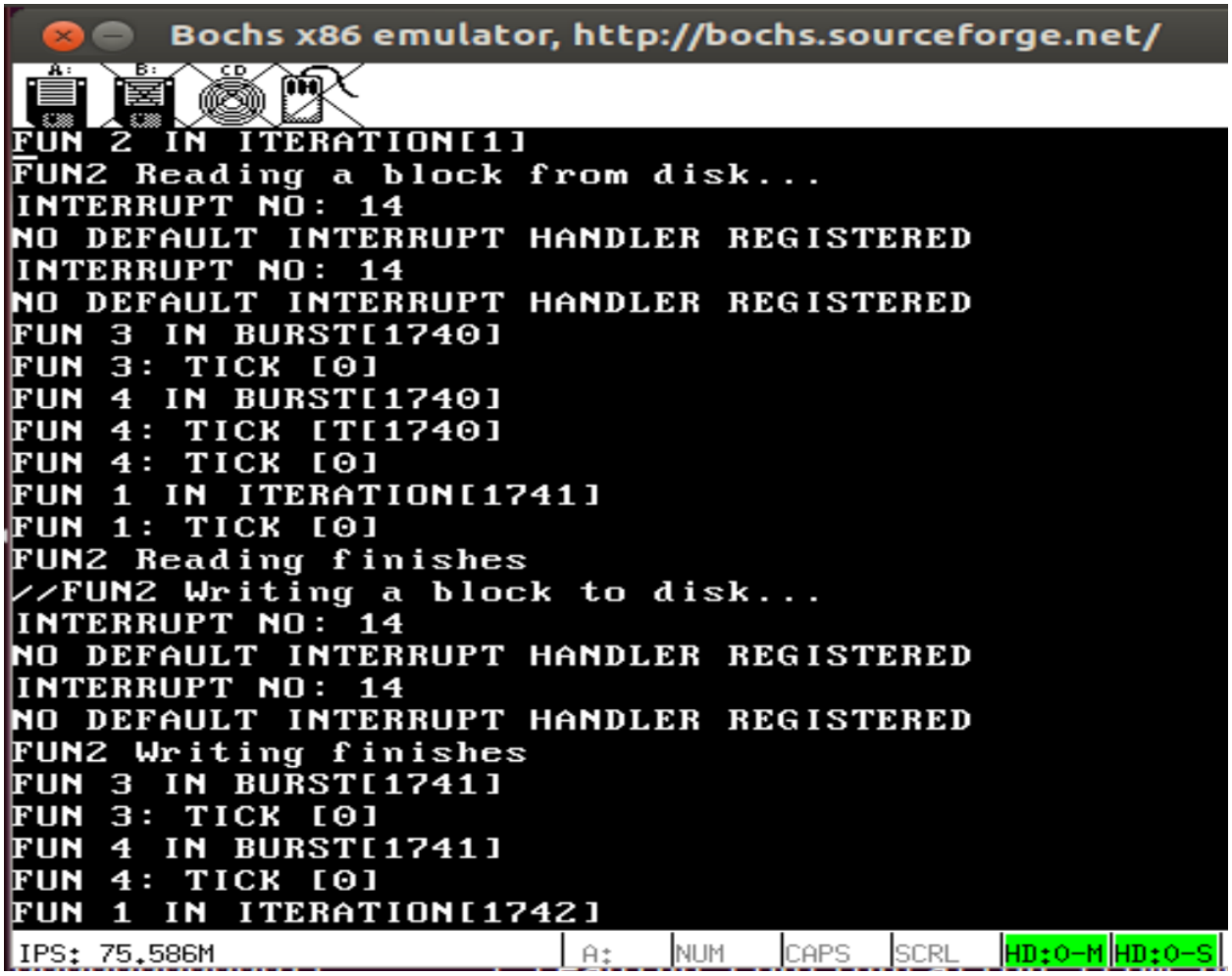
Function write will issue and write to both MASTER and SLAVE disks.

```
void MirrorDisk::write(unsigned long _block_no, unsigned char * _buf)
{
    // issue write to MASTER
    issue_write(MASTER, _block_no);
    wait_until_ready();

    // write to MASTER
    unsigned short tmpw;
    for(int i=0;i<256;++i) {
        tmpw = _buf[i<<1] | (_buf[i<<1 + 1]<<8);
        Machine::outportw(0x1F0, tmpw);
    }

    // issue write to SLAVE
    issue_write(SLAVE, _block_no);
    wait_until_ready();

    // write to SLAVE
    for(int i=0;i<256;++i) {
        tmpw = _buf[i<<1] | (_buf[i<<1 + 1]<<8);
        Machine::outportw(0x1F0, tmpw);
    }
}
```

```
void MirrorDisk::issue_write(DISK_ID _disk_id, unsigned long _block_no)
{
    Machine::outportb(0x1F1, 0x00); // send NULL to port 0x1F1
    Machine::outportb(0x1F2, 0x01); // send sector count to port 0x1F2
    Machine::outportb(0x1F3, (unsigned char)_block_no);  // low 8 bits
    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 | (
_disk_id<<4));
    Machine::outportb(0x1F7, 0x30);
}
```

