Machine Problem 5

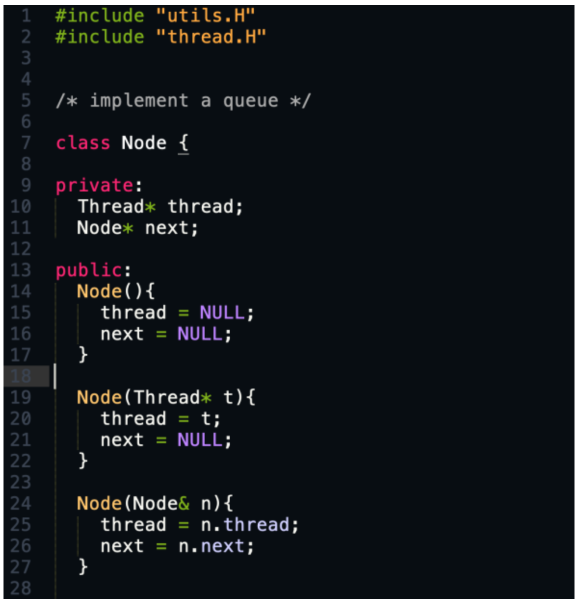
Jicheng Lu, 525004048

1. Introduction

In Machine Problem 5, we would add scheduling of multiple kernel-level threads. The emphasis is on scheduling. We would first implement a simple First-In-First-Out (FIFO) scheduler, and then deal with the handling of interrupts (option 1) and Round-Robin scheduling (option 2). Note that I submitted two sets of code, each of which corresponds the two options.

2. FIFO and Handling of Interrupts

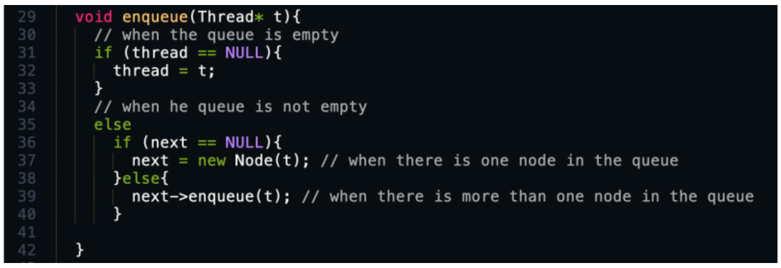
We first deal with the FIFO scheduler. Before developing the scheduler, we need to define a ready queue, where we could put the threads. In order to do this, we use link list data structure and define a new class "Node" in "node.H". The code implementation is presented in Fig 2.1.



```
#include "utils.H"
#include "thread.H"


/* implement a queue */

class Node {

private:
  Thread* thread;
  Node* next;

public:
  Node(){
    thread = NULL;
    next = NULL;
  }

  Node(Thread* t){
    thread = t;
    next = NULL;
  }

  Node(Node& n){
    thread = n.thread;
    next = n.next;
  }

```

Figure 2.1. Class "Node" in "node.H".

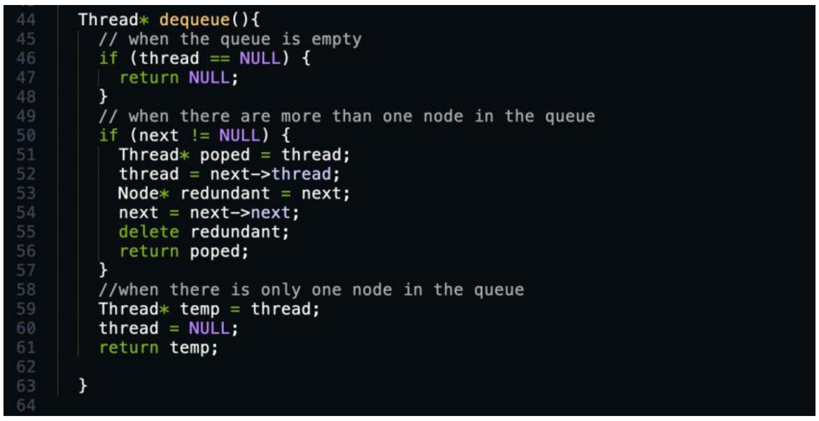
Then we define two operations: enqueue and dequeue. The "enqueue" operation is to put a thread at the end of the ready queue. Note that we use recursive method to ensure we can find the end of queue. The code implementation is presented in Fig 2.2.



```cpp
void enqueue(Thread* t){
  // when the queue is empty
  if (thread == NULL){
    thread = t;
  }
  // when he queue is not empty
  else
    if (next == NULL){
      next = new Node(t); // when there is one node in the queue
    }else{
      next->enqueue(t); // when there is more than one node in the queue
    }

}
```

