MP5: Kernel-Level Thread Scheduling

Naveen Babu Krishnasamy Jeyakumar
UIN: 335007146

CSCE611: Operating System

Assigned Tasks:

Main: Kernel-Level Thread Scheduling **-** Completed.
Bonus1: Correct handling of interrupts **-** Completed.
Bonus2: Round-Robin Scheduling **-** Completed.
Files Modified: scheduler.H, scheduler.C, thread.C, kernel.C

System Design:

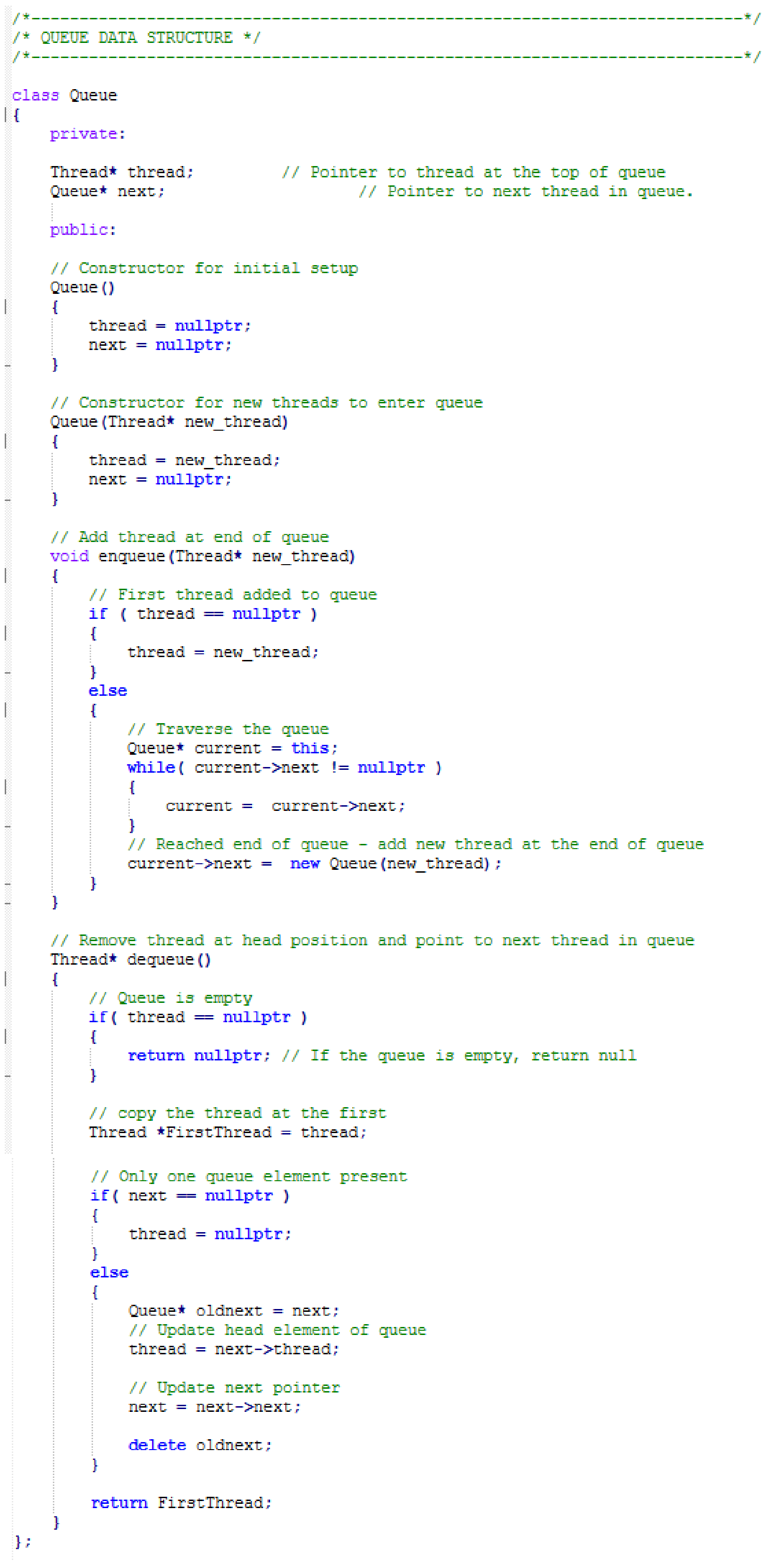
FIFO Scheduler: The FIFO Scheduler Implementation uses a Queue class to manage the ready queue of threads as a linked list. This class provides constructors and methods for enqueuing (adding) and dequeuing (removing) threads. When adding a thread, the queue checks if it's empty; if not, it traverses to the end to append the new thread. When removing, it dequeues the front thread and updates the pointer to the next thread. To handle terminating threads, the terminate method searches by thread ID to remove the specified thread from the queue. The thread_shutdown method releases memory for the terminated thread, and the yield method switches to the next thread in the queue.

