MP5: Simple Kernel Threads

Sathvik Kote Revanasiddappa
UIN: 634005706
CSCE611: Operating System

Assigned Tasks

Main: Add scheduling of multiple kernel-level threads : Completed.

Bonus:

- Correct handling of interrupts: Completed
- Round-Robin Scheduling: Completed
- Processes: Did not attempt.

System Design

Objective

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: The first come first scheduler is implemented using a queue. The Queue holds the threads in the order of their addition. The scheduler holds two pointers head and tail pointing to the start and end of the queue respectively. When a thread is added, a node is created in a queue and pushed at the tail end. Whenever the scheduler schedules a thread, it pops a thread from the head.
- Terminating Functions: To gracefully terminate a thread, we have to delete all the space allocated to the thread and remove any information present in the ready queue. When a scheduler picks a thread from the ready queue, it pops a node form the head. It treats every thread thats dispatched as terminating thread and hence as no information regarding a thread.
- Interrupt Handling: Interrupt handlers are disabled to ensure mutual exclusion on ready queue. Enabling and disabling of threads were achieved using the interrupts_enabled, enable_interrupts and disable_interrupts methods provided in machine.H.
- Round Robin Scheduler: This scheduler inherits properties of FCFS scheduler and simple timer. A Interrupt has to be raised after every 50 ms to dispatch a different thread. When a new thread is dispatched the ticks are reset to 0 to ensure equal time quantum to all threads. Before dispatching a new thread, an end of Interrupt message is sent to the master Interrupt controller.

Implementation Details

Mac Support

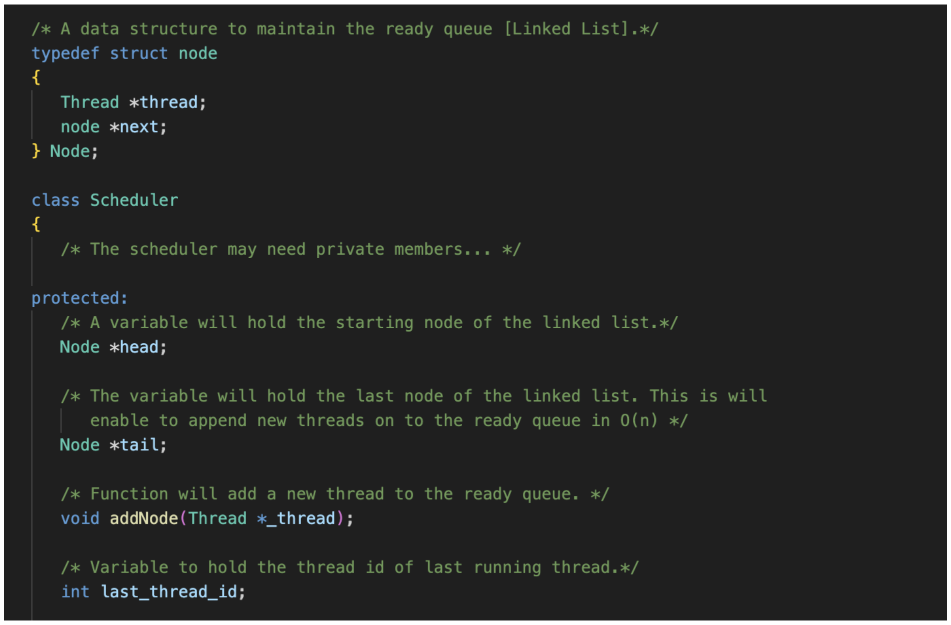
I have modified the below files to support the functioning of MP3 problem on Mac M2.

- copykernel.sh
- bochsrc.bxrc

Scheduler.H

FCFS Scheduler is implemented using a Queue. Queue is implemented using a linked list. A node in a linked list holds a thread and link to the next node.

- struct node: Represents a node in a linked list.
- Node *head: A variable will hold the starting node of the queue.
- Node *tail: The variable will point to the last node of the linked list. This is will enable to append new threads on to the ready queue in O(1).
- void addNode(Thread *_thread): Function will add a new thread to the ready queue.
- last_thread_id: Variable to hold the thread id of last running thread.

