MP5 Design

Clarification: I finish the basic requirement and also Option1 and Option2.

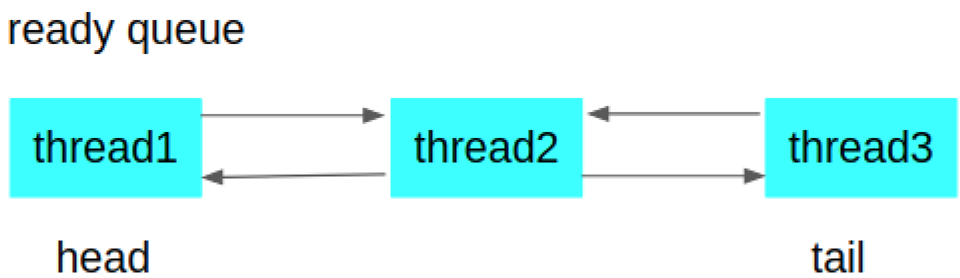
First, I design a FIFO scheduler for kernel level threads. To be more specific, I define a double linklist structure to support the ready queue, and also add head, tail and size attributes in class Scheduler. So with this ready queue, I can just pick up the first thread and put the switch-out thread at the tail in the ready queue when do FIFO scheduling. Also, when the thread returns, finishes, I will do the corresponding cleaning work and scheduling to other threads in the ready queue. Plus, I will also discuss a problem I encountered when the ready queue is empty and how to deal with it.

Second, for Option1, in order to support interrupt, I enable interrupts in thread_start, because the interrupt is disabled during initialization; and also enable interrupts / disable interrupts to protect the critical sections, like the add, resume, yield, terminate in Scheduler.

Third, for Option2, to implement RR Scheduling, based on the SimpleTimer, whenever 50 ms passes, I will schedule the next thread in the ready queue. Since I will perform context switching inside the SimpleTimer::handle_interrupt(), before this handle_interrupt() performs, I will send EOI message first.

Part 1 FIFO scheduler, basic requirement

(1)Define a ready queue in Scheduler, based on the double link list.




```
// Double linklist structure to support the ready queue
typedef struct Node {
      Thread *t;
      Node *prev;
      Node *next;
}Node;

class Scheduler {

   /* The scheduler may need private members... */
   Node *_head;
   Node *_tail;
   unsigned int _size;
```

(2)Main method implementation of Scheduler

Constructor:

```
Scheduler::Scheduler():_head(NULL),_tail(NULL),_size(0) {
  Console::puts("Constructed Scheduler.\n");
}
```

yield:

I use disble_interrupts and enable_interrupts to protect the Critical Section, notice this is a single-thread system.

If there are threads in the ready queue, pick the head and switch to it, FIFO scheduling.

Also update the ready queue structure.

```
void Scheduler::yield() {

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

    if (_size) {

        Thread *t = _head->t;
        Node *tmp = _head;
        if (_size == 1) {
            _head = NULL;
            _tail = NULL;
        }else {
            _head = _head->next;
            _head->prev = NULL;
        }
        --_size;
        delete tmp;
        Thread::dispatch_to(t);
        //Console::puts("inside yield() after dispatch_to\n");
    }

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

}
```

add:
Just add a new thread node at the tail of the ready queue.

```
void Scheduler::add(Thread * _thread) {
    assert(_thread != NULL);

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

    Node *n = new Node();
    if (!n) {
        Console::puts("new Node() failed
        return;
    }
    n->t = _thread;

    if (_size) {
        if (_size == 1) {
            n->prev = _head;
            n->next = NULL;
            _head->next = n;
            _tail = n;
        }else {
            n->next = NULL;
            n->prev = _tail;
            _tail->next = n;
            _tail = n;
        }
    }else {
```

```
    }else {
        n->prev = NULL;
        n->next = NULL;
        _head = n;
        _tail = n;
    }

    ++_size;

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

}
```

resume:

I just call add.

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

terminate:

If the to-terminate thread is in the ready queue, delete it from the queue.

