MP5: Kernel-Level Thread Scheduling

Vishnuvasan Raghuraman
UIN: 234009303
CSCE611: Operating System

Assigned Tasks

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

System Design

This machine problem aims to achieve scheduling for multiple kernel-level threads. We'll be implementing the following components:

- A FIFO scheduler for managing threads that do not terminate(non-terminating).
- Scheduler mechanisms to handle threads with functions that terminate(terminating).
- Proper handling of interrupts.
- Implementation of a Round-Robin scheduler.

Implmentation of First-In-First-Out Scheduler

In the implementation of the FIFO Scheduler, we introduce a class named 'Queue' to manage operations on the ready queue of threads. This queue is structured as a linked list of thread objects. Within the 'Queue' class, constructors and public functions are declared to facilitate the addition (enqueue) and removal (dequeue) of threads from the ready queue.

When performing the enqueue operation, the system first checks if there are already threads present in the ready queue. If there are, it traverses the queue to reach the end and then appends the new thread accordingly.

For the dequeue operation, the thread at the front of the queue is removed, and the pointer is adjusted to point to the next thread in line. This ensures the orderly execution of threads based on the First-In-First-Out principle.

Managing Threads with Terminating Thread Functions:

To manage threads with terminating functions, we begin by extracting the thread from the ready queue. This involves iterating through the queue and dequeuing the thread that matches the Thread ID, utilizing the 'terminate' method. Subsequently, we release the allocated memory for the thread via the 'thread_shutdown' method. Conclusively, we invoke the 'yield' method to transition execution to the subsequent thread in the ready queue. This ensures the seamless handling of threads with functions that terminate.

Bonus 1 - Interrupt Handling

To ensure mutual exclusion, it's imperative to appropriately enable and disable interrupts, preventing disruptions while adding or removing threads from the ready queue. Consequently, modifications were implemented in scheduler.C to address this concern. Specifically, interrupts are disabled prior to executing any operations on the ready queue, and re-enabled once these operations are completed. This enables seamless management of threads within the queue.

The enabling and disabling of interrupts are facilitated through the utilization of the 'interrupts_enabled', 'enable_interrupts', and 'disable_interrupts' methods provided in machine.H.

Bonus 2 - Implementation of Round Robin Scheduler

For the Round Robin Scheduler implementation with a time quantum of 50 ms, we define a class called 'RRScheduler', inheriting attributes from both 'Scheduler' and 'InterruptHandler' classes. This 'RRScheduler' class encompasses all methods defined within the 'Scheduler' class. Additionally, a timer-based interrupt is set up with a frequency of 5 ticks to monitor the time quantum. Upon the completion of the 50 ms time slice, the corresponding interrupt handler is invoked to preempt the current thread. An End-of-Interrupt message is dispatched to the master interrupt controller to signal the completion of interrupt handling. Subsequently, the scheduler dequeues the thread at the top of the ready queue and allocates CPU time to it, while adjusting the pointer to indicate the next thread in the ready queue. This mechanism ensures the smooth execution of threads under the Round Robin scheduling strategy.

Code Description

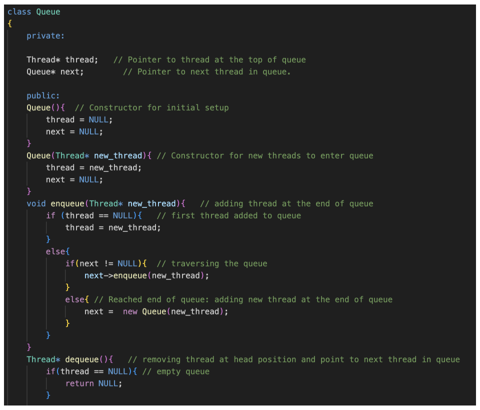
A Silicon(M2) Mac was used for this assignment and the native environment setup for such a machine from Piazza was followed. The below files were modified for this particular MP.

- scheduler.H
- scheduler.C
- thread.C
- kernel.C

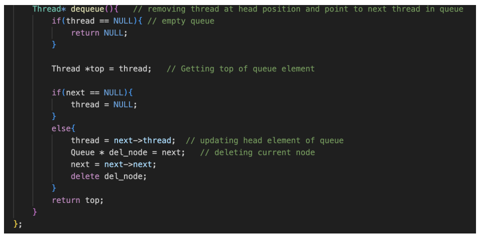
The functions implemented are given as below:

scheduler.H-class Queue: Within the class Queue, we define data structures and functions dedicated to managing the ready queue. In the class Queue, we define the following:

- Thread * thread: This member represents the thread at the top of the ready queue.
- Queue * next: This member points to the next Queue object in the ready queue, indicating the subsequent thread.
- Queue(): This constructor initializes the variables required for setup.
- Queue(Thread* new_thread): This constructor sets up subsequent threads in the ready queue.
- void enqueue(Thread* new_thread): This method adds a thread to the end of the ready queue.
- Thread* dequeue(): This method removes the thread at the top of the ready queue and updates the pointer to indicate the next thread in line.