(2.3)**Testing result**

As expected and shown below, write operations are issued to both disks, and read operations are supposed to return to the caller as soon as the first of the 2 disks is ready to return the data.


Bochs x86 emulator, http://bochs.sourceforge.net/
A:
B:
CD
FUN 2 IN ITERATION[1]
FUN2 Reading a block from disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN 3 IN BURST[1740]
FUN 3: TICK [0]
FUN 4 IN BURST[1740]
FUN 4: TICK [T[1740]
FUN 4: TICK [0]
FUN 1 IN ITERATION[1741]
FUN 1: TICK [0]
FUN2 Reading finishes
//FUN2 Writing a block to disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN2 Writing finishes
FUN 3 IN BURST[1741]
FUN 3: TICK [0]
FUN 4 IN BURST[1741]
FUN 4: TICK [0]
FUN 1 IN ITERATION[1742]
IPS: 75.586M
A:
NUM
CAPS
SCRL
HD:0-M
HD:0-S

3.(**OPTION2**)Using Interrupts for Concurrency

Assume the order for disk request and interrupt could be guaranteed to be FIFO, which means Interrupt of first disk request will occur first.

(3.1)**class Disk, disk.H and disk.C**

```
class Disk : public SimpleDisk {
private:

     DISK_ID       disk_id; // This disk is either MASTER or SLAVE

     unsigned int disk_size; // In Byte

     void issue_operation(DISK_OPERATION _op, unsigned long _block_no);

protected:
     /* -- HERE WE CAN DEFINE THE BEHAVIOR OF DERIVED DISKS */

     virtual bool is_ready();

     virtual void wait_until_ready() {
        while (!is_ready()) { /* wait */; }
     }
```

```
public:

   Disk(DISK_ID _disk_id, unsigned int _size);

   unsigned int size();

   /* DISK OPERATIONS */

   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. */

};
```

