MP6: Primitive Disk Device Driver

Vishnuvasan Raghuraman
UIN: 234009303
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus 1: Did not Attempt.
Bonus 2: Did not Attempt.
Bonus 3: Did not Attempt.
Bonus 4: Did not Attempt.

System Design

The objective of this machine problem is to enhance a basic device driver by implementing a layer that enables blocking read and write operations. This enhancement should ensure that users can call 'read' and 'write' operations without concern for premature returns or system lock-ups due to device wait times, all without resorting to busy waiting in the device driver code.

Implmentation of Blocking Disk

A class named 'Queue' is introduced to manage operations on both the ready queue and the queue containing blocked threads. This queue is implemented as a linked list of thread objects. The 'Queue' class declares constructors and public functions to add (enqueue) and remove (dequeue) threads from the queue. For the enqueue operation, it first checks if a thread already exists in the queue. If so, it traverses to the end of the queue and appends the new thread. For the dequeue operation, it removes the thread at the top of the queue and updates the pointer to point to the next thread in the queue.

For this machine problem, we create a 'BlockingDisk' device driver that inherits from 'SimpleDisk' and facilitates I/O operations (read and write) without resorting to busy waiting in the driver code. Leveraging the scheduler, threads relinquish the CPU if the disk isn't prepared for I/O operations. When a thread initiates a disk I/O operation, it activates the 'BlockingDisk::wait_until_ready()' method, which places the current thread into a distinct 'blocked_queue' and yields the CPU to the subsequent thread in the ready queue.

When a thread finishes execution and yields, it triggers the 'Scheduler::yield()' method. Within this method, we verify if the disk is prepared and if there's at least one thread in the blocked queue. If a blocked thread exists in the queue and the disk is ready, we dequeue the top thread from the blocked queue and promptly dispatch it to complete the I/O operation. If no threads are present in the blocked queue or if the disk isn't ready, we dequeue the top thread from the ready queue (which comprises threads not engaged in I/O operations and thus not blocked) and dispatch it as usual. (Note: Adjustments were made in 'simple_disk.C' to mimic disk I/O delay.)

This approach ensures that if the driver isn't ready, the scheduler yields the CPU, allowing other threads to execute in the meantime. By doing so, we prevent busy waiting in the device driver.

Code Description

A Silicon(M2) Mac was used for this assignment and the native environment setup for such a machine from Piazza was followed. The below files were modified for this particular MP.

- scheduler.H
- scheduler.C
- thread.C
- kernel.C
- blocking_disk.H
- blocking_disk.C
- simple_disk.C
- queue.H
- makefile

The functions implemented are given as below:

queue.H-class Queue: Within the class Queue, we define data structures and functions dedicated to managing the ready queue. In the class Queue, we define the following:

- Thread * thread: This member represents the thread at the top of the ready queue.
- Queue * next: This member points to the next Queue object in the ready queue, indicating the subsequent thread.
- Queue(): This constructor initializes the variables required for setup.
- Queue(Thread* new_thread): This constructor sets up subsequent threads in the ready queue.
- void enqueue(Thread* new_thread): This method adds a thread to the end of the ready queue.
- Thread* dequeue(): This method removes the thread at the top of the ready queue and updates the pointer to indicate the next thread in line.

```
/*--------------------------------------------------------------------------*/
/* QUEUE DATA STRUCTURE */
/*--------------------------------------------------------------------------*/

class Queue
{
    private:
    Thread* thread;                 // Pointer to thread at the top of queue
    Queue* next;                    // Pointer to next thread in queue.
    public:
    Queue(){ // Constructor for initial setup
        thread = nullptr;
        next   = nullptr;
    }

    Queue(Thread* new_thread){  // Constructor for new threads to enter queue
        thread = new_thread;
        next = nullptr;
    }

    // adding thread at the end of queue
    void enqueue(Thread* new_thread){
        // first thread added to queue
        if ( thread == nullptr ){
            thread = new_thread;
        }
        else{
            // traversing the queue
            if( next != nullptr ){
                next->enqueue(new_thread);
            }
            else{
                // reached end of queue - adding new thread at the end of queue
                next =  new Queue(new_thread);
            }
        }
    }
```

```
    // removing the thread at head position and point to next thread in queue
    Thread* dequeue(){
        // empty queeu
        if( thread == nullptr ){
            return nullptr;
        }
        // retrieving top of queue element
        Thread *top = thread;
        // when only one queue element present
        if( next == nullptr ){
            thread = nullptr;
        }
        else{
            // updating head element of queue
            thread = next->thread;
            // deleting current node
            Queue * del_node = next;
            // updating next pointer
            next = next->next;
            delete del_node;
        }
        return top;
    }
};

#endif
```

scheduler.H-class Scheduler: In the class Scheduler, we declare the following:

