MP 5: Kernel Level Thread Scheduling

Pranav Anantharam

UIN: 734003886

CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus Option 1: Completed.
Bonus Option 2: Completed.
Bonus Option 3: Did not attempt.

System Design

The goal of this machine problem is to perform scheduling of multiple kernel-level threads. To this end, we implement the following in this machine problem:

1. FIFO scheduler to handle non-terminating threads
2. Scheduler mechanisms to handle threads with terminating functions
3. Interrupt Handling
4. Round-Robin scheduler

FIFO Scheduler Implementation:

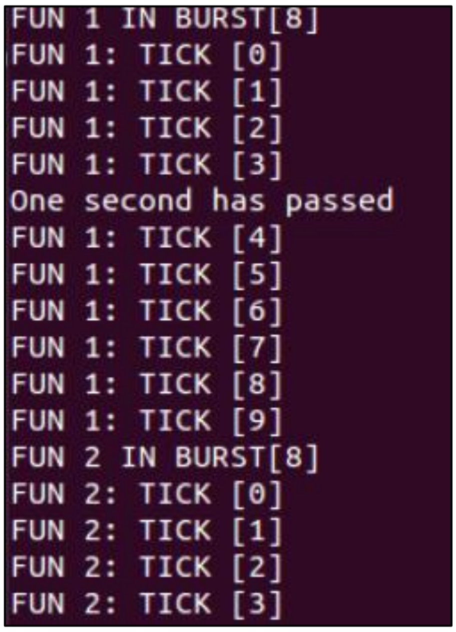
A class 'Queue' is introduced to handle operations on the ready queue of threads. The queue is implemented as a linked list of thread objects. The class 'Queue' declares constructors and public functions to add (enqueue) and remove (dequeue) threads from the ready queue. To perform the enqueue operation, we first check if there exists a thread in the ready queue already. If a thread exists in the ready queue, we traverse to the end of the queue and append the new thread. To perform the dequeue operation, the thread at the top of the queue is removed and the pointer is updated to point to the next thread in the queue.

Handling Threads with terminating thread functions:

To handle threads with terminating thread functions, we must first remove the thread from the ready queue. We iterate through the ready queue and dequeue the thread with matching Thread ID (using 'terminate' method). Next, the memory allocated for the thread is freed using the 'thread_shutdown' method. Finally, we call the 'yield' method to dispatch to the next thread in the ready queue.

Bonus Section Option 1 - Handling Interrupts:

To ensure mutual exclusion, we must enable and disable interrupts appropriately so there is no disruption while adding threads or removing threads from the ready queue. And so, changes were made in scheduler.C wherein interrupts were disabled before performing any operations on the ready queue, and interrupts were re-enabled after completing operations on the ready queue of threads. Enabling and disabling of threads were achieved using the 'interrupts_enabled', ‘enable_interrupts’ and ‘disable_interrupts’ methods provided in machine.H. The interrupt handling implementation was verified using the below output logs (generated when interrupts are enabled):



```
FUN 1 IN BURST[8]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
One second has passed
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[8]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
```

Bonus Section Option 2 - Round Robin Scheduler Implementation:

To implement a round-robin scheduler with a 50 ms time quantum, we create a class ‘RRScheduler’ which inherits properties from classes ‘Scheduler’ and ‘InterruptHandler’. The class ‘RRScheduler’ supports all the methods defined by class ‘Scheduler’. In addition, a timer-based interrupt with tick frequency of ‘5’ is created to track the time quantum and the corresponding interrupt handler is registered. When 50 ms has passed, the interrupt handler calls methods to pre-empt the current thread. An End-of-Interrupt message is sent to the master interrupt controller to indicate the interrupt has been handled. Next, the thread at the top of the ready queue is dequeued and given CPU time. The pointer points to the next thread in the ready queue.

List of files modified

The following files were modified:

1. scheduler.H
2. scheduler.C
3. thread.C
4. kernel.C

Code Description

1. **scheduler.H : class Queue:**

In class Queue, data structures and functions for managing the ready queue are defined:

a) Thread * thread - Pointer to the thread at the top of the ready queue
b) Queue * next - Pointer to next Queue type object (next thread in the ready queue)
c) Queue() - Constructor for initial setup of the variables
d) Queue(Thread* new_thread) - Constructor for setting up subsequent threads

e) void enqueue(Thread * new_thread) - Function to add a thread to the end of the ready queue

f) Thread* dequeue() - Function to remove the thread at the top of the ready queue and point to the next thread

```
/*-----------------------------------------------------------------------------*/
/* QUEUE DATA STRUCTURE */
/*-----------------------------------------------------------------------------*/

class Queue
{
    private:

    Thread* thread;                 // Pointer to thread at the top of queue
    Queue* next;                    // Pointer to next thread in queue.

    public:

    // Constructor for initial setup
    Queue()
    {
        thread = NULL;
        next   = NULL;
    }

    // Constructor for new threads to enter queue
    Queue(Thread* new_thread)
    {
        thread = new_thread;
        next = NULL;
    }

    // Add thread at end of queue
    void enqueue(Thread* new_thread)
    {
        // First thread added to queue
        if ( thread == NULL )
        {
            thread = new_thread;
        }
```

```
    else
    {
        // Traverse the queue
        if( next != NULL )
        {
            next->enqueue(new_thread);
        }
        else
        {
            // Reached end of queue - add new thread at the end of queue
            next =  new Queue(new_thread);
        }
    }
}

// Remove thread at head position and point to next thread in queue
Thread* dequeue()
{
    // Queue is empty
    if( thread == NULL )
    {
        return NULL;
    }

    // Get top of queue element
    Thread *top = thread;

    // Only one queue element present
    if( next == NULL )
    {
        thread = NULL;
    }
```

```
        else
        {
            // Update head element of queue
            thread = next->thread;

            // Delete current node
            Queue * del_node = next;

            // Update next pointer
            next = next->next;

            delete del_node;
        }

        return top;
    }
};
```

2. **scheduler.H : class Scheduler:**

   In class Scheduler, data structures and functions for FIFO scheduling of kernel-level threads and management of the ready queue are declared.

