MP5: Kernel-Level Thread Scheduling

Geetesh Challur
UIN: 834006606
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1 Correct handling of interrupts: Completed.
Bonus Option 2 Round Robin Scheduling: Completed.
Bonus Option 3 Processes: Did not attempt.

System Design

- A scheduler is implemented, which handles smooth dispatching of the thread whenever the current running thread dispatches a new thread, if it's terminated or if it's pre-empted.
- A ready queue is implemented using a linked list, which contains all the processes to be executed in the order.
- The scheduler class supports all the functions required to terminate a thread, resume a blocked thread, and switch to a new thread.
- The logistics of how the scheduler implements all this functionality is described in detail below.

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.H: : The ready queue is implemented using a linked list.

- A struct ready queue is defined, representing a node of the ready queue that stores the address of the thread and pointer to the next node.
- Head and tail of the ready queue are defined.

```
//   The ready queue is implemented using a linked list
  struct ready_queue
  {
      Thread * thread;
      ready_queue * next;
  };
   ready_queue * head;
   ready_queue * tail;
```

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

- Interrupts are disabled in these schduler functions because they may modify the ready queue which requires mutual exclusion.
- Checking if the ready queue is empty.
- If the ready queue is not empty, the first thread in the queue is dispatched.
- Interrupts are enabled again before leaving the function.

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

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

```
Scheduler::Scheduler() {

  // Initializing the private members
  head = nullptr;
  tail = nullptr;

}

void Scheduler::yield() {
  // Interrupts are diabled 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 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);
}

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

  // Adding the thread to the end of ready queue when the ready queue is empty.
  if(head == nullptr) {
    head = new ready_queue;
    head->thread = _thread;
    head->next = nullptr;
    tail = head;
    if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }

    return;
  }

  // Adding the thread to the end of ready queue when the ready queue is not empty.
  ready_queue * new_thread = new ready_queue;
  new_thread->thread = _thread;
  tail->next = new_thread;
  tail = new_thread;
  if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
  return;
}
```

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

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

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

- If the currently running thread is trying to terminate itself, yield is called.(Thread suicide)
- If the currently running thread is trying to terminate another thread in the ready queue. We iterate through the list to delete the thread and remove the respective node in the queue
- If the thread to terminate id not found in the ready queue, we assert false

```
void Scheduler::add(Thread * _thread) {
  if (Machine::interrupts_enabled()) { Machine::disable_interrupts(); }
  // Adding the thread to the end of ready queue when the ready queue is empty.
  if(head == nullptr) {
    head = new ready_queue;
    head->thread = _thread;
    head->next = nullptr;
    tail = head;
    if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
    return;
  }

  // Adding the thread to the end of ready queue when the ready queue is not empty.
  ready_queue * new_thread = new ready_queue;
  new_thread->thread = _thread;
  tail->next = new_thread;
  tail = new_thread;
  if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
  return;
}

void Scheduler::terminate(Thread * _thread) {
  if (Machine::interrupts_enabled()) { Machine::disable_interrupts(); }
  int id = _thread->ThreadId();

  // If the current running thread is trying to terminate itself.(Thread suicide)
  if(_thread == Thread::CurrentThread()) {
    Console::puts("Thread"); Console::puti(Thread::CurrentThread()->ThreadId()); Console::puts(" suicide.\n");
    if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
    yield();
    return;
  }

  // If the current running thread is trying to terminate another thread in the ready queue.
  //Iterating through the list to remove delete the thread and remove the respective node in the queue
  ready_queue * temp = head;
  if(temp->thread == _thread) {
    head = head->next;
    delete temp;
    if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
    return;
  }
  while(true){
    // If the thread to terminate id not found in the ready queue
    if(temp== tail){
      assert(false);
    }
    if(temp->next->thread == _thread) {
      ready_queue * temp2 = temp->next;
      temp->next = temp->next->next;
      delete temp2;
      delete _thread;
      if (!Machine::interrupts_enabled()) { Machine::enable_interrupts(); }
      return;
    }
  }

}
```

thread.C: thread_shutdown : The terminating functions are dealt by modifying the thread shutdown function in this file

- Thread shutdown function explains what happens when the thread function ends.
- Here, we call the terminate function of the scheduler to terminate the thread as soon as it returns from execution. Thread suicide happens here