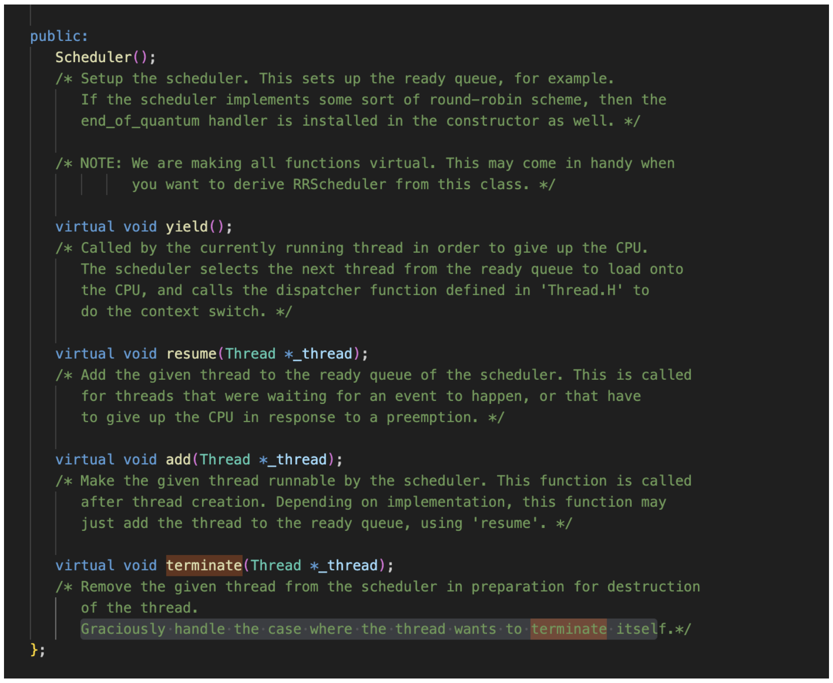


```
/* A data structure to maintain the ready queue [Linked List].*/
typedef struct node
{
   Thread *thread;
   node *next;
} Node;

class Scheduler
{
   /* The scheduler may need private members... */

protected:
   /* A variable will hold the starting node of the linked list.*/
   Node *head;

   /* The variable will hold the last node of the linked list. This is will
      enable to append new threads on to the ready queue in O(n) */
   Node *tail;

   /* Function will add a new thread to the ready queue. */
   void addNode(Thread *_thread);

   /* Variable to hold the thread id of last running thread.*/
   int last_thread_id;
```

Figure 1: Scheduler Variables

- Scheduler(): Constructor for the scheduler.
- 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): Graciously handle the case where the thread wants to terminate itself.

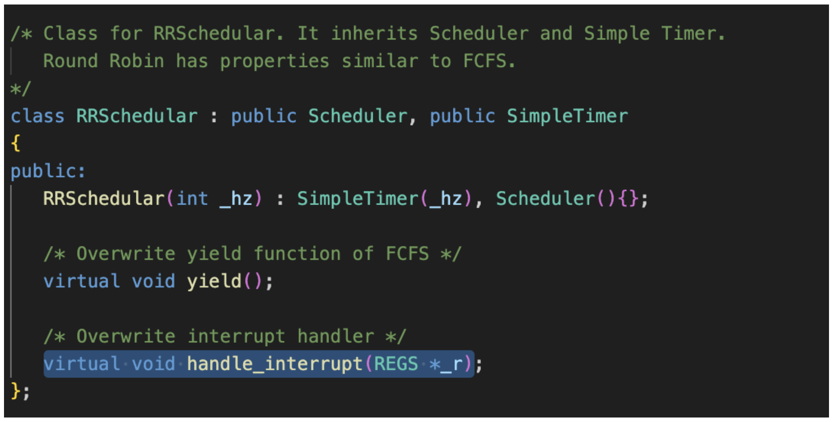


```
public:
  Scheduler();
  /* Setup the scheduler. This sets up the ready queue, for example.
     If the scheduler implements some sort of round-robin scheme, then the
     end_of_quantum handler is installed in the constructor as well. */

  /* 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.*/
};
```

Figure 2: Scheduler methods

Along with the FCFC scheduler, scheduler.H has RR Scheduler. This scheduler, as described before inherits FCFS scheduler and Simple Timer.