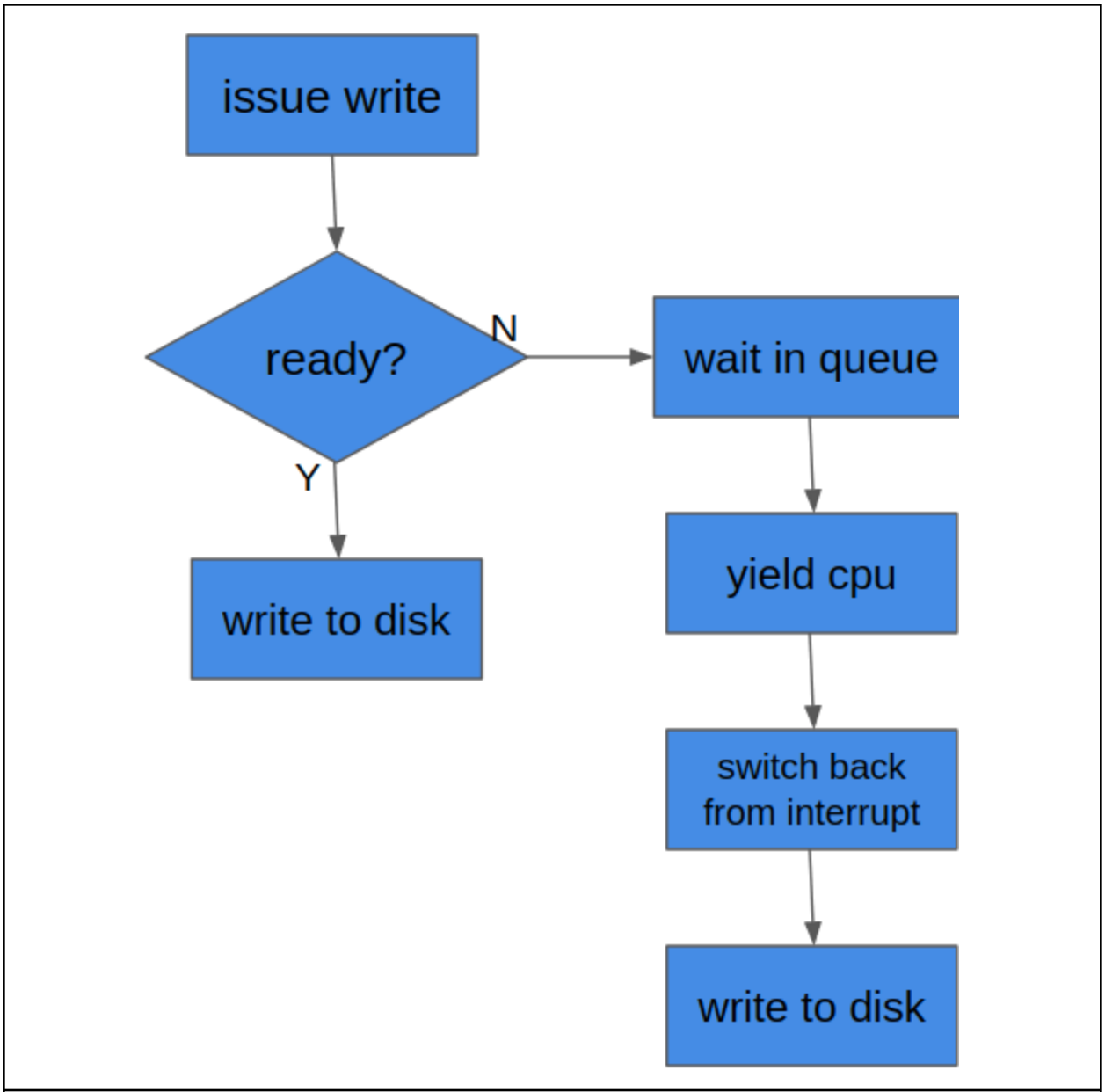
Read Process

1) Disable interrupt
2) Add current thread into the waiting queue, designed for threads to be waiting for the Interrupt 14 handler to deal with.
3) Yield CPU
4) Switched back from Interrupt 14 handler, and read the data from disk.

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

    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    SYSTEM_SCHEDULER->add_wait(Thread::CurrentThread());
    issue_operation(READ, _block_no);
    SYSTEM_SCHEDULER->yield();

    /* 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);
    }

    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();

}
```

Write Process

Notice, originally I expect the write process should be almost the same as the Read, but for unknown reason, if I directly yield CPU after issue_write, Interrupt 14 will never occur. I further notice, actually just after issue_write, the disk is already ready for write, so at least in most cases, if not all, it is unnecessary to yield CPU and wait for Interrupt 14 handler to deal with, I can just directly write to the disk if it is already ready after issue read.



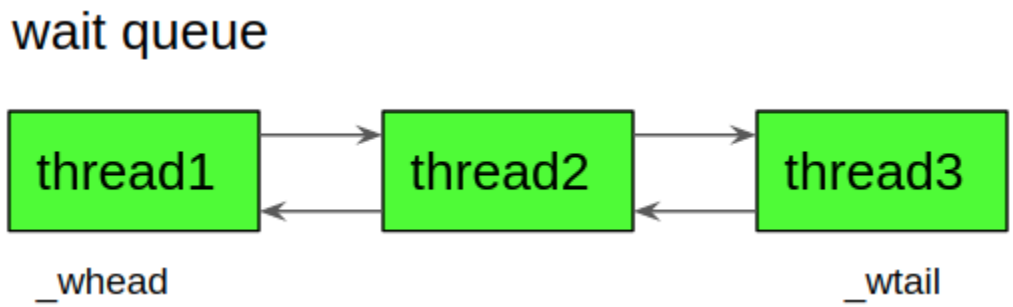
write process

