MP5: Kernel-Level Thread Scheduling

Oliver Carver
UIN: 630003314
CSCE410: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1: Completed.
Bonus Option 2: Completed.
Bonus Option 3: Plan submitted to Professor.

System Design

In this machine problem I implemented a simple kernel thread scheduling system. I did this by implementing the Scheduler class and adjusting the implementation of the Thread class. The scheduler class implements a FCFS scheduling algorithm. In order to achieve this, I created a utility struct called Queue which implements a simple FIFO queue. The queue is implemented using a singly linked list with a head and tail pointer. I created two private utility functions with the Scheduler class called enqueue and dequeue which implement their respectively named functionalities for the queue and work using pointers to threads. I edited the Thread class to have a member called next to be able to implement this queue. Threads are added at the tail and popped from the head in order to achieve FIFO. Additionally, I implemented bonus options 1 and 2. To complete bonus option 1, I added code which conditionally enables and disable interrupts in functions that require mutual exclusion. To complete bonus option 2, I implemented a new EOQTimer class derived from SimpleTimer, which implements round robin functionality.

Code Description

I changed the files thread.H, thread.C, scheduler.H, scheduler.C, simple_timer.H, simple_timer.C, interrupts.C, and kernel.C. Additionally, I had to change other utility files. I modified my makefile to support compiling the kernel as an .elf file. I reused my linker.ld file from previous MPs, in order to support .elf files. I added my .gdbinit file from previous MPs. I re-used the ROM image from previous MPs. I replaced the copykernel.sh file with the one from my previous MPs. To compile the code, use make and then copykernel.sh. After that, run it with bochs. Output should display within the terminal.

scheduler.C: Scheduler : This method is the constructor for the scheduler. It is very simple, only initializing a static member variable

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

scheduler.C: enqueue : This function enqueues a thread into the FIFO queue. It will either initialize the linked list or add the thread to the end of the queue via. the tail pointer.

```
void Scheduler::enqueue(Thread * _thread) {
    if (!queue.tail) {
        queue.head = queue.tail = _thread;
        return;
    }

    queue.tail->next = _thread;
    queue.tail = _thread;
}
```

scheduler.c: dequeue : This function dequeues a thread from the FIFO queue. It will then reset the pointers if the queue is now empty and removes the thread from the linked list.

```
Thread * Scheduler::dequeue() {
    if (!queue.head)
        return NULL;

    Thread *head = queue.head;
    queue.head = queue.head->next;

    if (!queue.head)
        queue.tail = NULL;

    return head;
}
```

scheduler.C: yield : This function is used to give up the CPU and transfer the CPU to another thread. It disables interrupts, calls dequeue to retrieve the next thread, then calls dispatch_to to transfer the CPU to that thread that was at the front of the queue. It then re-enables interrupts if disabled when it returns.

```
void Scheduler::yield() {
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    Thread *thread = dequeue();
    if (thread) {
        // The current thread will hang here
        // When we get back the CPU, we start here and then
        // immediately re-enable interrupts.
        // So while a terminating thread will never finish its
        // yield and so will never re-enable interrupts, the thread
        // it passes the CPU off to will resume here then enable interrupts
        thread->dispatch_to(thread);
    }

    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();
}
```

scheduler.C: resume : This function is used to add the given thread to the ready queue. The function disables/enables interrupts to ensure mutual exclusion then enqueues the thread, adding it to the back.

```
void Scheduler::resume(Thread * _thread) {
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    enqueue(_thread);

    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();
}
```

scheduler.C: add : This function is used to add a thread to the scheduler for the first time. It simply wraps around resume.

```
void Scheduler::add(Thread * _thread) {
    resume(_thread);
}
```