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

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


    if (_thread && _size) {
        Node * current = _head;
        while (current) {
            if (current->t == _thread)  break;
            current = current->next;
        }

        if (current) {

            if (_size == 1) {
                delete current;
                _head = NULL;
                _tail = NULL;
            }else if (_size == 2) {
                if (current == _head) {
                    _head = _tail;
                    _head->prev = NULL;
                }else { // current == _tail
                    _tail = _head;
                    _head->next = NULL;
                }
```

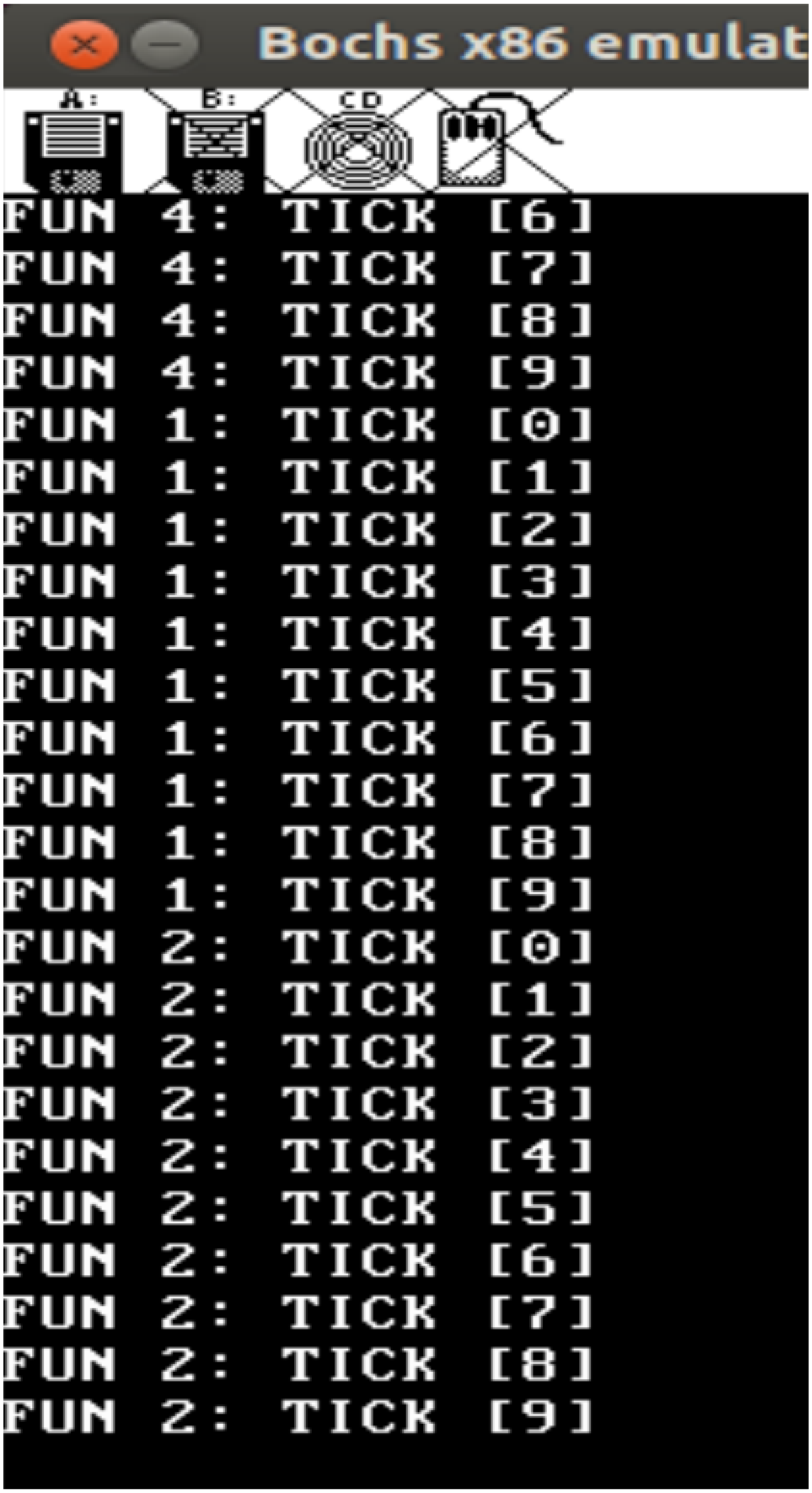
```
                    _head->next = NULL;
                }
                delete current;
            }else {
                if (current == _head) {
                    _head = _head->next;
                    _head->prev = NULL;
                }else if (current == _tail) {
                    _tail = _tail->prev;
                    _tail->next = NULL;
                }else {
                    current->prev->next = current->next;
                    current->next->prev = current->prev;
                }
                delete current;
            }
            --_size;
        } // end if (current)
    }

