MP5: Kernel-Level Thread Scheduling

Vinay Balamurali
UIN: 936002032
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus 1: Completed.
Bonus 2: Completed.
Bonus 3: Did not Attempt.

System Design

This machine problem has multiple aims but the crux is to implement a Kernel-level scheduler to manage the execution of threads on the CPU.

FIFO scheduler

Initially, we a implement a First-In-First-Out Scheduler (FIFO), wherein, the kernel maintains a Ready Queue of threads to be dispatched. The thread at the head of the queue is dispatched to the CPU when the currently executing thread yields execution. Before the the running thread yields execution, it also adds itself back to the end of the Ready Queue.

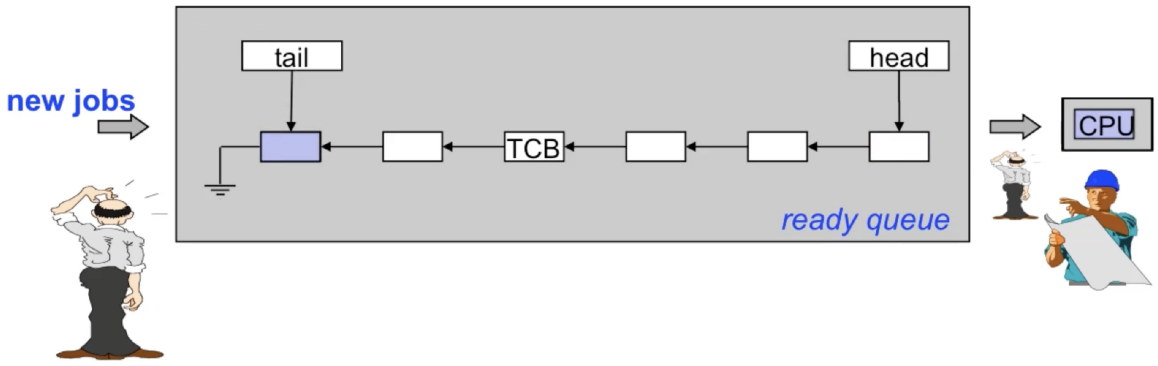

Figure 1: FIFO Scheduling Mechanism (source: CSCE 611 lectures)

Terminating threads

The thread functions in `kernel.C` are non-terminating by default. After some point in the execution, they pass control to the next thread by yielding the CPU and adding itself back to the ready queue. By defining the macro `_TERMINATING_FUNCTIONS_`, threads can complete their execution which means that they need to be terminated and cleaned up appropriately. This is done by modifying the thread management code such that when the thread returns, the thread is removed from the Ready queue and its context deleted. Post that, we yield the CPU to ensure the next thread gets executed.

Bonus 1: Interrupt Handling

The scheduler needs to enforce mutual exclusion when accessing the Ready Queue. Hence, methods in the class `Scheduler` always disable interrupts at the start of any operation, and re-enable them before returning from the method. Interrupts are also required to be enabled when a thread is created and it's context is pushed on the stack.

Bonus 2: Round Robin Scheduling

A Round Robin Scheduler has been implemented with a 50ms time quantum. This means that we need to set a frequency of 20Hz. The design uses the FIFO queue except that whenever the time quantum is hit, the current threads yeilds CPU to the head of queue and adds itself back to the end of queue. Determination of the time quantum, is done by implementing a class called `EOQTimer`. This class derives from class `SimpleTimer` and overides the interrupt handler appropriately to ensure that the preemption works correctly. Details are given in the implementation section.