scheduler.C: terminate : ****FUNCTION SIGNATURE CHANGED!****. This function is used to terminate a thread within the ready queue, or to gracefully handle the shutting down of a currently running thread. It disables interrupts to ensure mutual exclusion, then checks if the requested thread for termination is the currently running thread. If so, it deletes that thread, which will call its destructor. Then, it yields the CPU, effectively transferring control to the next thread in the queue. If the requested thread for termination is not the currently running thread, it searches the linked list for the requested thread then deletes it, also updating the pointers. It also sets the thread pointer to null. In addition to setting the thread pointer to null, the function signature has been changed. The pointer to the thread that is passed in is updated to be passed in by reference, so that the pointer can be properly updated to null.

```
void Scheduler::terminate(Thread *& _thread) {
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    if (_thread == NULL) return;

    // If a thread is returning it'll call terminate and pass itself
    // as the argument. It's still technically the CurrentThread and
    // the current_thread variable reflects this. We just check if the
    // requested thread is the current thread, if it is, we just yield
    if (_thread == Thread::CurrentThread()) {
        delete _thread;
        _thread = NULL;
        // Once we do this yield we'll never return to here
        yield();
        return;
    }

    // In the case that it isn't the current thread, then we need to
    // terminate some thread that's currently queued.
    if (queue.head == queue.tail) {
        if (queue.head == _thread) {
            delete _thread;
            _thread = NULL;
            queue.head = queue.tail = NULL;
        }
        return;
    }

    Thread *prev = queue.head;
    while (prev && prev->next != _thread) {
        prev = prev->next;
    }

    if (!prev || prev->next != _thread)
        return;

    prev->next = _thread->next;

    if (_thread == queue.tail)
        queue.tail = prev;

    delete _thread;
    _thread = NULL;

    if (!Machine::interrupts_enabled())
        Machine::enable_interrupts();
}
```

thread.C: thread_shutdown : This function is called by the thread once it returns. It disables interrupts then calls terminate from the scheduler and passes in the currently running thread to shut it down.

```
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.
    */

    // Disable interrupts for mutual exclusion
    if (Machine::interrupts_enabled())
        Machine::disable_interrupts();

    // Ask the scheduler to terminate the thread
    Scheduler::scheduler->terminate(current_thread);
}
```

thread.C: thread_start : This function is called by threads as they start up. It simply enables interrupts.

```
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. */
    Machine::enable_interrupts();
}
```

thread.C: ~Thread : This function is the destructor for threads. It simply deletes the stack.

```
// We define a destructor to destroy the allocated stack
Thread::~Thread() {
    delete[] stack;
}
```

1 Bonus Option 1

Bonus option 1 was implemented by adding code which called the Machine::disable_interrupts and Machine::enable_interrupts functions to functions that require mutual exclusion, specifically in the scheduler and thread functions. Examples of this can be found in the implementations of those functions.

2 Bonus Option 2

Bonus option 2 was implemented by taking advantage of the features of a FIFO queue. Round Robin essentially uses a FIFO queue, with the added caveat that if a thread has not finished executing, it should be immediately requeued, being added to the end of the queue. FCFS also uses a FIFO queue. As a result, the base scheduler class need not be changed, as it implicitly supports round robin already. By using a linked list instead of an array, we make our job much easier for ourselves. This round robin functionality is then implemented within the EOQTimer class' function handle_interrupts, which takes care of the whole requeueing businesses. So by using a linked list, we need not to even derive a new round robin scheduler, the functionality we have to add in our EOQTimer handles round robin for us already. We solve the issue of handling interrupts as described in the handout by editing the dispatch_interrupts function. We send an EOI signal before handing off to the interrupt_handler, then we return to avoid sending another EOI.

thread.C: dispatch_to : This function dispatches to another thread using low level code. I added the functionality of setting the scheduler running variable to running. This is because the scheduler only starts running officially once the first thread is dispatched to. This is only really relevant in the case that the EOQTimer is being used.

```
void Thread::dispatch_to(Thread * _thread) {
/* Context-switch to the given thread. Calls the low-level context switch code
   in thread_low.asm.
   NOTE: This call does not return until after the current thread is switched back in.
   NOTE: We don't consider the system start thread as an actual thread. Therefore, we will
         not return from this function ever when the system start code (in kernel.C) starts up
         the first thread.
*/

    /* The value of 'current_thread' is modified inside 'threads_low_switch_to()'. */

    if (Scheduler::running == false)
        Scheduler::running = true;

    threads_low_switch_to(_thread);

    /* The call does not return until after the thread is context-switched back in. */
}
```