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

}
```

Thread::thread_shutdown

In order to deal with thread-return correctly, I call Scheduler::terminate(), delta the current_thread, and Scheduler::yield() in the Thread::thread_shutdown. Because thread_shutdown() will be called when the thread returns from the thread function, I do the corresponding cleaning work and also schedule the next thread here.

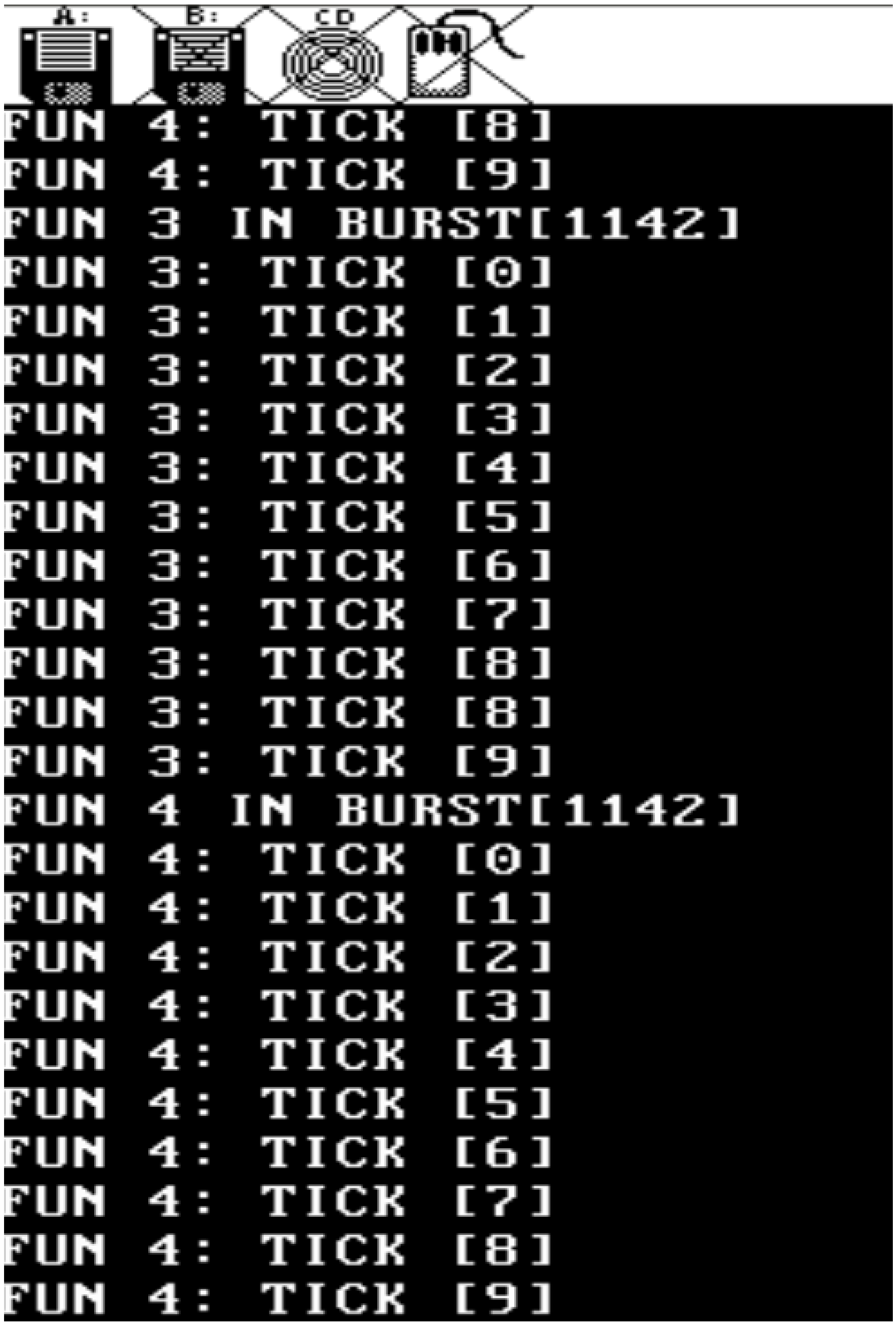
```
static void thread_shutdown() {
    /* This function should be called when the threa
     function.
        It terminates the thread by releasing memory
eld by the thread.
        This is a bit complicated because the thread
h the scheduler.
     */

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

    SYSTEM_SCHEDULER->terminate(current_thread);

    delete current_thread;