Figure 2.2. enqueue in “node.H”.

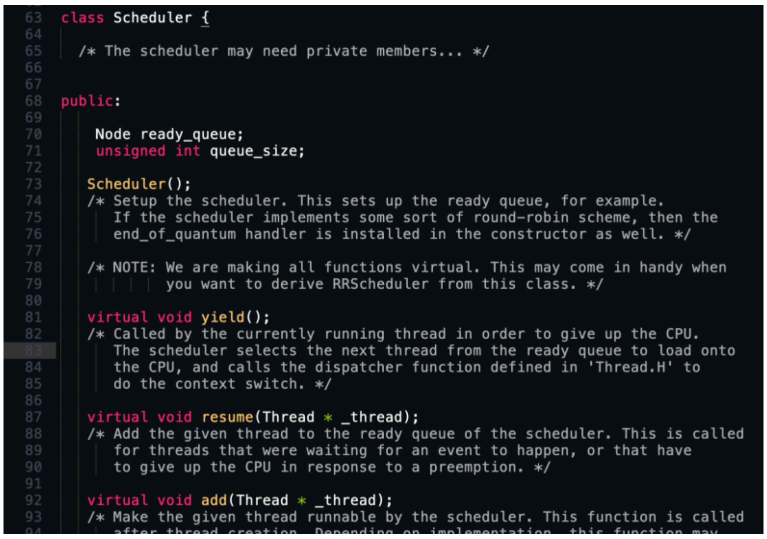
Next, we define another operation: dequeue. This is to get the first thread in the ready queue. The code implementation is presented in Fig 2.3.



```cpp
Thread* dequeue(){
  // when the queue is empty
  if (thread == NULL) {
    return NULL;
  }
  // when there are more than one node in the queue
  if (next != NULL) {
    Thread* poped = thread;
    thread = next->thread;
    Node* redundant = next;
    next = next->next;
    delete redundant;
    return poped;
  }
  //when there is only one node in the queue
  Thread* temp = thread;
  thread = NULL;
  return temp;

}
```

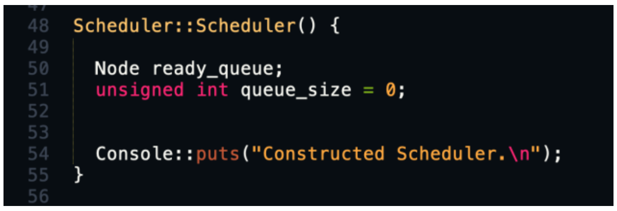
Figure 2.3. dequeue in “node.H”.

After defining our ready queue and its operations, we can deal with the operations related to the scheduler. It consists of five parts: constructor, yield, resume, add, and terminate. We start from the constructor. When we setup the scheduler, we initialize a ready queue and its size. The code implementation is presented in Fig 2.4 and Fig. 2.5.



```cpp
class Scheduler {

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


public:

    Node ready_queue;
    unsigned int queue_size;

   Scheduler();
   /* Setup the scheduler. This sets up the ready queue, for example.
      If the scheduler implements some sort of round-robin scheme, then the
      end_of_quantum handler is installed in the constructor as well. */

   /* NOTE: We are making all functions virtual. This may come in handy when
            you want to derive RRScheduler from this class. */

   virtual void yield();
   /* Called by the currently running thread in order to give up the CPU.
      The scheduler selects the next thread from the ready queue to load onto
      the CPU, and calls the dispatcher function defined in 'Thread.H' to
      do the context switch. */

   virtual void resume(Thread * _thread);
   /* Add the given thread to the ready queue of the scheduler. This is called
      for threads that were waiting for an event to happen, or that have
      to give up the CPU in response to a preemption. */

   virtual void add(Thread * _thread);
   /* Make the given thread runnable by the scheduler. This function is called
      after thread creation. Depending on implementation, this function may
```