   a) Queue ready_queue - Queue data object for handling the ready queue of threads
   b) int qsize - Variable to track the number of threads waiting in the FIFO ready queue

```
/*--------------------------------------------------------------------------*/
/* SCHEDULER */
/*--------------------------------------------------------------------------*/

class Scheduler {

  /* The scheduler may need private members... */
  Queue ready_queue;
  int qsize;
```

3. **scheduler.H : class RRScheduler:**

   In class RRScheduler, data structures and functions for Round-Robin scheduling of kernel-level threads and management of the ready queue are declared. The RRScheduler class is created by inheriting classes Scheduler and InterruptHandler.

   a) Queue ready_rr_queue - Queue object for handling the ready queue for the Round-Robin scheduler
   b) int rr_queue – Variable to track the number of threads waiting in the Round-Robin ready queue
   c) int ticks - Variable to track the number of ticks that occurred since last update
   d) int hz - Variable to indicate frequency of update of ticks
   e) void set_frequency(int _hz) - Function to set the interrupt frequency for the timer

```
/*-----------------------------------------------------------------------*/
/* ROUND ROBIN SCHEDULER */
/*-----------------------------------------------------------------------*/

// Inherited Scheduler and Interrupt Handler classes
class RRScheduler: public Scheduler, public InterruptHandler
{
    Queue ready_rr_queue;               // Ready queue for Round-Robin scheduler
    int rr_qsize;                       // Robin-Robin ready queue size
    int ticks;                          // Number of ticks since last update
    int hz;                             // Frequency of update of ticks

    void set_frequency( int _hz );      // Set the interrupt frequency for the timer
```

4. **scheduler.C : Scheduler():**

This method is the constructor for class Scheduler. The FIFO queue size is initialized to zero.

```
Scheduler::Scheduler()
{
    qsize = 0;
    Console::puts("Constructed Scheduler.\n");
}
```

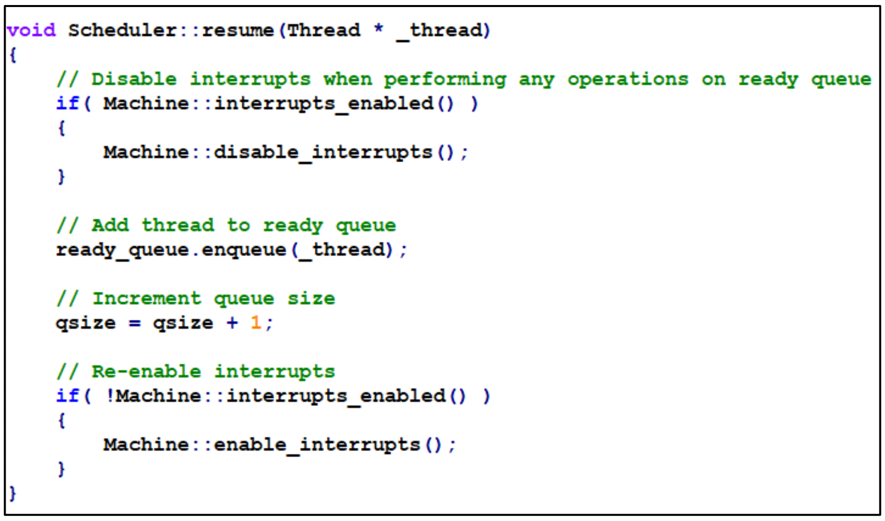
5. **scheduler.C : Scheduler :: yield():**

This method is used to pre-empt the currently running thread and give up the CPU. A new thread is selected for CPU time by dequeuing the top element in the ready queue and dispatching to it. Moreover, interrupts are disabled before the start of any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void Scheduler::yield()
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    if( qsize == 0 )
    {
        // Console::puts("Queue is empty. No threads available. \n");
    }
    else
    {
        // Remove thread from queue for CPU time
        Thread * new_thread = ready_queue.dequeue();

        // Decrement queue size
        qsize = qsize - 1;

        // Re-enable interrupts
        if( !Machine::interrupts_enabled() )
        {
            Machine::enable_interrupts();
        }

        // Context-switch and give CPU time to new thread
        Thread::dispatch_to(new_thread);
    }
}
```

6. **scheduler.C : Scheduler :: resume(Thread * _thread):**

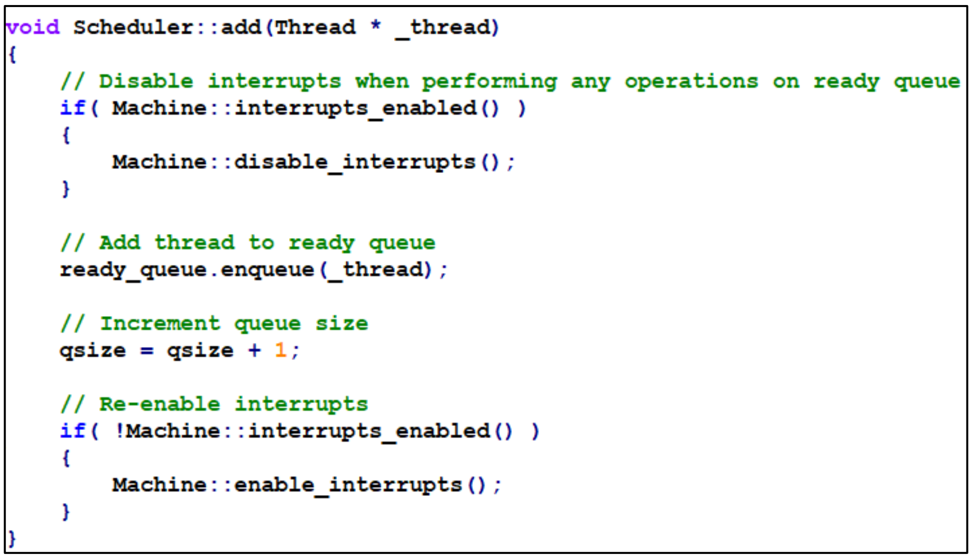
This method is used to add a thread to the ready queue of the FIFO scheduler. This method is called for threads that were waiting for an event to happen, or that have to give up the CPU in response to a pre-emption. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.