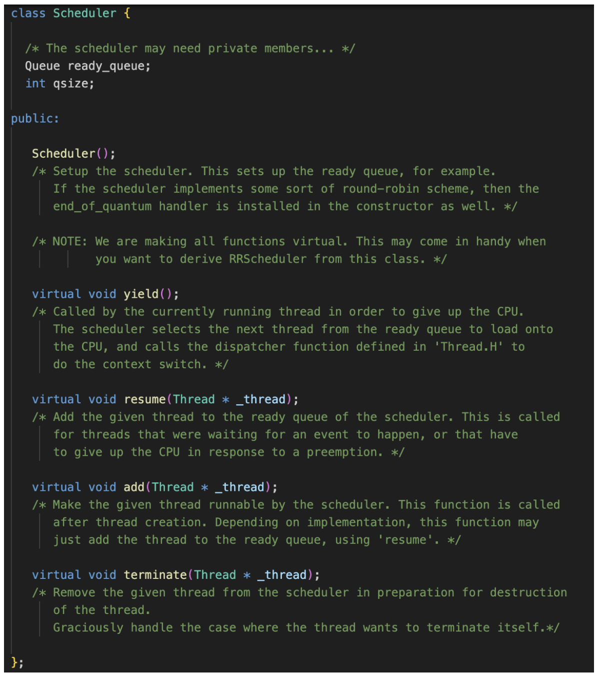
```
class Queue
{
    private:

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

    public:
    Queue(){  // Constructor for initial setup
        thread = NULL;
        next = NULL;
    }
    Queue(Thread* new_thread){ // Constructor for new threads to enter queue
        thread = new_thread;
        next = NULL;
    }
    void enqueue(Thread* new_thread){   // adding thread at the end of queue
        if (thread == NULL){   // first thread added to queue
            thread = new_thread;
        }
        else{
            if(next != NULL){  // traversing the queue
                next->enqueue(new_thread);
            }
            else{ // Reached end of queue: adding new thread at the end of queue
                next =  new Queue(new_thread);
            }
        }
    }
    Thread* dequeue(){   // removing thread at head position and point to next thread in queue
        if(thread == NULL){ // empty queue
            return NULL;
        }
```



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

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

        if(next == NULL){
            thread = NULL;
        }
        else{
            thread = next->thread;  // updating head element of queue
            Queue * del_node = next;   // deleting current node
            next = next->next;
            delete del_node;
        }
        return top;
    }
};
```

scheduler.H-class Scheduler: In the class Scheduler, we declare the following:

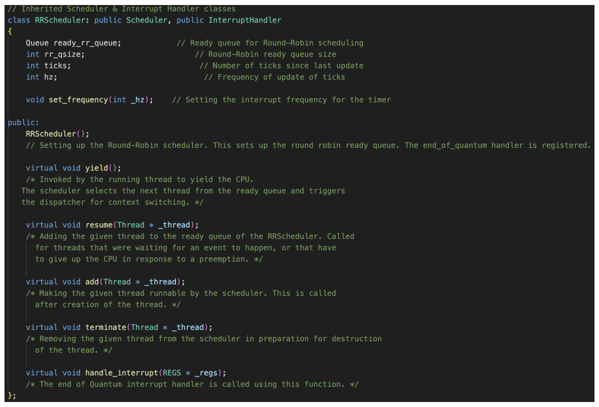
- Queue ready_queue: This Queue data object is responsible for managing the ready queue of threads.
- int qsize: This variable tracks the number of threads waiting in the FIFO ready queue.



```
class Scheduler {

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

public:

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

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

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

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

   virtual void add(Thread * _thread);
   /* Make the given thread runnable by the scheduler. This function is called
      after thread creation. Depending on implementation, this function may
      just add the thread to the ready queue, using 'resume'. */

   virtual void terminate(Thread * _thread);
   /* Remove the given thread from the scheduler in preparation for destruction
      of the thread.
      Graciously handle the case where the thread wants to terminate itself.*/

};
```

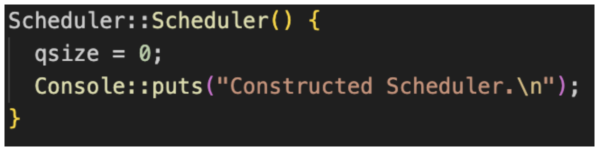
scheduler.H-class RRScheduler: In the class RRScheduler, which inherits from the Scheduler and InterruptHandler classes, we declare the following:

- `Queue ready_rr_queue`: This Queue object manages the ready queue for the Round-Robin scheduler.
- `int rr_queue`: This variable tracks the number of threads waiting in the Round-Robin ready queue.
- `int ticks`: This variable counts the number of ticks that have occurred since the last update.
- `int hz`: This variable indicates the frequency of updates for ticks.
- `void set_frequency(int _hz)`: This function is used to set the interrupt frequency for the timer.



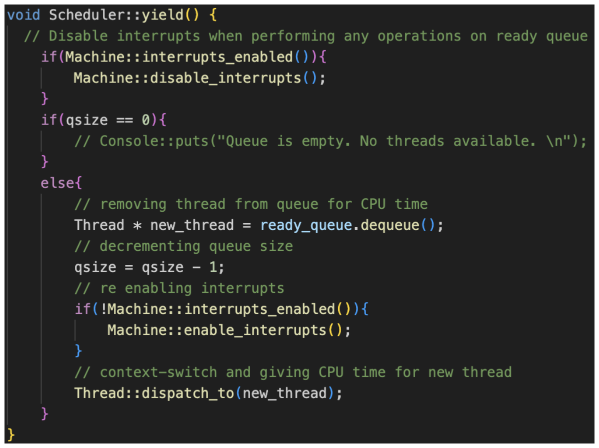
```
// Inherited Scheduler & Interrupt Handler classes
class RRScheduler: public Scheduler, public InterruptHandler
{
    Queue ready_rr_queue;            // Ready queue for Round-Robin scheduling
    int rr_qsize;                        // Round-Robin ready queue size
    int ticks;                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