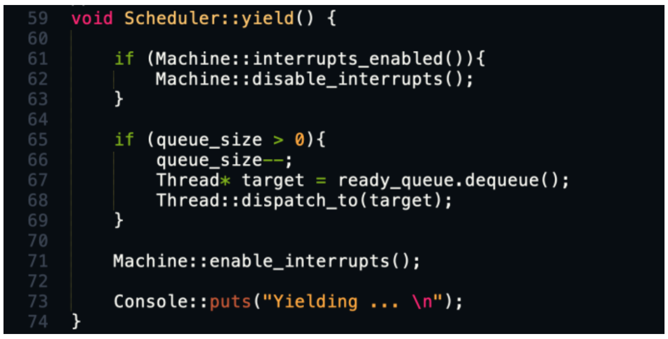
Figure 2.4. Modification in “scheduler.H”.



```
Scheduler::Scheduler() {

  Node ready_queue;
  unsigned int queue_size = 0;


  Console::puts("Constructed Scheduler.\n");
}
```

Figure 2.5. Setup a scheduler in “scheduler.C”.

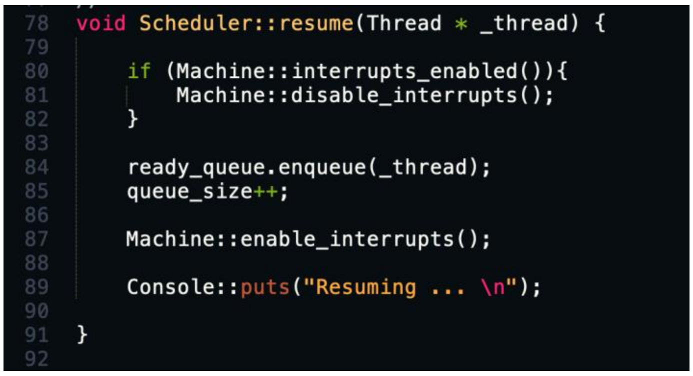
Then we define the “yield” operation. This is to let the scheduler select the next thread from the ready queue and load it onto the CPU. This can be done by using the “enqueue” and “dispatch_to” operations. The code implementation is presented in Fig 2.6.



```
void Scheduler::yield() {

    if (Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }

    if (queue_size > 0){
        queue_size--;
        Thread* target = ready_queue.dequeue();
        Thread::dispatch_to(target);
    }

    Machine::enable_interrupts();

    Console::puts("Yielding ... \n");
}
```

Figure 2.6. yield operation in “scheduler.C”.

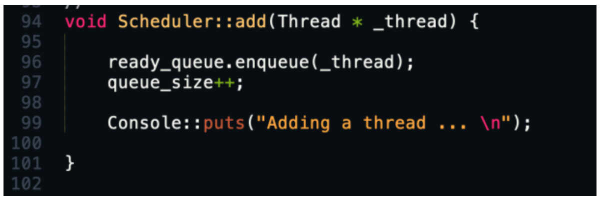
The “resume” operation is to add the given thread at the end of the ready queue. This can be done by using the “enqueue” operation. The code implementation is presented in Fig 2.7.



```
void Scheduler::resume(Thread * _thread) {

    if (Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }

    ready_queue.enqueue(_thread);
    queue_size++;

    Machine::enable_interrupts();

    Console::puts("Resuming ... \n");

}
```

Figure 2.7. resume operation in “scheduler.C”.