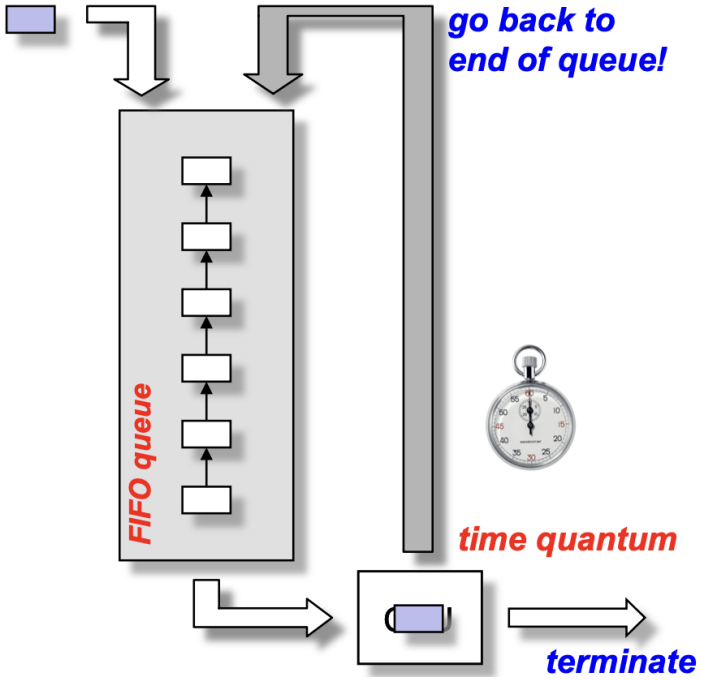

Figure 2: Round Robin Scheduling Mechanism (source: CSCE 611 lectures)

Code Description

0.1 Files Added/Modified

- scheduler.H
- scheduler.C
- thread.C
- simple_timer.H
- EOQTimer.H
- EOQTimer.C
- kernel.C
- makefile

scheduler.H : This header file provides method declarations for the class FIFO scheduler as well as the Round Robin Scheduler. Additionally, it provides an implementation for a Queue Data structure as a linked list which holds elements having the pointer to a thread.

Implementation of Class `Node`:

- `Thread *threadPtr` : Pointer to the thread to be executed.
- `Node *next` : Pointer to the next node in the linked list.

```
// Data structure to hold a Node to be inserted in the ready queue.
// The data is a pointer to the thread to be executed.
// Has a self pointer to the next node in the linked list.
class Node
{
public:

   Thread *threadPtr;  // Pointer to the 'to be executed' thread.
   Node *next;  // Pointer to next node.

public:

   // Constructor.
   Node(Thread *_thread)
   {
      this->threadPtr = _thread;
      this->next = nullptr;
   }

};
```

Figure 3: Class `Node`

The class `schedulerQueue` implements the Ready Queue and provides methods to add and remove items. Implementation of Class `schedulerQueue`:

- `Node *head` : Pointer to head of queue.
- `Node *tail` : Pointer to tail of queue.
- `unsigned int length` : Number of elements in the queue.
- `schedulerQueue(void)` : Constructor to initialize pointers to null and number of elements to 0.
- `void enqueue(Thread *_thread)` : Method to add an element at the tail of the queue.
- `Thread* dequeue(void)` : Method to remove an element from the beginning of the queue.
- `void removeThreadById(Thread* _thread)` : A special method to remove an element based on the thread ID.
- `unsigned int fetchSize(void)` : Method to fetch the size of the queue.

```
// Class to implement the ready queue.
class schedulerQueue
{

private:
   Node *head;  // 'Head' or 'front' of queue.
   Node *tail;  // 'Tail' or 'rear' of queue.
   unsigned int length;  // size of queue.

public:

   // Constructor.
   schedulerQueue(void)
   {
      head = nullptr;
      tail = nullptr;
      length = 0;
   }


   // Method to add an Thread to the ready queue
   // at the tail.
   void enqueue(Thread *_thread)
   {
      Node *newNode = new Node(_thread);
      if (tail == nullptr)
      {
         tail = newNode;
         head = tail;
         length = 1;
         return;
      }
```

Figure 4: Class `schedulerQueue`

The following items have been modified in class `Scheduler`:

- `schedulerQueue *readyQueue` : protected member to hold a reference to the Ready Queue. This pointer has been made protected so that derived class can access the member variable directly.

```
class Scheduler {

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

  schedulerQueue *readyQueue = nullptr;

public:
```

Figure 5: Class `Scheduler`

A new class `RoundRobinScheduler` has been introduced to implement the Round Robin Scheduler. It derives from class `scheduler`. Only one method is needed to be overriden. For the others we re-use the base class method. Members:

- `RoundRobinScheduler(void)` : Default constructor which calls the base class constructor.
- `virtual void yield(void) override` : Method to intimate Master Interrupt Controller and yield execution by calling the base class method.

```
// Class to implement the Round Robin Scheduling mechanism.
class RoundRobinScheduler: public Scheduler
{
public:
   // Constructor.
   RoundRobinScheduler(void);

   // Method to signal EOI to the Master Interrupt Controller
   // and call base class yield.
   virtual void yield(void) override;
};
```

Figure 6: Class `RoundRobinScheduler`

scheduler.C: Scheduler::Scheduler() : This is the constructor of this class. The Ready Queue is initialized within this method.

```
// Constructor.
Scheduler::Scheduler(void)
{
    readyQueue = new schedulerQueue();

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

} // Scheduler::Scheduler
```

Figure 7: Constructor of class `Scheduler`

scheduler.C: Scheduler::yield(void) : This method first disables all interrupts and pops the element from the head of the Ready Queue. It then utilizes the thread management library to dispatch to the popped thread. It enables all interrupts before returning.

```
// Method to start executing the thread at the
// head of the ready queue.
void Scheduler::yield(void)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    if (readyQueue->fetchSize() == 0)
    {
        Console::puts("No thread present in the Ready Queue to yield to!\n");
        Machine::enable_interrupts();
        return;
    }

    // Fetch current thread to be dispatched.
    Thread *readyThread = readyQueue->dequeue();

    Machine::enable_interrupts();

    // Dispatch to new thread.
    Thread::dispatch_to(readyThread);

} // Scheduler::yield
```

Figure 8: `yield` method of class `Scheduler`

scheduler.C: Scheduler::add(Thread * _thread) : This method first disables all interrupts and adds a new thread to be executed at the tail of the ready queue. It enables all interrupts before returning.

scheduler.C: Scheduler::resume(Thread * _thread) : This method is the same as `add` for all purposes. Hence, we just call add internally from here. It is primarily meant for those threads waiting on events. This method is called when a thread needs to yield execution. But before that it needs to be added back the end of the Ready Queue, for which it calls this method.

```
// Method to add a thread back to the ready queue
// when it was currently executing.
void Scheduler::resume(Thread * _thread)
{
    // Just call 'add' again.
    add(_thread);

} // Scheduler::resume


// Method to add a thread to the ready Queue
// at the tail.
void Scheduler::add(Thread * _thread)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    readyQueue->enqueue(_thread);

    Machine::enable_interrupts();

} // Scheduler::add
```

Figure 9: `add` and `resume` methods of class `Scheduler`

scheduler.C: Scheduler::terminate(Thread * _thread) : This method is called when a terminating thread is required to release its resources. But the thread could have been added back to the ready queue and a search is required to be performed to delete it. This necessitates traversing the entire linked list and finding a match by Thread id.

```
// Method to terminate a given thread once it is done
// executing and remove it from the ready queue.
void Scheduler::terminate(Thread * _thread)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Special method to remove a thread based on Thread ID.
    readyQueue->removeThreadById(_thread);

    Machine::enable_interrupts();
}
```

Figure 10: `terminate` method of class `Scheduler`

scheduler.C: RoundRobinScheduler::yield(void) : This method first signals the Master Interrupt Controller with an End of Message (EOI) signal. Post that, it calls the base class `yield`. There is no other change to this class. Maxiumum effort has been made to re-use code.

```
/*--------------------------------------------------------------------------*/
/* METHODS FOR CLASS   RoundRobinScheduler  */
/*--------------------------------------------------------------------------*/


// Constructor.
RoundRobinScheduler::RoundRobinScheduler() : Scheduler() {};


// Method to signal EOI to the Master Interrupt Controller
// and call base class yield.
void RoundRobinScheduler::yield(void)
{
    /* Send an EOI message to the master interrupt controller. */
    Machine::outportb(0x20, 0x20);

    Scheduler::yield();

} //  RoundRobinScheduler::yield
```