- RRSchedular(int _hz) : SimpleTimer(_hz), Scheduler(): It takes hertz as an input, which is passed to the parent claass simple timer.
- virtual void yield(): Override the parent implementation.
- virtual void handle_interrupt(REGS *_r): Override the parent implementation.



```
/* Class for RRSchedular. It inherits Scheduler and Simple Timer.
   Round Robin has properties similar to FCFS.
*/
class RRSchedular : public Scheduler, public SimpleTimer
{
public:
   RRSchedular(int _hz) : SimpleTimer(_hz), Scheduler(){};

   /* Overwrite yield function of FCFS */
   virtual void yield();

   /* Overwrite interrupt handler */
   virtual void handle_interrupt(REGS *_r);
};
```

Figure 3: RR Scheduler

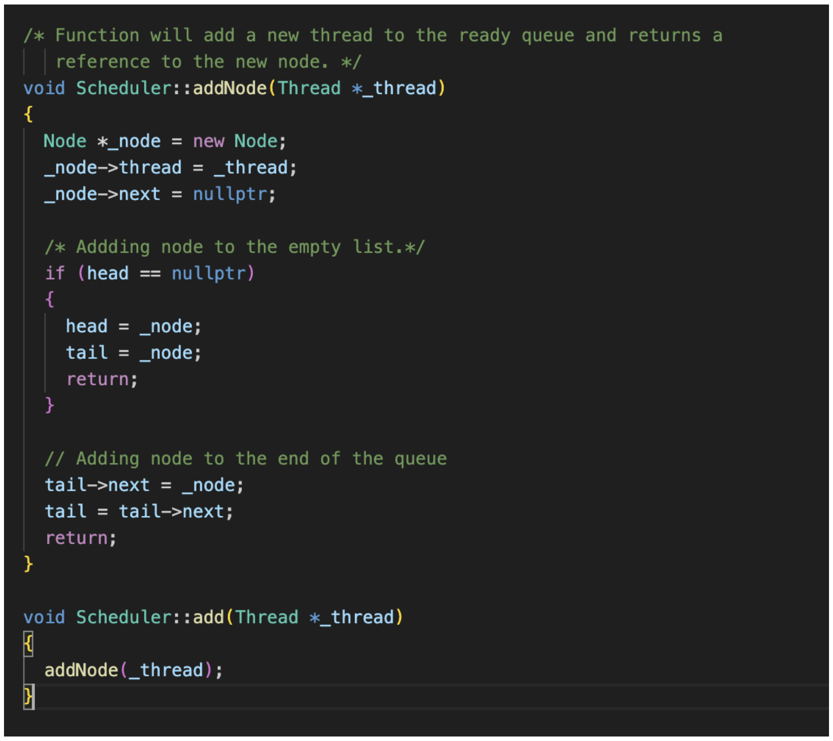
Scheduler.C

The scheduler has an private function called addNode. The function will create a node using a given thread pointer and add the node to the queue.

The yield function will find the first node in the raady queue and removes it from the queue. The selected node is deleted from the queue and its thread is dispacted.

The resume method is used to pause an thread execution and add it to the ready queue. It adds the thread to the queue and notes it ID in last_thread_id. This although not used in this implementation, is useful when the threads are preempted.

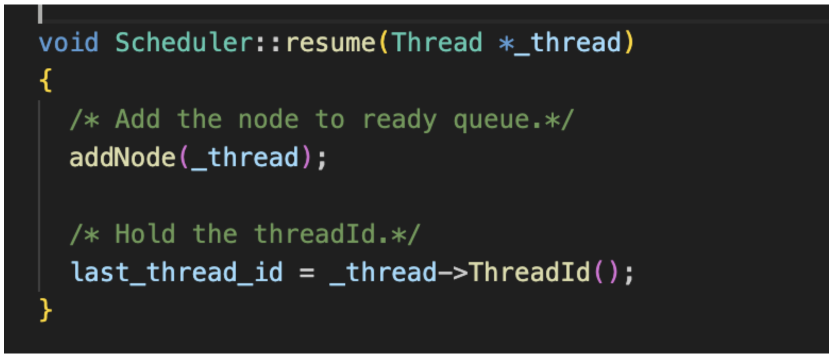
- void Scheduler::addNode(Thread *_thread): The method creates an node and add it to the queue. If the queue is empty, it will update both head and tail. But if it's empty, it will add to tail and update it.



```
/* Function will add a new thread to the ready queue and returns a
   reference to the new node. */
void Scheduler::addNode(Thread *_thread)
{
  Node *_node = new Node;
  _node->thread = _thread;
  _node->next = nullptr;

  /* Addding node to the empty list.*/
  if (head == nullptr)
  {
    head = _node;
    tail = _node;
    return;
  }

  // Adding node to the end of the queue
  tail->next = _node;
  tail = tail->next;
  return;
}

void Scheduler::add(Thread *_thread)
{
  addNode(_thread);
}
```