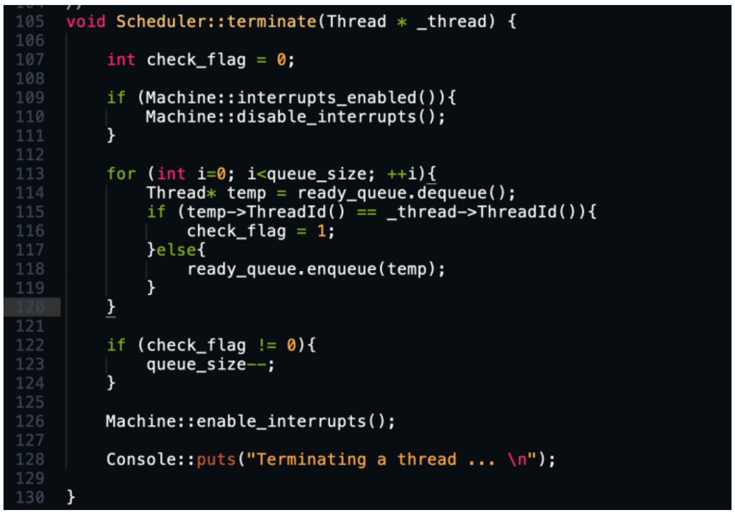
The “add” operation is to add a thread to the end of the ready queue. It is called after a thread is created. This can be done by using the “enqueue” operation. The code implementation is presented in Fig 2.8.



```
void Scheduler::add(Thread * _thread) {

    ready_queue.enqueue(_thread);
    queue_size++;

    Console::puts("Adding a thread ... \n");

}
```

Figure 2.8. add operation in “scheduler.C”.

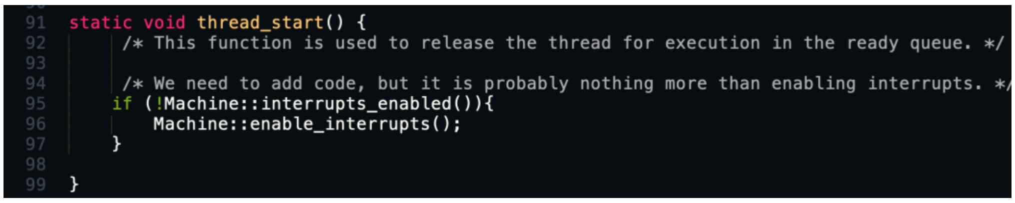
Finally, we identify the thread we want to terminate by using its ID. Once we find the thread in the ready queue, we get it using “dequeue” operation and remove it. The code implementation is presented in Fig 2.9.



```
void Scheduler::terminate(Thread * _thread) {

    int check_flag = 0;

    if (Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }

    for (int i=0; i<queue_size; ++i){
        Thread* temp = ready_queue.dequeue();
        if (temp->ThreadId() == _thread->ThreadId()){
            check_flag = 1;
        }else{
            ready_queue.enqueue(temp);
        }
    }

    if (check_flag != 0){
        queue_size--;
    }

    Machine::enable_interrupts();

    Console::puts("Terminating a thread ... \n");

}
```

Figure 2.9. terminate operation in “scheduler.C”.

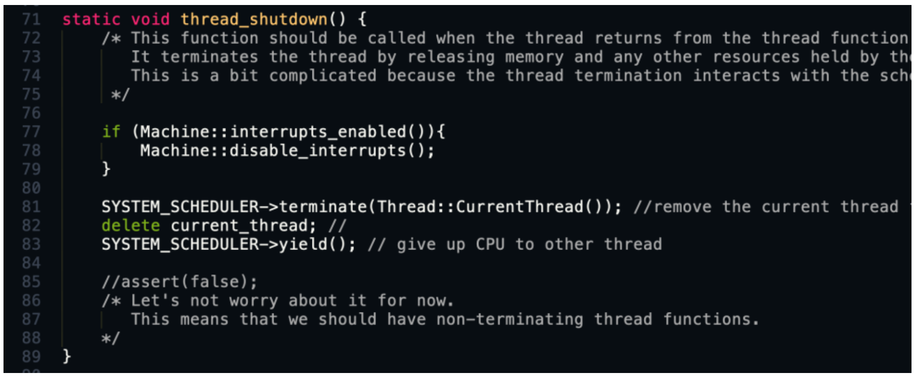
After completing the scheduler, we can implement these operations in “thread.C”. In “thread_start()” function, we simply enable the interrupts. The code implementation is presented in Fig 2.10.



```
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();
    }

}
```

Figure 2.10. thread_start() in “thread.C”.

