MP5: Kernel-Level Thread Scheduling

Po-Chieh Wang
UIN: 933004975
CSCE 611: Operating System

Assigned Tasks

Main: Completed.
Bonus option 1: Completed.
Bonus option 2: Completed.
Bonus option 3: Did not attempt.

System Design

The goal of this machine problem is to implement scheduling algorithms of kernel-level threads. In the main part, I created a linked-list with two pointers to do FIFO scheduling. In bonus 1 part, when we enter the critical section, which is doing scheduling, we should turn off the interrupt to do mutual exclusion. Therefore, I turned off the interrupt every time before scheduling and turned it on after scheduling. In bonus 2 part, I implemented two classes to complete Round-Robin scheduling. The first one is called EOQTimer, which inherits from SimpleTimer but will call the scheduling class after 1 qauntum. The second one is called RRScheduler, which inherits from Scheduler. The only difference is that when the thread terminates, it will call RRScheduler to reset the time in case of affecting the next running thread's quantum. By building these two classes, the system now can do round-robin scheduling of kernel-level threads.

Code Description

Main and bonus 1: In these parts, I implemented FIFO scheduler and correct handling of interrupts by modifying Scheduler.H, Scheduler.C, Thread.H, and Thread.C.

Scheduler.H: Queue: This class built a linked-list to store the information of ready threads.

```
class Queue {
   public:
      static Queue * head;
      static Queue * tail;
      Thread * curr_thread;
      Queue * next;
      Queue(Thread * _thread);
      static void enqueue(Queue * _queue);
      static Thread * dequeue();
};
```

Scheduler.H: Scheduler: The only modification here is to give it a private member of the queue.

```
class Scheduler {
   Queue * queue;
   /* The scheduler may need private members... */
```

Scheduler.C: Queue::Queue: This constructor sets the current thread and the next thread.

```
Queue::Queue(Thread * _thread){
  curr_thread = _thread;
  next = NULL;
}
```

Scheduler.C: Queue::enqueue: This function pushes the input thread to the end of the queue.

```
void Queue::enqueue(Queue * _queue){
  if(head == NULL){
    head = _queue;
    tail = _queue;
  }
  else{
    tail->next = _queue;
    tail = _queue;
  }
}
```

Scheduler.C: Queue::dequeue: This function pops the first thread in the queue.

```
Thread * Queue::dequeue(){
  Thread * removed_thread = head->curr_thread;
  Queue * temp = head;
  if(head == tail){
    head = NULL;
    tail = NULL;
  }
  else{
    head = head->next;
  }
  delete temp;
  return removed_thread;
}
```

Scheduler.C: Scheduler::yield: When this function is called, it first closes the interrupts because it is in the critical section. Then it will pop the first thread in the queue and call dispatch_to function to run it. If the thread in its first time to run, it will turn on the interrupts. If it is not, it will

return from yield and turn on interrupts.

```
void Scheduler::yield() {
  if(queue->head != NULL){
    if(Machine::interrupts_enabled()){
      Machine::disable_interrupts();
    }
    Thread * next_thread = Queue::dequeue();
    Thread::dispatch_to(next_thread);
    if(!Machine::interrupts_enabled()){
      Machine::enable_interrupts();
    }
  }
}
```

Scheduler.C: Scheduler::resume: This function also turns off the interrupts at first and turns on in the end. This function puts the input thread at the end of the queue.

```
void Scheduler::resume(Thread * _thread) {
  if(Machine::interrupts_enabled()){
      Machine::disable_interrupts();
    }
  Queue * new_queue = new Queue(_thread);
  Queue::enqueue(new_queue);
  if(!Machine::interrupts_enabled()){
      Machine::enable_interrupts();
    }
}
```

Scheduler.C: Scheduler::add: This function basically does the same thing as resume.

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

Scheduler.C: Scheduler::terminate: This function also turns off the interrupts at first and turns on in the end. Also, it releases the stack of the thread by calling delete_stack function. After that, it calls yield function to pass on the CPU.

```
void Scheduler::terminate(Thread * _thread) {
  if(Machine::interrupts_enabled()){
      Machine::disable_interrupts();
    }
  Thread * current_thread = _thread;
  current_thread->delete_stack();
  yield();
  if(!Machine::interrupts_enabled()){
      Machine::enable_interrupts();
    }
}
```

Thread.H: Thread: I added two functions. yield_thread (for Round-Robin scheduling) and delete_stack function (for terminating threads).

```
    static void yield_thread();

    void delete_stack();
```

Thread.C: Thead::thread_shutdown: This function used the external parameter to call the terminate function.

```
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 t
       This is a bit complicated because the thread termination interacts with the schedu
     */

    SYSTEM_SCHEDULER->terminate(current_thread);
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */
}
```

Thread.C: Thead::thread_start: This function enables interrupts.

```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
    /* We need to add code, but it is probably nothing more than enabling interrupts. */
}
```

Thread.C: Thread::delete_stack: This function deletes stack inside the thread.

```
void Thread::delete_stack(){
    delete stack;
}
```

Bonus 2: In this part, I implemented Round-Robin scheduling by modifying Thread.C, Thread.H, Scheduler.C, Scheduler.H, SimpleTimer.C and SimpleTimer.H.