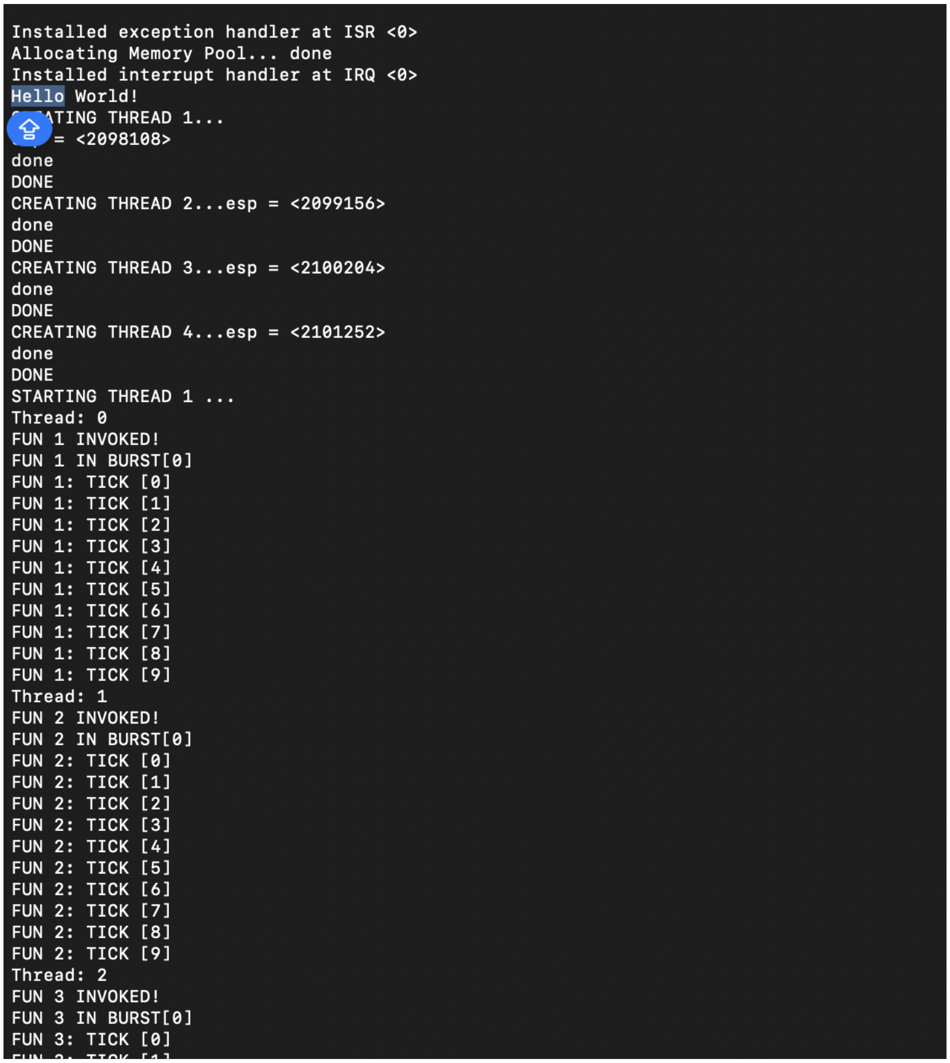
```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */
    // Calling the terminate function of the scheduler to terminate the thread as soon as it returns from exectuition. Thread suicide happens here
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());

    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
}
```

Testing

To test the code for just the FIFO scheduler, the TA or instructor has to comment _USES_RR_SCHEDULER_ from kernel.C which is currently uncommented.

The _USES_SCHEDULER_ in kernel.C is uncommented to test if the scheduler is implemented correctly. We can see below that the thread dispatching occurs as expected.



```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Hello World!
CREATING THREAD 1...
esp = <2098108>
done
DONE
CREATING THREAD 2...esp = <2099156>
done
DONE
CREATING THREAD 3...esp = <2100204>
done
DONE
CREATING THREAD 4...esp = <2101252>
done
DONE
STARTING THREAD 1 ...
Thread: 0
FUN 1 INVOKED!
FUN 1 IN BURST[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]
Thread: 1
FUN 2 INVOKED!
FUN 2 IN BURST[0]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Thread: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
```

The _TERMINATING_FUNCTIONS_ is uncommented to test if the functions are terminated correctly. After a while fun1 and fun2 are terminated and we can see below that only fun3 and fun4 are running.

```
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[18]
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]
FUN 4 IN BURST[18]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[19]
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]
FUN 4 IN BURST[19]
FUN 4: TICK [0]
```

Bonus Option 1: Correct handling of interrupts (including testing)

- The code in thread.C is modified to enable the interrupts even after the threads start running.
- This is done by enabling the interrupts in the thread_start function which is used to release the thread for execution in the ready queue.
- And while executing the functions from the scheduler classes, the interrupts are disabled since they modify the ready queue, which requires mutual exclusion. They are enabled again before returning from these functions.
- Since the interrupts are enabled, we can see below that the simpler timer message is appearing every second.

```
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
One second has passed
FUN 3: TICK [9]
FUN 4 IN BURST[60]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
```

Bonus Option 2: Round robin scheduler (including testing)

‘

- A new variable _USES_RR_SCHEDULER is defined. Uncomment this if you want to test the round-robin scheduler(It is already uncommented in the submitted file).
- simple_timer.C file is updated to implemented this functionality.
- EOQ timer is defined here inherited from the SimpleTimer class.
- The handle interrupt function is overridden to switch the thread after a time quantum has passed.
- As soon as the time quantum has passed, we switch the thread by resuming the current thread and yielding.
- the interrupts file is changed so that the system knows whether the thread is pre-tempted forcibly because of the end of quantum or if it voluntarily gave up the CPU.
- In the screenshot below, we can see that fun4 is pre-empted after tick 6 because of the end of time quantum, and it executes fun3, which is at the start of the ready queue.
- In the second screenshot, we can see that fun4 is executed again starting from tick 7, as soon as fun3 ends.

```
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
one time quantum has passed. Switching the thread
FUN 3 IN BURST[20]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
```

```
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
```