```
void Scheduler::resume(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_queue.enqueue(_thread);

    // Increment queue size
    qsize = qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

7. **scheduler.C : Scheduler :: add(Thread * _thread):**

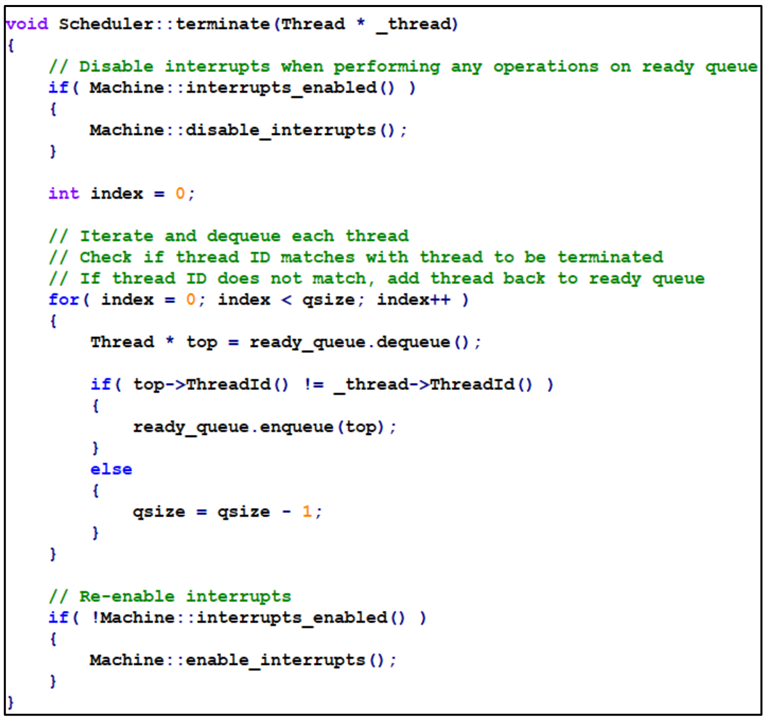
This method is used to add a thread to the ready queue of the FIFO scheduler. The thread is made runnable by the scheduler. This method is called on thread creation. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.



```
void Scheduler::add(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_queue.enqueue(_thread);

    // Increment queue size
    qsize = qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

8. **scheduler.C : Scheduler :: terminate(Thread * _thread):**

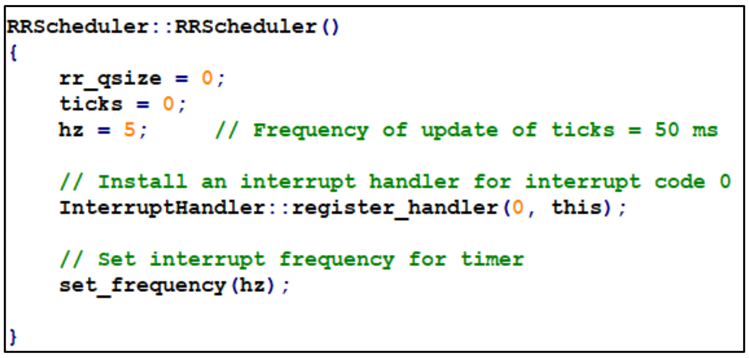
This method is used to remove a thread from the ready queue. We iterate through the ready queue; dequeue each thread and compare the Thread ID. If the Thread ID does not match with the thread ID of the thread to be terminated, then we enqueue the threads back into the ready queue. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.



```
void Scheduler::terminate(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    int index = 0;

    // Iterate and dequeue each thread
    // Check if thread ID matches with thread to be terminated
    // If thread ID does not match, add thread back to ready queue
    for( index = 0; index < qsize; index++ )
    {
        Thread * top = ready_queue.dequeue();

        if( top->ThreadId() != _thread->ThreadId() )
        {
            ready_queue.enqueue(top);
        }
        else
        {
            qsize = qsize - 1;
        }
    }

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

9. **scheduler.C : RRScheduler():**

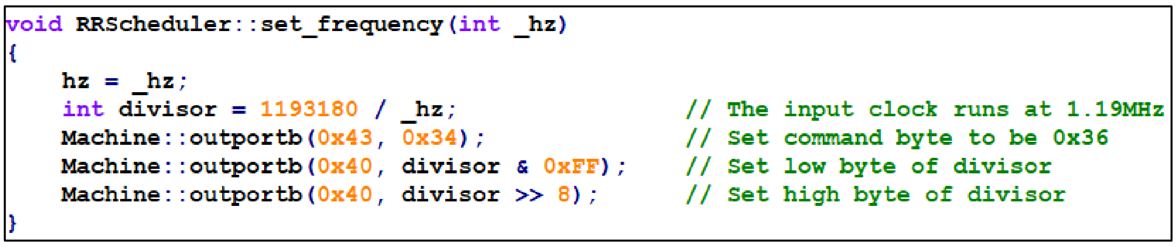
This method is the constructor for class RRScheduler. The Round-Robin queue size is initialized to zero. The timer variables are also initialized. Next, an interrupt handler is registered and the timer frequency of interrupts is set.



```
RRScheduler::RRScheduler()
{
    rr_qsize = 0;
    ticks = 0;
    hz = 5;     // Frequency of update of ticks = 50 ms

    // Install an interrupt handler for interrupt code 0
    InterruptHandler::register_handler(0, this);

    // Set interrupt frequency for timer
    set_frequency(hz);

}
```