Figure 4: Scheduler Add Node

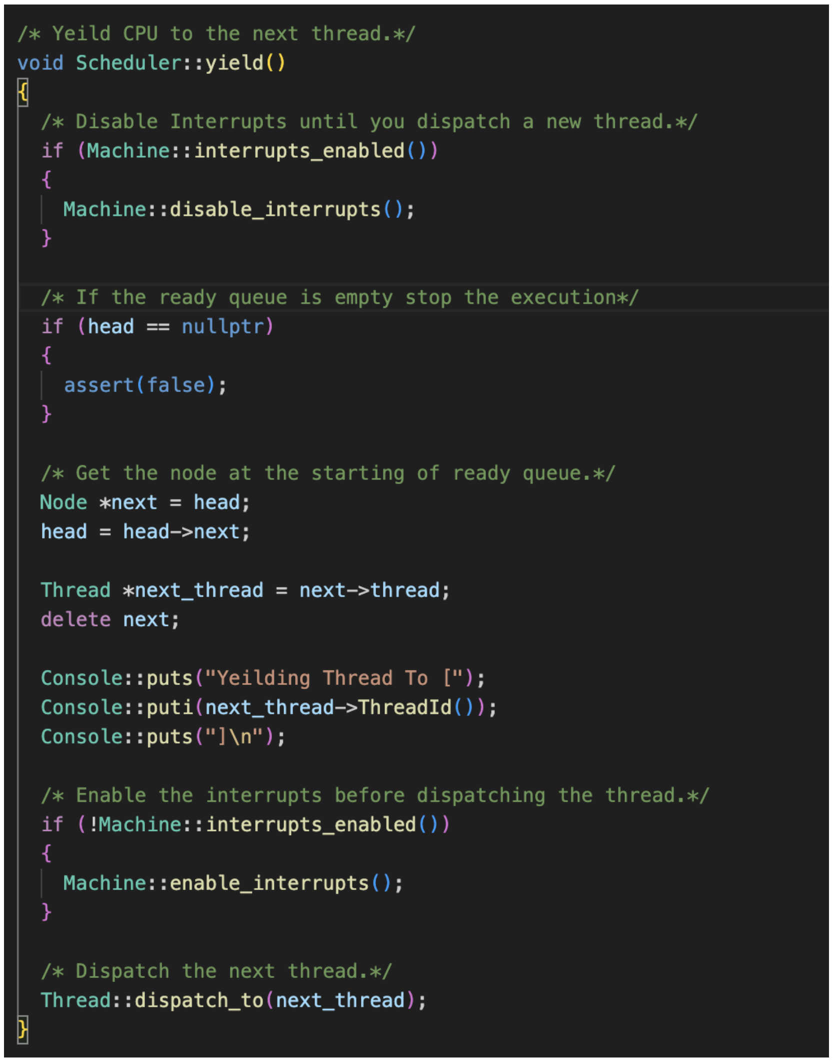
- void Scheduler::resume(Thread *_thread): The method will add the thread to the end of the queue.



```
void Scheduler::resume(Thread *_thread)
{
  /* Add the node to ready queue.*/
  addNode(_thread);

  /* Hold the threadId.*/
  last_thread_id = _thread->ThreadId();
}
```

Figure 5: Scheduler resume

- void Scheduler::yield(): The method will select the first node in the ready queue and delete it from the list. The selected node is then dispacted. In this case, if the queue is empty. The functionality is halted. Disable the interrupts before altering the ready queue.



```
/* Yeild CPU to the next thread.*/
void Scheduler::yield()
{
  /* Disable Interrupts until you dispatch a new thread.*/
  if (Machine::interrupts_enabled())
  {
    Machine::disable_interrupts();
  }

  /* If the ready queue is empty stop the execution*/
  if (head == nullptr)
  {
    assert(false);
  }

  /* Get the node at the starting of ready queue.*/
  Node *next = head;
  head = head->next;

  Thread *next_thread = next->thread;
  delete next;

  Console::puts("Yeilding Thread To [");
  Console::puti(next_thread->ThreadId());
  Console::puts("]\n");

  /* Enable the interrupts before dispatching the thread.*/
  if (!Machine::interrupts_enabled())
  {
    Machine::enable_interrupts();
  }

  /* Dispatch the next thread.*/
  Thread::dispatch_to(next_thread);
}
```

Figure 6: Scheduler yield

- void Scheduler::terminate(Thread *_thread): Because of the way the scheduler is implemented, there is no requirement of this method.Every thread that is dispacted from the scheduler is considered as terminate node and hence removed from the queue.