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

public:
    RRScheduler();
    // Setting up the Round-Robin scheduler. This sets up the round robin ready queue. The end_of_quantum handler is registered.

    virtual void yield();
    /* Invoked by the running thread to yield the CPU.
   The scheduler selects the next thread from the ready queue and triggers
   the dispatcher for context switching. */

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

    virtual void add(Thread * _thread);
    /* Making the given thread runnable by the scheduler. This is called
      after creation of the thread. */

    virtual void terminate(Thread * _thread);
    /* Removing the given thread from the scheduler in preparation for destruction
      of the thread. */

    virtual void handle_interrupt(REGS * _regs);
    /* The end of Quantum interrupt handler is called using this function. */
};
```

scheduler.C-Scheduler: This method serves as the constructor for the Scheduler class. It initializes the FIFO queue size to zero.



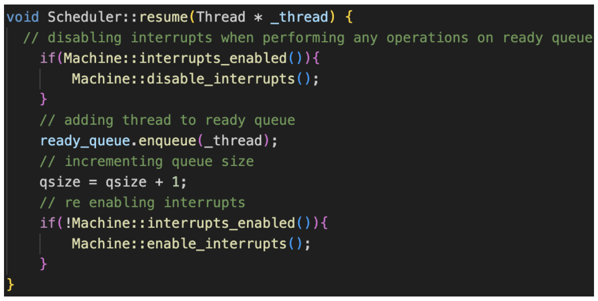
```
Scheduler::Scheduler() {
  qsize = 0;
  Console::puts("Constructed Scheduler.\n");
}
```

scheduler.C-Scheduler::yield: This method facilitates the pre-emption of the currently running thread, allowing it to relinquish the CPU. A new thread is chosen for CPU execution by dequeuing the top element from the ready queue and transitioning control to it. Additionally, interrupts are disabled prior to any operations on the ready queue, ensuring uninterrupted processing. Once operations on the ready queue are completed, interrupts are re-enabled to resume normal system functionality.



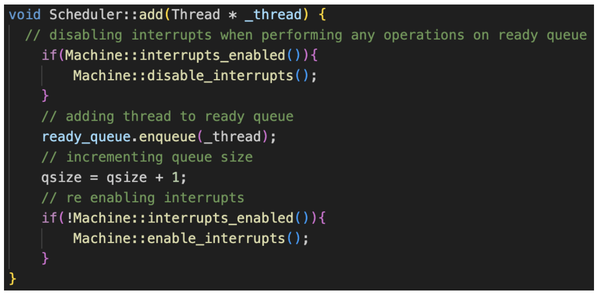
```
void Scheduler::yield() {
  // Disable interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    if(qsize == 0){
        // Console::puts("Queue is empty. No threads available. \n");
    }
    else{
        // removing thread from queue for CPU time
        Thread * new_thread = ready_queue.dequeue();
        // decrementing queue size
        qsize = qsize - 1;
        // re enabling interrupts
        if(!Machine::interrupts_enabled()){
            Machine::enable_interrupts();
        }
        // context-switch and giving CPU time for new thread
        Thread::dispatch_to(new_thread);
    }
}
```

scheduler.C:Scheduler::resume: This method is employed to include a thread into the ready queue of the FIFO scheduler. It's invoked for threads either waiting for an event or yielding the CPU due to pre-emption. To maintain integrity, interrupts are disabled before executing any actions on the ready queue and are subsequently re-enabled upon completion of these operations.



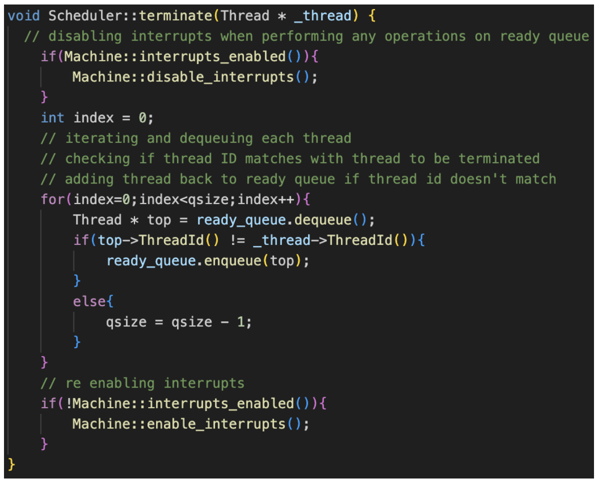
```
void Scheduler::resume(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_queue.enqueue(_thread);
    // incrementing queue size
    qsize = qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-Scheduler::add: This method is employed to add a thread into the ready queue of the FIFO scheduler, marking the thread as runnable. It's typically invoked upon thread creation. To ensure smooth operation, interrupts are disabled before any actions are taken on the ready queue and then re-enabled once those operations are finished.