Bonus 1 - Correct handling of interrupts: In the first bonus section on interrupt handling, mutual exclusion is achieved by enabling and disabling interrupts to prevent disruptions during queue modifications. Updates in scheduler.C ensure interrupts are disabled before queue operations and re-enabled afterward using interrupts_enabled, enable_interrupts, and disable_interrupts from machine.H.

Bonus 2 - Round-Robin Scheduling: The second bonus section details the round-robin scheduler, implemented in the RRScheduler class, which inherits from Scheduler and InterruptHandler. This class supports all Scheduler methods and includes a 50 ms time quantum managed by a timer-based interrupt with a tick frequency of 5. After 50 ms, the interrupt handler pre-empts the current thread and sends an End-of-Interrupt message to the master interrupt controller. The next thread in the ready queue is then dequeued and allocated CPU time, ensuring fair time-sharing across threads.

Code Description:

Scheduler.H - This code provides a thread scheduling system with FIFO and Round-Robin policies for managing kernel-level threads. The Queue class handles thread queuing using linked lists, supporting enqueue and dequeue operations. The Scheduler class implements basic FIFO scheduling, while the RRScheduler class extends it to a time-sliced Round-Robin approach, where threads receive CPU time in fixed intervals using a timer-based interrupt handler. The system ensures efficient thread management with controlled interrupt handling for mutual exclusion during queue operations.



```
/*---------------------------------------------------------------------------*/
/* 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 = nullptr;
        next = nullptr;
    }

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

    // Add thread at end of queue
    void enqueue(Thread* new_thread)
    {
        // First thread added to queue
        if ( thread == nullptr )
        {
            thread = new_thread;
        }
        else
        {
            // Traverse the queue
            Queue* current = this;
            while( current->next != nullptr )
            {
                current =  current->next;
            }
            // Reached end of queue - add new thread at the end of queue
            current->next =  new Queue(new_thread);
        }
    }

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

        // copy the thread at the first
        Thread *FirstThread = thread;

        // Only one queue element present
        if( next == nullptr )
        {
            thread = nullptr;
        }
        else
        {
            Queue* oldnext = next;
            // Update head element of queue
            thread = next->thread;

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

            delete oldnext;
        }

        return FirstThread;
    }
};
```

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

class Scheduler {

private:

  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.*/

};
/*--------------------------------------------------------------------------*/
/* ROUND ROBIN SCHEDULER */
/*--------------------------------------------------------------------------*/

// Inherited Scheduler and Interrupt Handler classes
class RRScheduler: public Scheduler, public InterruptHandler
{
    Queue rr_ready_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

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

    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 to do the context switch. */

    virtual void resume(Thread * _thread);
    /* Add the given thread to the ready queue of the RRScheduler. 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. */

    virtual void terminate(Thread * _thread);
    /* Remove 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 method. */
};