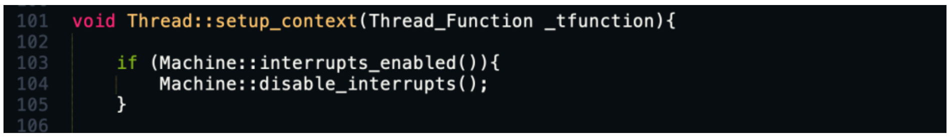
In the “thread_start()” function, we remove the thread that we want to shut down and give up CPU to the next thread. The code implementation is presented in Fig 2.11.



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

    if (Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }

    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread()); //remove the current thread
    delete current_thread; //
    SYSTEM_SCHEDULER->yield(); // give up CPU to other thread

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

Figure 2.11. thread_shutdown() in “thread.C”.

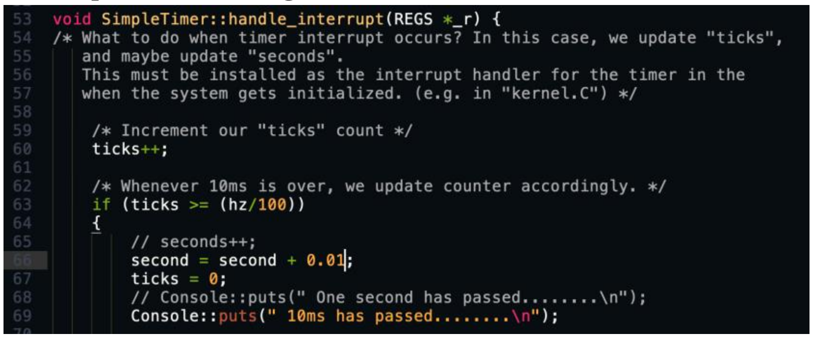
Before running the program, we want to add code to handle the interrupts correctly. We notice that interrupts are disabled when we created the threads. However, we need to re-enable interrupts at some point and let the periodic clock update message present again. In order to do this, we disable the interrupts at the start of “yield”, “resume” and “terminate” operations in the scheduler, and re-enable interrupts after we complete these operations, respectively. This can be seen in Fig 2.6, Fig 2.7, and Fig 2.9. Moreover, we also disable the interrupts at the beginning of the “thread_shutdown” and “setup_context” in “thread.C”.



```
void Thread::setup_context(Thread_Function _tfunction){

    if (Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
```

Figure 2.12. setup_context() in “thread.C”.

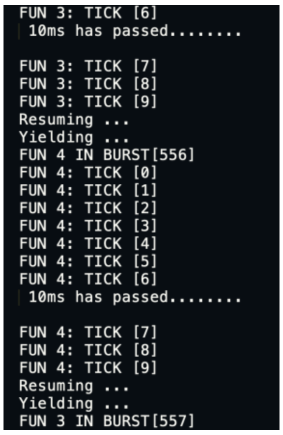
Furthermore, instead of showing clock message per second, we modify the “simple timer” and let the message show per 10ms. Since we set hz equal to 100 in the “kernel.C” (i.e., each tick is 10ms), we update the ticks whenever 10ms is over. The code implementation is presented in Fig 2.13.



```
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 10ms is over, we update counter accordingly. */
    if (ticks >= (hz/100))
    {
        // seconds++;
        second = second + 0.01;
        ticks = 0;
        // Console::puts(" One second has passed........\n");
        Console::puts(" 10ms has passed........\n");
```

Figure 2.13. Modification in “simple_timer.C”.

Then we start the program and see the clock message again in the console. Fig 2.14 presents a snapshot from the console. Since function 3 and function 4 are the two non-terminating threads, we can see that one thread give up the CPU to the other after every 10 steps. Moreover, we can see the clock message appear on the screen every 10ms. This means that we have enabled the interrupts.