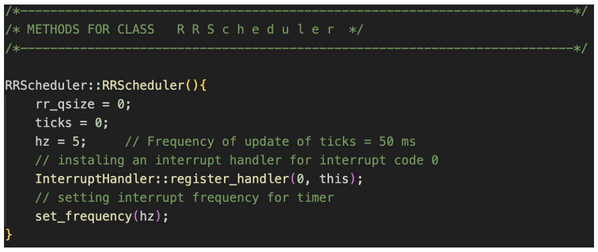
```
void Scheduler::add(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_queue.enqueue(_thread);
    // incrementing queue size
    qsize = qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-Scheduler::terminate: This method is employed to remove a thread from the ready queue. It involves iterating through the queue, dequeuing each thread, and comparing their Thread IDs. If a match is not found with the ID of the thread to be terminated, those threads are enqueued back into the ready queue. To ensure consistency, interrupts are disabled before any actions are taken on the ready queue and then re-enabled once these operations are completed.



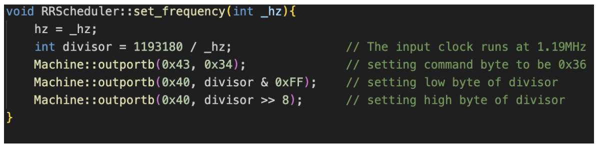
```
void Scheduler::terminate(Thread * _thread) {
  // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    int index = 0;
    // iterating and dequeuing each thread
    // checking if thread ID matches with thread to be terminated
    // adding thread back to ready queue if thread id doesn't match
    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 enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-RRScheduler: This method serves as the constructor for the RRScheduler class. It initializes the Round-Robin queue size to zero and sets up the timer variables accordingly. Additionally, it registers an interrupt handler and configures the frequency of timer interrupts.



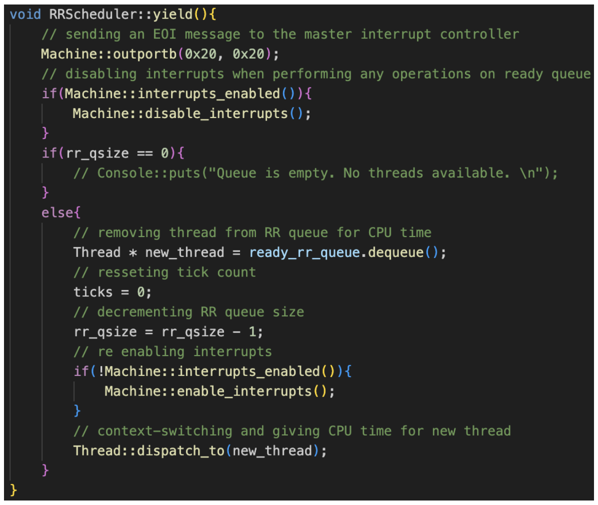
```
/*--------------------------------------------------------------------------*/
/* METHODS FOR CLASS   R R S c h e d u l e r  */
/*--------------------------------------------------------------------------*/

RRScheduler::RRScheduler(){
    rr_qsize = 0;
    ticks = 0;
    hz = 5;     // Frequency of update of ticks = 50 ms
    // instaling an interrupt handler for interrupt code 0
    InterruptHandler::register_handler(0, this);
    // setting interrupt frequency for timer
    set_frequency(hz);
}
```

scheduler.C-RRScheduler::set_frequency: This function is employed to set the frequency for the timer.



