CSCE 611: Operating Systems

Fall 2022

Machine Problem 5: Kernel-Level Thread Scheduling

Name: Rajesh Satpathy

UIN: 931006158

Goal:

Implement a thread scheduler that can handle multiple kernel-level threads for the given kernel.

Problem Statement:

The scheduler has to be implemented by implementing thread addition, yielding, resuming, and terminating. The scheduler should have a ready queue that can modify the thread and control the block.

Scope

- Complete the basic functionalities of the thread scheduler to hand control from one thread to another.
- Handle the terminating of threads.
- Enable and Disable interrupts to ensure mutual exclusion of threads. (Option 1)
- The idea behind extending the solution to a Round Robin Scheduler. (Option 2)

Implementation:

The scheduler is implemented using a custom class of FIFO Queue that is implemented at scheduler.H. The FIFO Queue is basically a linked list that supports enqueue and dequeue operations. The thread scheduler supports the following functions in scheduler.C:

- **Constructor:** Sets up the scheduler. Initializes the FIFO Queue and set the queue size to 0.
- **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. Dequeues the thread from the FIFO queue.
- **resume:** 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. Adds the thread to the FIFO queue.
- **add:** Make the given thread runnable by the scheduler. This function is called after thread creation. This function just adds the thread to the ready queue, using 'resume'. Adds the thread to the FIFO queue.
- **terminate:** Remove the given thread from the scheduler in preparation for the destruction of the thread. This uses the yield function to handle self-termination.

An extern System Scheduler is added to thread.C. Thread functions are updated in thread.C file as follows:

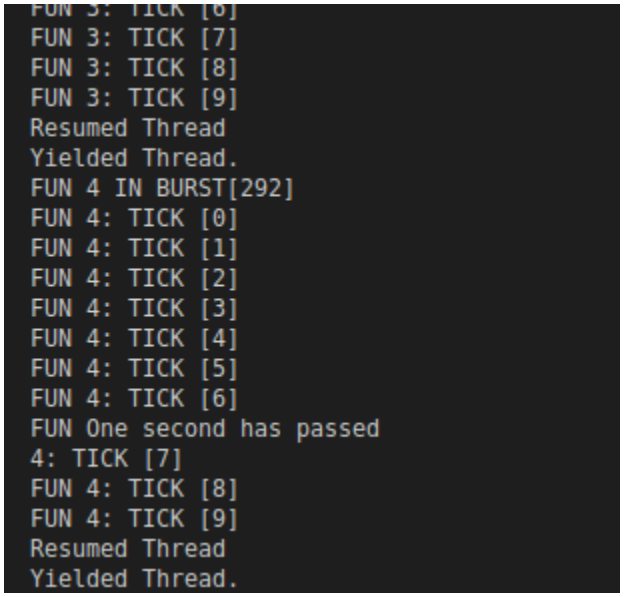
- **thread_shutdown:** This function is called when the thread returns from the thread function. It terminates the thread by releasing memory through the thread scheduler and deleting it.
- **thread_start:** This function mainly just enables interrupts, so that thread gets added when interrupt is raised for a new thread.

Other functions are already implemented.

Option 1: The interrupts are correctly handled for enabling mutual exclusion by,

1. Disabling interrupts whenever there is a thread add, resume, yield, or terminate operation on the scheduler's FIFO queue.
2. Enabling interrupts after the thread operation is done.

These steps are mainly followed to make operations within the functions not be affected by other threads that might try accessing common resources by raising interrupts. We can see in the image below that the periodic clock update is functioning as expected.



