CSCE 611: Operating Systems
MP4: Design Document

Shweta Sharma
UIN: 433003780

Design Idea:

Kernel-Level Thread Scheduling – MP5:

The following have been implemented in the project:

- FIFO Scheduler
- Terminating of threads
- Option 1: Interrupt management
- Option 2: Round-Robin Scheduling using End-of-Quantum Timer

FIFO Scheduler:

A pointer-based list with the front and tail pointers is used to implement the ready queue. The queue is initially empty with its front and tail pointer pointing to null. A variable thread_count is used to maintain the number of threads in the system.

The Scheduler provides the following functionality:

- Scheduler() - Constructs the Scheduler. The initial thread_count is zero and both the front of the queue and tail of the queue points to null.

```
Scheduler::Scheduler() {
  //adding dummy node to the ready queue
  Scheduler::thread_count = 0;
  Scheduler::ready_queue_front = Scheduler::ready_queue_tail = NULL;
  Console::puts("Constructed Scheduler.\n");
}
```

- add(thread) – Adds the thread to the end of the ready queue after the tail node. If the front and tail pointer are null, they now point to the new thread added to the queue. We disable interrupts before entering the critical section. In the critical section we add the thread_node to the scheduler queue. After adding the thread_node to the queue, we increment thread_count by 1.

```
void Scheduler::add(Thread * _thread) {
  Node* new_ready_thread_node = new Node;
  new_ready_thread_node -> _thread = _thread;
  new_ready_thread_node -> _next_node = NULL;

  bool interrupts_enabled = Machine::interrupts_enabled();
  Console::puts("Initial Interrupt State: ");
  Console::puti(interrupts_enabled);
  Console::puts("\n");

  if(interrupts_enabled){
    Machine::disable_interrupts();
    Console::puts("Disabled Interrupts Before Critical Section.\n");
  }

  if(Scheduler::ready_queue_front==NULL){
    //only dummy node in ready queue
    Scheduler::ready_queue_front = new_ready_thread_node;
    Scheduler::ready_queue_tail = new_ready_thread_node;
  }else{
    Scheduler::ready_queue_tail -> _next_node = new_ready_thread_node;
    Scheduler::ready_queue_tail = new_ready_thread_node;
  }
  Scheduler::thread_count++;

  if(interrupts_enabled){
    Machine::enable_interrupts();
    Console::puts("Enabled Interrupts After critical section.\n");
  }

  //Scheduler::printReadyQueue();
  Console::puts("Added Thread To Ready Queue.\n");
}
```

- resume(thread) – Adds thread to the end of the queue using add(thread) functionality.

```
74
75  void Scheduler::resume(Thread * _thread) {
76      Scheduler::add(_thread);
77  }
78
```

- yield() – Scheduler passes the control to the thread at the front of the ready queue. It throws an assertion error if the queue is empty. If the ready queue only contains one thread, the front and tail nodes are made to point to null.

```
56  void Scheduler::yield() {
57      if(Scheduler::ready_queue_front==NULL){
58        Console::puts("Ready Queue Is Empty.\n");
59        assert(false);
60      }
61
62      //adding current thread to end of ready queue not needed, as current thread is premempted by kernal before yielding.
63
64      //dispatch to next node
65      Node* next_ready_thread = Scheduler::ready_queue_front;
66      if(Scheduler::ready_queue_front == Scheduler::ready_queue_tail){
67        Scheduler::ready_queue_tail = NULL;
68      }
69      Scheduler::ready_queue_front = Scheduler::ready_queue_front -> _next_node;
70
71      Console::puts("Yielding To Next Thread In Ready Queue.\n");
72      Thread::dispatch_to(next_ready_thread -> _thread);
73  }
```

