MP6: Primitive Disk Device Driver

Cheng-Yun Cheng
UIN: 633002216
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1: Completed.
Bonus Option 2: Completed.
Bonus Option 3: Completed.
Bonus Option 4: Completed.

System Design

The goal of the machine problem is to implement a kernel-level device driver.

- Main part: When the read and write functions are called, the current thread would be put into a block queue and give up the CPU. Every time a thread yield the CPU, the scheduler would check the status of the disk. If the disk is ready, the scheduler will resume the first thread in the block queue (put it into the ready queue).

- Option 1, support for disk mirroring: A MirroredDisk class is implemented. When issuing a read operation, the data transfer would be done when one of the disks is ready. Write operations would be issued to both disks.

- Option 2, using interrupts for concurrency: Rather than checking the state of the disk at regular intervals, interrupt is used to put the thread which is waiting for transferring data into the ready queue. I found that the interrupt will occur after the write operation is finished. Therefore, I only used interrupt for read operation, and busy-waiting is used for write operation.

- Option 3, design of a thread-safe disk system: If multiple threads can access the disk, we should consider when to issue the operation. If there is already a thread issuing an I/O operation, then another thread cannot issue an operation at this time. It should issue the operation after the last one is finished. To do so, when a thread wants to execute an I/O operation, put it into the block queue and yield the CPU. A boolean variable need_issue is set to determine that whether the first thread in the block queue is able to call the issue_operation function. If need_issue is true, then dispatch to this thread. On the other hand, dispatch to the next thread in the ready queue. There is a simple example.

  1. Thread 2 issues an I/O operation, need_issus is set as false.
  2. Thread 2 waits for the disk and yields the CPU.
  3. Thread 3 wants to request I/O operation, but there is already an issue. Therefore, thread 3 yields.
  4. Disk is ready and Thread 2 transfers data. need_issue is set as true.
  5. Now Thread 3 can issue the operation.

Code Description (Main)

For the main part, I modified the blocking_disk.h, blocking_disk.c, and kernel.c. Also, queue.h, scheduler.h, and scheduler.c are added. Also, the issue_operation function in SimpleDisk is moved to protected part and modified as a virtual function.

kernel.c :

- _USES_SCHEDULER_ is used.
- Include blocking_disk.h.
- SimpleDisk is replaced with BlockingDisk.

blocking_disk.h: Data member :

- Queue block_queue: A queue to store threads which are waiting for the disk.
- wait_CPU: If the disk is ready and the thread which issued I/O operation is waiting for execution in the ready queue, wait_CPU is set as true.

blocking_disk.c: wait_until_ready : Put the current thread into the block queue and call the yield function to give up the CPU.

```
void BlockingDisk::wait_until_ready(){
  block_queue.enqueue(Thread::CurrentThread());
  Console::puts("Wait for disk, yield\n");
  SYSTEM_SCHEDULER->yield();
}
```

blocking_disk.c: disk_ready : Return true if the block queue is not empty and the disk is ready.

```
bool BlockingDisk::disk_ready(){
  return SimpleDisk::is_ready() && !block_queue.is_empty();
}
```

blocking_disk.c: is_wait_CPU : Return wait_CPU.

```
bool BlockingDisk::is_wait_CPU(){
  return wait_CPU;
}
```

blocking_disk.c: dequeue : Set wait_CPU as true. Then pop the first thread in the block queue and return it.

```
Thread * BlockingDisk::dequeue(){
  wait_CPU = true;
  return block_queue.dequeue();
}
```

blocking_disk.c: read : Call the read function in SimpleDisk. After finishing it, set wait_CPU as false.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  SimpleDisk::read(_block_no, _buf);
  wait_CPU = false;
  Console::puts("READ DONE\n");
}
```

blocking_disk.c: write : Call the write function in SimpleDisk. After finishing it, set wait_CPU as false.

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  SimpleDisk::write(_block_no, _buf);
  wait_CPU = false;
  Console::puts("WRITE DONE\n");
}
```

scheduler.h: Scheduler : The most part of Scheduler is as same as the Scheduler in MP5, except yield function.

- Data member: Queue ready_queue
- Member function: yield, resume, add, terminate

scheduler.c: yield : Check whether the disk is ready or not first. If yes, pop the thread from the block queue and put it into the ready queue. Then, get the next thread from the ready queue and call the dispatch_to function to invoke the context switch.

```
void Scheduler::yield() {
  if(SYSTEM_DISK->disk_ready() && !SYSTEM_DISK->is_wait_CPU()){
    resume(SYSTEM_DISK->dequeue());
    Console::puts("Disk is ready, put thread into ready queue\n");
  }

  Thread * next = ready_queue.dequeue();

  Thread::dispatch_to(next);
}
```

queue.h: Queue : Queue is implement by array. It is as same as the Queue class in MP5. The member functions include is_empty, enqueue, dequeue, and remove.