- Queue ready_queue: This Queue data object is responsible for managing the ready queue of threads.
- int qsize: This variable tracks the number of threads waiting in the ready queue.

```
class Scheduler {

  /* The scheduler might need private members. */
  Queue ready_queue;
  int qsize;
```

scheduler.C-Scheduler: This method serves as the constructor for the Scheduler class. It initializes the ready queue size to zero.

```
Scheduler::Scheduler() {
  qsize = 0;
  Console::puts("Constructed Scheduler.\n");
}
```

scheduler.C-Scheduler::yield: This method facilitates preempting the currently running thread and relinquishing the CPU. A new thread is chosen for CPU time by dequeuing the top element from the ready queue and dispatching to it. Additionally, interrupts are disabled before any operations on the ready queue start, and interrupts are re-enabled after completing operations on the ready queue. To handle blocking read/write operations without resorting to busy waiting, changes were made to the 'yield' method. Utilizing the 'check_blocked_thread_in_queue' method, we verify if the disk is ready and if at least one thread exists in the blocked threads queue. If this condition holds true, we dequeue the top thread from the blocked threads queue (threads that block due to I/O operations) and dispatch it immediately. If the condition is false, we dequeue the top thread from the ready queue (threads that do not perform I/O operations) and dispatch it.

```
void Scheduler::yield() {
  // Disable interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // checking if disk is ready and blocked threads exist in the queue
    // if hte disk is ready, immediately dispatch to the top thread in blocked queue
    if(SYSTEM_DISK->check_blocked_thread_in_queue()){
        // re enabling interrupts
        if(!Machine::interrupts_enabled()){
            Machine::enable_interrupts();
        }
        Thread::dispatch_to(SYSTEM_DISK->get_top_thread());
    }
    // if the disk is not ready or blocked threads do not exist in queue
    // checking the regular FIFO ready queue
    else{
        if(qsize == 0){
            Console::puts("Queue is empty. No threads available. \n");
        }
        else{
            // removing thread from queue for CPU time
            Thread * new_thread = ready_queue.dequeue();
            // decrementing queue size
            qsize = qsize - 1;
            // re enabling interrupts
            if(!Machine::interrupts_enabled()){
                Machine::enable_interrupts();
            }
            // context-switch and giving CPU time for new thread
            Thread::dispatch_to(new_thread);
        }
    }
}
```

scheduler.C:Scheduler::resume: This method is employed to include a thread into the ready queue of the scheduler. It's invoked for threads either waiting for an event or yielding the CPU due to pre-emption. To maintain integrity, interrupts are disabled before executing any actions on the ready queue and are subsequently re-enabled upon completion of these operations.

```
void Scheduler::resume(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_queue.enqueue(_thread);
    // incrementing queue size
    qsize = qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-Scheduler::add: This method is employed to add a thread into the ready queue of the scheduler, marking the thread as runnable. It's typically invoked upon thread creation. To ensure smooth operation, interrupts are disabled before any actions are taken on the ready queue and then re-enabled once those operations are finished.

```
void Scheduler::add(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_queue.enqueue(_thread);
    // incrementing queue size
    qsize = qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-Scheduler::terminate: This method is employed to remove a thread from the ready queue. It involves iterating through the queue, dequeuing each thread, and comparing their Thread IDs. If a match is not found with the ID of the thread to be terminated, those threads are enqueued back into the ready queue. To ensure consistency, interrupts are disabled before any actions are taken on the ready queue and then re-enabled once these operations are completed.

```
void Scheduler::terminate(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    int index = 0;
    // iterating and dequeuing each thread
    // checking if thread ID matches with thread to be terminated
    // adding thread back to ready queue if thread id doesn't match
    for(index=0;index<qsize;index++){
        Thread * top = ready_queue.dequeue();
        if(top->ThreadId() != _thread->ThreadId()){
            ready_queue.enqueue(top);
        }
        else{
            qsize = qsize - 1;
        }
    }
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

blocking_disk.H-class BlockingDisk: Within the 'BlockingDisk' class, data structures and functions for kernel-level thread scheduling and management of the blocked queue are defined. This class inherits from the 'SimpleDisk' class.

- Queue blocked_queue: A Queue data object for managing the queue of blocked threads.
- int blocked_queue_size: A variable to monitor the number of threads in the blocked queue.

