Anuska Pant
CSCE 611 - OPERATING SYSTEMS

Machine Problem 5: Kernel-Level Thread Scheduling

Bonus: Option 1 and Option 2

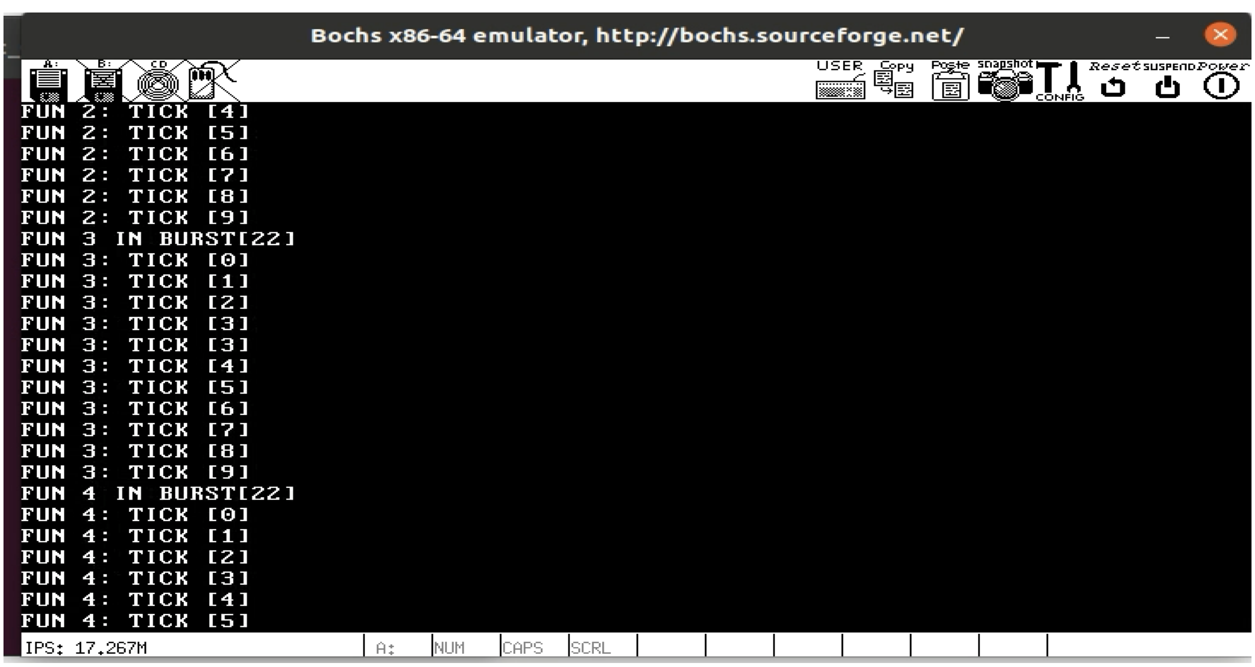
Introduction

The objective of this machine problem was to create scheduling of multiple kernel-level threads. We are to create a scheduler that allocates the CPU on behalf of the running threads.

Implementation of Scheduler

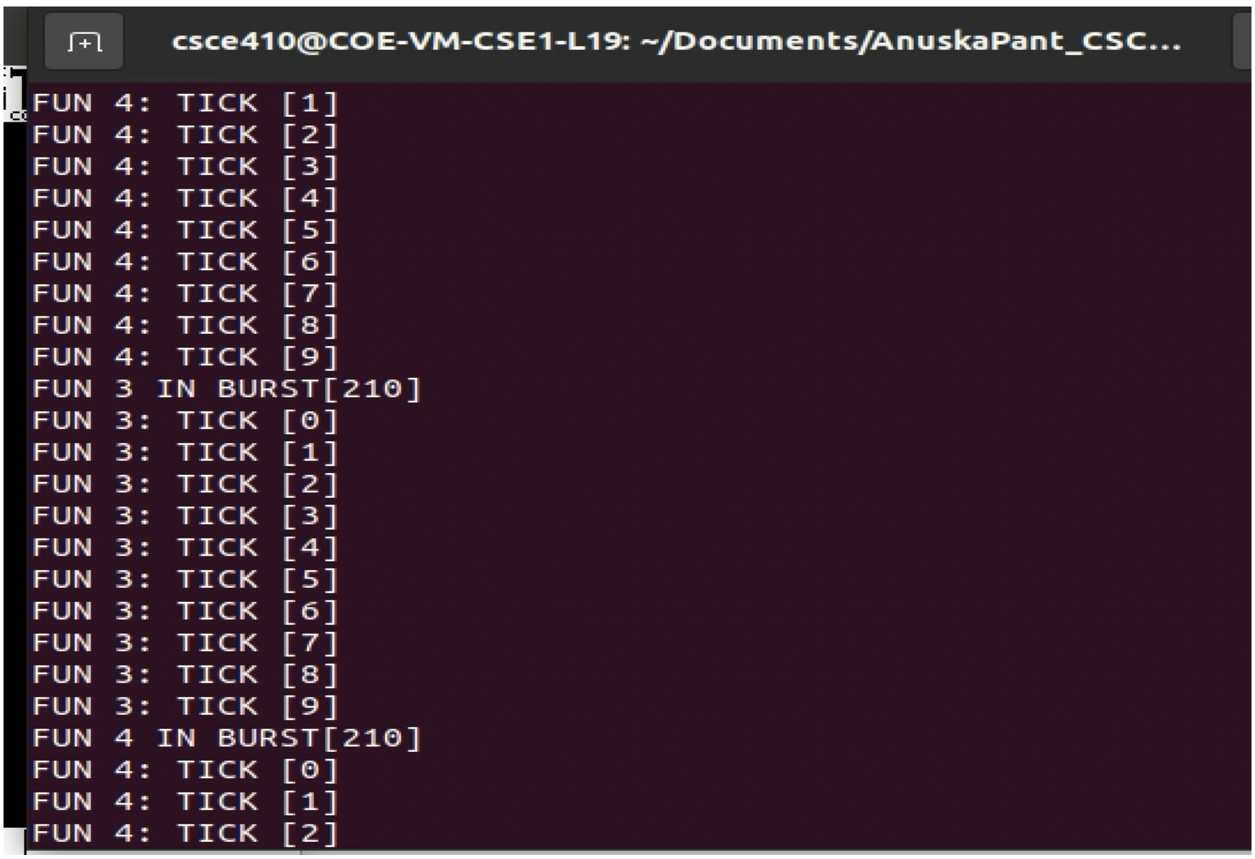
Initially I created a queue data structure for the threads and it uses FIFO methodology. We have functionalities like enqueue and dequeue to add and remove threads from the queue which is used later to implement the scheduler methods like add, resume, yield and terminate.