Option 2: Round Robin Scheduling

I could not implement this part because of facing errors I could not resolve. Hence, I am presenting the idea for the implementation:

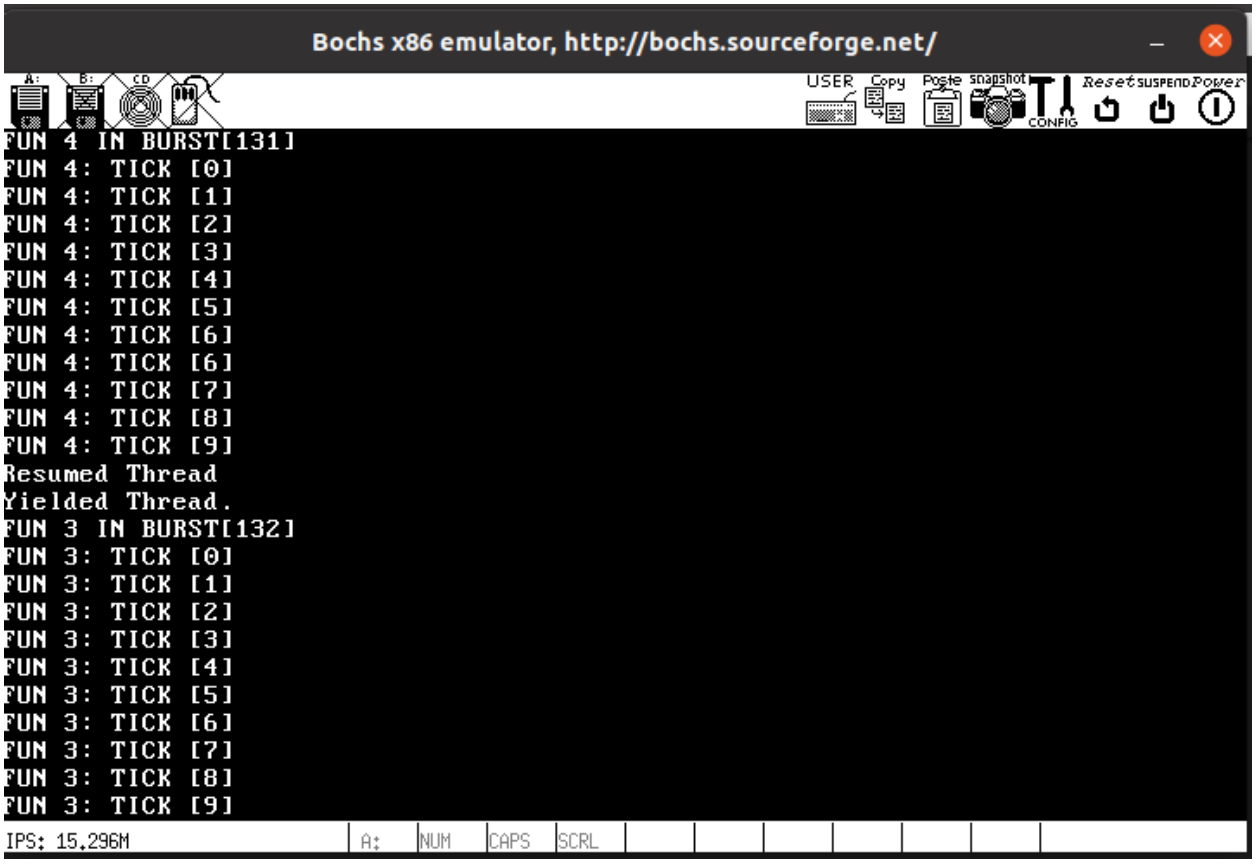
1. Create a derived class in scheduler.H RRSCheduler.
2. The derived class contains variables, set_quantum_time (To set the quantum)
3. An extension of the interrupt handler to raise an interrupt every time the quantum hits the set quantum time and passes control to the next thread.

4. scheduler.C implements the methods for RRSCheduler::set_quantum_time() and RRSCheduler::interrupt_handler(Regs *r).
5. A #define RRScheduler is added to kernel.C to check the implementation of the Round Robin Scheduler.

Design Decisions:

FIFO Queue: A data structure of a linked list is created to a mock FIFO queue that is flexible to change in the Queue size.

Output from testing:



It is observed in the output that threads FUN 1 and FUN 2 are terminated after a while and only threads FUN 3 and FUN 4 keep running after the #define _TERMINATING_FUNCTIONS_ is uncommented.