    SYSTEM_SCHEDULER->yield();
}
```

Testing for FIFO Scheduler, just the basic requirement of this MP.
(1)With #define _USES_SCHEDULER_
As shown below, Thread1 ~ Thread4 are running one by one in FIFO Scheduling. It is correct.

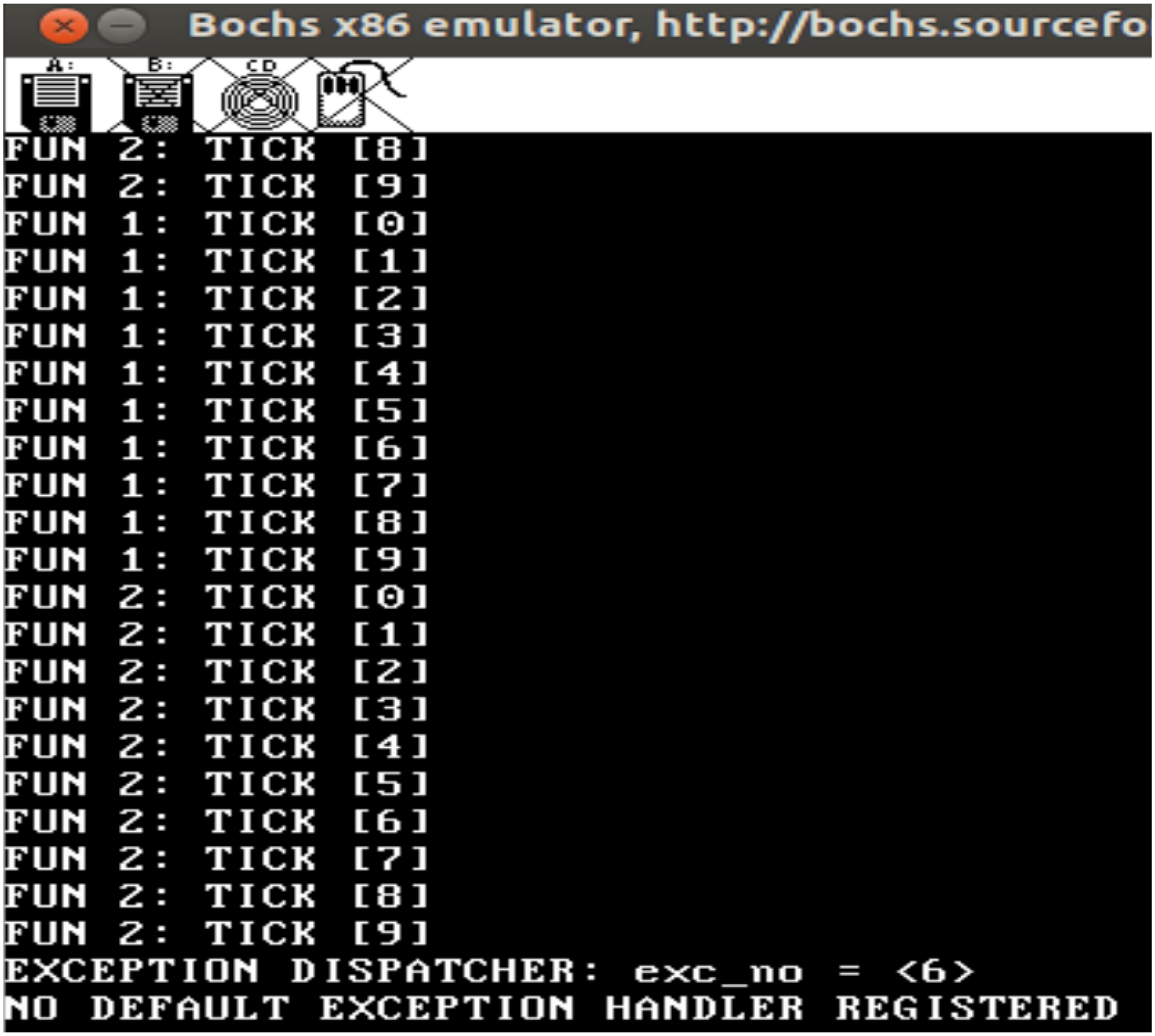


(2)With both #define _USES_SCHEDULER_ and #define _TERMINATING_FUNCTIONS_

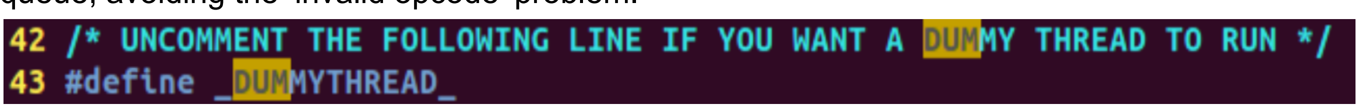
At first, Thread1 - Thread4 will run one by one just as in Testing case (1), but since Thread1 and Thread2 will terminate after 10 loops, Thread3 and Thread4 will run one by one later. This is also correct because Thread1 and Thread2 terminate successfully.



