MP6: Primitive Disk Device Driver

Geetesh Challur
UIN: 834006606
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1 Support for Disk Mirroring: Completed.
Bonus Option 2 Using Interrupts for Concurrency: Did not attempt.
Bonus Option 3 Design of a thread-safe disk system: Completed.
Bonus Option 4 Implementation of a thread-safe disk system: Did not attempt.

System Design

- A simple FIFO scheduler is used to handle the scheduling of threads when I/O operations such as read and write are used.
- On top of a simple I/O device, a layer on kernel level device drivers isimplemented
- . The layer was placed on top of this device to enable the same blocking read and write operations, eliminating busy waiting.
- A disk queue is implemented using a linked list, which contains all the processes blocked by the disk to be executed in the order.
- The scheduler first checks for blocked threads in disk queue and then proceeds to check the ready queue before dispatching.

Code Description

Describe your code setup here, any instruction about how to compile the code. For example, "I changed map.h, map.c, code.h, code.c for this machine problem. To compile the code use this and that in the kernel.c file."

scheduler.C: yield : The following are done in this function step by step

- If the disk queue is not empty, thread from disk queue is dispatched. Else, thread from ready queue is dispatched.
- Dispatch the next thread
- Thread deque operation
- Or Resume the thread from ready queue
- Dipatch the next thread

```
void Scheduler::yield() {
  // Interrupts are disabled in these schduler functions because they may modify the ready queue which requires mutual exclusion.
  // if (Machine::interrupts_enabled()) { Machine::disable_interrupts(); }

  // // Checking if the ready queue is empty.
  // if(head == nullptr) {
  //   Console::puts("No thread in empty queue to yield the current one.\n");
  //   assert(false);
  // }

  //If the disk queue is not empty, thread from disk queue is dispatched. Else, thread from ready queue is dispatched.
  if(blocking_disk->is_ready()&&blocking_disk->head != nullptr) {
        //Resume the thread from disk queue
        // Console::puts("Resume the thread from disk queue\n");
        // Thread deque operation
        Thread * temp = blocking_disk->head->thread;
        blocking_disk->head = blocking_disk->head->next;
        //Execute the next thread
        Thread::dispatch_to(temp);
  }else{
        //Resume the thread from ready queue
        // Console::puts("Resume the thread from ready queue\n");
        // If the ready queue is not empty, the first thread in the queue is dispatched.
        ready_queue * temp = head;
        head = head->next;
        // Interrupts are enabled again before leaving the function.
        // if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
        Thread::dispatch_to(temp->thread);
  }


}
```

Configuring project. Code can still execute in --dry mode. Do you want to continue?

blocking_disk.C: wait_until_ready, constructor, is_ready : The following are done in this function step by step

- Adding the thread to the end of disk queue when the disk queue is not empty.
- Adding the thread to the end of disk queue when the disk queue is not empty.

```
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size) {
    // Intialising the head and teal of the disk queue
    head = nullptr;
    tail = nullptr;

}



/*--------------------------------------------------------------------------*/
/* SIMPLE_DISK FUNCTIONS */
/*--------------------------------------------------------------------------*/
void BlockingDisk::wait_until_ready() {

    if ( !is_ready() ) {
       // Adding the thread to the end of disk queue when the disk queue is not empty.
        Thread * thread = Thread::CurrentThread();
        if(head == nullptr) {
            head = new disk_queue;
            head->thread = thread;
            head->next = nullptr;
            tail = head;
            // if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
        }else{
            // Adding the thread to the end of disk queue when the disk queue is not empty.
            disk_queue * new_thread = new disk_queue;
            new_thread->thread = thread;
            tail->next = new_thread;
            tail = new_thread;
        }
        // Yeilding the control to the next thread in the ready queue.
        SYSTEM_SCHEDULER->yield();
    }
}
```