RRSchedular

RRSchedular Overrides two methods of its parents classes. Handle interrupt from SimpleTimer and yield from FCFS Scheduler.

- void RRSchedular::handle_interrupt(REGS *_r): It carries most of the functionality from the parent. Along with that, the scheduler will resume the current thread and yield the execution to a new thread.
- void RRSchedular::yield(): This method sends and EOI message to master interrupt controller. It also disables all the interrupts before handling datastructures related to the scheduler. Similar to the FCFS scheduler, it dispactes the first node in the ready queue. If the queue is empty, then it dispactes an ideal thread.

thread.C

- void Thread::shutdow_thread(): Identify space allocated to the thread and free it.

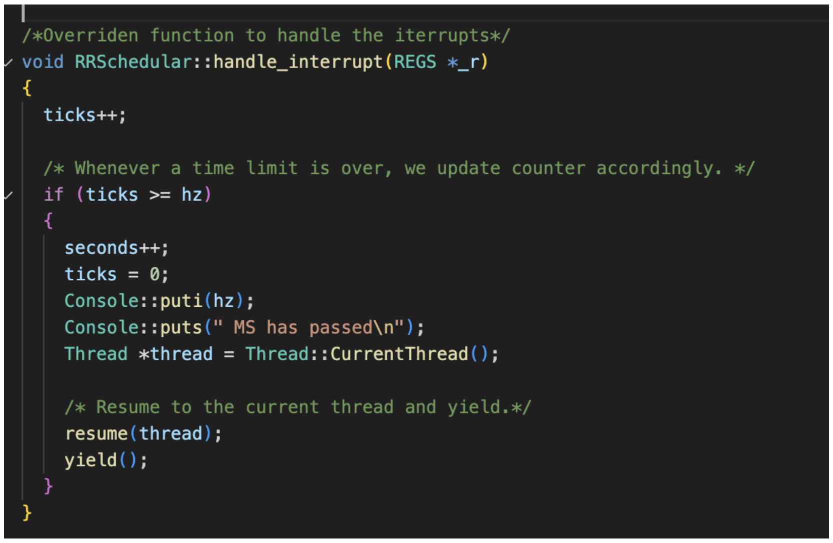

```
/*Overriden function to handle the iterrupts*/
void RRSchedular::handle_interrupt(REGS *_r)
{
    ticks++;

    /* Whenever a time limit is over, we update counter accordingly. */
    if (ticks >= hz)
    {
        seconds++;
        ticks = 0;
        Console::puti(hz);
        Console::puts(" MS has passed\n");
        Thread *thread = Thread::CurrentThread();

        /* Resume to the current thread and yield.*/
        resume(thread);
        yield();
    }
}
```

Figure 7: RR Scheduler Handle Interrupt

- static void thread_start(): Enable the interrupts before the execution of the thread.