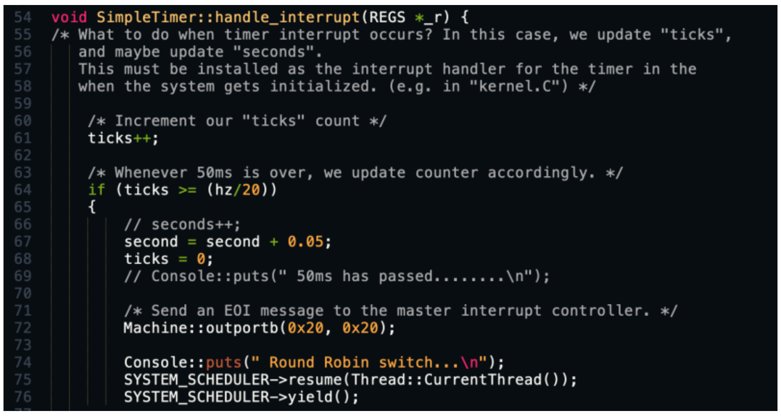


```
FUN 3: TICK [6]
 10ms has passed........

FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
Resuming ...
Yielding ...
FUN 4 IN BURST[556]
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]
 10ms has passed........

FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Resuming ...
Yielding ...
FUN 3 IN BURST[557]
```

Figure 2.14. A snapshot from the screen.

3. Round-Robin Scheduling

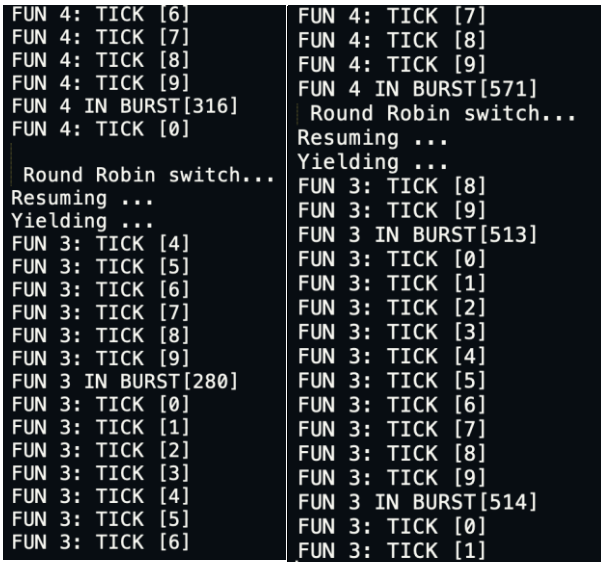
In this section, we develop a round-robin scheduler based on the FIFO scheduler. We do this by changing the interrupt handling code. The code modification in "simple_timer.C" is given in Fig 3.1. We need to implement a round-robin with a 50ms time quantum. We do this by changing condition where we need to update the "ticks". Since each tick represents 10ms given hz is 100, we reset the counter every 5 ticks. This means we handle the interrupt every 50ms. Then we make a thread switch every 50ms. First, we need to send an EOI message to the master interrupt controller to let it know we have handled the interrupt. We perform the thread switch by "resume" and "yield" operations that we have defined in the scheduler. Moreover, in the "kernel.C", we comment the "pass_on_cpu" function, because we switch threads based on the time quantum.



```
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 50ms is over, we update counter accordingly. */
    if (ticks >= (hz/20))
    {
        // seconds++;
        second = second + 0.05;
        ticks = 0;
        // Console::puts(" 50ms has passed........\n");

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

        Console::puts(" Round Robin switch...\n");
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
```

Figure 3.1. Modification in "simple_timer.C".

The two snapshots are presented in Fig 3.2. We can see that the thread is switching when the timer expires. Note that the two snapshots are not continuous.



```
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[316]
FUN 4: TICK [0]

 Round Robin switch...
Resuming ...
Yielding ...
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 3 IN BURST[280]
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 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[571]
 Round Robin switch...
Resuming ...
Yielding ...
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[513]
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 3 IN BURST[514]
FUN 3: TICK [0]
FUN 3: TICK [1]
```

Figure 3.2. Two snapshots from the console screen.