MP6: Primitive Disk Device Driver

Nikhil Nunna
UIN: 525008698
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Option I: Omitted.
Option II: Omitted.
Option III: Omitted.
Option IV: Omitted.

System Design

The goal of this program was to create an Scheduler that could handle threads that were waiting for IO operations. This is done by managing an IO queue or list of threads. When a thread is awaiting an IO operation it is passed off into IO queue and it yields the CPU. This prevents CPU clock cycles from being wasted by an idle thread waiting for an IO operation. Whenever we add another thread to the back of the ready queue we check if there is a thread waiting in the IO queue, if there is we then we check if the IO operation is done, if it is then we can add it to the ready queue. If not we leave the thread on the IO the queue.

Code Description

The code compiles and runs normally. All the grader needs to do is run 'make', './copykernel.sh' and then 'bochs -f bochsrc.bxrc'. As an alternative to this, I've created test.sh which can be run using './test.sh'. The test.sh shell script just runs make, copykernel and the bochs command for you.

test.sh: code block : This shell file compiles and runs the project

```
#!/bin/bash

make
./copykernel.sh
bochs -f bochsrc.bxrc
```

scheduler.C: constructor : In this function, I set initialize the ready queue and the IO queue by setting up the head and tail pointers.

```
Scheduler::Scheduler() {
  head = nullptr;
  tail = nullptr;
  iohead = nullptr;
  iotail = nullptr;
  Console::puts("Constructed Scheduler.\n");
}
```

scheduler.C: resume : In this function, I append the thread to resume to the back of the ready queue/list if there are other threads in the list otherwise I put it at the front of the list. I then do the same thing for IO after checking there are processes waiting in it and that IO operations are complete

```
void Scheduler::resume(Thread * _thread) {
  // Check if anything in ready queue
  // if nothing place at head of ready queue
  if (head == nullptr)
  {
    head = _thread;
    head->setnextthread(nullptr);
    tail = _thread;
  }
  // If others in queue append to end of queue
  // then set as new end of queue
  else
  {
    tail->setnextthread(_thread);
    tail = _thread;
    tail->setnextthread(nullptr);
  }

  // Thread ptr to access thread on io queue
  Thread * io_thread = nullptr;

  if (iohead != nullptr)
  {
    if ((Machine::inportb(0x1F7) & 0x08) != 0)
    {
      // get the next thread in io queue
      io_thread = iohead;
      // remove the current one from queue
```

```
        iohead = iohead->getnextthread();
        // check if next is nullptr
        if (iohead == nullptr) {
          iotail = nullptr;
        }

        // append the thread to the back of the ready queue.
        tail->setnextthread(io_thread);
        tail = io_thread;
        tail->setnextthread(nullptr);
      }
    }

}
```

scheduler.C: add_to_queue : In this function, I add a thread to the IO queue and yield the CPU to the next thread on the ready queue.

```
void Scheduler::add_to_ioqueue(Thread * _thread){
  // Check if anything in IO queue
  // if nothing place at head of ready queue
  if (iohead == nullptr)
  {
    iohead = _thread;
    iohead->setnextthread(nullptr);
    iotail = _thread;
  }
  // If others in queue append to end of queue
  // then set as new end of queue
  else
  {
    iotail->setnextthread(_thread);
    iotail = _thread;
    iotail->setnextthread(nullptr);
  }
  // yield cpu to next thread.
  yield();
}
```

Blocking_Disk.C: constructor : In this function, I just use as a wrapper for the constructor from SimpleDisk class they have the same functionality.

```
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size) {
}
```

Blocking_Disk.C: issue_operation : In this function, I issue the operation for the emulated disk to move to the correct location for me to write or read from.

```
void BlockingDisk::issue_operation(DISK_OPERATION _op, 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);
                              /* 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: read : In this function, I call the issue operation function to move to the correct location then if the disk operation is not done I add the thread to the IO queue, when the IO operation is done and the thread is back on the ready queue I read 512 bytes from the disk.

```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  // SimpleDisk::read(_block_no, _buf);

  issue_operation(DISK_OPERATION::READ, _block_no);
  // after issuing operation add to io queue in scheduler
  // have the io queue get checked everytime resume is called
  // if there is something on the io queue call ((Machine::inportb(0x1F7) & 0x08) != 0)
  // this will tell us if there is an operation going on
  // if there is then leave on io queue otherwise pop off and add back to ready queue
  if(!is_ready())
  {
    SYSTEM_SCHEDULER->add_to_ioqueue(current_thread);
  }


  /* 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: write : In this function, I call the issue operation function to move to the correct location then if the disk operation is not done I add the thread to the IO queue, when the IO operation is done and the thread is back on the ready queue I write 512 bytes to the disk.

```
void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
  // -- REPLACE THIS!!!
  //SimpleDisk::write(_block_no, _buf);

  issue_operation(DISK_OPERATION::WRITE, _block_no);

  if(!is_ready())
  {
    SYSTEM_SCHEDULER->add_to_ioqueue(current_thread);
  }

  /* 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);
  }
}
```

kernel.C: fun2 : I modified this function when printing read and write to block so I could clearly see when thread 1 or function 2 was running.

```
void fun2() {
     Console::puts("THREAD: "); Console::puti(Thread::CurrentThread()->ThreadId());
     Console::puts("\n");

     Console::puts("FUN 2 INVOKED!\n");

     unsigned char buf[DISK_BLOCK_SIZE];
     int   read_block  = 1;
     int   write_block = 0;

     for(int j = 0;; j++) {

         Console::puts("FUN 2 IN ITERATION["); Console::puti(j); Console::puts("]\n");

         /* -- Read */
         Console::puts("FUN 2 Reading a block from disk...\n");
         SYSTEM_DISK->read(read_block, buf);

         /* -- Display */
         for (int i = 0; i < DISK_BLOCK_SIZE; i++) {
              Console::putch(buf[i]);
         }

         Console::puts("FUN 2 Writing a block to disk...\n");
         SYSTEM_DISK->write(write_block, buf);

         /* -- Move to next block */
         write_block = read_block;
         read_block  = (read_block + 1) % 10;

         /* -- Give up the CPU */
         pass_on_CPU(thread3);
     }
}
```

Testing

I just used the tests given to prove that my implementation of the IO queue worked. I had to make a change to the test where I printed FUN 2 before the reading and writing prints in order to clearly tell where exactly fun 2 was running. I believe this level of testing is enough for my implementation as I can see that the process is being removed from the ready queue and added back on from the IO queue once the IO operation is complete. I also added some debug prints when I was working on was resolving an issue where threads where not being removed from the ready queue, but these were resolved after I found the issue in my add_to_queue function where I was checking the head of the ready queue not the head of the IO queue which led me adding new threads to the tail rather than initializing the head and changing it from nullptr. So considering the given tests were enough to test my full implementation and correct any errors I believe that this is enough testing.