kernel.c

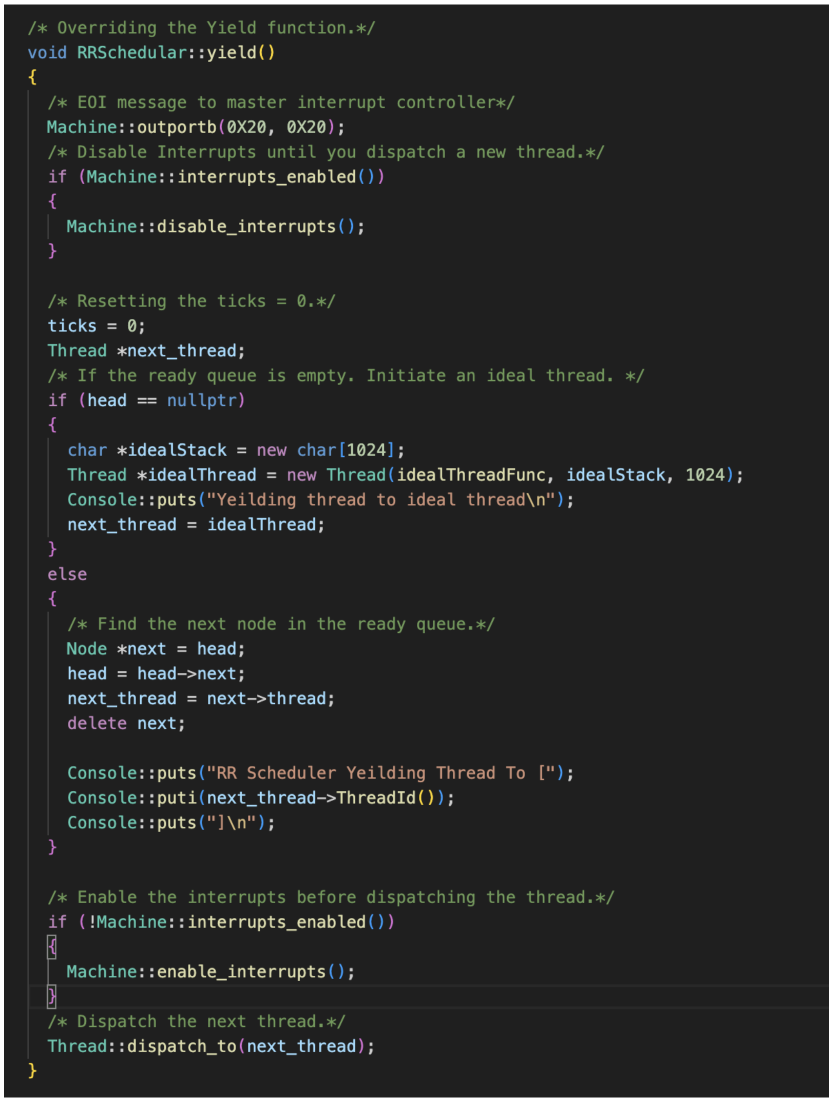
Added macros for supporting RR Scheduler. Along with that added an macro which helps in additional testing of macro.

Testing

All the logs for each test case is present in output.txt, rroutput.txt and rr_additional_output.txt.

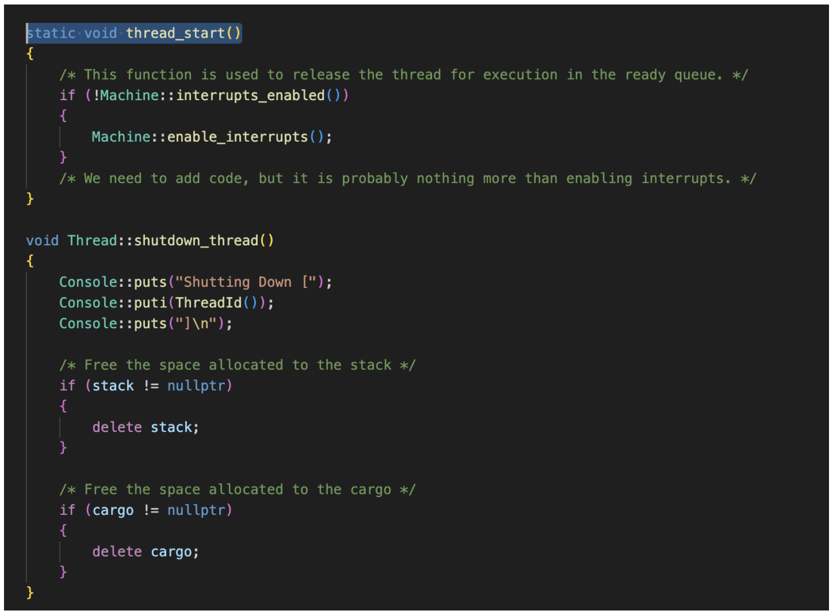
- FIFO Scheduling with terminating function("_USES_SCHEDULER_" and "_TERMINATING_FUNCTIONS_" defined) : Thread 1 and Thread 2 executes and terminates. Thread 3 and 4 continues executing. Works as expected.
- RR Scheduling with given threads: ("_RR_SCHEDULER_" and "_TERMINATING_FUNCTIONS_" defined): Thread 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.
- RR Scheduler with all terminating functions and ideal threads ("_RR_SCHEDULER_", "_RR_SCHEDULER_TESTIN "_TERMINATING_FUNCTIONS_" defined)

All tests are passing.