simple_timer.C: EOQTimer::handle_interrupts : This function implements an override for the default SimpleTimer handle_interrupts function. Instead of printing, it preempts the currently running thread. It calls resume to add the thread to the ready queue again and then yields the CPU. This essentially implements round robin behavior. It preempts the thread according to a time quantum that's passed in when the timer is being constructed.

```
void EOQTimer::handle_interrupt(REGS *_r) {
    if (Scheduler::running)
        ticks++;

    if (ticks >= hz) {
        ticks = 0;
        Scheduler::scheduler->resume(Thread::CurrentThread());
        Scheduler::scheduler->yield();
    }
}
```

interrupts.C: dispatch_interrupts : This function is nearly identical to the normal dispatch function. The only modification made is within the else block where we send the EOI message to inform the interrupt controller that the interrupt has been handled. After the handler is called we return to prevent sending another EOI message.

```
void InterruptHandler::dispatch_interrupt(REGS * _r) {

  /* -- INTERRUPT NUMBER */
  unsigned int int_no = _r->int_no - IRQ_BASE;

  //Console::puts("INTERRUPT DISPATCHER: int_no = ");
  //Console::putui(int_no);
  //Console::puts("\n");

  assert((int_no >= 0) && (int_no < IRQ_TABLE_SIZE));

  /* -- HAS A HANDLER BEEN REGISTERED FOR THIS INTERRUPT NO? */

  InterruptHandler * handler = handler_table[int_no];

  if (!handler) {
    /* --- NO DEFAULT HANDLER HAS BEEN REGISTERED. SIMPLY RETURN AN ERROR. */
    Console::puts("INTERRUPT NO: ");
    Console::puti(int_no);
    Console::puts("\n");
    Console::puts("NO DEFAULT INTERRUPT HANDLER REGISTERED\n");
    //    abort();
  }
  else {
    /* -- HANDLE THE INTERRUPT */
    Machine::outportb(0x20, 0x20);
    handler->handle_interrupt(_r);
    return;
  }

  /* This is an interrupt that was raised by the interrupt controller. We need
       to send and end-of-interrupt (EOI) signal to the controller after the
       interrupt has been handled. */

  /* Check if the interrupt was generated by the slave interrupt controller.
       If so, send an End-of-Interrupt (EOI) message to the slave controller. */

  if (generated_by_slave_PIC(int_no)) {
    Machine::outportb(0xA0, 0x20);
  }

  /* Send an EOI message to the master interrupt controller. */
  Machine::outportb(0x20, 0x20);

}
```

Testing

I assumed that the default testing provided enough coverage for all the normal test cases. My added test is extremely minimal. I add a singular test case to test the code path for deleting a thread that in the ready queue, as to my knowledge that is the only code path not tested by the normal testing that was provided. For testing round robin, I added an extra definition that defines "_USES_RR", comment this out to not use RR. This define essentially just makes the pass_on_cpu function do nothing and instantiates an EOQTimer instead of SimpleTimer.

```
#define _CUSTOM_TEST

Thread * testThread1 = NULL;
Thread * testThread2 = NULL;
Thread * testThread3 = NULL;

void test1() {
    for(int i = 0;; i++) {
        Console::puts("Test thread 1 running "); Console::puti(i); Console::puts("\n");
    }
}

void test2() {
    for(int i = 0;; i++) {
        Console::puts("Test thread 2 running "); Console::puti(i); Console::puts("\n");

        if (testThread1 != NULL) {
            SYSTEM_SCHEDULER->terminate(testThread1);
            Console::puts("Terminated test thread 1!\n");
        }
    }
}

void test3() {
    for (int i = 0;; i++) {
        Console::puts("test thread 3 running "); Console::puti(i); Console::puts("\n");
    }
}
```

```
#ifdef _CUSTOM_TEST
    char * testStack1 = new char[1024];
    char * testStack2 = new char[1024];
    char * testStack3 = new char[1024];

    testThread1 = new Thread(test1, testStack1, 1024);
    testThread2 = new Thread(test2, testStack2, 1024);
    testThread3 = new Thread(test3, testStack3, 1024);

    SYSTEM_SCHEDULER->add(testThread2);
    SYSTEM_SCHEDULER->add(testThread3);

    Thread::dispatch_to(testThread1);

    return 1;

#endif
```