Testing (Main)

For the machine problem, I only use the provided test. Only thread 2 requests I/O operation. The result shows that after issuing the read and write operation, thread 2 yields the CPU. After the disk is ready, thread 2 is put into ready thread again to wait for transfer data.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Thread2 Reading a block from disk...
Wait for disk, yield
Disk is ready, put thread into ready queue
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
```

```
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
READ DONE
Thread20000000000000000000000000000000000000
Thread2 Writing a block to disk...
Wait for disk, yield
Disk is ready, put thread into ready queue
FUN 3 IN BURST[1]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
```

Code Description (Option 1)

I added the MirroredDisk class in blocking_disk.h and blocking_disk.c.

blocking_disk.h: MirroredDisk : The MirroredDisk class is derived from the SimpleDisk class, and most functions in MirroredDisk is as same as the functions in BlockingDisk.

```
class MirroredDisk : public SimpleDisk{
private:
    Queue block_queue;
    bool wait_CPU;
protected:
    virtual void issue_operation(DISK_OPERATION _op, unsigned long _block_no, DISK_ID _disk_id);
    virtual void wait_until_ready();
    void _wtite(unsigned long _block_no, unsigned char * _buf, DISK_ID _disk_id);
public:
    MirroredDisk(unsigned int _size);
    bool disk_ready();
    bool is_wait_CPU();
    Thread * dequeue();
    virtual void read(unsigned long _block_no, unsigned char * _buf);
    virtual void write(unsigned long _block_no, unsigned char * _buf);
};
```

blocking_disk.c: issue_operation : The function is as same as the issue_operation function in SimpleDisk, except a DISK_ID argument. This argument is used to determine which disk we will issue operation to.

```
void MirroredDisk::issue_operation(DISK_OPERATION _op, unsigned long _block_no, DISK_ID _disk_id){
  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);
                                /* send low 8 bits of block number */
  Machine::outportb(0x1F4, (unsigned char)(_block_no >> 8));
                                /* send next 8 bits of block number */
  Machine::outportb(0x1F5, (unsigned char)(_block_no >> 16));
                                /* send next 8 bits of block number */
  unsigned int disk_no = _disk_id == DISK_ID::MASTER ? 0 : 1;
  Machine::outportb(0x1F6, ((unsigned char)(_block_no >> 24)&0x0F) | 0xE0 | (disk_no << 4));
                                /* send drive indicator, some bits,
                                   highest 4 bits of block no */

  Machine::outportb(0x1F7, (_op == DISK_OPERATION::READ) ? 0x20 : 0x30);

}
```

blocking_disk.c: _write : A DISK_ID argument is added to determine which disk would do the operation..

```
void MirroredDisk::_wtite(unsigned long _block_no, unsigned char * _buf, DISK_ID _disk_id){
  issue_operation(DISK_OPERATION::WRITE, _block_no, _disk_id);
  wait_until_ready();
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++) {
    tmpw = _buf[2*i] | (_buf[2*i+1] << 8);
    Machine::outportw(0x1F0, tmpw);
  }
}
```

blocking_disk.c: read : The difference between this function and the read function in SimpleDisk is that the read operation is issued to both disks.

```
void MirroredDisk::read(unsigned long _block_no, unsigned char * _buf){
  issue_operation(DISK_OPERATION::READ, _block_no, DISK_ID::MASTER);
  issue_operation(DISK_OPERATION::READ, _block_no, DISK_ID::DEPENDENT);

  wait_until_ready();

  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);
  }
  wait_CPU = false;
  Console::puts("READ DONE\n");
}
```

blocking_disk.c: write : Call _write function for the master disk and the dependent disk.

```
void MirroredDisk::write(unsigned long _block_no, unsigned char * _buf){
  _wtite(_block_no, _buf, DISK_ID::MASTER);
  wait_CPU = false;
  Console::puts("WRITE MASTER DONE\n");

  _wtite(_block_no, _buf, DISK_ID::DEPENDENT);
  wait_CPU = false;
  Console::puts("WRITE DEPENDENT DONE\n");
}
```

Testing (Option 1)

I only use the provided test. To do this test, please uncomment _MIRRORED_DISK in kernel.c and scheduler.c.