Thread.C: Thread::yield_thread: This function yields the current thread by calling resume and yield function inside the scheduler.

```
void Thread::yield_thread(){
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
}
```

Scheduler.H: RRScheduler: This class inherits from Scheduler, it has a pointer to EOQTimer and its own yield function.

```
class RRScheduler: public Scheduler {

   Queue * queue;
   EOQTimer * eoq_timer;

public:
   RRScheduler(EOQTimer * _eoq_timer);
   void yield();
};
```

Scheduler.C: RRScheduler::RRScheduler: This constructor sets the eoq_timer.

```
RRScheduler::RRScheduler(EOQTimer * _eoq_timer){
  eoq_timer = _eoq_timer;
}
```

Scheduler.C: RRScheduler::yield: This function does the same thing as yield in scheduler, the only difference is that when this function is called, it will reset the tick inside the eoq_timer to 0.

```
void RRScheduler::yield(){
  if(queue->head != NULL){
    if(Machine::interrupts_enabled()){
      Machine::disable_interrupts();
    }
    eoq_timer->reset_tick();
    Thread * next_thread = Queue::dequeue();
    Thread::dispatch_to(next_thread);
    if(!Machine::interrupts_enabled()){
      Machine::enable_interrupts();
    }
  }
}
```

SimpleTimer.H: EOQTimer: This class inherits from SimpleTimer, it has its own functions of handle_interrupt and reset_tick.

```
class EOQTimer: public SimpleTimer{

private:

  /* How long has the system been running? */
  unsigned long seconds;
  int           ticks;   /* ticks since last "seconds" update.    */

  /* At what frequency do we update the ticks counter? */
  int hz;                /* Actually, by defaults it is 18.22Hz.
                            In this way, a 16-bit counter wraps
                            around every hour.                    */
  void set_frequency(int _hz);
  /* Set the interrupt frequency for the simple timer. */

public:
  EOQTimer(int _hz);
  void handle_interrupt(REGS *_r);
  void reset_tick();
};
```

SimpleTimer.C: EOQTimer::handle_interrupt: This function first tells the PIC that interrupts are handled. When the ticks reach the setting quantum (50ms), it will call the function of yield_thread.

```
void EOQTimer::handle_interrupt(REGS *_r){
    ticks++;
    Machine::outportb(0x20, 0x20);
    /* Whenever a second is over, we update counter accordingly. */
    if (ticks >= (hz / 20) )
    {
        Console::puts("Context Switch!\n");
        Thread::yield_thread();
    }

}
```

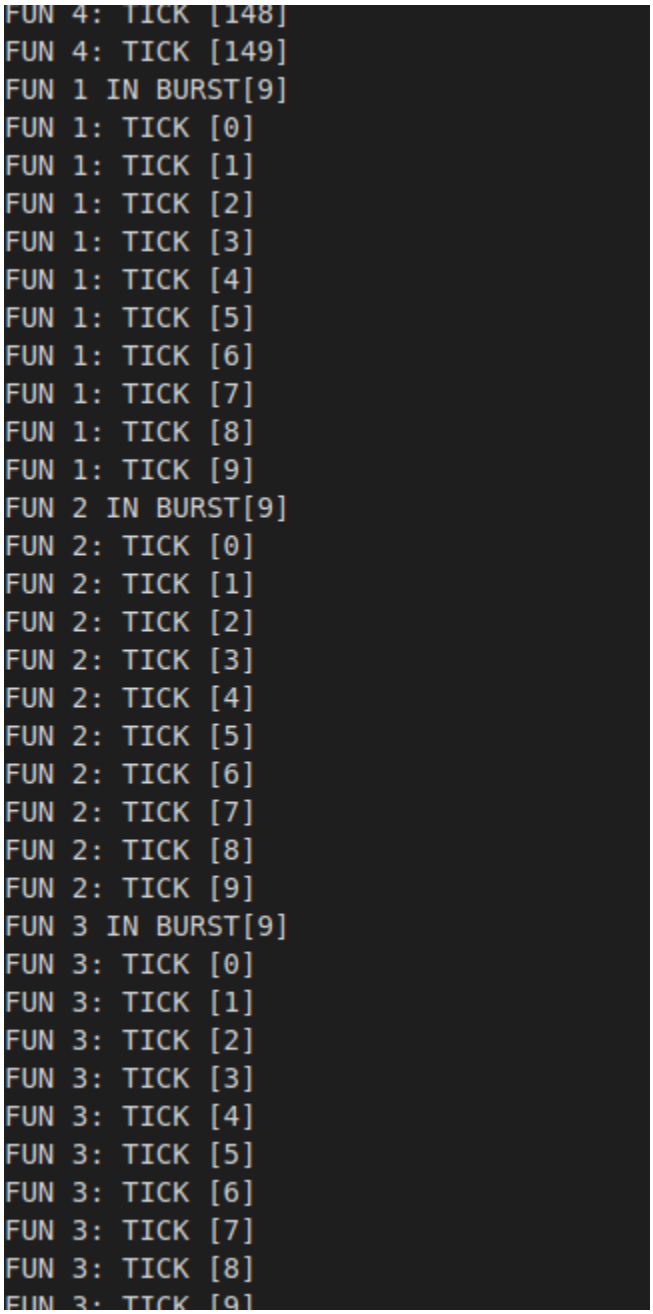
SimpleTimer.C: EOQTimer::reset_tick: This function sets ticks to 0.

