CSCE 611: Operating Systems
MP6: Design Document

Shweta Sharma
UIN: 433003780

Design Idea:

Primitive Disk Device Driver – MP5:

I have implemented the following:

- Blocking Drive
- Option 3: Design of thread-safe disk system
- Option 4: Implementation of a thread-safe disk system

Blocking Drive:

The design of the blocking disk is done keeping in mind the options 3 and 4. To ensure the read and write operations don't block the CPU, I have implemented a thread blocking queue. Whenever a read/write operation is issued, the thread gives up the CPU and is added at the end of a blocked queue. Anytime the thread yields, the scheduler first checks if the front of the blocked queue is ready to take the CPU again. If yes, this thread is unblocked and given the CPU, otherwise, the CPU is given to the next thread in ready queue. Thus, both the blocked and ready queue work in FCFS fashion.

The Scheduler provides the following functionality:

• read(read block, buffer) – Reads the block into the buffer, using the SimpleDisk read api, that internally calls the wait_until_ready() function.

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

}
```

• write(write block, buffer) – Writes the buffer into the block, using the SimpleDisk write api, that internally calls the wait_until_ready() function.

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

• ready() – returns if the disk is ready using the SimpleDisk ready api

```
bool BlockingDisk::ready() {
  return SimpleDisk::is_ready();
}
```

• wait_until_ready() – overwrites the SimpleDisk wait_until_ready api. Instead of busy waiting, the thread is added to the end of the Blocking Disk Queue, and the thread give up the CPU to the next thread.

```
void BlockingDisk::wait_until_ready() {
  if(!BlockingDisk::ready()){
    Thread* curr_thread = Thread::CurrentThread();
    disk_queue -> enqueue(curr_thread);
    Console::puts("Disk not Available. Thread Added to Blocked Disk Queue.\n");
    SYSTEM_SCHEDULER -> yield();
  }
}
```

Option 3 and 4: Design and Implementation of thread-safe disk system

The current design of device drivers handles the scenario of multi-threading on a uni-core system. As in this machine problem, we are given only one processor, multiprogramming is achieved by multi-threading solely. When multiple threads perform the Read/Write operations, the threads are added to the blocked queue and yield the CPU. When doing so, the interrupts are enabled and disabled in the critical section, to ensure mutual exclusion.
For a multiprocessor system, to prevent race conditions between multiple CPUs, along with the above, protection need to be implemented using thread-safe queue and I/O device locking. The thread-safe queue will ensure multiple process don't incorrectly update the queue and the lock on the I/O will make sure that the system device is not given up if busy. However, this implementation can't be tested as we only have one processor in the current setup.

Updates to Scheduler:
A queue is used to maintain the ready threads with the scheduler in a preemptive-FCFS fashion (as implemented in MP5). Apart from this, the scheduler also maintains a pointer to the Blocking Disk (which internally references the Blocked Disk Queue). The BlockingDisk is registered with the Scheduler, at the start of the kernel. When yielding, the Scheduler checks this Blocked Disk Queue for a ready thread that is ready to re-acquire the CPU.

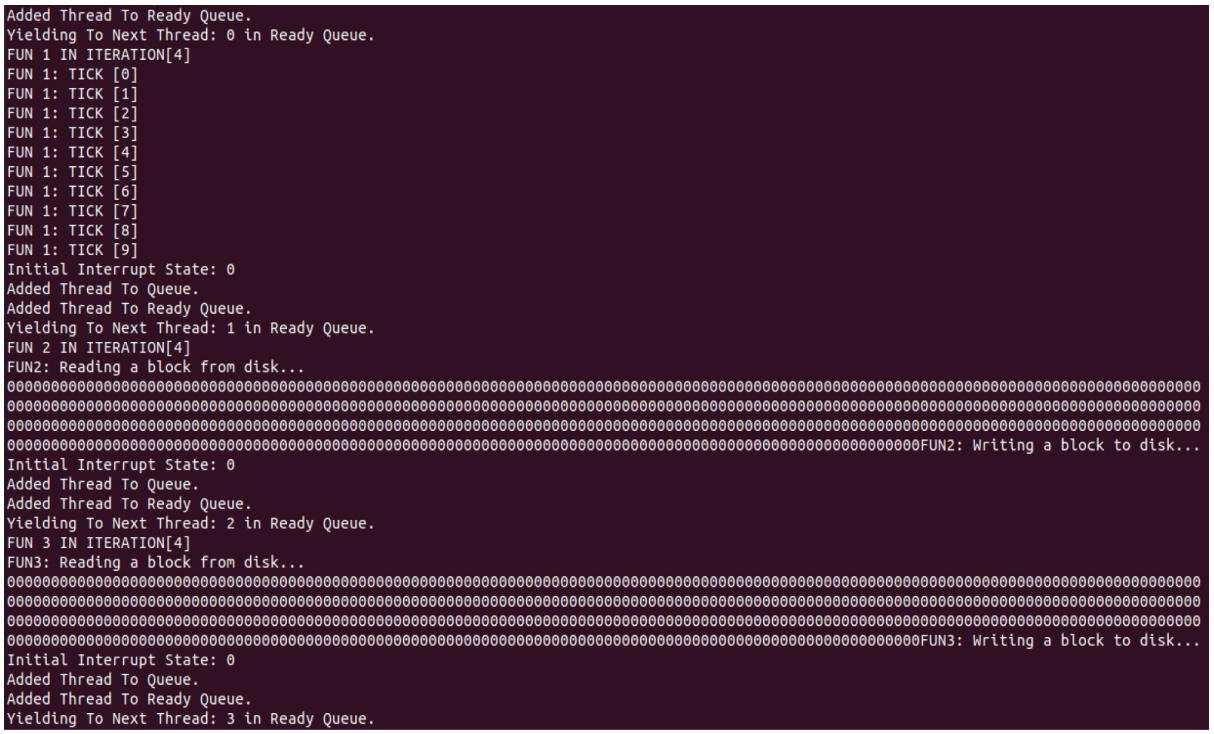
```
void Scheduler::yield() {
    //yield to a blocked thread if ready before yielding to next ready thread