```
 87 void Disk::write(unsigned long _block_no, unsigned char * _buf) {
 88
 89     if (Machine::interrupts_enabled())
 90         Machine::disable_interrupts();
 91
 92     issue_operation(WRITE, _block_no);
 93     if (is_ready()) {
 94     // just ready for to be written so we could just write to disk directly
 95     // and unnecessary to deal with it in InterruptHandler
 96         int i;
 97         unsigned short tmpw;
 98         for (i = 0; i < 256; i++) {
 99             tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
100             Machine::outportw(0x1F0, tmpw);
101         }
102
103     }else {
```

```
104     // in this case, Interrupt14 will occur
105         SYSTEM_SCHEDULER->add_wait(Thread::CurrentThread());
106         SYSTEM_SCHEDULER->yield();
107         // write data to port
108         int i;
109         unsigned short tmpw;
110         for (i = 0; i < 256; i++) {
111             tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
112             Machine::outportw(0x1F0, tmpw);
113         }
114     }
115
116     if (!Machine::interrupts_enabled())
117         Machine::enable_interrupts();
118
119 }
```

(3.2)**class Scheduler, scheduler.H and scheduler.C**

Based on my Scheduler of MP5, I add the waiting queue into it in order to manage/schedule the threads waiting for Interrupt14.




```
void Scheduler::add_wait(Thread *_thread) {
    assert(_thread!=NULL && _wsize>=0);

    Node *n = new Node;
    assert(n != NULL);
    n->t = _thread;

    if (_wsize) {
        n->next = NULL;
        if (_wsize == 1) {
            n->prev = _whead;
            _whead->next = n;
        }else {
            n->prev = _wtail;
            _wtail->next = n;
        }
        _wtail = n;

    }else {
        n->prev = NULL;
        n->next = NULL;
        _whead = n;
        _wtail = n;
    }