```
class BlockingDisk : public SimpleDisk {
    Queue * blocked_queue;
    int blocked_queue_size;
```

blocking_disk.C-BlockingDisk: This method serves as the constructor for the 'BlockingDisk' class. It initializes the blocked threads queue and sets the size of the blocked threads queue to zero.

```
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size) {
    blocked_queue = new Queue();
    blocked_queue_size = 0;
}
```

blocking_disk.C-get_top_thread: This method retrieves the blocked thread positioned at the top of the blocked queue by executing a 'dequeue' operation.

```
Thread * BlockingDisk::get_top_thread(){
  // getting the thread at the top of the blocked queue and returning
  Thread *top = this->blocked_queue->dequeue();
  this->blocked_queue_size--;
  return top;
}
```

blocking_disk.C-wait_until_ready: This method appends the currently running thread to the end of the blocked queue through an 'enqueue' operation. Subsequently, the thread yields the CPU to the next thread.

```
void BlockingDisk::wait_until_ready(){
  // adding current thread to end of blocked threads queue
  this->blocked_queue->enqueue(Thread::CurrentThread());
  this->blocked_queue_size++;
  SYSTEM_SCHEDULER->yield();
}
```

blocking_disk.C-check_blocked_thread_in_queue: This method verifies whether the disk is prepared for data transfer and whether threads exist in the blocked queue. If both conditions are met, it returns true; otherwise, it returns false.

```
bool BlockingDisk::check_blocked_thread_in_queue(){
  // checking if disk is ready to transfer data and threads exist in blocked state
  return ( SimpleDisk::is_ready() && (this->blocked_queue_size > 0) );
}
```

simple_disk.C-is_ready: Minor modifications were made to this method to mimic a disk I/O delay.

```
static int delay = 1;
bool SimpleDisk::is_ready() {
  if(delay == 0){
    delay += 1;
    return false;
  }
  delay -= 1;
  return ((Machine::inportb(0x1F7) & 0x08) != 0);
}
```

thread.C : thread_start: Changes have been implemented in this method to enable interrupts at the beginning of the thread by utilizing the 'enable_interrupts' method.

```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */

    /* We need to add code, but it is probably nothing more than enabling interrupts. */
    // enabling interrupts
    Machine::enable_interrupts();

}
```

thread.C : thread_shutdown: This method is utilized to terminate a thread, releasing the memory and other associated resources held by the thread. To accommodate threads with terminating functions, the following adjustments were made. Initially, the currently running thread is terminated. Subsequently, memory is freed up using the 'delete' operation. Finally, the 'yield' method is invoked to relinquish CPU control and select the next thread in the ready queue.

```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */

    // assert(false);
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */

    // terminating currently running thread
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());

    // deleting thread and freeing space
    delete current_thread;

    // current thread gives up CPU and the next thread is selected
    SYSTEM_SCHEDULER->yield();
}
```

kernel.C: There were some changes made in this file to test different cases. They were as follows:

- Support for scheduling has been enabled by uncommenting the '_USES_SCHEDULER' macro.
- To utilize the BlockingDisk device, 'SimpleDisk' objects were replaced with 'BlockingDisk' objects.
- To address the stack overflow issue triggered when interrupts are enabled, the stack size allocated for Thread 2 was augmented from 1024 bytes to 4096 bytes.

```
#define _USES_SCHEDULER_
/* This macro is defined when we want to force the code below to use
   a scheduler.
   Otherwise, no scheduler is used, and the threads pass control to each
   other in a co-routine fashion.
*/
```

```
/* -- DISK DEVICE -- */

// SYSTEM_DISK = new SimpleDisk(DISK_ID::MASTER, SYSTEM_DISK_SIZE);
SYSTEM_DISK = new BlockingDisk(DISK_ID::MASTER, SYSTEM_DISK_SIZE);
```

```
/* -- A POINTER TO THE SYSTEM DISK */
// SimpleDisk * SYSTEM_DISK;
BlockingDisk * SYSTEM_DISK;
```