```
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Thread2 Reading a block from disk...
Wait for disk, yield
Disk is ready, put thread into ready queue
THREAD: 2
FUN 3 INVOKED!
```

```
FUN 1: TICK [9]
READ DONE
Thread20000000000000000000000000000000000
Thread2 Writing a block to disk...
Wait for disk, yield
Disk is ready, put thread into ready queue
FUN 3 IN BURST[1]
```

```
FUN 1: TICK [8]
FUN 1: TICK [9]
WRITE MASTER DONE
Wait for disk, yield
Disk is ready, put thread into ready queue
FUN 3 IN BURST[2]
FUN 3: TICK [0]
```

```
FUN 1: TICK [8]
FUN 1: TICK [9]
WRITE DEPENDENT DONE
Thread2 finish
FUN 3 IN BURST[3]
FUN 3: TICK [0]
```

Code Description (Option 2)

I changed scheduler.c, blocking_disk.h, and blocking_disk.c. Also, an interrupt handler is added in kernel.c and enable_interrupt is added in thread.c.

blocking_disk.h: data member : bool done is added. It is used to determine that when interrupt 14 occurs, the data transfer is done or not. For read operation, interrupt occurs before transferring data. However, for write operation, interrupt occurs after transferring data.

blocking_disk.c: is_done : Return done.

```
bool BlockingDisk::is_done(){
  return done;
}
```

blocking_disk.c: read : Most parts are the same. In the read function, interrupt is disabled. done is set as false after isseu_operation and set as true after data transfer.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
#ifdef _INTERRUPT
  bool enable = false;
  if(Machine::interrupts_enabled()){
    Machine::disable_interrupts();
    enable = true;
  }
#endif

  SimpleDisk::issue_operation(DISK_OPERATION::READ, _block_no);
  done = false;

  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);
  }

  wait_CPU = false;
  done = true;
  Console::puts("READ DONE\n");

#ifdef _INTERRUPT
  if(enable)
    Machine::enable_interrupts();
#endif
}
```

blocking_disk.c: write : Most parts are the same. In the write function, interrupt is disabled. done is set as false after isseu_operation and set as true after data transfer. Besides, busy_waiting is used.

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
#ifdef _INTERRUPT
bool enable = false;
  if(Machine::interrupts_enabled()){
    Machine::disable_interrupts();
    enable = true;
  }
#endif

  SimpleDisk::issue_operation(DISK_OPERATION::WRITE, _block_no);
  done = false;

  while(!is_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);
  }

  wait_CPU = false;
  done = true;
  Console::puts("WRITE DONE\n");

#ifdef _INTERRUPT
  if(enable)
    Machine::enable_interrupts();
#endif
}
```

scheduler.c: yield : Interrupt is disabled, and just pop a thread in the ready queue and dispatch to it.

```
void Scheduler::yield() {
#ifdef _INTERRUPT
  bool enable = false;
  if(Machine::interrupts_enabled()){
    Machine::disable_interrupts();
    enable = true;
  }
#endif

  Thread * next = ready_queue.dequeue();
  Thread::dispatch_to(next);

#ifdef _INTERRUPT
  if(enable)
    Machine::enable_interrupts();
#endif
}
```

kernel.c: DiskHandler : It is used for handling interrupt 14. When interrupt occurs, if done is false (read operation), pop a thread in the block queue and add it into the ready queue. If done is true (write operation), do nothing.

```
#ifdef _INTERRUPT
    /* interrupt handler for disk */
    class DiskHandler : public InterruptHandler{
    public:
      virtual void handle_interrupt(REGS * _r) {
        Console::puts("Disk Interrupt Handler\n");
        if(!((BlockingDisk*)SYSTEM_DISK)->is_done()){
            Console::puts("put into ready queue\n");
            Thread * t = ((BlockingDisk*)SYSTEM_DISK)->dequeue();
            SYSTEM_SCHEDULER->resume(t);
        }
      }
    } disk_handler;
    InterruptHandler::register_handler(14, &disk_handler);
#endif
```

Testing (Option 2)

I only use the provided test. To do this test, please uncomment the _INTERRUPT in kernel.c, schduler.c, blocking_disk.c, and thread.c.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Thread2 Reading a block from disk...
Wait for disk, yield
Disk Interrupt Handler
put into ready queue
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
```

```
FUN 1: TICK [8]
FUN 1: TICK [9]
READ DONE
Thread200000000000000000000000000000
Thread2 Writing a block to disk...
WRITE DONE
Disk Interrupt Handler
Thread2 finish
FUN 3 IN BURST[1]
FUN 3: TICK [0]
FUN 3: TICK [1]
```

Code Description (Option 4)

I changed scheduler.c, blocking_disk.h, and blocking_disk.c.