    ++_wsize;
}
```

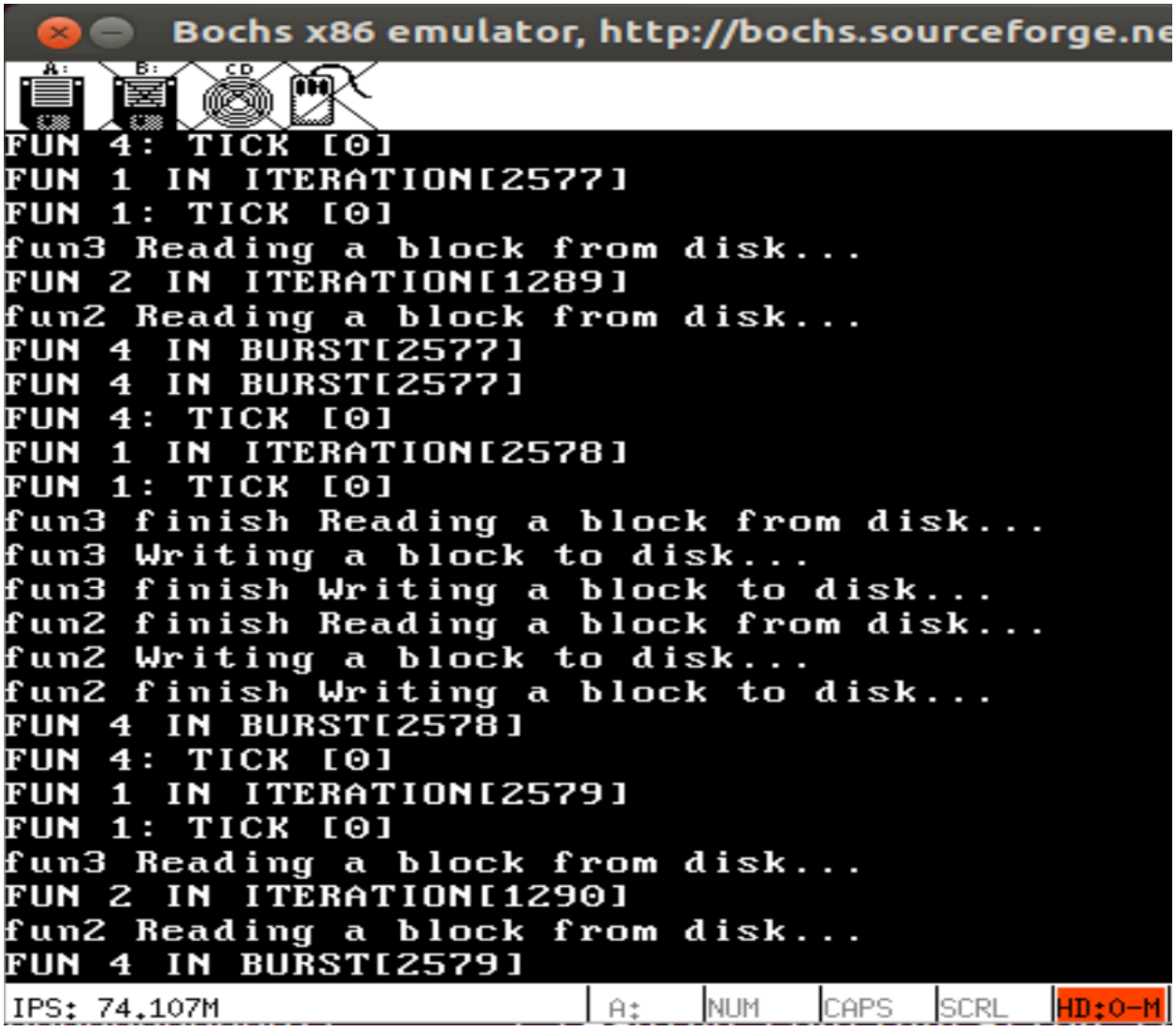
```
void Scheduler::switch_to_wait() {

    assert(_wsize>=0);
    if (_wsize==0)  return;

    Thread *t = _whead->t;

    // attain the first Node in wait queue
    Node *tmp = _whead;
    if (_wsize == 1) {
        _whead = NULL;
        _wtail = NULL;
    }else {
        _whead = _whead->next;
        _whead->prev = NULL;
    }
    --_wsize;
    delete tmp;

    Thread::dispatch_to(t);

}
```

(3.3)**class DiskBack, disk_back.H and disk_back.C**

Add the Interrupt 14 Handler.

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

    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    if (SYSTEM_SCHEDULER->wait_size()) {
        SYSTEM_SCHEDULER->raw_add(Thread::CurrentThread());
        SYSTEM_SCHEDULER->switch_to_wait();
    }

    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();

}
```

(3.4)**Testing**


Bochs x86 emulator, http://bochs.sourceforge.ne
A:
B:
CD
FUN 4: TICK [0]
FUN 1 IN ITERATION[2577]
FUN 1: TICK [0]
fun3 Reading a block from disk...
FUN 2 IN ITERATION[1289]
fun2 Reading a block from disk...
FUN 4 IN BURST[2577]
FUN 4 IN BURST[2577]
FUN 4: TICK [0]
FUN 1 IN ITERATION[2578]
FUN 1: TICK [0]
fun3 finish Reading a block from disk...
fun3 Writing a block to disk...
fun3 finish Writing a block to disk...
fun2 finish Reading a block from disk...
fun2 Writing a block to disk...
fun2 finish Writing a block to disk...
FUN 4 IN BURST[2578]
FUN 4: TICK [0]
FUN 1 IN ITERATION[2579]
FUN 1: TICK [0]
fun3 Reading a block from disk...
FUN 2 IN ITERATION[1290]
fun2 Reading a block from disk...
FUN 4 IN BURST[2579]
IPS: 74.107M
A:
NUM
CAPS
SCRL
HD:0-M

4.(**OPTION3**)Design a thread-safe disk system

Based on Round Robin Scheduling and Lock mechanism, I design my thread-safe disk system. I will design two versions, (1)one is a basic thread-safe disk system with a lock associated with the whole disk system; (2)another is an improvement, which has a smaller granularity lock associated with block_id.