A problem to discuss: What happens when the ready queue is empty?
Without infinite thread, the ready queue of the Scheduler will be empty, the system output will be like this:



This is expected, because no thread any more in the ready queue after finishing the current thread, so it is an invalid opcode for the system. In order to prevent the OS from this exception when there is no infinite threads, we can add a dummy thread to run infinitely in the kernel.C, since there will be always at least one thread in the ready queue, avoiding the ‘invalid opcode’ problem.

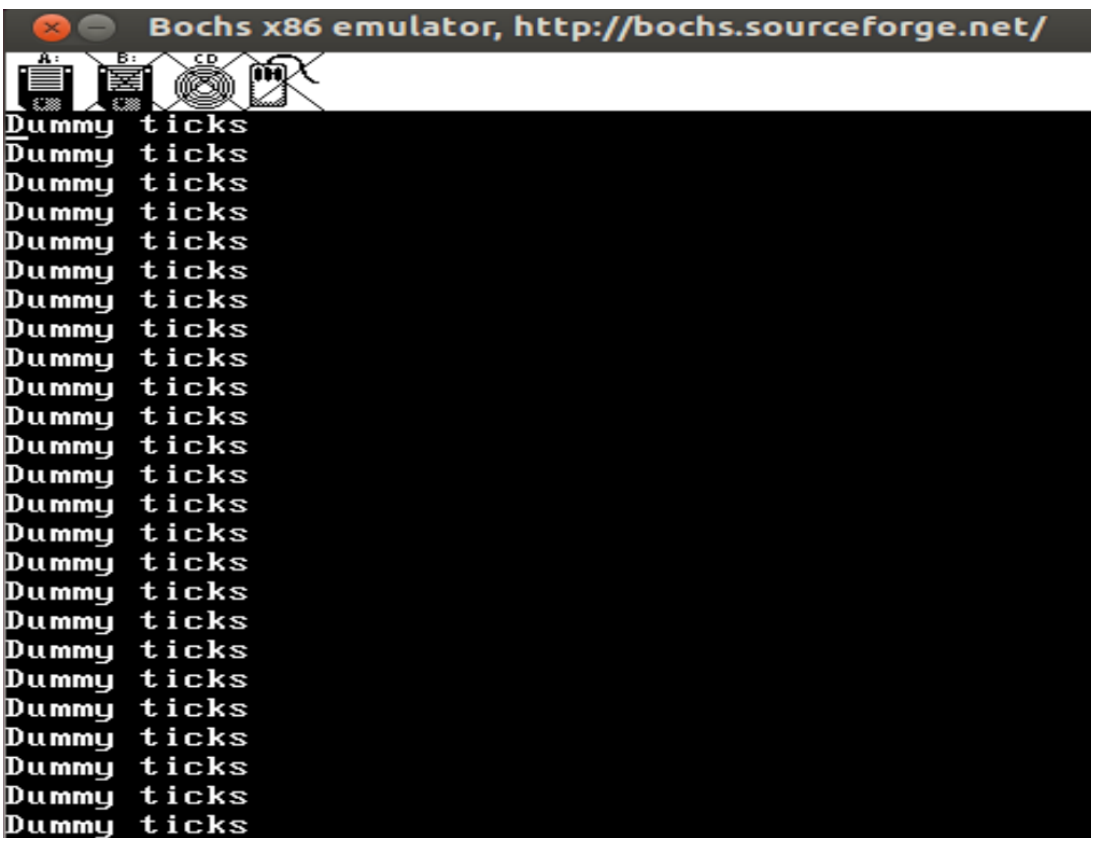