- terminate(thread) – Terminates the thread by removing it from the ready queue and yields to the next thread in queue.We disable interrupts before entering the Critical Section. In the critical section, we traverse the queue to see where the thread is and delete it from the list. We then decrement the number of threads by 1. Post this critical section, we re-enable the interrupts. After this, we yield to the next thread using the yield() function defined above.

```
113 void Scheduler::terminate(Thread * _thread) {
114   //remove the _thread from the ready queue, in case the thread sucides, before thread shutdown.
115
116   if(Scheduler::ready_queue_front==NULL){
117       return;
118     }
119
120   bool interrupts_enabled = Machine::interrupts_enabled();
121   //Console::puts("Initial Interrupt State: ");Console::puti(interrupts_enabled);Console::puts("\n");
122   if(interrupts_enabled){
123     Console::puts("Disabled Interrupts Before Critical Section.\n");
124     Machine::disable_interrupts();
125   }
126
127   Node* node = Scheduler::ready_queue_front;
128
129   if(Scheduler::ready_queue_front -> _thread -> ThreadId() == _thread -> ThreadId()){
130     Scheduler::ready_queue_front = Scheduler::ready_queue_front -> _next_node;
131     delete node;
132   }
133   else {
134     while(node->_next_node!=NULL){
135       Node* nextNode = node -> _next_node;
136       if(nextNode -> _thread -> ThreadId() == _thread -> ThreadId()){
137         node -> _next_node = nextNode -> _next_node;
138         delete nextNode;
139         break;
140       }
141       node = nextNode;
142     }
143   }
144
145   if(interrupts_enabled){
146     Machine::enable_interrupts();
147     Console::puts("Enabled Interrupts After Critical Section.\n");
148   }
149
150   Scheduler::thread_count--;
151   Console::puts("Thread Terminated.\n");
152   Scheduler::yield();
153 }
154
```

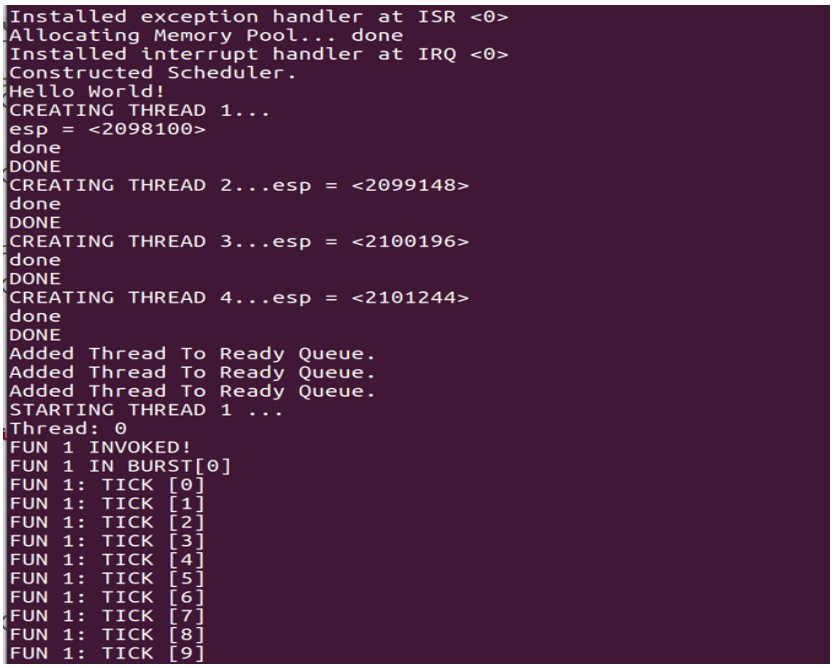
Termination of threads:

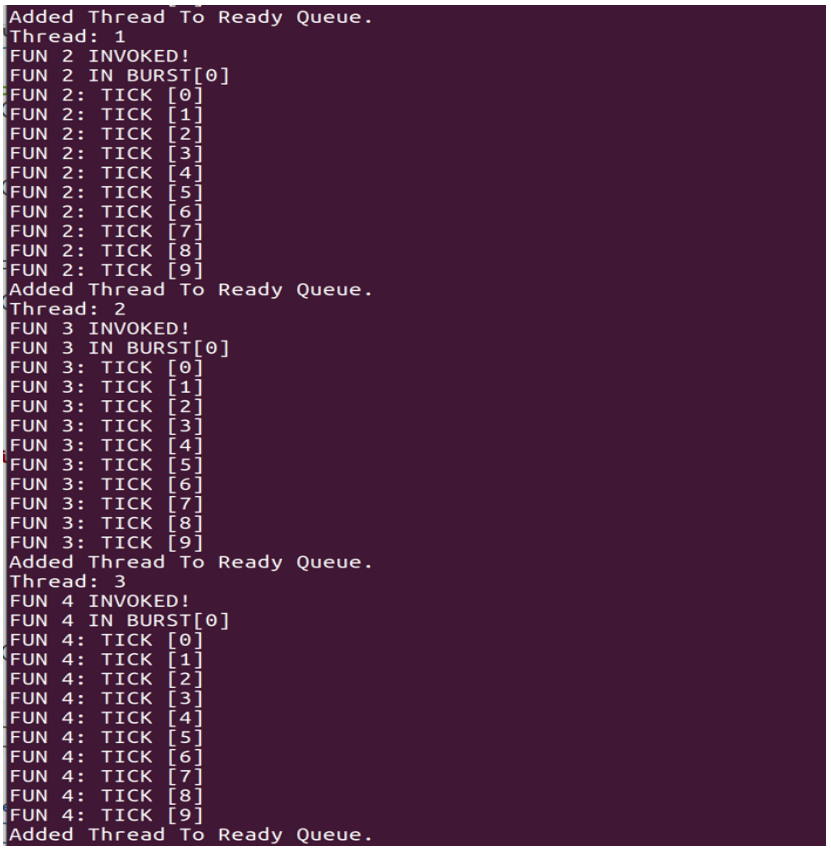
Threads are terminated by thread_shutdown api in the thread class. The threads may be terminated after task completion or on suiciding. To handle this, the thread is removed from the ready queue by the scheduler before deleting the thread.

```
74  static void thread_shutdown() {
75      /* This function should be called when the thread returns from the thread function.
76         It terminates the thread by releasing memory and any other resources held by the thread.
77         This is a bit complicated because the thread termination interacts with the scheduler.
78       */
79
80      //assert(false);
81      /* Let's not worry about it for now.
82         This means that we should have non-terminating thread functions.
83      */
84
85      // Terminate and remove address space of current thread
86      SYSTEM_SCHEDULER -> terminate(Thread::CurrentThread());
87      delete current_thread;
88  }
```

Test Screenshots / Output – FIFO Scheduler with terminating threads:

As seen below, every thread completes 10 ticks in a burst before passing the CPU to the next thread in the queue. Also, threads 0 and 1 run for bursts 0 to 9 each, while threads 2 and 3 run infinitely, unless the operating system is switched off at bursts 51 and 50 respectively.





```
FUN 1 IN BURST[9]
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]
Added Thread To Ready Queue.
FUN 2 IN BURST[9]
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]
Added Thread To Ready Queue.
FUN 3 IN BURST[9]
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]
Added Thread To Ready Queue.
FUN 4 IN BURST[9]
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]
```

```
Added Thread To Ready Queue.
FUN 4 IN BURST[9]
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]
Added Thread To Ready Queue.
FUN 3 IN BURST[10]
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]
Added Thread To Ready Queue.
FUN 4 IN BURST[10]
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]
Added Thread To Ready Queue.
FUN 3 IN BURST[11]
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]
```

```
Added Thread To Ready Queue.
FUN 3 IN BURST[11]
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]
Added Thread To Ready Queue.
FUN 4 IN BURST[11]
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]
Added Thread To Ready Queue.
FUN 3 IN BURST[12]
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]
Added Thread To Ready Queue.
FUN 4 IN BURST[12]
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]
```

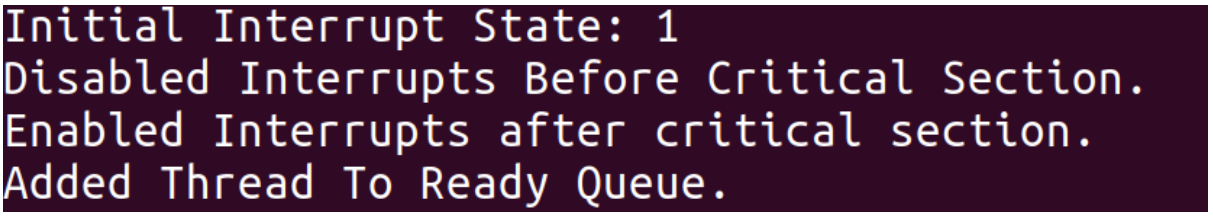
```
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
Added Thread To Ready Queue.
FUN 4 IN BURST[50]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Added Thread To Ready Queue.
FUN 3 IN BURST[51]
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]
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
```

Option 1 - Interrupt management:

- Interrupt is enabled on the starting thread. This is also written to the EFLAGS while populating the thread stack by checking if the interrupt is enabled. (0=interrupt disabled, 1=interrupt enabled).
- Further, to ensure mutual exclusion, the interrupt is disabled and enabled before and after critical sections of manipulating the ready queue.

Test Screenshots / Output:

The below log shows clearly that the interrupt is enabled at thread start and disabled in the critical section. Once the critical section is over, the thread is enabled again.