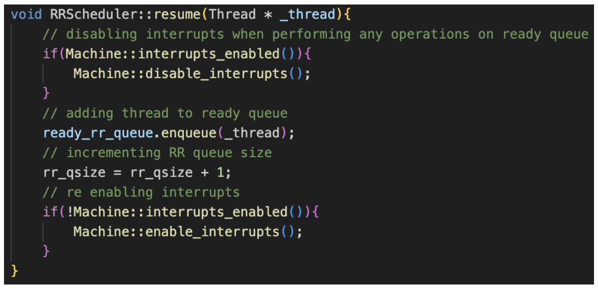
```
void RRScheduler::set_frequency(int _hz){
    hz = _hz;
    int divisor = 1193180 / _hz;               // The input clock runs at 1.19MHz
    Machine::outportb(0x43, 0x34);             // setting command byte to be 0x36
    Machine::outportb(0x40, divisor & 0xFF);   // setting low byte of divisor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

scheduler.C-RRScheduler::yield: This method facilitates the pre-emption of the currently running thread, allowing it to yield the CPU. A new thread is chosen for CPU execution by dequeuing the top element from the Round-Robin ready queue and transitioning control to it. Initially, an End-of-Interrupt message is dispatched to the master interrupt controller before interrupts are disabled. Subsequently, interrupts are disabled before any operations commence on the ready queue, and re-enabled once these operations are finalized.



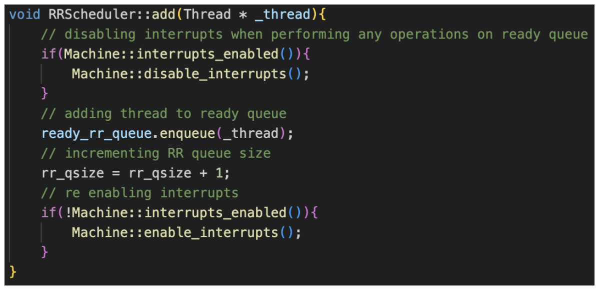
```
void RRScheduler::yield(){
    // sending an EOI message to the master interrupt controller
    Machine::outportb(0x20, 0x20);
    // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    if(rr_qsize == 0){
        // Console::puts("Queue is empty. No threads available. \n");
    }
    else{
        // removing thread from RR queue for CPU time
        Thread * new_thread = ready_rr_queue.dequeue();
        // resseting tick count
        ticks = 0;
        // decrementing RR queue size
        rr_qsize = rr_qsize - 1;
        // re enabling interrupts
        if(!Machine::interrupts_enabled()){
            Machine::enable_interrupts();
        }
        // context-switching and giving CPU time for new thread
        Thread::dispatch_to(new_thread);
    }
}
```

scheduler.C-RRScheduler::resume: This method is utilized to include a thread into the ready queue of the Round-Robin scheduler. It's invoked for threads either waiting for an event to occur or yielding the CPU due to pre-emption. Interrupts are disabled prior to executing any actions on the ready queue and are subsequently enabled upon completion of these operations.



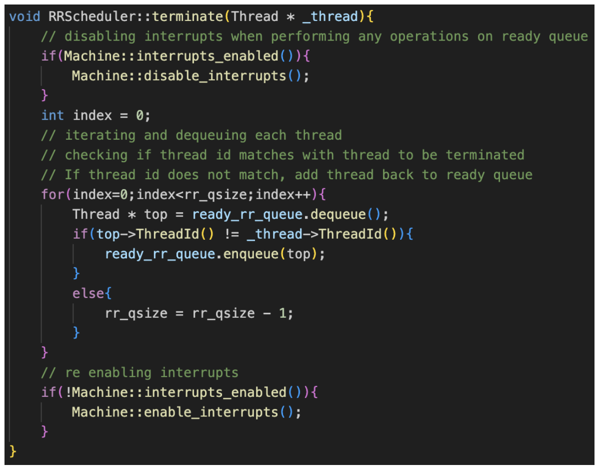
```
void RRScheduler::resume(Thread * _thread){
    // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_rr_queue.enqueue(_thread);
    // incrementing RR queue size
    rr_qsize = rr_qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-RRScheduler::add: This method is utilized to add a thread into the ready queue of the Round-Robin scheduler, which is made runnable by the scheduler. This method is invoked on thread creation. Interrupts are disabled prior to executing any actions on the ready queue and are subsequently re-enabled upon completion of these operations.



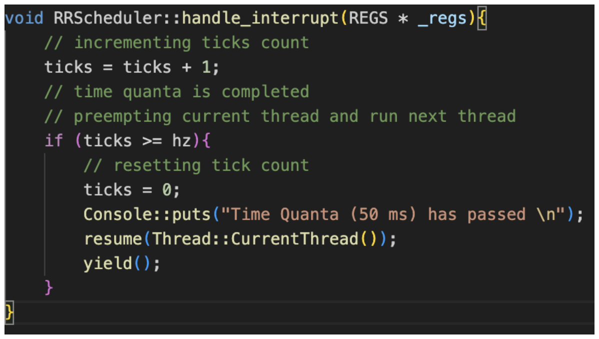
```
void RRScheduler::add(Thread * _thread){
    // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    // adding thread to ready queue
    ready_rr_queue.enqueue(_thread);
    // incrementing RR queue size
    rr_qsize = rr_qsize + 1;
    // re enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-RRScheduler::terminate: This method is employed to delete a thread from the Round-Robin ready queue. It entails iterating through the queue, dequeuing each thread, and comparing their Thread IDs. If there's no match with the ID of the thread to be terminated, those threads are enqueued back into the ready queue. To maintain consistency, interrupts are disabled before any actions are taken on the ready queue and then re-enabled once these operations are completed.



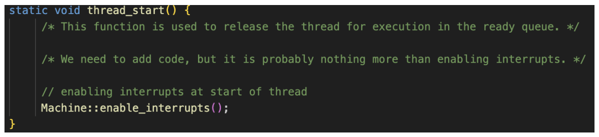
```
void RRScheduler::terminate(Thread * _thread){
    // disabling interrupts when performing any operations on ready queue
    if(Machine::interrupts_enabled()){
        Machine::disable_interrupts();
    }
    int index = 0;
    // iterating and dequeuing each thread
    // checking if thread id matches 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 enabling interrupts
    if(!Machine::interrupts_enabled()){
        Machine::enable_interrupts();
    }
}
```

scheduler.C-RRScheduler::handle_interrupt: This method is employed to manage interrupts triggered by the timer. It first increments the tick count. Subsequently, it checks if a time quantum of 50 milliseconds has elapsed. Upon completion of a time quantum, the ticks counter is reset, and the currently running thread is preempted to dispatch to the next thread. These operations are executed using the 'resume' and 'yield' methods.



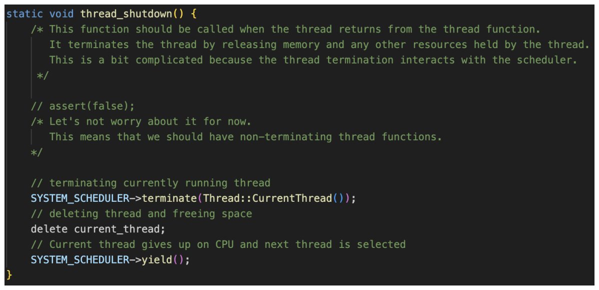
```
void RRScheduler::handle_interrupt(REGS * _regs){
    // incrementing ticks count
    ticks = ticks + 1;
    // time quanta is completed
    // preempting current thread and run next thread
    if (ticks >= hz){
        // resetting tick count
        ticks = 0;
        Console::puts("Time Quanta (50 ms) has passed \n");
        resume(Thread::CurrentThread());
        yield();
    }
}
```

thread.C : thread_start: Changes have been implemented in this method to enable interrupts at the beginning of the thread by utilizing 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. */

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

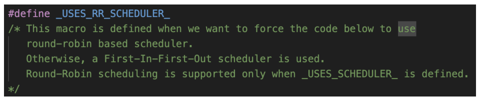
thread.C : thread_shutdown: This method is utilized to terminate a thread, releasing the memory and other associated resources held by the thread. To accommodate threads with terminating functions, the following adjustments were made. Initially, the currently running thread is terminated. Subsequently, memory is freed up using the 'delete' operation. Finally, the 'yield' method is invoked to relinquish CPU control and select the next thread in the ready queue.



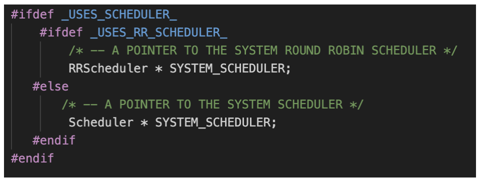
```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */

