MP6: Primitive Device Driver

Completed	@April 16, 2023
Files & media	
Status	Done
Type	MP

Sofia Ortega

UIN: 830002159

CSCE410: Operating System

Assigned Tasks

Main: Completed

Bonus Option 1: Did not Attempt

Bonus Option 2: Did not attempt

Bonus Option 3: Did not attempt

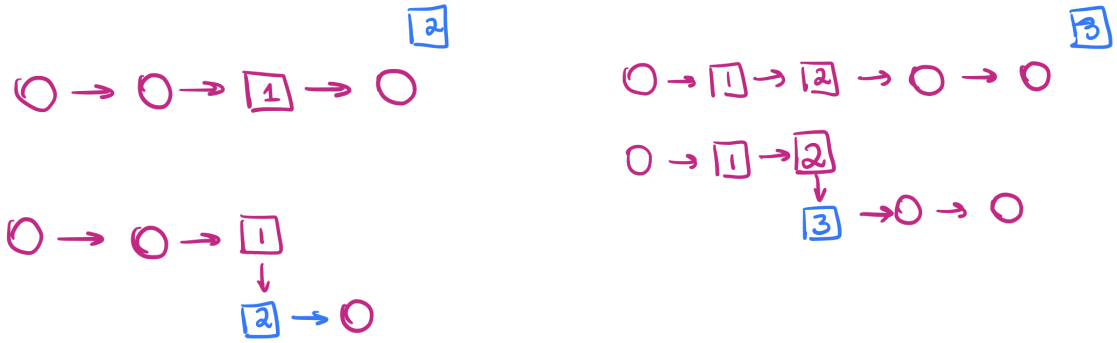
Bonus Option 4: Did not attempt

System Description

The Scheduler remains largely the same from MP5. Only additions to the interface occurred. When a disk operation occurred, rather than continuing to poll, I yielded the CPU through the scheduler. The scheduler detected that it was a disk operation. Thus, it placed it into BOTH the disk queue and ready queue.

I did not use the `resume` function in my Scheduler. Rather, I always assumed that the thread wanted to be added back into the ready queue during my `yield()`. When I wish to terminate a thread, I set a flag `terminated` in my thread to remove it from the queue.

The disk queue kept track of all the threads waiting for the disk to issue its operation. In essence, it allowed disk threads to “cut in line” into the ready queue. See Figures below.



This was a good trade off between long waiting times, but high CPU utilization

Code Description

Scheduler.C

yieldDisk()

This method is called when a function has to wait for a disk operation. Rather than busy waiting, it yields the CPU and notifies the scheduler

We set the thread `bool disk` to true to indicate this is a thread waiting on a disk operation. We find the future head, the future running thread.

Check to see if the disk queue is empty or not. If empty, make sure to add disk queue’s head to ready queue. If not empty, just insert new thread into ready queue (cutting in

line with the other queues).

At the end, print out both the ready and the disk queue for testing purposes.

```
void Scheduler::yieldDisk() {
  // have thread cut in line with rest of disk queues
  curr_thread->disk = true;

  // get next head
  head = head->next;
  while(head->terminated) head = head->next;



  // add to disk queue
  if(diskHead == nullptr) {
    diskHead = curr_thread;
    diskTail = curr_thread;
    curr_thread->next = nullptr;


    // add diskHead to ready queue
    tail->next = curr_thread;
    tail = diskTail;

  } else {
    // insert into ready queue
    curr_thread->next = diskTail->next;
    diskTail->next = curr_thread;

    if(diskTail == tail) tail = curr_thread;

    diskTail = curr_thread;

  }
```

```
Console::puts("yielded in disk queue \n");
printQueue();
printDiskQueue();


curr_thread = head;
curr_thread->dispatch_to(head);


}
```

yield()

Very similar to MP5 yield. Note that we automatically add threads back into the ready queue. (Does the function of resume within). If we want to remove a thread from the ready queue, we simply set `thread->terminated` to false, and the `yield()` function will automatically remove it.

New material starts with `if(old_thread->disk)`. Here, we handle if our thread was waiting for a disk operation. If it is yielding normally (by calling `yield()` rather than `yieldDisk`) we know that it was able to handle the disk operation and is ready to continue normally. Thus, we set our `disk` flag as false. We make sure to remove the thread from the disk queue. Later on, we will add the thread to the back of the ready queue, as it is now a *normal* thread.

```
void Scheduler::yield() {

  // temp var for old curr thread
  Thread* old_thread = curr_thread;

  // find new head
  Thread* new_head = head->next;
  while(new_head->terminated) {
    new_head = new_head->next;
  }

  if(old_thread->disk) {
    old_thread->disk = false; // disk operation successful

    // remove from disk queue
    while(diskHead != diskTail->next) {
      diskHead = diskHead->next;
    }

    if(diskHead == diskTail->next) {
      // disk queue is empty
      diskTail = nullptr;
      diskHead = nullptr;
    }
  }

  // add old head to the back of the queue
  if(!old_thread->terminated) {
    tail->next = head;
    tail = tail->next;
  }
```

```
    tail->next = nullptr;

    // start queue in correct place
    head = new_head;

    // update curr thread
    curr_thread = head;


    // context switch
    old_thread->dispatch_to(head);


    printQueue();
    printDiskQueue();
    Console::puts("Yielded\n");

}
```

resume()