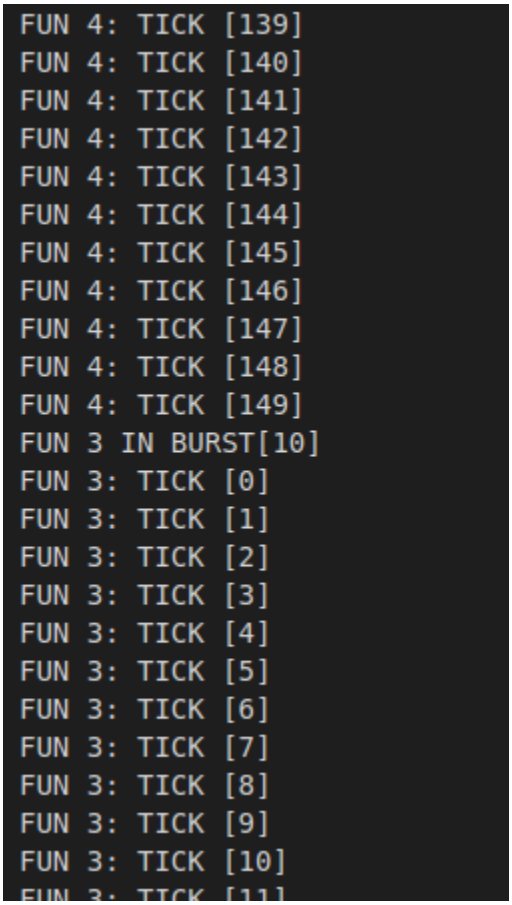
```
void EOQTimer::reset_tick(){
    ticks = 0;
}
```

Testing

I think the tests provided in Kernel.C are enough for main part and bonus 1, nothing needs to be added. I only changed the for loop inside function 3 and 4 to 150 for testing Round-Robin

scheduling. The test result shows that the kernel-thread scheduling and correct interrupts are correct. Functions voluntarily give up CPU after 10 cycles, function 1 and 2 finished in burst 10.





For bonus 2 part, in order to test the round-robin scheduling to preempt after 1 quantum. I did some modifications in Kernel.C. I changed the scheduler to RRScheduler and the timer to EOQTimer. The result shows that the RRScheduler performs correctly and the thread will be preempted when the time is up.

```
#ifdef _RR_SCHEDULER_

EOQTimer timer(100);
InterruptHandler::register_handler(0, &timer);
SYSTEM_SCHEDULER = new RRScheduler(&timer);

#else

SimpleTimer timer(100); /* timer ticks every 10ms. */
InterruptHandler::register_handler(0, &timer);
/* The Timer is implemented as an interrupt handler. */
#ifdef _USES_SCHEDULER_

    /* -- SCHEDULER -- IF YOU HAVE ONE -- */

    SYSTEM_SCHEDULER = new Scheduler();

#endif
```

```
FUN 4: TICK [54]
FUN 4: TICK [55]
FUN 4: TICK [56]
FUN 4: TICK [57]
FUN 4: TICK [58]
FUN 4: TICK [59]
FUN 4: TICK [60]
FUN 4: TICK [61]
FUN 4: TICK [62]
FUN 4: TICK [63]
FUN 4: TICK [64]
FUN 4: TICK [65]
Context Switch!
FUN 3: TICK [113]
FUN 3: TICK [114]
FUN 3: TICK [115]
FUN 3: TICK [116]
FUN 3: TICK [117]
FUN 3: TICK [118]
FUN 3: TICK [119]
FUN 3: TICK [120]
FUN 3: TICK [121]
FUN 3: TICK [122]
FUN 3: TICK [123]
FUN 3: TICK [124]
FUN 3: TICK [125]
FUN 3: TICK [126]
FUN 3: TICK [127]
FUN 3: TICK [128]
FUN 3: TICK [129]
FUN 3: TICK [130]
FUN 3: TICK [131]
FUN 3: TICK [132]
```