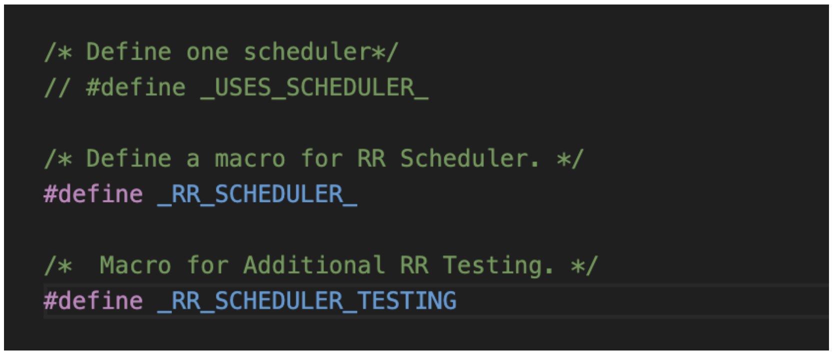
```
/* Overriding the Yield function.*/
void RRSchedular::yield()
{
  /* EOI message to master interrupt controller*/
  Machine::outportb(0X20, 0X20);
  /* Disable Interrupts until you dispatch a new thread.*/
  if (Machine::interrupts_enabled())
  {
    Machine::disable_interrupts();
  }

  /* Resetting the ticks = 0.*/
  ticks = 0;
  Thread *next_thread;
  /* If the ready queue is empty. Initiate an ideal thread. */
  if (head == nullptr)
  {
    char *idealStack = new char[1024];
    Thread *idealThread = new Thread(idealThreadFunc, idealStack, 1024);
    Console::puts("Yeilding thread to ideal thread\n");
    next_thread = idealThread;
  }
  else
  {
    /* Find the next node in the ready queue.*/
    Node *next = head;
    head = head->next;
    next_thread = next->thread;
    delete next;

    Console::puts("RR Scheduler Yeilding Thread To [");
    Console::puti(next_thread->ThreadId());
    Console::puts("]\n");
  }

  /* Enable the interrupts before dispatching the thread.*/
  if (!Machine::interrupts_enabled())
  {
    Machine::enable_interrupts();
  }
  /* Dispatch the next thread.*/
  Thread::dispatch_to(next_thread);
}
```

Figure 8: RR Scheduler Yield



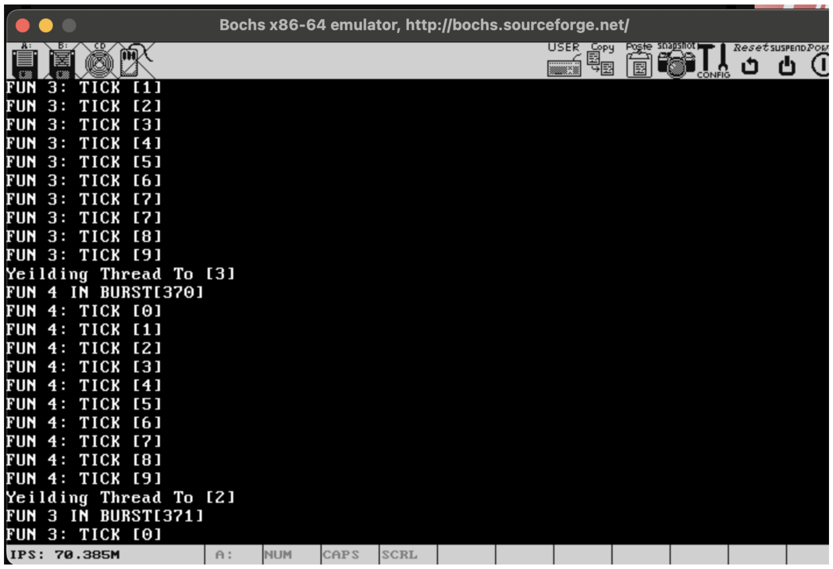
```
static void thread_start()
{
    /* This function is used to release the thread for execution in the ready queue. */
    if (!Machine::interrupts_enabled())
    {
        Machine::enable_interrupts();
    }
    /* We need to add code, but it is probably nothing more than enabling interrupts. */
}

void Thread::shutdown_thread()
{
    Console::puts("Shutting Down [");
    Console::puti(ThreadId());
    Console::puts("]\n");

    /* Free the space allocated to the stack */
    if (stack != nullptr)
    {
        delete stack;
    }

    /* Free the space allocated to the cargo */
    if (cargo != nullptr)
    {
        delete cargo;
    }
}
```