    if(Scheduler::blocking_disk!=NULL && Scheduler::blocking_disk->ready() && !Scheduler::blocking_disk->disk_queue->empty()) {
        Thread* next_disk_thread = Scheduler::blocking_disk-> disk_queue -> dequeue();
        Console::puts("Yielding To Disk Thread: "); Console::puti(next_disk_thread->ThreadId()); Console::puts(" in Blocked Queue.\n");
        Thread::dispatch_to(next_disk_thread);
    }else{

        if(Scheduler::ready_queue->empty()){
            Console::puts("Ready Queue Is Empty.\n");
            assert(false);
        }
        //adding current thread to end of ready queue not needed, as current thread is premempted by kernal before yielding.

        //dispatch to next node
        Thread* next_ready_thread = Scheduler::ready_queue -> dequeue();

        //Scheduler::ready_queue->printQueue();
        Console::puts("Yielding To Next Thread: "); Console::puti(next_ready_thread->ThreadId()); Console::puts(" in Ready Queue.\n");
        Thread::dispatch_to(next_ready_thread);
    }
}
```

Test Screenshots / Output:

```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
BlockingDisk Initiated.
Hello World!
CREATING THREAD 1...
esp = <2098148>
done
DONE
CREATING THREAD 2...esp = <2099196>
done
DONE
CREATING THREAD 3...esp = <2100244>
done
DONE
CREATING THREAD 4...esp = <2101292>
done
DONE
Initial Interrupt State: 1
Disabled Interrupts Before Critical Section.
Enabled Interrupts After critical section.
```

As seen below, threads 0 and 1 perform read and write operations. When starting the read/write operation, if the disk is not available, it is added into the Blocked Disk Queue. This is shown using the log “Disk not Available. Thread Added to Blocked Disk Queue.” When yielding, the blocked queue is checked and if a thread is ready to resume, it is given the CPU. This is shown using the log message: “Yielding To Disk Thread: 1 in Blocked Queue.”

```
Enabled Interrupts After critical section.
Added Thread To Queue.
Added Thread To Ready Queue.
STARTING THREAD 1 ...
THREAD: 0
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]
Initial Interrupt State: 0
Added Thread To Queue.
Added Thread To Ready Queue.
Yielding To Next Thread: 1 in Ready Queue.
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
FUN2: Reading a block from disk...
Initial Interrupt State: 0
Added Thread To Queue.
Disk not Available. Thread Added to Blocked Disk Queue.
Yielding To Disk Thread: 1 in Blocked Queue.
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000FUN2: Writing a block to disk...
Initial Interrupt State: 0
Added Thread To Queue.
Added Thread To Ready Queue.
Yielding To Next Thread: 2 in Ready Queue.
```



Documents changed:

I have changed the following files:

- **queue.C** (new file) – to implement scheduler ready queue and blocking disk queue
- **queue.H** (new file) – to implement scheduler ready queue and blocking disk queue
- **kernel.C** – as per instructions in the handout for the compulsory parts, options 3 & 4
- **makefile** – updated to compile queue.H/C
- **scheduler.C** – as per instructions in the handout for the compulsory parts & options 3 & 4
- **scheduler.H** – as per instructions in the handout for the compulsory parts & options 3 & 4
- **blocking_disk.C** – as per instructions in the handout for the compulsory parts & options 3 & 4
- **blocking_disk.H** – as per instructions in the handout for the compulsory parts & options 3 & 4