10. **scheduler.C : RRScheduler :: set_frequency(int _hz):**

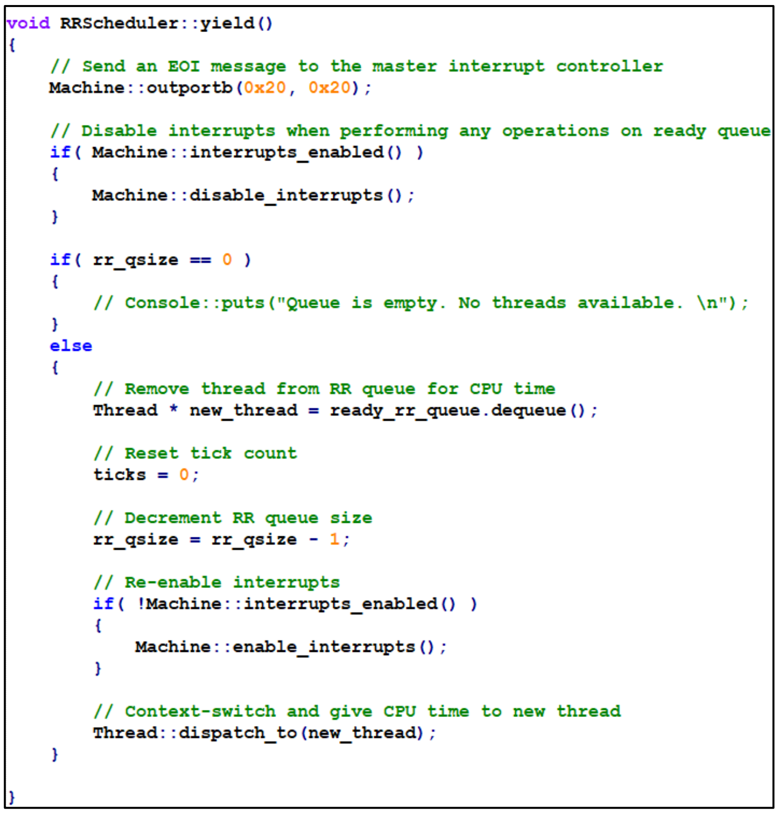
This method is used to set the frequency for the timer.



```
void RRScheduler::set_frequency(int _hz)
{
    hz = _hz;
    int divisor = 1193180 / _hz;                // The input clock runs at 1.19MHz
    Machine::outportb(0x43, 0x34);              // Set command byte to be 0x36
    Machine::outportb(0x40, divisor & 0xFF);    // Set low byte of divisor
    Machine::outportb(0x40, divisor >> 8);      // Set high byte of divisor
}
```

11. **scheduler.C : RRScheduler :: yield():**

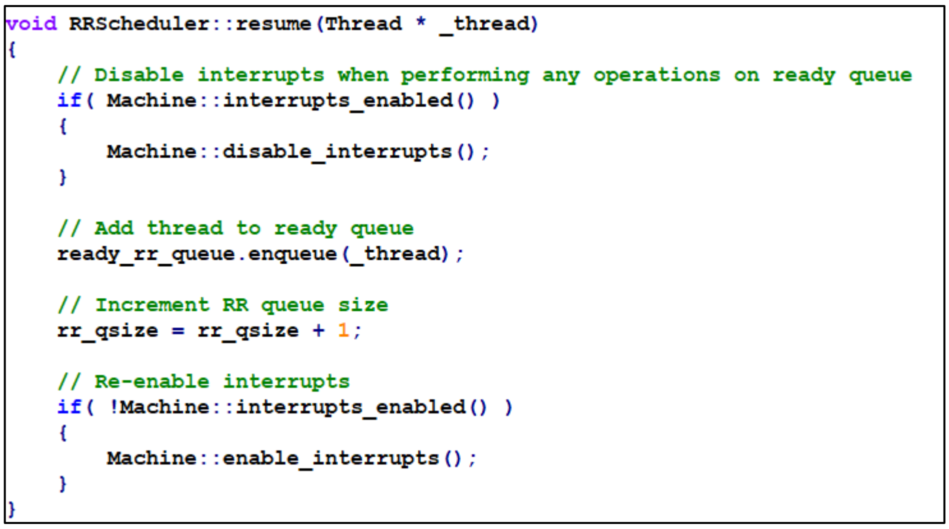
This method is used to pre-empt the currently running thread and give up the CPU. A new thread is selected for CPU time by dequeuing the top element in the Round-Robin ready queue and dispatching to it. First, an End-of-Interrupt message is sent to the master interrupt controller before interrupts are disabled. Next, interrupts are disabled before the start of any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.



```
void RRScheduler::yield()
{
    // Send an EOI message to the master interrupt controller
    Machine::outportb(0x20, 0x20);

    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    if( rr_qsize == 0 )
    {
        // Console::puts("Queue is empty. No threads available. \n");
    }
    else
    {
        // Remove thread from RR queue for CPU time
        Thread * new_thread = ready_rr_queue.dequeue();

        // Reset tick count
        ticks = 0;

        // Decrement RR queue size
        rr_qsize = rr_qsize - 1;

        // Re-enable interrupts
        if( !Machine::interrupts_enabled() )
        {
            Machine::enable_interrupts();
        }

        // Context-switch and give CPU time to new thread
        Thread::dispatch_to(new_thread);
    }

}
```

