MP 6: Primitive Disk Device Driver

Pranav Anantharam

UIN: 734003886

CSCE611: 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 Design

The goal of this machine problem is to add a layer on top of a simple device driver to support blocking read and write operations as the basic implementation, but without busy waiting in the device driver code. The user should be able to call 'read' and 'write' operations without worrying that the call may either return prematurely or tie up the entire system waiting for the device to return.

Blocking Disk Implementation:

A class 'Queue' is introduced to handle operations on the ready queue and blocked threads queue. The queue is implemented as a linked list of thread objects. The class 'Queue' declares constructors and public functions to add (enqueue) and remove (dequeue) threads from a queue. To perform the enqueue operation, we first check if there exists a thread in the queue already. If a thread exists in the queue, we traverse to the end of the queue and append the new thread. To perform the dequeue operation, the thread at the top of the queue is removed and the pointer is updated to point to the next thread in the queue.

In this machine problem, we implement a 'BlockingDisk' device driver which inherits from 'SimpleDisk' and supports I/O operations (read and write operations) without busy waiting in the device driver code. Using the scheduler, threads yield the CPU if the disk is not ready for I/O operations.

When a thread requests a disk I/O operation, it triggers the 'BlockingDisk::wait_until_ready()' method which enqueues the current thread into a separate 'blocked_queue' and yields the CPU to the next thread in the ready queue.

When a thread completes execution and yields, it triggers the 'Scheduler::yield()' method. In this method, we check if the disk is ready and that there is at least one thread present in the blocked queue. If there exists a blocked thread in the blocked queue and the disk is ready, then we dequeue the thread at the top of the blocked queue and dispatch to it immediately to complete the I/O operation. If there are no threads present in the blocked queue or if the disk is not ready, then we dequeue the thread at the top of the ready queue (which contains threads which do not perform I/O operations and do not get blocked) and dispatch to it as usual. (Note: Changes were made in 'simple_disk.C' to simulate disk I/O delay)

In this manner, if the driver is not ready, the scheduler yields the CPU so that other threads can be executed in the meantime and we avoid busy waiting in the device driver.

List of files modified

The following files were modified:

1. scheduler.H
2. scheduler.C
3. blocking_disk.H
4. blocking_disk.C
5. simple_disk.C
6. queue.H
7. thread.C
8. kernel.C
9. makefile

Code Description

1. **scheduler.H : class Scheduler:**

   In class Scheduler, data structures and functions for scheduling of kernel-level threads and management of the ready queue are declared.

   a) Queue ready_queue - Queue data object for handling the ready queue of threads
   b) int qsize - Variable to track the number of threads waiting in the ready queue

```
/*--------------------------------------------------------------------------*/
/* SCHEDULER */
/*--------------------------------------------------------------------------*/

class Scheduler {

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

2. **scheduler.C : Scheduler():**

   This method is the constructor for class Scheduler. The ready queue size is initialized to zero.

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

3. **scheduler.C : Scheduler :: yield():**

   This method is used to pre-empt the currently running thread and give up the CPU. A new thread is selected for CPU time by dequeuing the top element in the ready queue and dispatching to it. Moreover, interrupts are disabled before the start of any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue. In order to handle blocking read/write operations without busy waiting, modifications were made to the 'yield' method. Using the 'check_blocked_thread_in_queue' method, we check if the disk is ready and that at least one thread is present in the blocked threads queue. If this condition is true, we dequeue the thread at the top of the blocked threads

queue (threads which block due to I/O operations) and dispatch to it immediately. If the condition is not true, we dequeue the top thread in the ready queue (threads which do not perform I/O operations) and dispatch to it.

```
void Scheduler::yield()
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Check if disk is ready and blocked threads exist in queue
    // If disk is ready, immediately dispatch to top thread in blocked queue
    if( SYSTEM_DISK->check_blocked_thread_in_queue() )
    {
        // Re-enable interrupts
        if( !Machine::interrupts_enabled() )
        {
            Machine::enable_interrupts();
        }

        Thread::dispatch_to(SYSTEM_DISK->get_top_thread());
    }

    // If disk is not ready or blocked threads do not exist in queue
    // Check the regular FIFO ready queue
    else
    {
        if( qsize == 0 )
        {
            Console::puts("Queue is empty. No threads available. \n");
        }
```

```
        else
        {
            // Remove thread from queue for CPU time
            Thread * new_thread = ready_queue.dequeue();

            // Decrement queue size
            qsize = qsize - 1;

            // Re-enable interrupts
            if( !Machine::interrupts_enabled() )
            {
                Machine::enable_interrupts();
            }