```
#ifdef _DUMMYTHREAD_
Thread * thread_dummy;
#endif
```

```
#ifdef _DUMMYTHREAD_
void fun_dummy() {
    Console::puts("Thread Dummy\n");
    for(;;) {
        Console::puts("Dummy ticks\n");
    }
}
#endif
```

```
#ifdef _DUMMYTHREAD_
    Console::puts("CREATING THREAD_DUMMY ...");
    char * stack_dummy = new char[1024];
    thread_dummy = new Thread(fun_dummy, stack_dummy, 1024);
    Console::puts("Done\n");
#endif
```

```
#ifdef _DUMMYTHREAD_
    SYSTEM_SCHEDULER->add(thread_dummy);
#endif
```

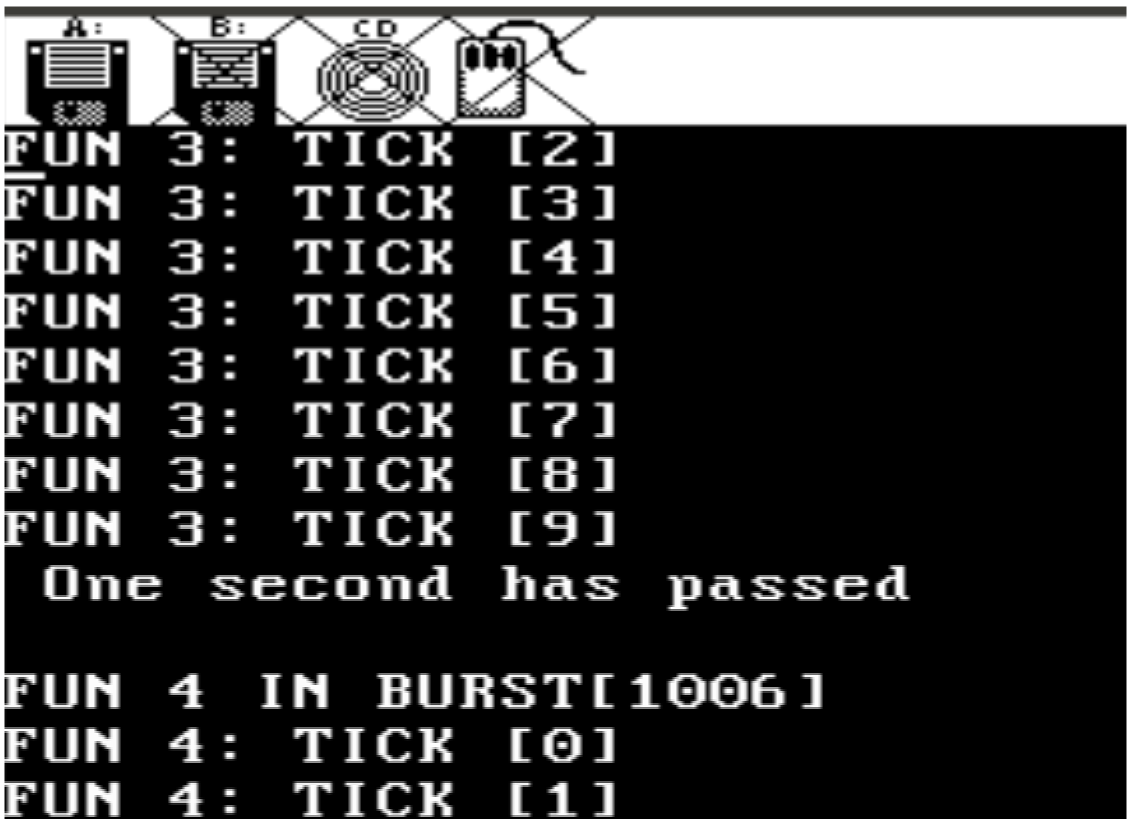


Part 2 Interrupt support, Option 1

In order to support interrupts, first I enable interrupts inside Thread::thread_start, since during the thread initialization, thread's interrupts will be disabled. I also disable interrupts before entering