12. **scheduler.C : RRScheduler :: resume(Thread * _thread):**

This method is used to add a thread to the ready queue of the Round-Robin scheduler. This method is called for threads that were waiting for an event to happen, or that have to give up the CPU in response to a pre-emption. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.



```
void RRScheduler::resume(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_rr_queue.enqueue(_thread);

    // Increment RR queue size
    rr_qsize = rr_qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

13. **scheduler.C : RRScheduler :: add(Thread * _thread):**

This method is used to add a thread to the ready queue of the Round-Robin scheduler. The thread is made runnable by the scheduler. This method is called on thread creation. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void RRScheduler::add(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    // Add thread to ready queue
    ready_rr_queue.enqueue(_thread);

    // Increment RR queue size
    rr_qsize = rr_qsize + 1;

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

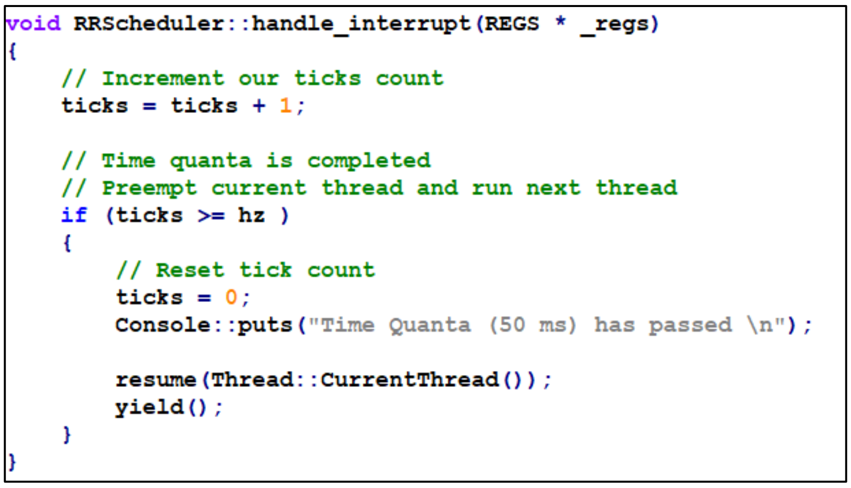
14. **scheduler.C : RRScheduler :: terminate(Thread * _thread):**

This method is used to remove a thread from the Round-Robin ready queue. We iterate through the ready queue; dequeue each thread and compare the Thread ID. If the Thread ID does not match with the thread ID of the thread to be terminated, then we enqueue the threads back into the ready queue. Interrupts are disabled before performing any operations on the ready queue, and interrupts are re-enabled after completing operations on the ready queue.

```
void RRScheduler::terminate(Thread * _thread)
{
    // Disable interrupts when performing any operations on ready queue
    if( Machine::interrupts_enabled() )
    {
        Machine::disable_interrupts();
    }

    int index = 0;

    // Iterate and dequeue each thread
    // Check if thread ID matches with thread to be terminated
    // If thread ID does not match, add thread back to ready queue

    for( index = 0; index < rr_qsize; index++ )
    {
        Thread * top = ready_rr_queue.dequeue();

        if( top->ThreadId() != _thread->ThreadId() )
        {
            ready_rr_queue.enqueue(top);
        }
        else
        {
            rr_qsize = rr_qsize - 1;
        }
    }

    // Re-enable interrupts
    if( !Machine::interrupts_enabled() )
    {
        Machine::enable_interrupts();
    }
}
```

15. **scheduler.C : RRScheduler :: handle_interrupt(REGS * _regs):**

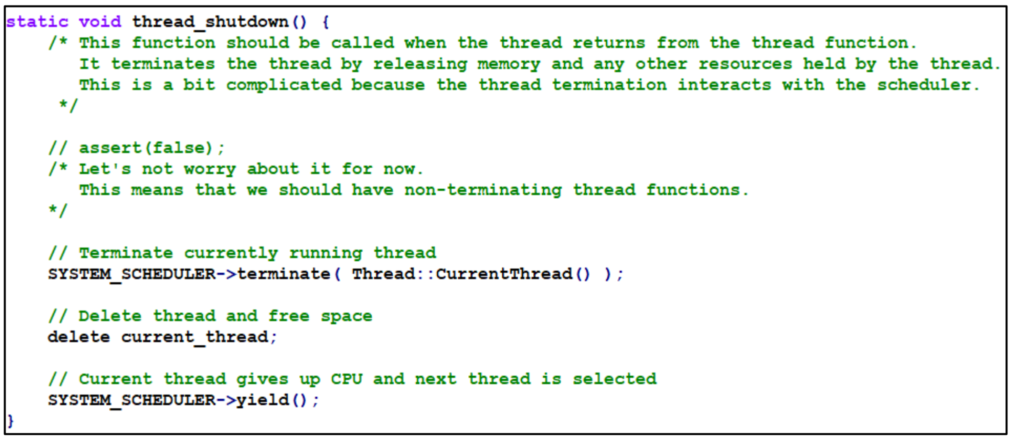
This method is used to handle interrupts raised by the timer. The number of ticks is incremented. Next, we check if a time quantum (50 ms) has been completed. If a time quantum is completed, we reset the ticks counter and pre-empt the currently running thread and dispatch to the next thread. In order to perform these operations, we use ‘resume’ and ‘yield’ methods.



```
void RRScheduler::handle_interrupt(REGS * _regs)
{
    // Increment our ticks count
    ticks = ticks + 1;

    // Time quanta is completed
    // Preempt current thread and run next thread
    if (ticks >= hz )
    {
        // Reset tick count
        ticks = 0;
        Console::puts("Time Quanta (50 ms) has passed \n");

        resume(Thread::CurrentThread());
        yield();
    }
}
```

16. **thread.C : thread_shutdown():**

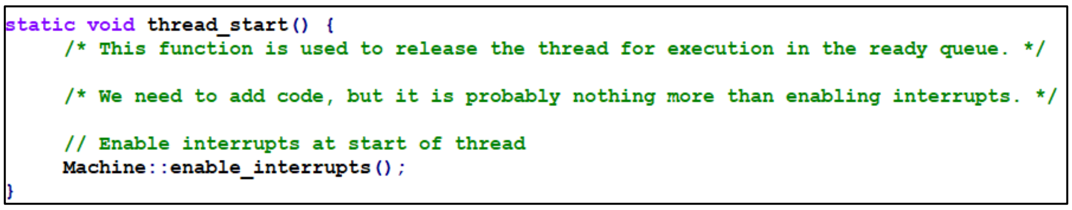
This method is used to terminate a thread by releasing the memory and other resources held by the thread. The below changes were made to handle threads with terminating thread functions. The currently running thread is first terminated. Next, memory is freed up using the ‘delete’ operation. Finally, the ‘yield’ method is called to give up CPU and select the next thread in the ready 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 currently running thread
    SYSTEM_SCHEDULER->terminate( Thread::CurrentThread() );

    // Delete thread and free space
    delete current_thread;

    // Current thread gives up CPU and next thread is selected
    SYSTEM_SCHEDULER->yield();
}
```

17. **thread.C : thread_start():**

Changes were made in this method to enable interrupts at the start of the thread using the ‘enable_interrupts’ method.