(4.1)Basic thread-safe disk system

Treat the disk system read/right process as Critical Section, and use lock mechanism to protect the Critical Section. To be specific, before Read/Write, acquire lock; after Read/Write, release lock.

```
read(block_id, buffer) {
    lock();
    issue_read(block_id);
    wait_until_ready();//no busy wait, if not ready, will yield cpu
    read_buffer();
    unlock();
}

write(block_id, buffer) {
    lock();
    issue_write(block_id);
    wait_until_ready();//no busy wait, if not ready, will yield cpu
    write_buffer();
    unlock();
}
```

(4.2)Improvement

Small granularity lock associated with block_id.

```
read(block_id, buffer) {
    lock(block_id);
    issue_read(block_id);
    wait_until_ready();//no busy wait, if not ready, will yield cpu
    read_buffer();
    unlock(block_id);
}

write(block_id, buffer) {
    lock(block_id);
    issue_write(block_id);
    wait_until_ready();//no busy wait, if not ready, will yield cpu
    write_buffer();
    unlock(block_id);
}
```

5.(**OPTION4**)Implement a thread-safe disk system

Due to limited time, I will implement a thread-safe disk system based on a lock, which is whole disk_system wide.
Also, I will add Round Robin Scheduling strategy, which I implemented in the last Machine Probelm.

Before read/write, I will lock the disk; after read/write, I will unlock the disk.

(5.1)Class **TSDisk**, ts_disk.H, ts_disk.C

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

    lock(&mutex);
    Console::puts("Reading block ");Console::puti(_block_no);Console::puts("\
n");
    issue_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);
    }
    Console::puts("Finish reading block ");Console::puti(_block_no);Console::
puts("\n");
    unlock(&mutex);

}
```

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

    lock(&mutex);
    Console::puts("Writing block ");Console::puti(_block_no);Console::puts("\
n");
    issue_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);
    }
    Console::puts("Finish writing block ");Console::puti(_block_no);Console::
puts("\n");
    unlock(&mutex);

}
```

(5.2)**Lock** based on atomic operation. **lock.H, lock.C**

```
int TSL(int *addr);

void lock(int *lck);
void unlock(int *lck);
```

