In this machine problem. I choose to accomplish **option 1** and **option 2** design. Basically, I implemented the idea of doubly linked list to perform the FIFO scheduler juts like a ready queue functionality. The simple idea would be that the scheduler would maintain a list of Thread structures, the next pointer for thread, and the previous pointer for the previous thread. At the beginning, I uncommented the "#define _USES_SCHEDULER_" and "#define _TERMINATING_FUNCTIONS_" to allow the code to use the scheduler and thread in kernel.C.

In kernel.C, replace the delete function definition to make sure the other file can call delete function here.

```
//replace the operator "delete"
void operator delete (void * p, size_t size) {
    MEMORY_POOL->release((unsigned long)p);
}
```

In thread.H/C

I "extern" MemPool and Scheduler, so that I can directly use them in the thread.C

extern MemPool* MEMORY_POOL;

extern Scheduler* SYSTEM_SCHEDULER;

After that, I implemented MEMORY_POOL AND SYSTEM_SCHEDULER here to implement thread_shutdown function and thread_start function. For shutdown, it was simple that just terminated the current_thread, and set current_thread to 0. And start function just easily set to enable the interrupts.

```
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.
    */
    if (Machine::interrupts_enabled()) {
        Machine::disable_interrupts();
    }
    Console::puts("Thread ");
    Console::putui(current_thread->ThreadId());
    Console::puts(" shut down\n");
    SYSTEM_SCHEDULER->resume(current_thread);
    SYSTEM_SCHEDULER->terminate(current_thread);
    MEMORY_POOL->release((unsigned long)(current_thread->stack_address()));
    MEMORY_POOL->release((unsigned long)current_thread);
    current_thread = 0;
    SYSTEM_SCHEDULER->yield();
}

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. */
     if (!Machine::interrupts_enabled()) {
         Machine::enable_interrupts();
     }
}
```

There were two more simple implementations in thread.C, which just return stack address and yield the thread.

```
unsigned long Thread::stack_address() {
     return (unsigned long) stack;
}

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

In scheduler.H/C

In header file, I created a Node class in order to deal with the doubly linked list mechanism here. And add head node and tail node in the public of Scheduler class.

```
class Node {
public:
    Thread* thread;
    Node* next;
    Node* pre;
    Node();
    Node(Thread* _thread);
};

class Scheduler {

   /* The scheduler may need private members... */

public:
    static Node* head;
    static Node* tail;
```

In scheduler.C, I did the important initializations first to set NULL to thread, next, and pre of Node class.

```
Node::Node() {
  thread = NULL;
  next = NULL;
  pre = NULL;
}

Node::Node(Thread* _thread) {
  thread = _thread;
  next = NULL;
  pre = NULL;
}

Node* Scheduler::head = NULL;
Node* Scheduler::tail = NULL;
```

Then in the scheduler constructor, I just set head point to the tail at the first. For yield() function, just got the current node and dispatched it.

```
Scheduler::Scheduler() {
  // assert(false);
  head = new Node();
  tail = new Node();
  head->next = tail;
  tail->pre = head;
  Console::puts("Constructed Scheduler.\n");
}

void Scheduler::yield() {
  // assert(false);
  if (Machine::interrupts_enabled()) {
    Machine::disable_interrupts();
  }
  Node* cur = head->next;
  // point to the second node in ready queue
  head->next = cur->next;

  cur->next = NULL;
  cur->pre = NULL;
  Thread::dispatch_to(cur->thread);

}
```

Then the resume() function and add() function were mostly serve the same purpose, so just call add() in the resume() function. Simply, just added one thread at the end.

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

void Scheduler::add(Thread * _thread) {
  // assert(false);
  if (!Machine::interrupts_enabled()) {
    Machine::enable_interrupts();
  }
  Node* new_node = new Node(_thread);
  Node* temp_node = tail->pre;
  temp_node->next = new_node;
  new_node->pre = temp_node;
  new_node->next = tail;
  tail->pre = new_node;
}
```

Terminate function just offered a method to find the target thread and remove the whole node.

```
void Scheduler::terminate(Thread * _thread) {
  // assert(false);
  if (head->next == tail) {
    // no thread to terminate
    return;
  }
  Node* temp_node = head->next;
  while (temp_node != tail) {
    if (temp_node->thread->ThreadId() == _thread->ThreadId()) {
      temp_node->pre->next = temp_node->next;
      temp_node->next->pre = temp_node->pre;
      // delete temp_node, and free memory
      // MEMORY_POOL->release((unsigned long)temp_node);
      delete temp_node;
      break;
    }
    temp_node = temp_node->next;
  }
}
```

For option 1 implementation, I just added enable interrupts after executing the thread in interrupts.C.

```
  else {
    // /* -- HANDLE THE INTERRUPT */
    // handler->handle_interrupt(_r);
  // }

  // make changes here; to set handler after sending EOI messages.

  /* 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);
    /* -- HANDLE THE INTERRUPT */
    handler->handle_interrupt(_r);
  }
```

For option 2 implementation, it is used for round robin, and required a time quantum of 50ms, which I decide to modify something under the simple_timer.C. I successfully made the ticks to 50ms interval, and reset the ticks every time and yield the thread.

```
void SimpleTimer::handle_interrupt(REGS *_r) {
/* What to do when timer interrupt occurs? In this case, we update "ticks",
   and maybe update "seconds".
   This must be installed as the interrupt handler for the timer in the
   when the system gets initialized. (e.g. in "kernel.C") */

    /* Increment our "ticks" count */
    ticks++;

    /* Whenever a second is over, we update counter accordingly. */
    // if (ticks >= hz )
    // {
    //     seconds++;
    //     ticks = 0;
    //     Console::puts("One second has passed\n");
    // }
    // 50ms interval, time quantum
    if (ticks >= hz/20 )
    {
        // seconds++;
        ticks = 0;
        Console::puts("One time quantum has passed\n");
        Thread::thread_yield();
    }
}
```