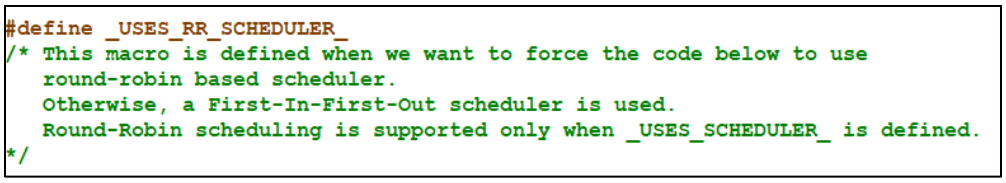
```
static void thread_start() {
    /* This function is used to release the thread for execution in the ready queue. */

    /* We need to add code, but it is probably nothing more than enabling interrupts. */

    // Enable interrupts at start of thread
    Machine::enable_interrupts();
}
```

18. **kernel.C : Code changes for testing:**

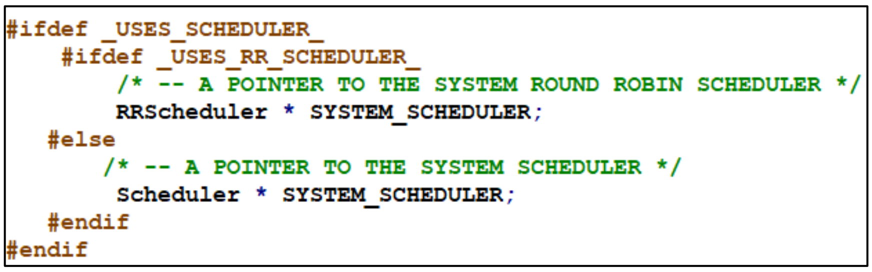
The following changes were made in kernel.C for the purpose of testing:

a) A new macro '_USES_RR_SCHEDULER' was introduced to enable support for Round-Robin scheduling

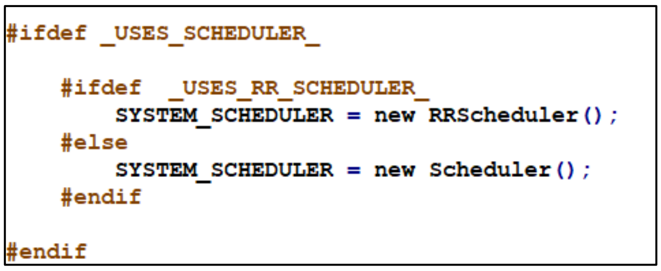
b) Code was fenced to support Round-Robin scheduling when '_USES_RR_SCHEDULER' is defined



```
#define _USES_RR_SCHEDULER_
/* This macro is defined when we want to force the code below to use
   round-robin based scheduler.
   Otherwise, a First-In-First-Out scheduler is used.
   Round-Robin scheduling is supported only when _USES_SCHEDULER_ is defined.
*/
```



```
#ifdef _USES_SCHEDULER_
    #ifdef _USES_RR_SCHEDULER_
        /* -- A POINTER TO THE SYSTEM ROUND ROBIN SCHEDULER */
        RRScheduler * SYSTEM_SCHEDULER;
   #else
       /* -- A POINTER TO THE SYSTEM SCHEDULER */
        Scheduler * SYSTEM_SCHEDULER;
   #endif
#endif
```



```
#ifdef _USES_SCHEDULER_

    #ifdef  _USES_RR_SCHEDULER_
        SYSTEM_SCHEDULER = new RRScheduler();
    #else
        SYSTEM_SCHEDULER = new Scheduler();
    #endif

#endif
```

Testing

Kernel level thread scheduling was performed and outputs were observed for the following cases:

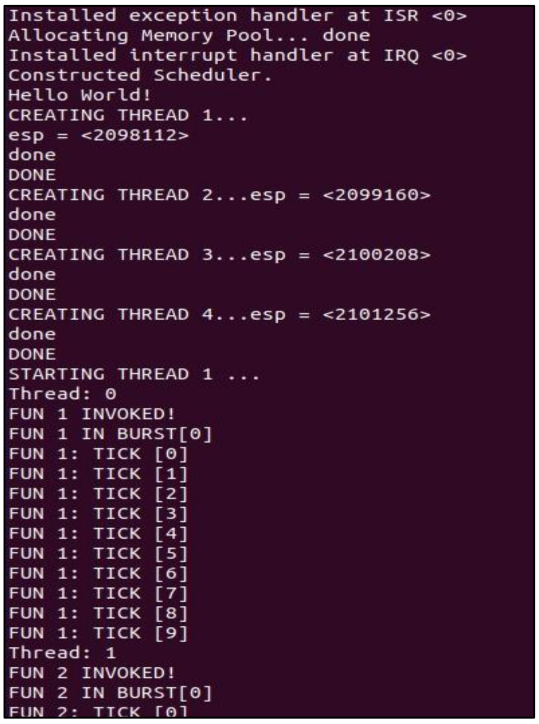
Serial No.	Testcase	Desired Result	Actual Result
1	FIFO scheduling to handle non-terminating threads ('_USES_SCHEDULER_' is defined, '_TERMINATING_FUNCTIONS_' is not defined)	Each thread executes for 10 ticks and passes the CPU to the next thread in continuous FIFO fashion	Each thread executes for 10 ticks and passes the CPU to the next thread in continuous FIFO fashion
2	FIFO scheduling to handle terminating threads ('_USES_SCHEDULER_' is defined, '_TERMINATING_FUNCTIONS_' is defined)	Thread 1 and Thread 2 execute and terminate. Thread 3 and Thread 4 continue to execute in continuous FIFO fashion	Thread 1 and Thread 2 execute and terminate. Thread 3 and Thread 4 continue to execute in continuous FIFO fashion