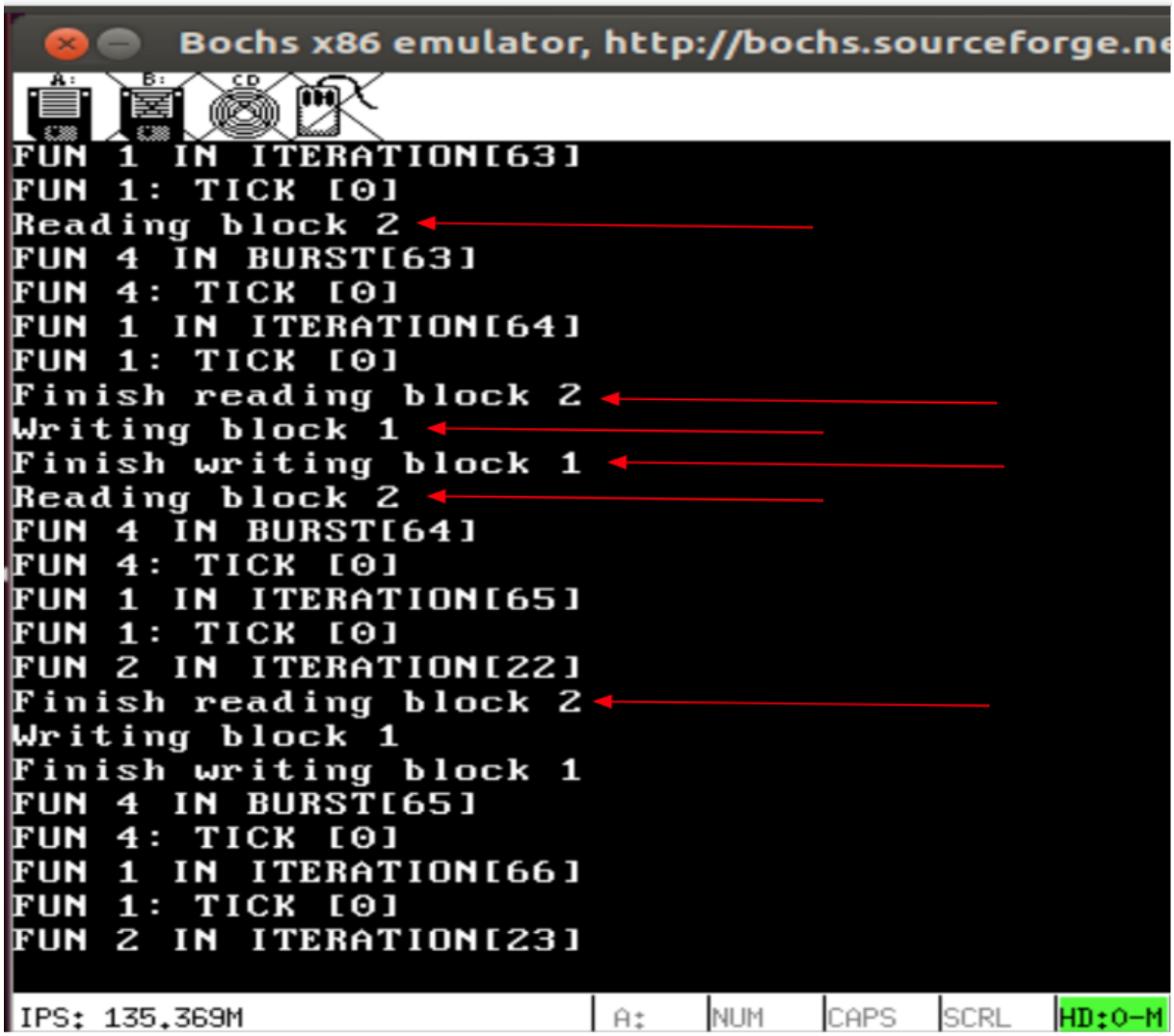
```
void lock(int *lck) {
    while (TSL(lck) != 0);
}

void unlock(int *lck) {
    *lck = 0;
}

int TSL(int *addr) {

    register int content = 1;
    // xchgl content, *addr
    // xchgl exchanges the values of its 2 operands,
    // while locking the memory bus to exclude other operations.
    asm volatile (
        "xchgl %0,%1" : "=r" (content),
        "=m" (*addr) : "0" (content),
        "m" (*addr)
    );
    return (content);

}
```

(5.3)**Round Robin** Scheduling based on simple_timer.

```
void SimpleTimer::handle_interrupt(REGS *_r) {
    /* Increment our "ticks" count */
    ticks++;

    if (ticks %5 == 0) {
        //Console::puts("50ms quantum ends\n");
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
    }
    /* Whenever a second is over, we update counter accordingly. */
    if (ticks >= hz )
    {
        seconds++;
        ticks = 0;
        //Console::puts("One second has passed\n");
    }
}
```

(5.4)**Testing**

As expected and shown below, anytime at most 1 thread can read/write on the disk, so thread-safe is guaranteed.


Bochs x86 emulator, http://bochs.sourceforge.ne
A:
B:
CD
FUN 1 IN ITERATION[63]
FUN 1: TICK [0]
Reading block 2
FUN 4 IN BURST[63]
FUN 4: TICK [0]
FUN 1 IN ITERATION[64]
FUN 1: TICK [0]
Finish reading block 2
Writing block 1
Finish writing block 1
Reading block 2
FUN 4 IN BURST[64]
FUN 4: TICK [0]
FUN 1 IN ITERATION[65]
FUN 1: TICK [0]
FUN 2 IN ITERATION[22]
Finish reading block 2
Writing block 1
Finish writing block 1
FUN 4 IN BURST[65]
FUN 4: TICK [0]
FUN 1 IN ITERATION[66]
FUN 1: TICK [0]
FUN 2 IN ITERATION[23]
IPS: 135.369M
A:
NUM
CAPS
SCRL
HD:0-M