Figure 11: class `RoundRobinScheduler` implementation

thread.C: Thread::thread_start() : This method enables all interrupts as part of the thread management library.

thread.C: Thread::thread_shutdown() : This method is called when a thread returns from a thread function. It releases all resources held by it by calling the method `terminate` and yields CPU to the head of queue.

```
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.
    */

   // Terminate current thread, delete its context and
   // yield CPU to the thread at the head of ready queue.
   SYSTEM_SCHEDULER->terminate(current_thread);
   delete current_thread;
   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. */
    Machine::enable_interrupts();
}
```

Figure 12: `start` and `shutdown` methods of the Thread Management Library

simple_Timer.H : The private members have been made protected so that derived classes can access the data structures. This is to prevent duplication of similar elements in the derived class.

```
/*-----------------------------------------------------------------------*/
/* S I M P L E   T I M E R */
/*-----------------------------------------------------------------------*/

class SimpleTimer : public InterruptHandler {

protected:

  /* 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. */
```

Figure 13: `start` and `SimpleTimer` modifications

EOQTimer.H : Contains class `EOQTimer` which implements End Of Quantum timer. It derives from class `SimpleTimer`.

```
#ifndef _EOQ_TIMER_H_
#define _EOQ_TIMER_H_

#include "simple_timer.H"

// Forward Declarations.
class Scheduler;

class EOQTimer : public SimpleTimer
{
private:
    // Reference to scheduler.
    Scheduler *SYSTEM_SCHEDULER;

public:
    // Constructor.
    EOQTimer(int _hz, Scheduler *_scheduler);

    // Method to override base class interrupt handler to handle
    // End Of Quantum preemption.
    virtual void handle_interrupt(REGS *_r) override;
};

#endif /* _EOQ_TIMER_H_ */
```

Figure 14: Header file containing class `EOQTimer`

EOQTimer.C: EOQTimer::EOQTimer(int _hz, Scheduler* _scheduler) : Parameterized constructor that receives a reference to the scheduler as argument and also sets the frequency of the timer by calling the base class constructor.

EOQTimer.C: EOQTimer::handle_interrupt(REGS *_r) : This is the interrupt handler. Since the time quantum is ***50ms***, the frequency that needs to be passed is ***20Hz***. Every time the interrupt handler is called, it is because the timer fires at an interval of ***50ms***. Every time quanta, the Interrupt handler add the currently executing thread back to the end of queue and yields the CPU to the next thread at the head of queue in a round robin fashion.

```
// Constructor.
EOQTimer::EOQTimer(int _hz, Scheduler* _scheduler) : SimpleTimer(_hz)
{
    Console::puts("Constructing EOQTimer\n");
    this->SYSTEM_SCHEDULER = _scheduler;

} // EOQTimer::EOQTimer


// Method to override base class interrupt handler to handle
// End Of Quantum preemption.
void EOQTimer::handle_interrupt(REGS *_r)
{
    Console::puts("Time quantum has passed.\n");
    Console::puts("Pre-empting current thread.\n");
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();

} // EOQTimer::handle_interrupt
```

Figure 15: Implementation of class `EOQTimer`

kernel.C : The `kernel.C` file has been modified accommodate the Round Robin Scheduler. The macro definition `_USES_ROUND_ROBIN_SCHEDULER_` instantiates an object of class `RoundRobinScheduler` and also triggers End-Of-Quantum timer interrupts. This macro needs to be defined to run the Round Robin Scheduler. Without this definition, the regular FIFO scheduler would run.

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

#ifdef _USES_SCHEDULER_
    /* -- SCHEDULER -- IF YOU HAVE ONE -- */
    #ifdef _USES_ROUND_ROBIN_SCHEDULER_
        SYSTEM_SCHEDULER = new RoundRobinScheduler();
    #else //  Use FIFO scheduler
        SYSTEM_SCHEDULER = new Scheduler();
    #endif
#endif
```

Figure 16: Usage of macro `_USES_ROUND_ROBIN_SCHEDULER_`

```
    /* -- KICK-OFF THREAD1 ... */
    Console::puts("STARTING THREAD 1 ...\n");
#ifdef _USES_ROUND_ROBIN_SCHEDULER_
    EOQTimer rrTimer(5, SYSTEM_SCHEDULER);
    InterruptHandler::register_handler(0, &rrTimer);
#endif
    Thread::dispatch_to(thread1);
```

Figure 17: Registration of End Of Quantum Timer

makefile : The `makefile` has been modified to compile and link class `EOQTimer`.