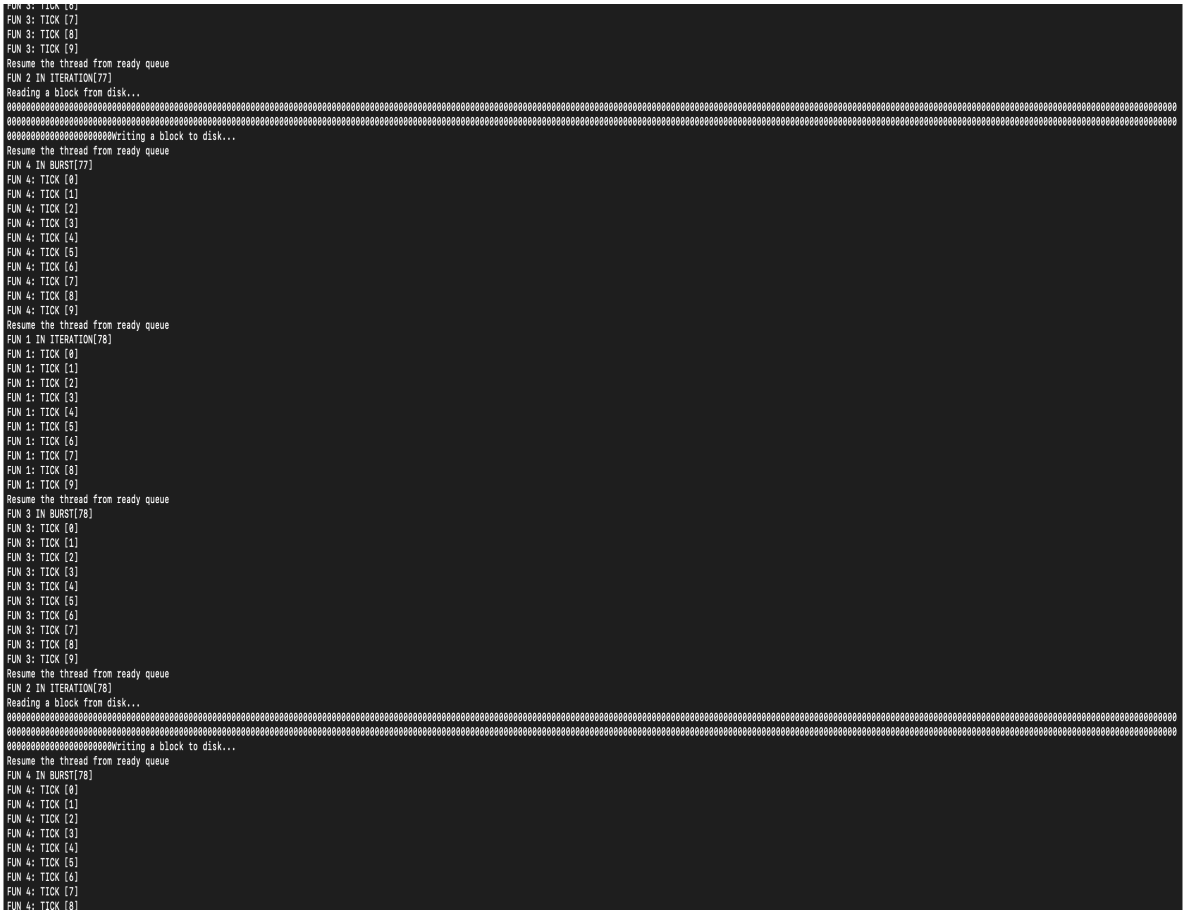
blocking_disk.H: : The following are done in this function step by step

- Defining disk queue
- Overloading the wait_until_ready() function inherited from the SimpleDisk class.
- Overloading the is_ready() function inherited from the SimpleDisk class.

```
class BlockingDisk : public SimpleDisk {
public:
  struct disk_queue
  {
      Thread * thread;
      disk_queue * next;
  };
   disk_queue * head;
   disk_queue * tail;




   BlockingDisk(DISK_ID _disk_id, unsigned int _size);
   /* Creates a BlockingDisk device with the given size connected to the
      MASTER or SLAVE slot of the primary ATA controller.
      NOTE: We are passing the _size argument out of laziness.
      In a real system, we would infer this information from the
      disk controller. */



   /* DISK OPERATIONS */

   // Overloading the wait_until_ready() function inherited from the SimpleDisk class.
   virtual void  wait_until_ready();
   // Overloading is_ready() function inherited from the SimpleDisk class.
   bool is_ready();
   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. */


};

#endif
```