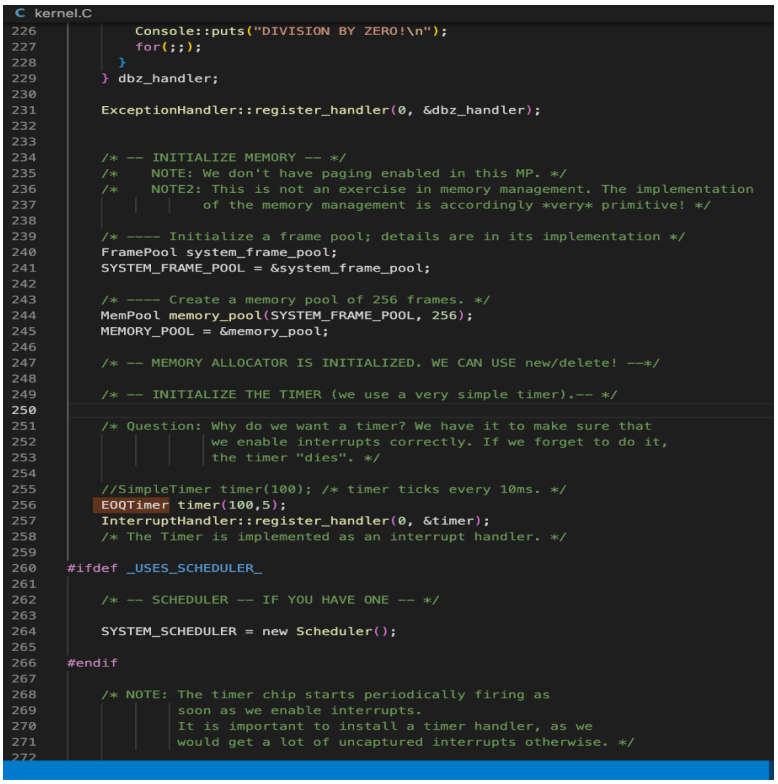
Option 2 – Round-Robin Scheduling using End-of-Quantum Timer:

- End-of-Quantum Timer (EOQTimer) is implemented as a derived class of the SimpleTimer class. Upon completion of the quantum time (50ms / 5 clock ticks), the scheduler adds the current thread to the tail of the ready queue using the resume function. It also yields the control from the current thread to the next thread at the front of the queue.
- When dispatching an interrupt, to let the interrupt controller (PIC) know that the interrupt has been handled, an EOI signal is sent and then the interrupt is handled by yielding to the next thread in the ready queue.
- To implement Round Robin scheduling, the kernel is updated to use the EOQTimer instead of the SimpleTimer. Also, interrupt is enabled after the start of the first thread, instead of at the start of the operating system.
- Further the makefile is updated to compile the new EOQTimer files created.

Handling interrupt in eoq_timer.C :

```
38  void EOQTimer::handle_interrupt(REGS *_r) {
39  /* What to do when timer interrupt occurs? In this case, we update "ticks",
40     and maybe update "seconds".
41     This must be installed as the interrupt handler for the timer in the
42     when the system gets initialized. (e.g. in "kernel.C") */
43
44      /* Increment our "ticks" count */
45      ticks++;
46
47      /* Call an interrupt every  after _quantum quantums have passed. */
48      if (ticks >= hz/20)//quantum )
49      {
50          //seconds++;
51          ticks = 0;
52          Console::puts("\nOne time quantum has passed, yielding to next thread\n");
53          Thread::yield_thread();
54      }
55  }
```