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

    // terminating currently running thread
    SYSTEM_SCHEDULER->terminate(Thread::CurrentThread());
    // deleting thread and freeing space
    delete current_thread;
    // Current thread gives up on CPU and next thread is selected
    SYSTEM_SCHEDULER->yield();
}
```

kernel.C: There were some changes made in this file to test different cases. They were as follows:

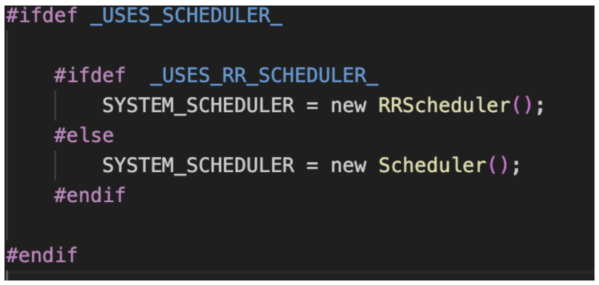
- Support for Round-Robin scheduling has been enabled by introducing a new macro named '_USES_RR_SCHEDULER'.
- The code has been fenced to accommodate Round-Robin scheduling when the '_USES_RR_SCHEDULER' macro is defined.



```
#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
```

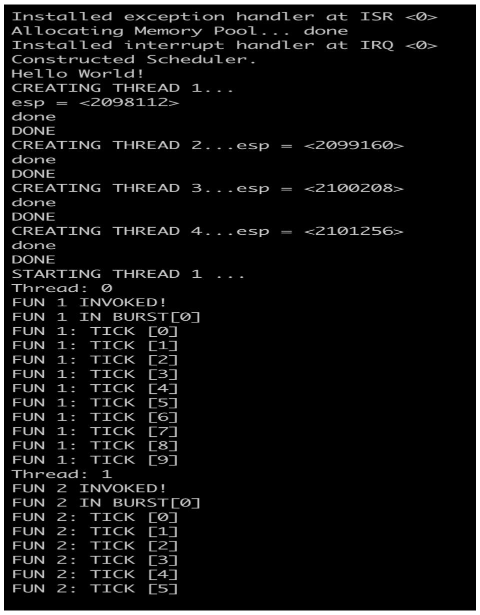


```
#ifdef _USES_SCHEDULER_

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

#endif
```

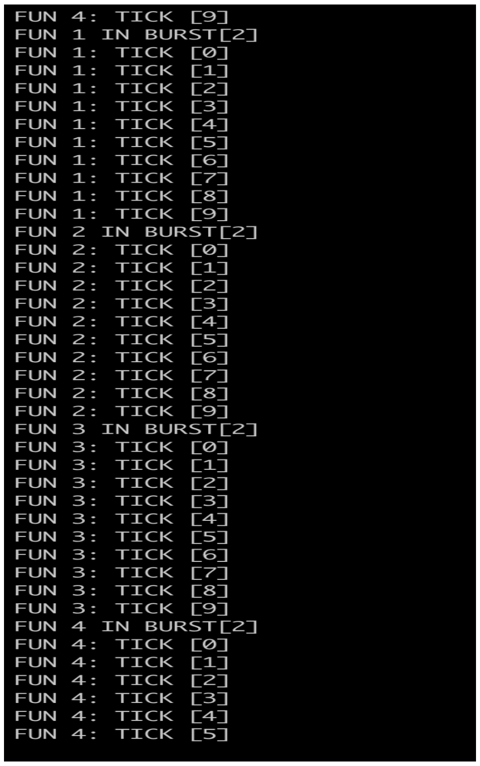
Testing

A few testcases were carried out by making modifications to the *kernel.C* file in the part of the code given below. This code block was changed based on the different test cases.

- Handling non-terminating threads by FIFO scheduling