3	Round-Robin scheduling to handle non-terminating threads ('_USES_SCHEDULER_' is defined, '_USES_RR_SCHEDULER' is defined, '_TERMINATING_FUNCTIONS_' is not defined)	Each thread executes for 50 ms; gets pre-empted by the Round-Robin scheduler and the next thread is executed	Each thread executes for 50 ms; gets pre-empted by the Round-Robin scheduler and the next thread is executed
4	Round-Robin scheduling to handle terminating threads ('_USES_SCHEDULER_' is defined, '_USES_RR_SCHEDULER' is defined, '_TERMINATING_FUNCTIONS_' is defined)	Thread 1 and Thread 2 execute and terminate. Thread 3 and Thread 4 continue to execute in round-robin fashion and get pre-empted every 50 ms by the scheduler	Thread 1 and Thread 2 execute and terminate. Thread 3 and Thread 4 continue to execute in round-robin fashion and get pre-empted every 50 ms by the scheduler

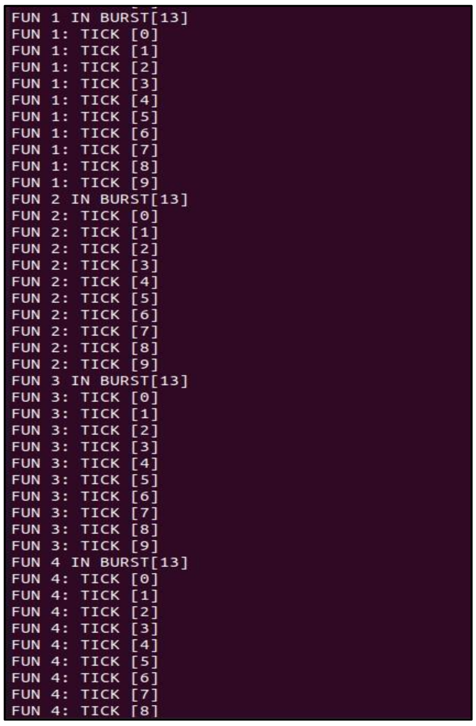
Modifications were made in kernel.C file to perform for FIFO and Round-Robin Scheduling and perform the above testcases; and verify the output.

Outputs:

1. FIFO scheduling to handle non-terminating threads)

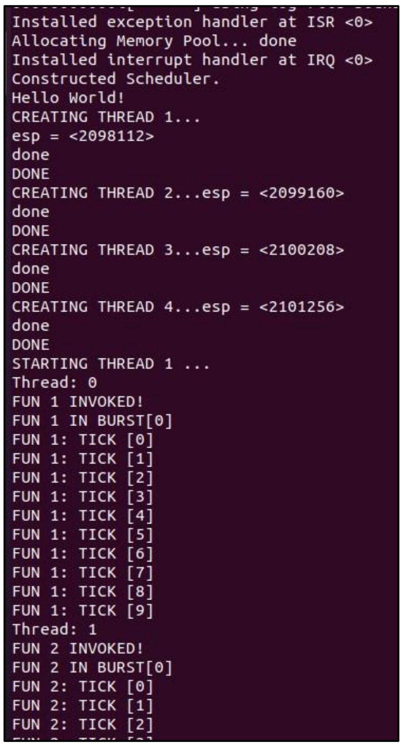


```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098112>
done
DONE
CREATING THREAD 2...esp = <2099160>
done
DONE
CREATING THREAD 3...esp = <2100208>
done
DONE
CREATING THREAD 4...esp = <2101256>
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]
```

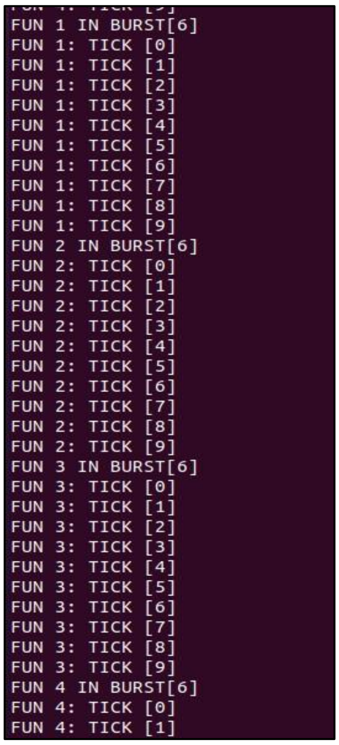


```
FUN 1 IN BURST[13]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN BURST[13]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 3 IN BURST[13]
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[13]
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]
```

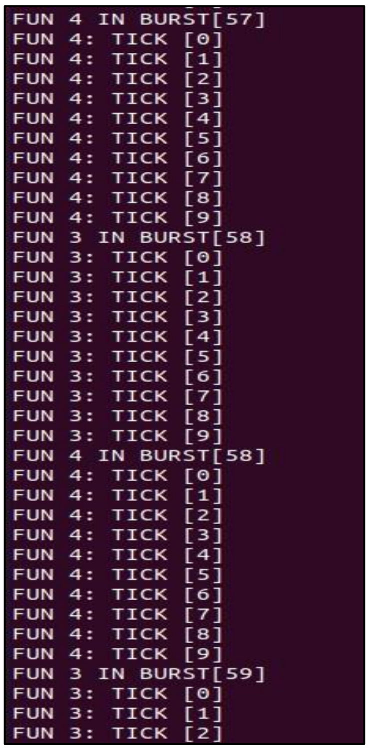
2. FIFO scheduling to handle terminating threads



```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098112>
done
DONE
CREATING THREAD 2...esp = <2099160>
done
DONE
CREATING THREAD 3...esp = <2100208>
done
DONE
CREATING THREAD 4...esp = <2101256>
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]
```