```
# ==== DEVICES =====

console.o: console.C console.H
    $(GCC) $(GCC_OPTIONS) -c -o console.o console.C

simple_timer.o: simple_timer.C simple_timer.H
    $(GCC) $(GCC_OPTIONS) -c -o simple_timer.o simple_timer.C

EOQTimer.o: EOQTimer.C EOQTimer.H
    $(GCC) $(GCC_OPTIONS) -c -o EOQTimer.o EOQTimer.C
```

Figure 18: Modifications to the `makefile`

Testing

Testing covers running the FIFO scheduler as well as the Round Robin Scheduler with both Terminating and Non-Terminating functions.

FIFO Scheduler

Comment out the macro `_USES_ROUND_ROBIN_SCHEDULER_`

Non-Terminating Functions

Comment out the macro `_TERMINATING_FUNCTIONS_`. Now, threads 1 and 2 will run infinitely but they would get preempted after 10 ticks. Detailed logs can be found under *../outputs/fifo-non-terminating.log*.

```
vinayb@Vinays-Laptop src % make run
Darwin
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098116>
done
DONE
CREATING THREAD 2...esp = <2099164>
done
DONE
CREATING THREAD 3...esp = <2100212>
done
DONE
CREATING THREAD 4...esp = <2101260>
done
DONE
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]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
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]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Thread: 2
FUN 3 INVOKED!
FUN 3 IN BURST[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]
```

Figure 19: Non Terminating Methods with the FIFO scheduler

Terminating Functions

Uncomment the macro `_TERMINATING_FUNCTIONS_`. Now threads 1 and 2 will run only for 10 bursts, post which they would terminate. Threads 3 and 4 would continue to run infinitely. It can be seen that in the figure below, Threads 1 and 2 end after some point in the execution. Detailed logs can be found under *../outputs/fifo-terminating.log*.

```
FUN 4: 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]
FUN 1: TICK [9]
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]
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]
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]
Cannot terminate given thread!
Cannot terminate given thread!
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]
FUN 4 IN BURST[10]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
```

Figure 20: Terminating Methods with the FIFO scheduler

Round Robin Scheduler

Uncomment the macro `_USES_ROUND_ROBIN_SCHEDULER_`

Non-Terminating Functions

Comment out the macro `_TERMINATING_FUNCTIONS_`. Now, threads 1 and 2 will run infinitely but even then, the End-Of-Quantum timer would fire and preempt the threads. Detailed logs can be found under *../outputs/rr-non-terminating.log*.

```
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 1 IN BURST[2]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
Time quantum has passed.
Pre-empting current thread.
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]
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 [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]
Time quantum has passed.
Pre-empting current thread.
Thread: 2
FUN 3 INVOKED!
FUN 3 IN BURST[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 3 IN BURST[1]
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[2]
```

Figure 21: Non-Terminating Methods with the Round Robin Scheduler

Terminating Functions

Uncomment the macro `_TERMINATING_FUNCTIONS_`. Now, threads 1 and 2 will run only for a finite time. Again, the End-Of-Quantum timer would determine which threads to run. The figure below demonstrates that a thread can get preempted once it is done executing and before the quantum expires. Detailed logs can be found under *../outputs/rr-terminating.log*.

```
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]
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]
Cannot terminate given thread!
Time quantum has passed.
Pre-empting current thread.
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 2 IN BURST[7]
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]
FUN 2 IN BURST[8]
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]
Time quantum has passed.
Pre-empting current thread.
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[7]
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]
```

Figure 22: Terminating Methods with the Round Robin Scheduler