```
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 2: TICK [3]
FUN 2: TICK [4]
FUN 2: TICK [5]
```

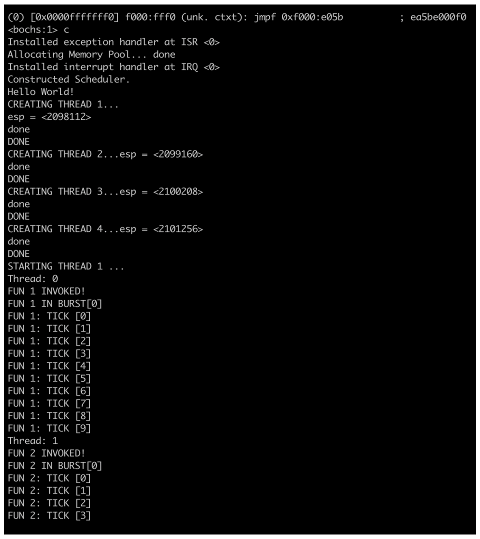


```
FUN 4: TICK [9]
FUN 1 IN BURST[2]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN BURST[2]
FUN 2: TICK [0]
FUN 2: TICK [1]
FUN 2: TICK [2]
FUN 2: TICK [3]
FUN 2: TICK [4]
FUN 2: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

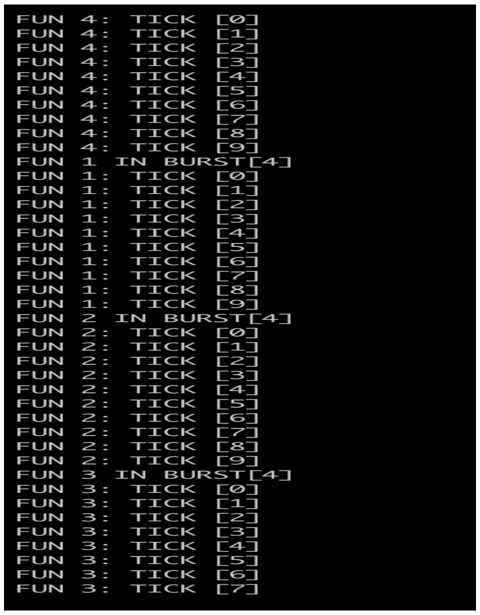
Expected Result: Each thread executes for 10 ticks before passing the CPU to the next thread in a continuous FIFO fashion.

Actual Result: Each thread executes for 10 ticks before passing the CPU to the next thread in a continuous FIFO fashion.

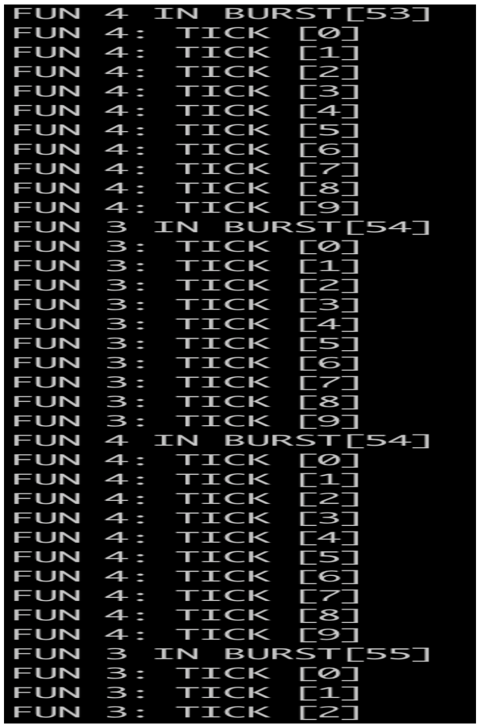
- Handling terminating threads by FIFO scheduling



```
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b         ; ea5be000f0
<bochs:1> c
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 2: TICK [3]
```



```
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 1 IN BURST[4]
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[4]
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[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]
```



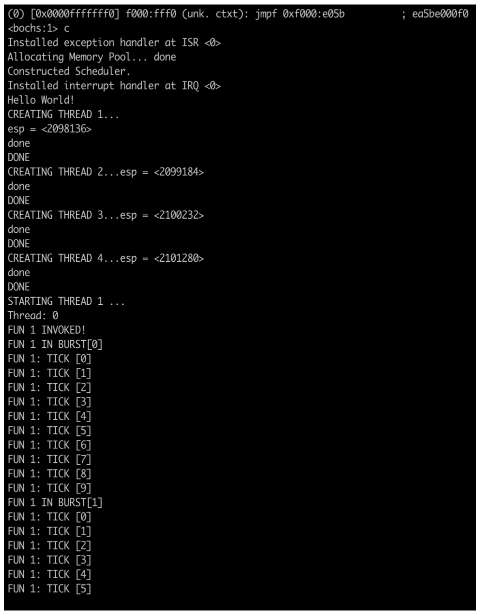
```
FUN 4 IN BURST[53]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[54]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[54]
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[55]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
```

Expected Result: Thread 1 and Thread 2 execute and terminate, while Thread 3 and Thread 4 continue to execute in a continuous FIFO fashion.