```
Console::puts("CREATING THREAD 2...");
char * stack2 = new char[4096];
thread2 = new Thread(fun2, stack2, 4096);
Console::puts("DONE\n");
```

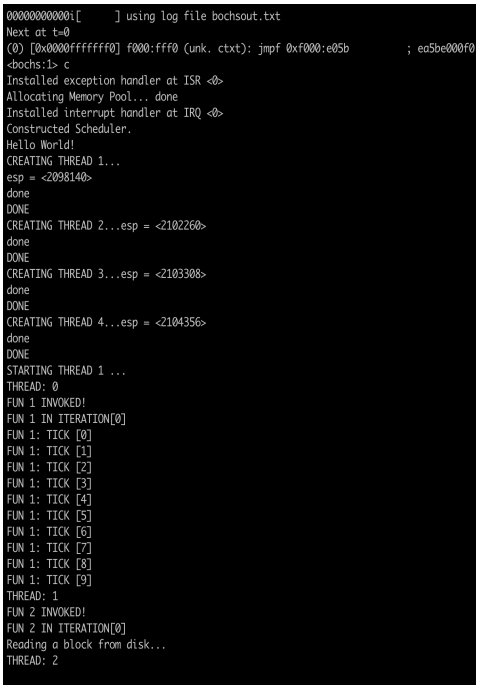
makefile: Changes were made in the makefile to incorporate the inclusion of 'scheduler.C', 'scheduler.H', and 'queue.H' files.

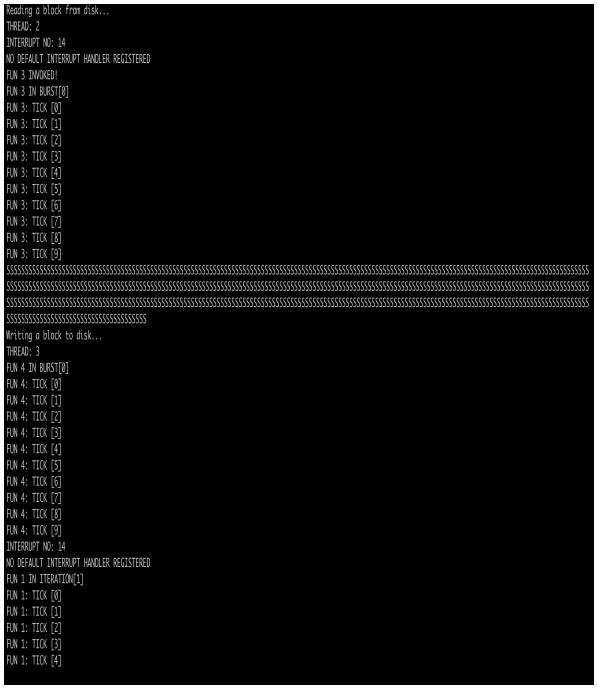
```
scheduler.o: scheduler.C scheduler.H thread.H
	$(GCC) $(GCC_OPTIONS) -c -o scheduler.o scheduler.C

queue.o: queue.H thread.H
	$(GCC) $(GCC_OPTIONS) -c -o queue.o

# ===== KERNEL MAIN FILE =====

kernel.o: kernel.C machine.H console.H gdt.H idt.H irq.H exceptions.H interrupts.H simple_timer.H frame_pool.H mem_pool.H thread.H simple_disk.H scheduler.H
	$(GCC) $(GCC_OPTIONS) -c -o kernel.o kernel.C

kernel.bin: start.o utils.o kernel.o \
   assert.o console.o gdt.o idt.o irq.o exceptions.o \
   interrupts.o simple_timer.o simple_keyboard.o frame_pool.o mem_pool.o \
   thread.o threads_low.o simple_disk.o blocking_disk.o \
   machine.o machine_low.o scheduler.o
	$(LD) -melf_i386 -T linker.ld -o kernel.bin start.o utils.o kernel.o \
   assert.o console.o gdt.o idt.o irq.o exceptions.o interrupts.o \
   simple_timer.o simple_keyboard.o frame_pool.o mem_pool.o \
   thread.o threads_low.o simple_disk.o blocking_disk.o \
   machine.o machine_low.o scheduler.o
```

Testing

The below testcase was executed for verification purposes.

- Thread 2 exclusively handles disk I/O operations, while other threads execute regular non-terminating functions.





From the above images, it can be seen that Thread 2, acts as the sole executor of disk I/O operations, and successfully completes the required tasks. When a disk read operation is initiated, Thread 2 yields the CPU, allowing another thread to take control for execution. After some time, control is restored to Thread 2, which then completes the disk write operation.

Expected Result: Thread 2 yields the CPU to another thread upon initiating a Disk I/O operation. This ensures that busy waiting is avoided. Once the Disk I/O operation is completed, the process resumes.

Actual Result: Thread 2 yields the CPU to another thread upon initiating a Disk I/O operation. This ensures that busy waiting is avoided. Once the Disk I/O operation is completed, the process resumes.