Figure 9: Thread



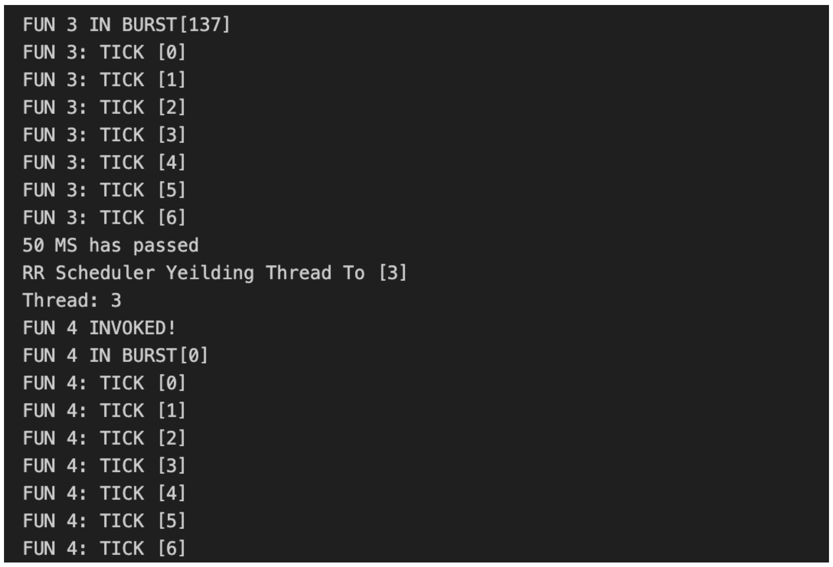
```
/* Define one scheduler*/
// #define _USES_SCHEDULER_

/* Define a macro for RR Scheduler. */
#define _RR_SCHEDULER_

/*  Macro for Additional RR Testing. */
#define _RR_SCHEDULER_TESTING
```

Figure 10: Kernel Macro



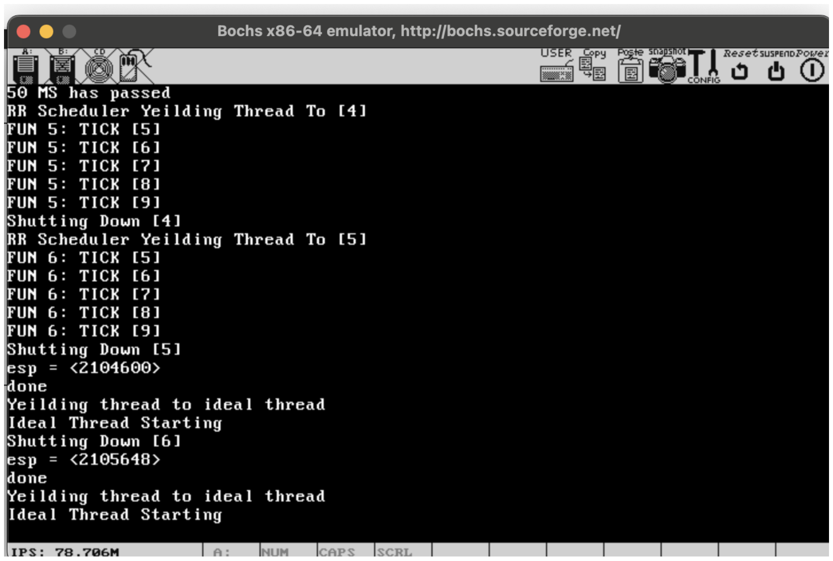
```
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
Yeilding Thread To [3]
FUN 4 IN BURST[370]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
Yeilding Thread To [2]
FUN 3 IN BURST[371]
FUN 3: TICK [0]
IPS: 70.385M    A:   NUM   CAPS   SCRL
```

Figure 11: FCFS Results



```
FUN 3 IN BURST[137]
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]
50 MS has passed
RR Scheduler Yeilding Thread To [3]
Thread: 3
FUN 4 INVOKED!
FUN 4 IN BURST[0]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
```

Figure 12: RR Results



Bochs x86-64 emulator, http://bochs.sourceforge.net/

```
50 MS has passed
RR Scheduler Yeilding Thread To [4]
FUN 5: TICK [5]
FUN 5: TICK [6]
FUN 5: TICK [7]
FUN 5: TICK [8]
FUN 5: TICK [9]
Shutting Down [4]
RR Scheduler Yeilding Thread To [5]
FUN 6: TICK [5]
FUN 6: TICK [6]
FUN 6: TICK [7]
FUN 6: TICK [8]
FUN 6: TICK [9]
Shutting Down [5]
esp = <2104600>
done
Yeilding thread to ideal thread
Ideal Thread Starting
Shutting Down [6]
esp = <2105648>
done
Yeilding thread to ideal thread
Ideal Thread Starting
```

Figure 13: RR additional Results