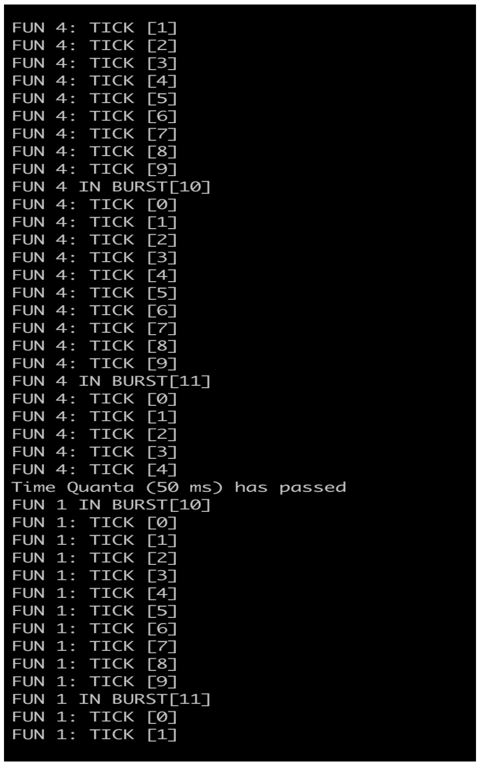
Actual Result: Thread 1 and Thread 2 execute and terminate, while Thread 3 and Thread 4 continue to execute in a continuous FIFO fashion.

NOTE: The thread numbering starts from 0 in the program(0,1,2,3).

- Handling non-terminating threads by RR scheduling



```
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b         ; ea5be000f0
<bochs:1> c
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 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
```

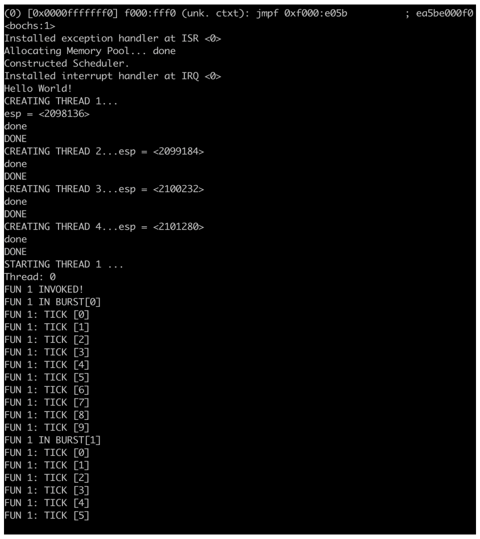


```
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[10]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
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]
Time Quanta (50 ms) has passed
FUN 1 IN BURST[10]
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[11]
FUN 1: TICK [0]
FUN 1: TICK [1]
```

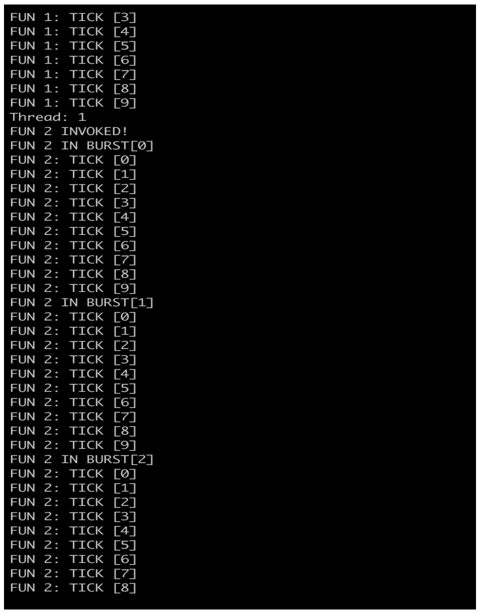
Expected Result: Each thread executes for 50 ms before being pre-empted by the Round-Robin scheduler, allowing the next thread to be executed.

Actual Result: Each thread executes for 50 ms before being pre-empted by the Round-Robin scheduler, allowing the next thread to be executed.

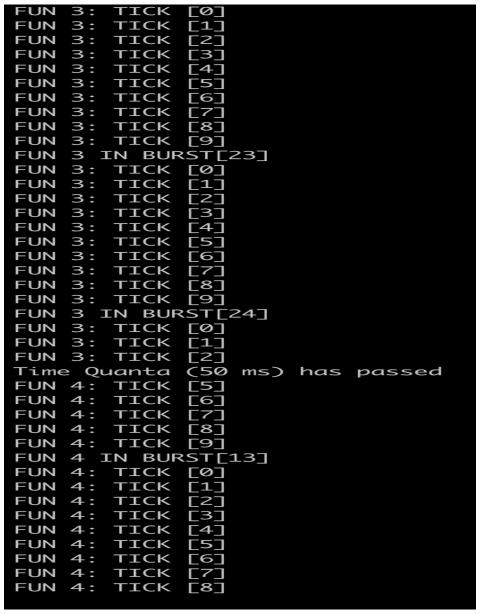
- Handling terminating threads by RR scheduling



```
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b         ; ea5be000f0
<bochs:1>
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 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 [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]
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]
FUN 2 IN BURST[2]
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 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[23]
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[24]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
Time Quanta (50 ms) has passed
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: 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]
```

Expected Result: Thread 1 and Thread 2 execute and terminate, while Thread 3 and Thread 4 continue to execute in a round-robin fashion, getting pre-empted every 50 ms by the scheduler.

Actual Result: Thread 1 and Thread 2 execute and terminate, while Thread 3 and Thread 4 continue to execute in a round-robin fashion, getting pre-empted every 50 ms by the scheduler.

NOTE: The thread numbering starts from 0 in the program(0,1,2,3).