            // Context-switch and give CPU time to new thread
            Thread::dispatch_to(new_thread);
        }
    }
}
```

4. **scheduler.C : Scheduler :: resume(Thread * _thread):**

This method is used to add a thread to the ready queue of the scheduler. This method is called for threads that were waiting for an event to happen, or that have to give up the CPU in response to a pre-emption. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void Scheduler::resume(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_queue.enqueue(_thread);

    // Increment queue size
    qsize = qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

5. **scheduler.C : Scheduler :: add(Thread * _thread):**

This method is used to add a thread to the ready queue of the scheduler. The thread is made runnable by the scheduler. This method is called on thread creation. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void Scheduler::add(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_queue.enqueue(_thread);

    // Increment queue size
    qsize = qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

6. **scheduler.C : Scheduler :: terminate(Thread * _thread):**

This method is used to remove a thread from the ready queue. We iterate through the ready queue; dequeue each thread and compare the Thread ID. If the Thread ID does not match with the thread ID of the thread to be terminated, then we enqueue the threads back into the ready queue. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void Scheduler::terminate(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    int index = 0;

    // Iterate and dequeue each thread
    // Check if thread ID matches with thread to be terminated
    // If thread ID does not match, add thread back to ready queue
    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-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

7. **blocking_disk.H : class BlockingDisk:**

In class BlockingDisk, data structures and functions for scheduling of kernel-level threads and management of the blocked queue are declared. It inherits from class SimpleDisk.

a) Queue blocked_queue - Queue data object for handling the queue of blocked threads
b) int blocked_queue_size - Variable to track the number of threads in the blocked queue

```
/*--------------------------------------------------------------------------*/
/* B l o c k i n g D i s k  */
/*--------------------------------------------------------------------------*/

class BlockingDisk : public SimpleDisk {

    Queue * blocked_queue;
    int blocked_queue_size;
```

8. **blocking_disk.C : BlockingDisk():**

This method is the constructor for class BlockingDisk. The blocked threads queue is initialized and the blocked threads queue size is set to zero.

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

9. **blocking_disk.C : get_top_thread():**

This method returns the blocked thread located at the top of the blocked queue by performing a ‘dequeue’ operation.

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

10. **blocking_disk.C : wait_until_ready():**

This method adds the current running thread to the end of the blocked queue by performing an ‘enqueue’ operation. The thread then yields the CPU to the next thread.

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

11. **blocking_disk.C : check_blocked_thread_in_queue():**

This method checks if the disk is ready to transfer data and if threads exist in the blocked queue. If the disk is ready to transfer data and threads exist in the blocked queue, a Boolean true is returned. Otherwise, a Boolean false is returned.

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

12. **simple_disk.C : is_ready():**

Changes were made to this method to simulate a disk I/O delay.

```
static int delay = 1;
bool SimpleDisk::is_ready()
{
    if( delay == 0 )
    {
        delay = delay + 1;
        return false;
    }

    delay = delay - 1;
    return ((Machine::inportb(0x1F7) & 0x08) != 0);
}
```

13. **queue.H : class Queue:**

In class Queue, data structures and functions for managing the ready queue and blocked queue are defined:

a) Thread * thread - Pointer to the thread at the top of the queue
b) Queue * next - Pointer to next Queue type object (next thread in the queue)
c) Queue() - Constructor for initial setup of the variables
d) Queue(Thread * new_thread) - Constructor for setting up subsequent threads
e) Void enqueue(Thread * new_thread) - Method to add a thread to the end of the queue
f) Thread * dequeue() - Function to remove the thread at the top of the queue and point to the next thread

```
/*---------------------------------------------------------------------------*/
/* 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:

    // Constructor for initial setup
    Queue()
    {
        thread = nullptr;
        next   = nullptr;
    }

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

```
// Add thread at end of queue
void enqueue(Thread* new_thread)
{
    // First thread added to queue
    if ( thread == nullptr )
    {
        thread = new_thread;
    }
    else
    {
        // Traverse the queue
        if( next != nullptr )
        {
            next->enqueue(new_thread);
        }
        else
        {
            // Reached end of queue - add new thread at the end of queue
            next =  new Queue(new_thread);
        }
    }
}
```

```
    // Remove thread at head position and point to next thread in queue
    Thread* dequeue()
    {
        // Queue is empty
        if( thread == nullptr )
        {
            return nullptr;
        }

        // Get top of queue element
        Thread *top = thread;

        // Only one queue element present
        if( next == nullptr )
        {
            thread = nullptr;
        }
        else
        {
            // Update head element of queue
            thread = next->thread;

            // Delete current node
            Queue * del_node = next;

            // Update next pointer
            next = next->next;

            delete del_node;
        }

        return top;
    }
};

#endif
```