blocking_disk.h: data member : bool issued is added. It means that whether an operation is issued.

blocking_disk.c: need_issued : Return true, if there is not an operation issue and there is a thread in the block queue. It means that the the first thread in the block queue needs to issue an operation.

```
bool BlockingDisk::need_issued(){
  return !issued && !block_queue.is_empty();
}
```

blocking_disk.c: get_head : Get and return the first thread in the block queue (not pop, it is still in the queue).

```
Thread * BlockingDisk::get_head(){
  return block_queue.get_head();
}
```

blocking_disk.c: read :

- First, add the current thread into the block queue and yield the CPU.
- Get CPU again (when it is time for issue operation), issue the operation and set issued as true. Then yield the CPU.
- Get CPU again (when the disk is ready), transfer the data. After that, set issued as false.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
  block_queue.enqueue(Thread::CurrentThread());
  Console::puts("Wait for issuing operation, yield\n");
  SYSTEM_SCHEDULER->yield();

  SimpleDisk::issue_operation(DISK_OPERATION::READ, _block_no);
  issued = true;
  done = false;

  Console::puts("Wait for disk, yield\n");
  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);
  }

  wait_CPU = false;
  done = true;
  issued = false;
  Console::puts("READ DONE\n");
}
```

blocking_disk.c: write : Like read function.

- First, add the current thread into the block queue and yield the CPU.
- Get CPU again (when it is time for issue operation), issue the operation and set issued as true. Then yield the CPU.
- Get CPU again (when the disk is ready), transfer the data. After that, set issued as false.

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
  block_queue.enqueue(Thread::CurrentThread());
  Console::puts("Wait for issuing operation, yield\n");
  SYSTEM_SCHEDULER->yield();

  SimpleDisk::issue_operation(DISK_OPERATION::WRITE, _block_no);
  done = false;
  issued = true;

  Console::puts("Wait for disk, yield\n");
  SYSTEM_SCHEDULER->yield();

  /* 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);
  }

  wait_CPU = false;
  done = true;
  issued = false;
  Console::puts("WRITE DONE\n");
}
```

scheduler.c: yield : First, check whether the disk is ready or not, if yes, pop the thread in the block queue and add it into the ready thread. Then, check whether a new operation can be issued, if yes, get the first thread (not pop) in the block queue and dispatch to it. If no, dispatch to the thread in the ready queue.

```
void Scheduler::yield() {
  if(SYSTEM_DISK->disk_ready() && !SYSTEM_DISK->is_wait_CPU()){
    resume(SYSTEM_DISK->dequeue());
    Console::puts("Disk is ready, put thread into ready queue\n");
  }
  if(SYSTEM_DISK->need_issued())
    Console::puts("Resume thread for issue operation\n");
  Thread * next = SYSTEM_DISK->need_issued() ?
                  SYSTEM_DISK->get_head() : ready_queue.dequeue();

  Thread::dispatch_to(next);
}
```

Testing (Option 4)

To test option 4, I modified the fun3 to let thread 3 do I/O operation too. To do this test, please uncomment the _THREAD_SAFE in kernel.c, scheduler.c, and blocking_disk.c.

- There is no issue at the begin. Therefore, the thread 2 can issue an operation. Then it waits for disk and dispatches CPU to thread 3. Thread 3 wants to execute read operation too, but it cannot issue operation now, so it yield the CPU.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Thread2 Reading a block from disk...
Wait for issuing operation, yield
Resume thread for issue operation
Wait for disk, yield
Disk is ready, put thread into ready queue
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
Thread3 Reading a block from disk...
Wait for issuing operation, yield
THREAD: 3
FUN 4 IN BURST[0]
FUN 4: TICK [0]
```

- After disk is ready, thread 2 finishes data transfer and wants to issue a write operation. Put thread 2 into the block queue and yield the CPU. Now, thread 3 can issue the read operation, so resume thread 3.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
READ DONE
Thread20000000000000000000000000000000000000000
Thread2 Writing a block to disk...
Wait for issuing operation, yield
Resume thread for issue operation
Wait for disk, yield
Disk is ready, put thread into ready queue
FUN 4 IN BURST[1]
FUN 4: TICK [0]
FUN 4: TICK [1]
```

- After disk is ready, thread 3 finishes its read operation.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
READ DONE
Thred3000000000000000000000000000000000000000000
Writing a block to disk...
Wait for issuing operation, yield
Resume thread for issue operation
Wait for disk, yield
Disk is ready, put thread into ready queue
FUN 4 IN BURST[2]
FUN 4: TICK [0]
FUN 4: TICK [1]
```