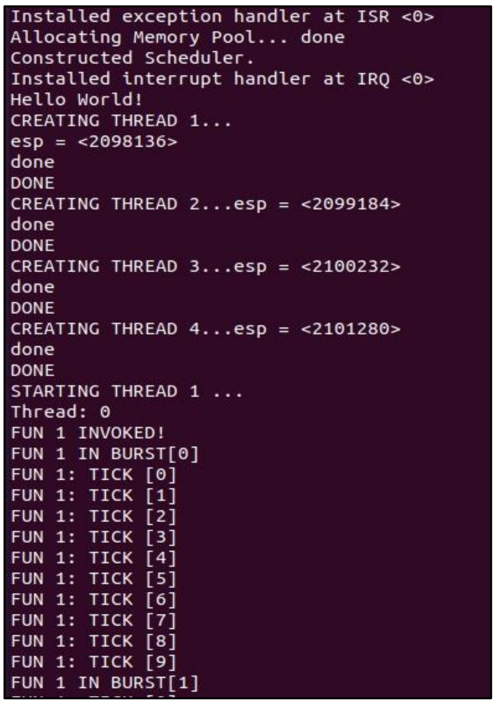


```
FUN 1 IN BURST[6]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN BURST[6]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
FUN 3 IN BURST[6]
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[6]
FUN 4: TICK [0]
FUN 4: TICK [1]
```

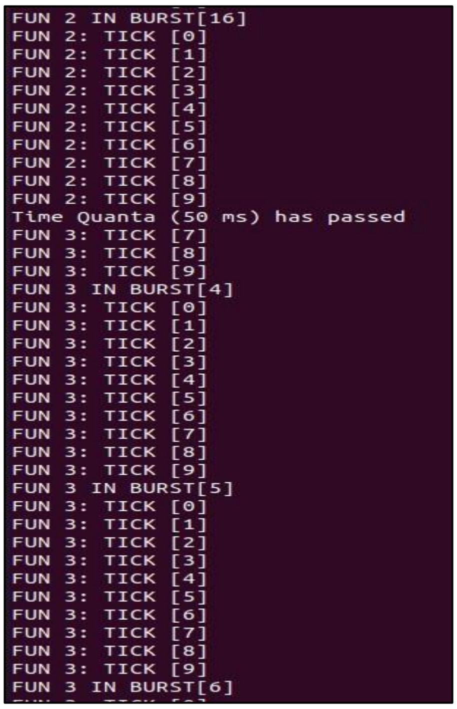


```
FUN 4 IN BURST[57]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[58]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[58]
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]
FUN 3 IN BURST[59]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
```

3. Round-Robin scheduling to handle non-terminating threads

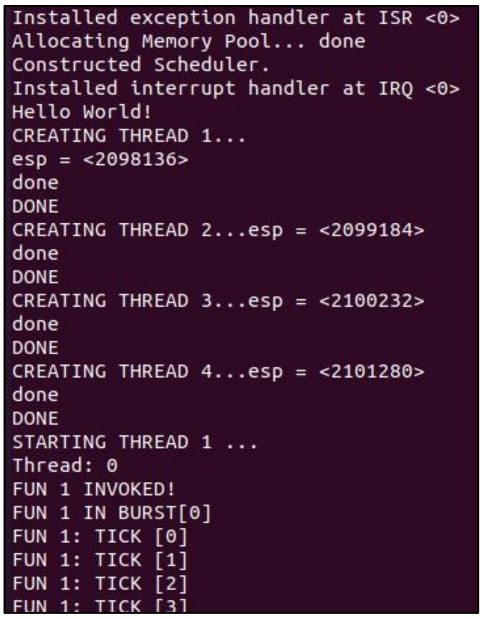


```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Constructed Scheduler.
Installed interrupt handler at IRQ <0>
Hello World!
CREATING THREAD 1...
esp = <2098136>
done
DONE
CREATING THREAD 2...esp = <2099184>
done
DONE
CREATING THREAD 3...esp = <2100232>
done
DONE
CREATING THREAD 4...esp = <2101280>
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]
FUN 1 IN BURST[1]
```

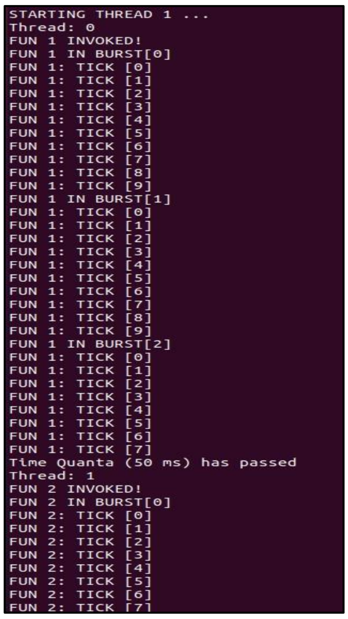


```
FUN 2 IN BURST[16]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
FUN 2: TICK [6]
FUN 2: TICK [7]
FUN 2: TICK [8]
FUN 2: TICK [9]
Time Quanta (50 ms) has passed
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 3 IN BURST[4]
FUN 3: TICK [0]
FUN 3: 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[5]
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[6]
```

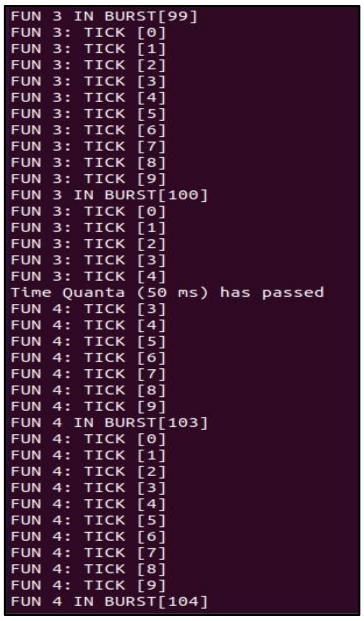
4. Round-Robin scheduling to handle non-terminating threads



```
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Constructed Scheduler.
Installed interrupt handler at IRQ <0>
Hello World!
CREATING THREAD 1...
esp = <2098136>
done
DONE
CREATING THREAD 2...esp = <2099184>
done
DONE
CREATING THREAD 3...esp = <2100232>
done
DONE
CREATING THREAD 4...esp = <2101280>
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]
```



```
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]
FUN 1 IN BURST[1]
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 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]
FUN 1: TICK [6]
FUN 1: TICK [7]
Time Quanta (50 ms) has passed
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 3 IN BURST[99]
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[100]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
Time Quanta (50 ms) has passed
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[103]
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]
FUN 4 IN BURST[104]
```