#endif
```

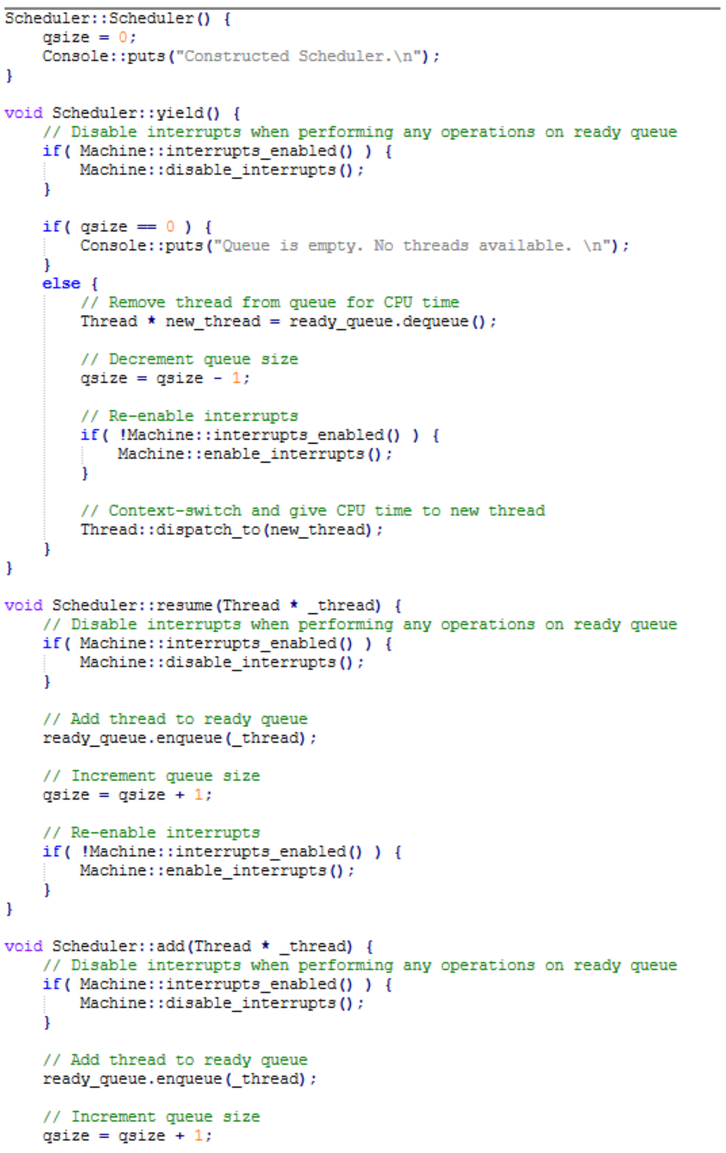
scheduler.C : Scheduler() - This method is the constructor for the Scheduler class, initializing the FIFO queue size to zero

scheduler.C : yield() - This method removes a thread from the queue, disables interrupts to prevent interference, and context-switches to the dequeued thread if the queue isn't empty. If the queue is empty, it logs a message.

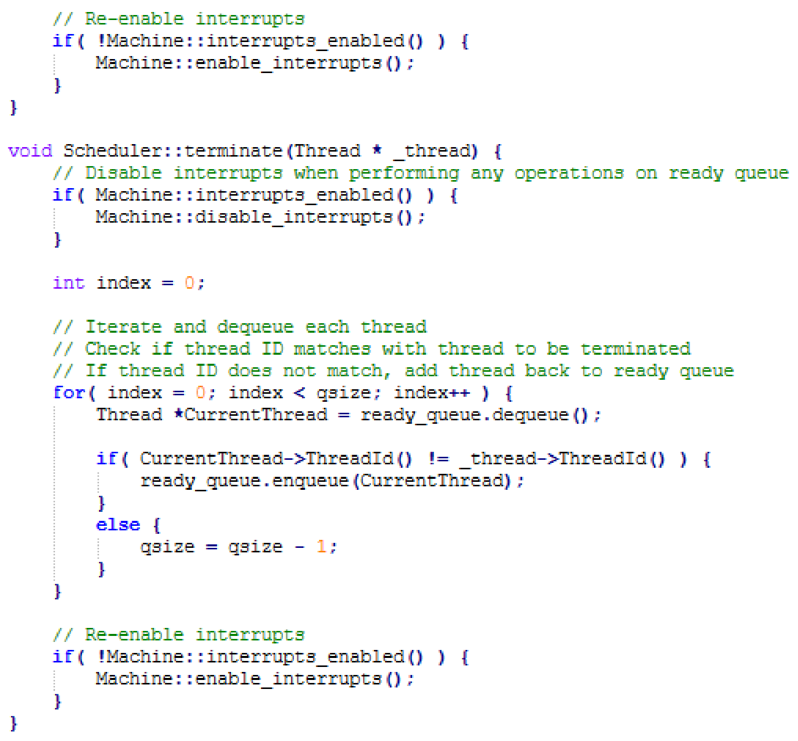
scheduler.C : resume() - This method adds a new thread to the queue, incrementing the queue size, while also disabling interrupts for safe operation and re-enabling them afterward.

scheduler.C : add() - This method enqueues a thread to the ready queue, incrementing the queue size while ensuring safe operations by disabling and re-enabling interrupts.

scheduler.C : terminate() - This method searches for a specific thread to remove by ID, iteratively dequeuing threads, and re-enqueues unmatched threads to maintain order, also protecting the process with interrupt control.