Critical Section, like adding the ready queue in Scheduler::run, dequeue in Scheduler::yield(), and enable interrupts after finishing the Critical Section.

Testing

As shown below, the timer interrupts will still occur. It is correct.



Part 3 RR Scheduling, Option 2

I support both RR Scheduling and voluntarily yielding CPU here. In order to implement 50ms quantum time of RR Scheduling, I do scheduling inside SimpleTimer::handle_interrupt() when 50ms passes.

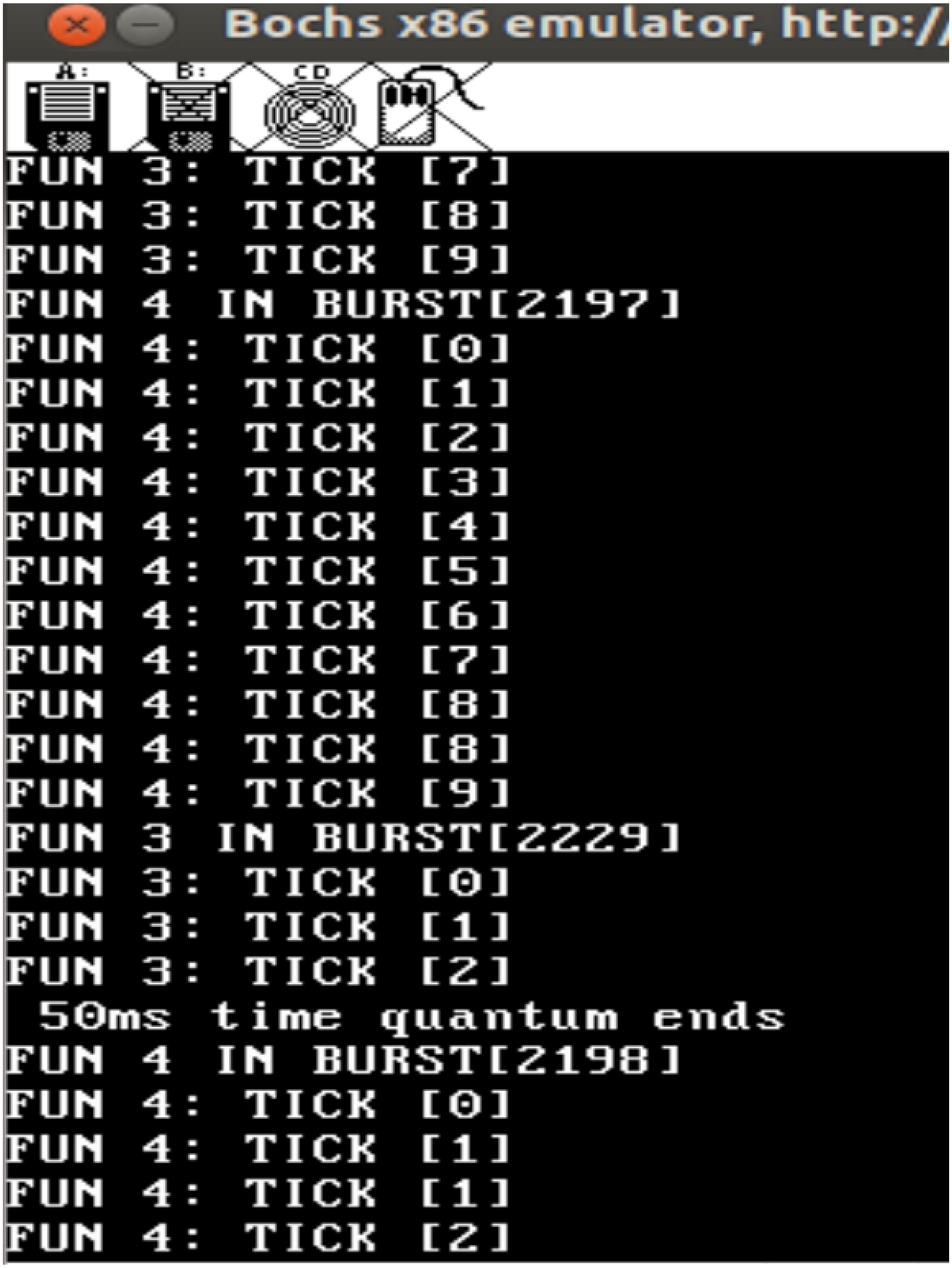
```
void SimpleTimer::handle_interrupt(REGS *_r) {
/* What to do when timer interrupt occurs? In this case, we update "ticks",
   and maybe update "seconds".
   This must be installed as the interrupt handler for the timer in the
   when the system gets initialized. (e.g. in "kernel.C") */