Testing

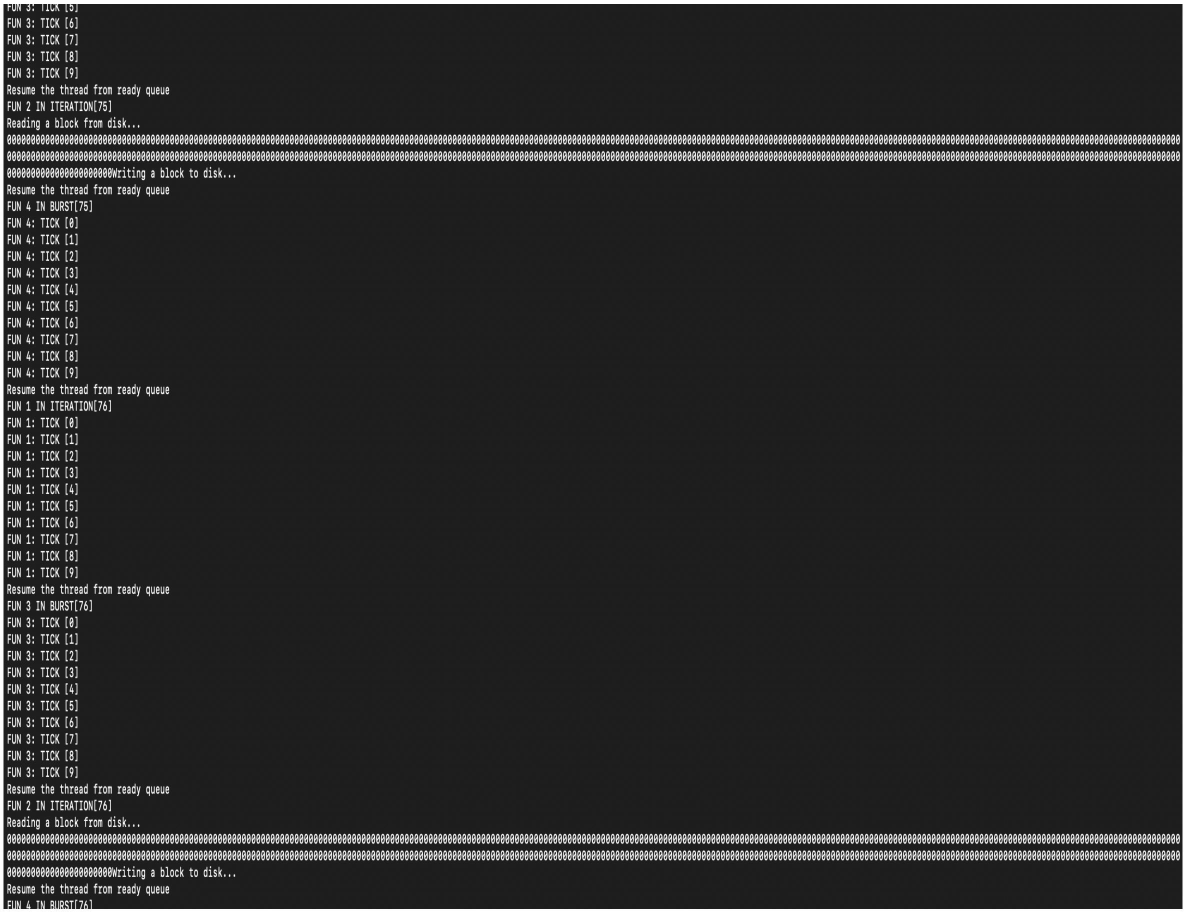
After a write operation, instead of busy waiting, the thread waits in the disk queue and then the next thread is dispatched. You can check below that the code executes as expected even after implemented this layer.



Here we can see that a read operation is called after thread 2 is executed. Soon after that 0 are printed as coded in thread 2 function and then a write operation is issued, and soon after that, thread 3 is run.

Bonus Option 1: Support for Disk Mirroring

- Separate mirror_disk.H and mirror_disk.C files are created in which we define a new class MirrorDisk derived from BlockingDisk.
- To test the code with this disk, uncomment _USES_MIRROR_DISK_ in kernel.C which used MirrorDisk instead of Blocking disk.
- We can see in the test screenshot that the code is expected just as it should, even after using MirrorDiskinstead of blocking disk.



Bonus Option 3: Design of a thread-safe disk system:

‘

- A mutex lock can be defined which takes care of disk when read and write is requested.
- A function int lock with parameter int mutex can be used, where a mutex variable such as *mutex can be defined.
- When the critical section is entered, mx(mutex) is grabbed by some thread then the current thread yields instead of busy waiting.
- Then we check for mutex before entering a critical section, which doesn't allow the thread to enter since it is already grabbed by the previous thread.
- This way we can manage to avoid race conditions.