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

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);
    }
}

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();
    }
}

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();
    }
}

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 *CurrentThread = ready_queue.dequeue();

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

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

scheduler.C : RRScheduler::RRScheduler() - Initializes the Round-Robin scheduler with default values, sets a 50 ms time quantum by installing an interrupt handler, and configures the timer frequency.

scheduler.C : RRScheduler::set_frequency - Sets the timer frequency by calculating and loading the divisor for a 1.19 MHz input clock, ensuring regular timer interrupts based on the specified frequency.

scheduler.C : RRScheduler::yield() - Handles thread yielding by pre-empting the current thread, resetting the tick count, dequeuing the next thread, and performing a context switch.

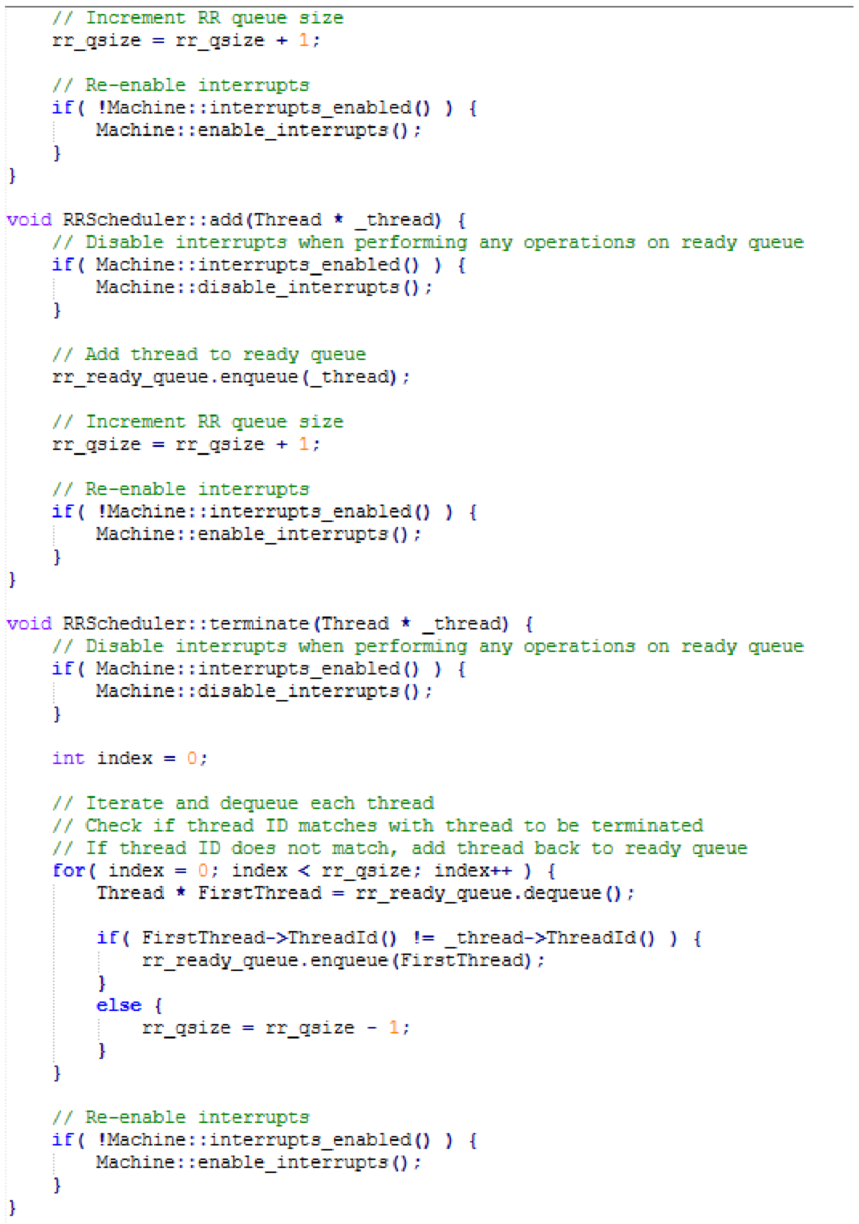
scheduler.C : RRScheduler::resume() - Adds a thread to the ready queue and increments the queue size, ensuring mutual exclusion by temporarily disabling interrupts.

scheduler.C : RRScheduler::add() - Adds a new thread to the Round-Robin ready queue, similarly managing the queue size and protecting operations with interrupt control.

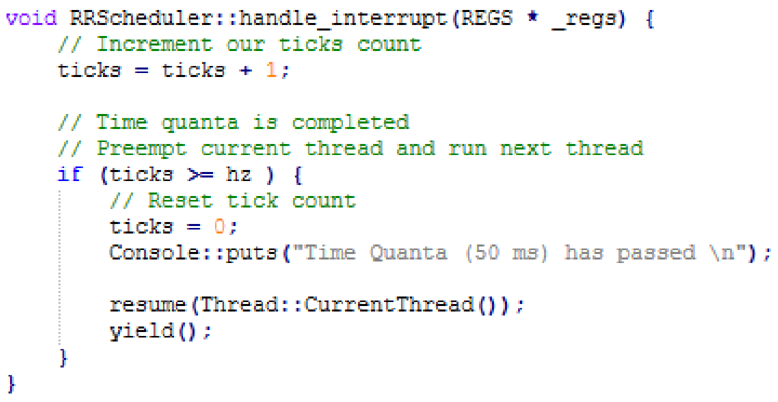
scheduler.C : RRScheduler::terminate() - Locates and removes a specific thread from the queue by its ID, maintaining safe queue operations with interrupt handling.

scheduler.C : RRScheduler::handle_interrupt() - Increments the tick count and checks for quantum expiration; if reached, it resets ticks, pre-empts the current thread, and triggers a context switch to the next thread in the queue.