Calling EOQTimer in kernel.C:



```
C kernel.C
226         Console::puts("DIVISION BY ZERO!\n");
227         for(;;);
228       }
229     } dbz_handler;
230
231     ExceptionHandler::register_handler(0, &dbz_handler);
232
233
234     /* -- INITIALIZE MEMORY -- */
235     /*    NOTE: We don't have paging enabled in this MP. */
236     /*    NOTE2: This is not an exercise in memory management. The implementation
237                  of the memory management is accordingly *very* primitive! */
238
239     /* ---- Initialize a frame pool; details are in its implementation */
240     FramePool system_frame_pool;
241     SYSTEM_FRAME_POOL = &system_frame_pool;
242
243     /* ---- Create a memory pool of 256 frames. */
244     MemPool memory_pool(SYSTEM_FRAME_POOL, 256);
245     MEMORY_POOL = &memory_pool;
246
247     /* -- MEMORY ALLOCATOR IS INITIALIZED. WE CAN USE new/delete! --*/
248
249     /* -- INITIALIZE THE TIMER (we use a very simple timer).-- */
250
251     /* Question: Why do we want a timer? We have it to make sure that
252                  we enable interrupts correctly. If we forget to do it,
253                  the timer "dies". */
254
255     //SimpleTimer timer(100); /* timer ticks every 10ms. */
256     EOQTimer timer(100,5);
257     InterruptHandler::register_handler(0, &timer);
258     /* The Timer is implemented as an interrupt handler. */
259
260   #ifdef _USES_SCHEDULER_
261
262     /* -- SCHEDULER -- IF YOU HAVE ONE -- */
263
264     SYSTEM_SCHEDULER = new Scheduler();
265
266   #endif
267
268     /* NOTE: The timer chip starts periodically firing as
269              soon as we enable interrupts.
270              It is important to install a timer handler, as we
271              would get a lot of uncaptured interrupts otherwise. */
272
```

Test Screenshots / Output:

As seen in the logs below:

- Every thread runs for the set quantum, before yielding to the next thread. Hence, instead of completing a burst (10 ticks) before giving up the CPU, they perform a different number of ticks in each quantum. Thread 0, 1, 2 and 3 perform 3, 4, 2, and 3 ticks of burst 0 in their first time-quantum and then give up the CPU.
- In the next quantum, these threads resume from where they left off, i.e., ticks 4, 5, 3, and 4 of bursts 0 respectively.
- Threads 0 and 1 terminate after completing 10 bursts at different time-quantums.
- Threads 2 and 3 are non-terminating threads and continue until the system is switched off.
- The non-terminating threads completed different amounts of bursts, when the system was turned off. Specifically, threads 2 and 3 are in bursts 34 and 37 respectively.

```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
SimpleTimer constructed
EOQTimer constructed/derived from  SimpleTimer
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098100>
done
DONE
CREATING THREAD 2...esp = <2099148>
done
DONE
CREATING THREAD 3...esp = <2100196>
done
DONE
CREATING THREAD 4...esp = <2101244>
done
DONE
Initial Interrupt State: 0
Added Thread To Ready Queue.
Initial Interrupt State: 0
Added Thread To Ready Queue.
Initial Interrupt State: 0
Added Thread To Ready Queue.
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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
Thread: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
Thread: 3
FUN 4 INVOKED!
FUN 4 IN BURST[0]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 1 IN BURST[1]
FUN 1: TICK [0]
FUN 1: TICK [
One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
```

```
FUN 2: TICK [9]
FUN 2 IN BURST[1]
FUN 2: TICK
One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
 [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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [9]
FUN 3 IN BURST[1]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4 IN BURST[1]
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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 1 IN BURST[2]
FUN 1: TICK [0]
FUN 1: TICK [1]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 2: TICK [7]
FUN 2: TICK [8]
```

```
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 1 IN BURST[9]
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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Disabled Interrupts Before Critical Section.
Enabled Interrupts After Critical Section.

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[10]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 1: TICK [9]
Disabled Interrupts Before Critical Section.
Enabled Interrupts After Critical Section.
Thread Terminated.
Yielding To Next Thread In Ready Queue.

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3 IN BURST[9]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[11]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4
One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
Thread Terminated.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [6]
```

```
One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
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
One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
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 4 IN BURST[36]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
3 IN BURST[33]
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]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[34]
FUN 3: TICK [0]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[37]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]

One time quantum has passed, yielding to next thread
Initial Interrupt State: 0
Added Thread To Ready Queue.
Yielding To Next Thread In Ready Queue.
========================================================================
Bochs is exiting with the following message:
[XGUI  ] POWER button turned off.
========================================================================
```

Documents changed:

I have changed the following files:

- **kernel.C** – as per instructions in the handout for the compulsory parts, options 1 & 2
- **scheduler.C** – as per instructions in the handout for the compulsory parts & option 1
- **scheduler.H** – as per instructions in the handout for the compulsory parts & option 1
- **thread.C** – as per instructions in the handout for the compulsory parts, options 1 & 2
- **eoq_timer.C (new file)** – as per instructions in Option 2
- **eoq_timer.H (new file)** – as per instructions in Option 2
- **interrupts.C** – as per instructions in Option 2
- **makefile** – updated to compile eoq_timer