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

    /* Whenever ticks 5 times, 50 ms passed */
    if (ticks%5 == 0) {
        Console::puts(" 50ms time quantum ends\n");
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
    }


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

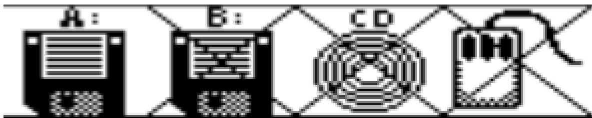
Since the thread will yield CPU inside the Timer interrupt handler, we need to send EOI message before performing the scheduling in InterruptHandler::dispatch_interrupt().

```
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 {
    /* 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);

  }

}
```

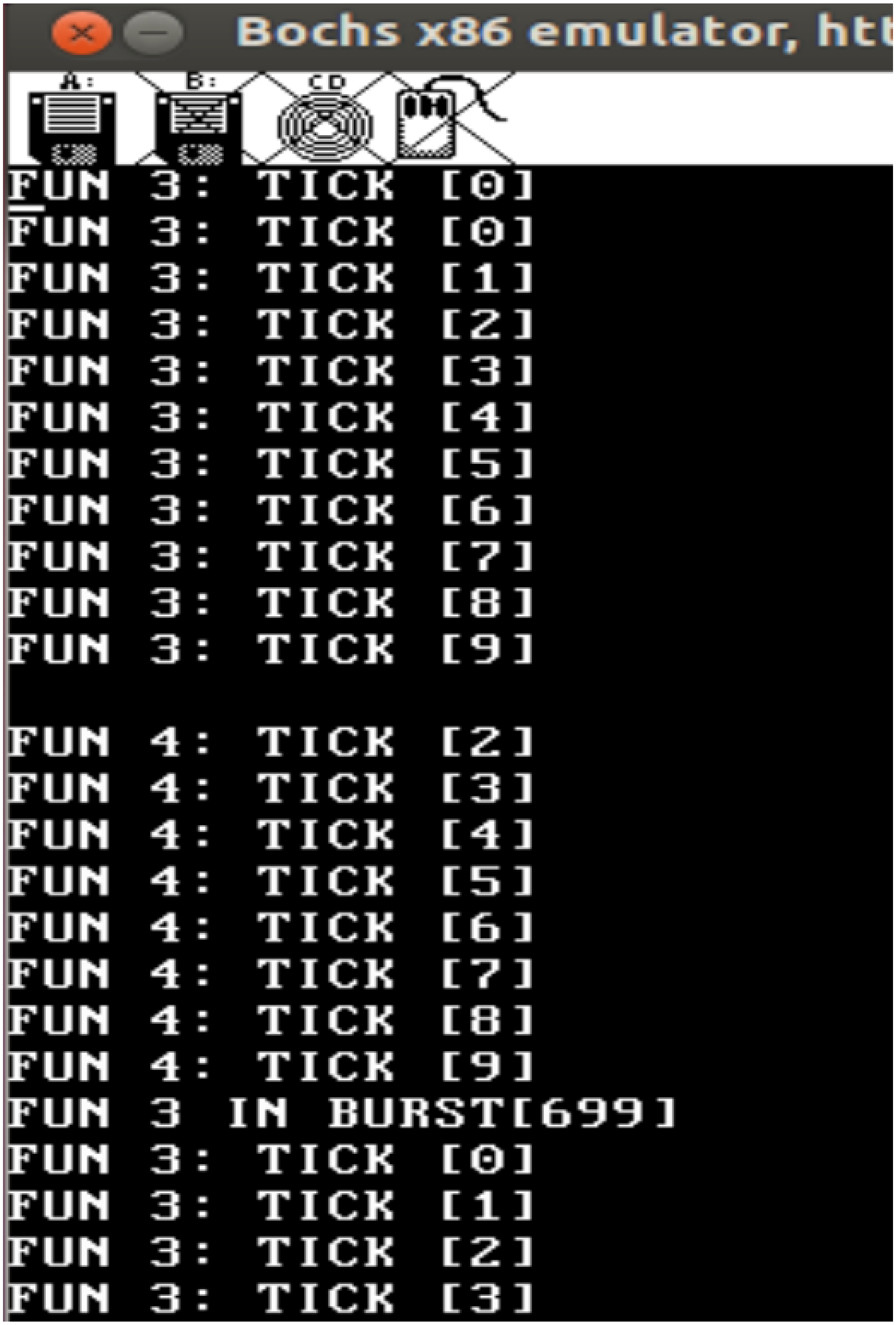
Testing RR Scheduling

As shown in the 3 screenshots below, my OS support RR Scheduling and also voluntarily yielding CPU.






```
FUN 4: TICK [0]
 50ms time quantum ends
FUN 3 IN BURST[2218]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
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: 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 [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[2219]
FUN 3: TICK [0]
```


Bochs x86 emulator, htt
A:
B:
CD
FUN 3: TICK [0]
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: 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[699]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]