```
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);
}

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
}

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 = rr_ready_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);
    }
}

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
    rr_ready_queue.enqueue(_thread);
```



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

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

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
    rr_ready_queue.enqueue(_thread);

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

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

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 * FirstThread = rr_ready_queue.dequeue();

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

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



```
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();
    }
}
```

thread.C : thread_shutdown() - The thread_shutdown function safely terminates the currently running thread by releasing its resources, including memory management and interaction with the system scheduler. After terminating the thread, it deletes the thread instance and yields control to the next thread in the 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();
}
```

thread.C : thread_start() - The thread_start function prepares a thread for execution by enabling interrupts, allowing it to enter the ready queue and participate in the scheduling process.

```
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();
}
```

kernel.C (Code Changes for Testing) - The following modifications were implemented in kernel.C for testing purposes: a new macro, _USES_RR_SCHEDULER, was introduced to facilitate Round-Robin scheduling support, and the code was encapsulated to enable Round-Robin scheduling functionality when _USES_RR_SCHEDULER is defined.

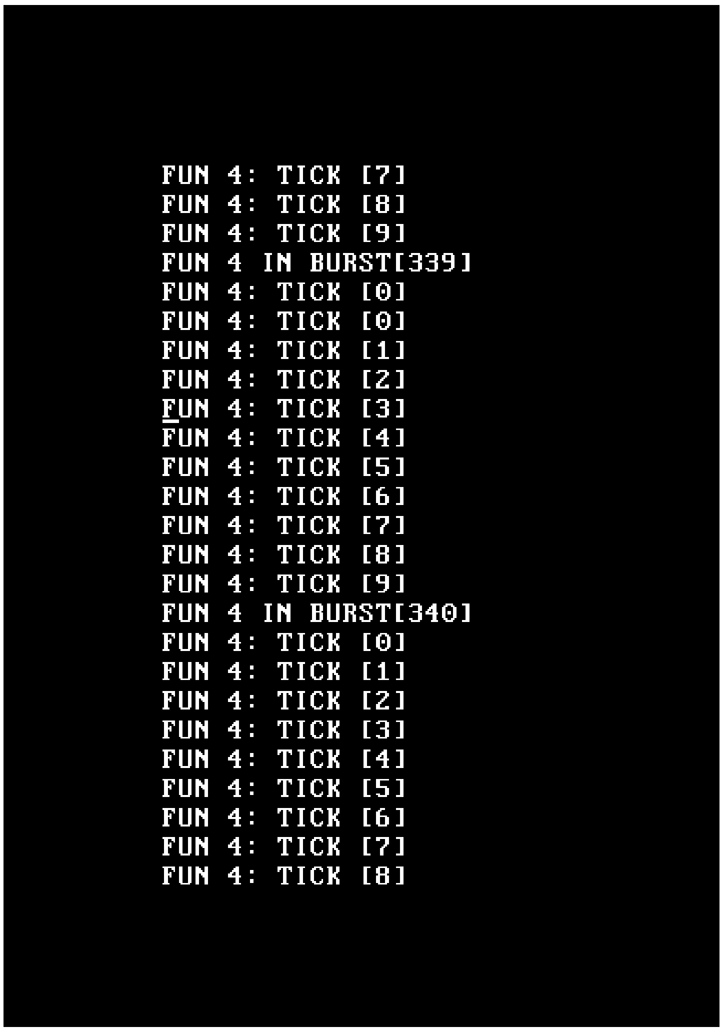
```
#define _USES_RR_SCHEDULER_
/* This macro is defined when we want to force the code below to use
   round-robin based scheduler.
   Otherwise, a First-In-First-Out scheduler is used, Round-Robin scheduling
   is supported only when _USES_SCHEDULER_ is defined.
*/
```

```
#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_SCHEDULER_
#ifdef  _USES_RR_SCHEDULER_
    SYSTEM_SCHEDULER = new RRScheduler();
#else
    SYSTEM_SCHEDULER = new Scheduler();
#endif
#endif /* _USES_SCHEDULER_ */
```

Testing and Output:

Kernel-level thread scheduling was tested with various scenarios. Changes were made in the kernel.C file to support these FIFO and Round-Robin scheduling tests and verify the results. In the first test, using FIFO scheduling for non-terminating threads, each thread ran for 10 ticks before passing the CPU to the next, working as expected. In the second test, with FIFO scheduling for terminating threads, Threads 1 and 2 ran and terminated, allowing Threads 3 and 4 to continue in FIFO order, also as expected. The third test used Round-Robin scheduling for non-terminating threads, where each thread ran for 50 ms before being pre-empted, which behaved correctly. Finally, in the fourth test, Round-Robin scheduling for terminating threads had Threads 1 and 2 run and terminate, while Threads 3 and 4 continued in a round-robin manner, pre-empted every 50 ms, matching the expected outcome.



```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 4 IN BURST[339]
FUN 4: TICK [0]
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[340]
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]
```