14. **thread.C : thread_shutdown():**

This method is used to terminate a thread by releasing the memory and other resources held by the thread. The below changes were made to handle threads with terminating thread functions. The currently running thread is first

terminated. Next, memory is freed up using the 'delete' operation. Finally, the 'yield' method is called to give up CPU 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.
    */

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

    // Delete thread and free space
    delete current_thread;

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

15. **thread.C : thread_start():**

Changes were made in this method to enable interrupts at the start of the thread using 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. */

    // Enable interrupts at start of thread
    Machine::enable_interrupts();
}
```

16. **kernel.C :**

The following changes were made in kernel.C for the purpose of testing:

a) The macro '_USES_SCHEDULER' was uncommented to enable support for the scheduler
b) Replaced 'SimpleDisk' objects with 'BlockingDisk' objects to make use of BlockingDisk device
c) Increased stack size allocated for Thread 2 from 1024 bytes to 4096 bytes to resolve stack overflow bug when interrupts are enabled

```
/* -- COMMENT/UNCOMMENT THE FOLLOWING LINE TO EXCLUDE/INCLUDE SCHEDULER CODE */

#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");
```

17. **makefile :**

Changes were made in the makefile to include ‘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

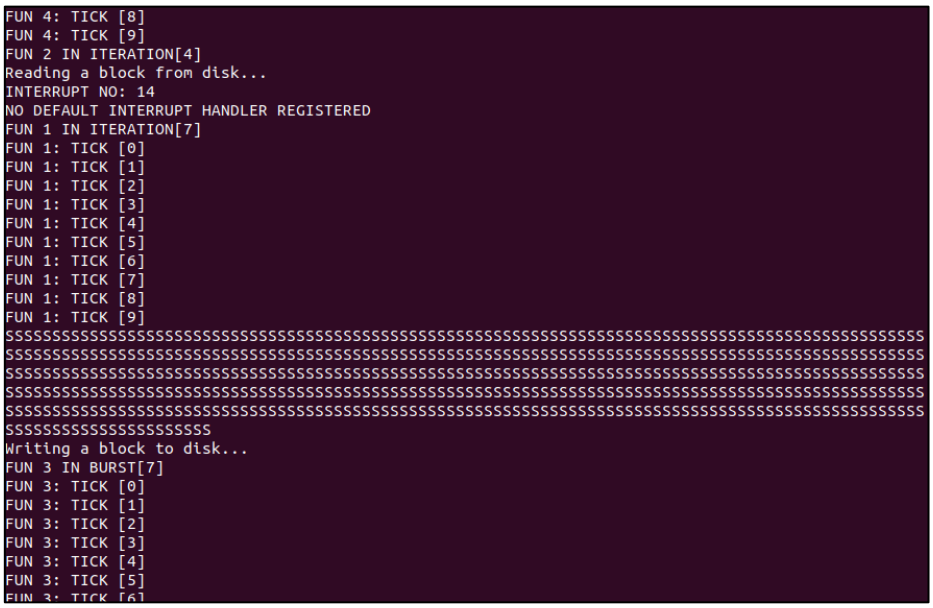
Serial No.	Testcase	Desired Result	Actual Result
1	Single thread (Thread 2) issues disk I/O operations; while other threads run regular non-terminating functions	Thread 2 yields CPU to another thread on initiating Disk I/O operation. Busy waiting is avoided. Disk I/O operation is completed.	Thread 2 yielded CPU to another thread on initiating Disk I/O operation. Busy waiting was avoided. Disk I/O operation was completed.

Outputs:

1. Single thread (Thread 2) issues disk I/O operations; Disk operations complete successfully
   Disk read operation is initiated, Thread 2 yields CPU and gives control to another thread for execution. After a while, control is returned to Thread 2 and Disk Write operation is completed.

```
Please choose one: [6] 6
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file bochsout.txt
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098140>
done
DONE
CREATING THREAD 2...esp = <2102260>
done
DONE
CREATING THREAD 3...esp = <2103308>
done
DONE
CREATING THREAD 4...esp = <2104356>
done
DONE
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 [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Reading a block from disk...
THREAD: 2
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN 3 INVOKED!
FUN 3 IN BURST[0]
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]
Writing a block to disk...
THREAD: 3
FUN 4 IN BURST[0]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]


FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 2 IN ITERATION[4]
Reading a block from disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
FUN 1 IN ITERATION[7]
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]
Writing a block to disk...
FUN 3 IN BURST[7]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]