In `yield()` we automatically add any running thread back into the ready queue. See **yield()** above for more information

```
void Scheduler::resume(Thread * _thread) {
  // handled in yield

}
```

printQueue() and printDiskQueue()

Two functions for testing purposes. Printed out both the *normal* ready queue and the disk queue

```
void Scheduler::printQueue() {
  Thread* curr = head;
  Console::puts("\n-----------------\nQueue: ");
  while(curr != nullptr) {
    Console::puts("[ Thread "); Console::putui(curr->ThreadId()); Console::puts("] ->");
    curr = curr->next;
  }
  Console::puts("\n");
}

void Scheduler::printDiskQueue() {
  Thread* curr = diskHead;
  Console::puts("\nDISK Queue: ");
  while(curr != diskTail->next) {
    Console::puts("[ Thread "); Console::putui(curr->ThreadId()); Console::puts("] ->");
    curr = curr->next;
  }
  Console::puts("\n-----------------------\n");

}
```

BlockingDisk

read()

Very similar to its parent class. Instead of just busy waiting, we yield the CPU, indicating to the scheduler that we are waiting for a **disk operation** by calling `yieldDisk`. We wait until `is_ready()` returns true.

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

  issue_operation(DISK_OPERATION::READ, _block_no);

  while (!is_ready()) {
    bool disk = true;
    Console::puts("\nNOT ready\n");
    SYSTEM_SCHEDULER->yieldDisk();
  }

    Console::puts("\nready it IS\n");

  /* 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);
  }

  SimpleDisk::read(_block_no, _buf);

}
```

write()

Very similar to its parent class. Instead of just busy waiting, we yield the CPU, indicating to the scheduler that we are waiting for a **disk operation** by calling `yieldDisk`. We wait until `is_ready()` returns true.

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

  issue_operation(DISK_OPERATION::WRITE, _block_no);

  while(!is_ready()) {
    Console::puts("\nNOT wridy\n");
    bool disk = true;
    SYSTEM_SCHEDULER->yieldDisk();
  }

    Console::puts("\nwridy it IS\n");

  /* 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);
  }

}
```

Thread

Added a `disk` flag to indicate whether the thread was currently handling a disk operation.

Testing

Test - Normal Operation

When a thread yields normally, do the other functions (non-disk operation) still run?

Yes. See below. Thread 3 yields, and Thread 4 begins.

```
Queue: [ Thread <3>] ->[ Thread <4>] ->[ Thread <0>] ->[ Thread <2>] ->[ Thread <5>
] ->

DISK Queue: [ Thread <5>] ->
----------------------
Yielded
FUN 3 IN BURST[45]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]

-----------------
Queue: [ Thread <4>] ->[ Thread <0>] ->[ Thread <2>] ->[ Thread <5>] ->[ Thread <3>
] ->

DISK Queue: [ Thread <5>] ->
----------------------
Yielded
FUN 4 IN BURST[45]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]

-----------------
Queue: [ Thread <0>] ->[ Thread <2>] ->[ Thread <5>] ->[ Thread <3>] ->[ Thread <4>
] ->

DISK Queue: [ Thread <5>] ->
----------------------
Yielded
```

Test - Disk Queue Initialization

Can the disk queue be initialized?

`fun2` yields for a disk operation. As seen below, It is added to the disk queue, and to the end of the ready queue, and successfully yields CPU to the next thread on the ready queue, thread 3

```
STARTING THREAD 1 ...
THREAD: 1
FUN 1 INVOKED!
FUN 1 IN ITERATION[0]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 2
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Reading a block from disk...

NOT ready
adding disk head as <2>
yielded in disk queue

-----------------
Queue: [ Thread <3>] ->[ Thread <4>] ->[ Thread <5>] ->[ Thread <0>] ->[ Thread <2>
] ->

DISK Queue: [ Thread <2>] ->
----------------------
THREAD: 3
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
```

Test - Cut Lines

Will a disk operation thread cut in line?

I added an extra `fun5` to `kernel.c` that was an exact copy of `fun2` in order to test the code when there where TWO functions who needed to read and write to the disk.

As seen in the image below. Thread 2 is preempted due to waiting for a disk operation. Instead of being put at the end of the ready queue, it cuts in line behind Thread 5, and is

kept track of by the disk queue.

```
-----------------
Queue: [ Thread <2>] ->[ Thread <5>] ->[ Thread <3>] ->[ Thread <4>] ->[ Thread <0>
] ->

DISK Queue: [ Thread <5>] ->
----------------------
Yielded
FUN 2 IN ITERATION[19]
Reading a block from disk...

NOT ready
yielded in disk queue

-----------------
Queue: [ Thread <5>] ->[ Thread <2>] ->[ Thread <3>] ->[ Thread <4>] ->[ Thread <0>
] ->

DISK Queue: [ Thread <5>] ->[ Thread <2>] ->
----------------------
```