Yield() : When the running thread yields we dequeue the top most thread from the queue and dispatch it when the queue isn't empty.
Resume(): To resume a thread we enqueue it back to the queue. Add() is implemented similarly.
The output after enabling macro _USES_SCHEDULER_, where the four threads run for infinite bursts one after the other running for 10 ticks each.



Terminating Thread Functions

I implemented the terminate function to allow for threads to terminate after the function returns. The thread is searched in the ready queue and when the thread with the matching id is found it's removed from the queue. The whole process involves a series of dequeue and enqueue until the required thread is removed. After this we remove the memory allocated to the thread and call yield to dispatch the next thread.
The output after enabling macro _TERMINATING_FUNCTIONS_. Here the threads 1 and 2 terminate after 10 bursts and thread 3 and 4 switch control between each other for infinite bursts.

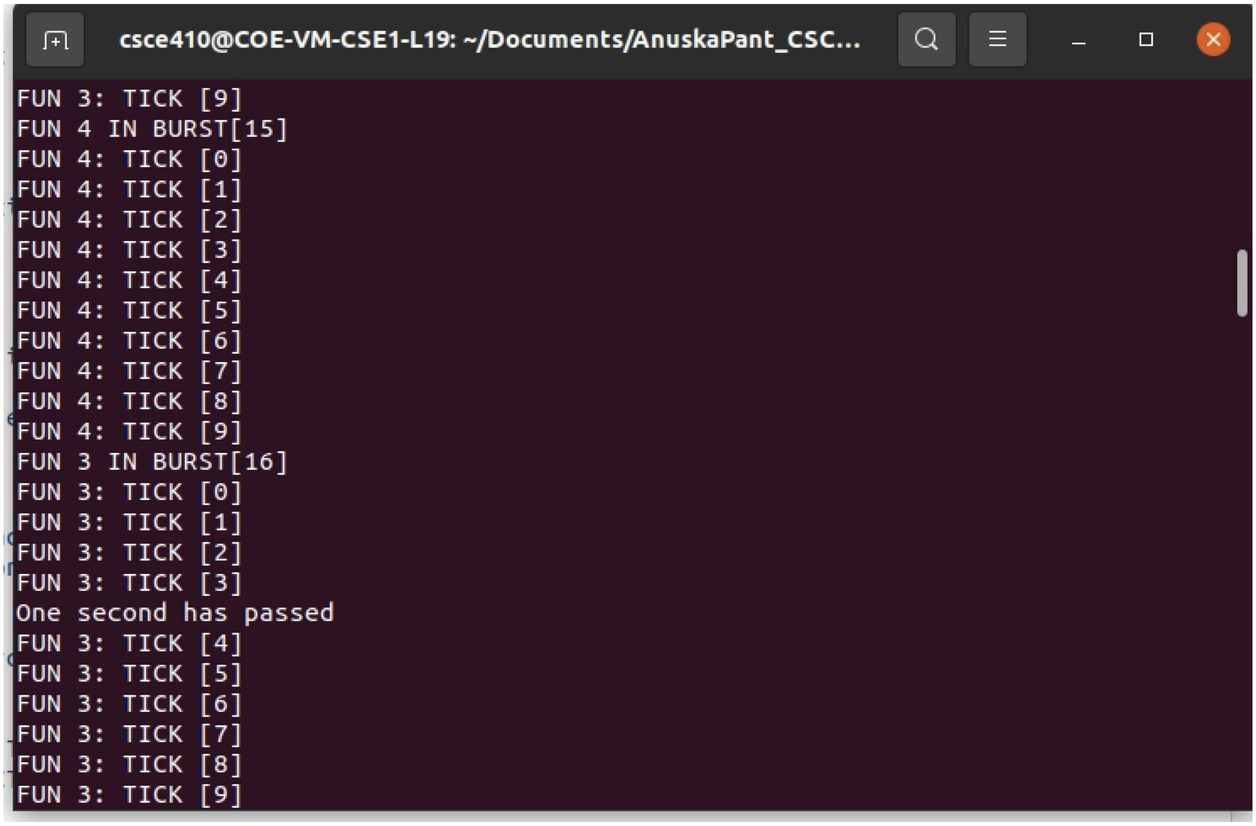


Bonus Points

Option 1: Correct handling of interrupts

scheduler.C and thread.C were modified for this implementation. We add Machine::enable_interrupts() to thread_start() in thread.C to enable interrupts at the start of each thread. The interrupts were also activated at the end of yield function and disabled at the beginning of yield, add and resume functions.

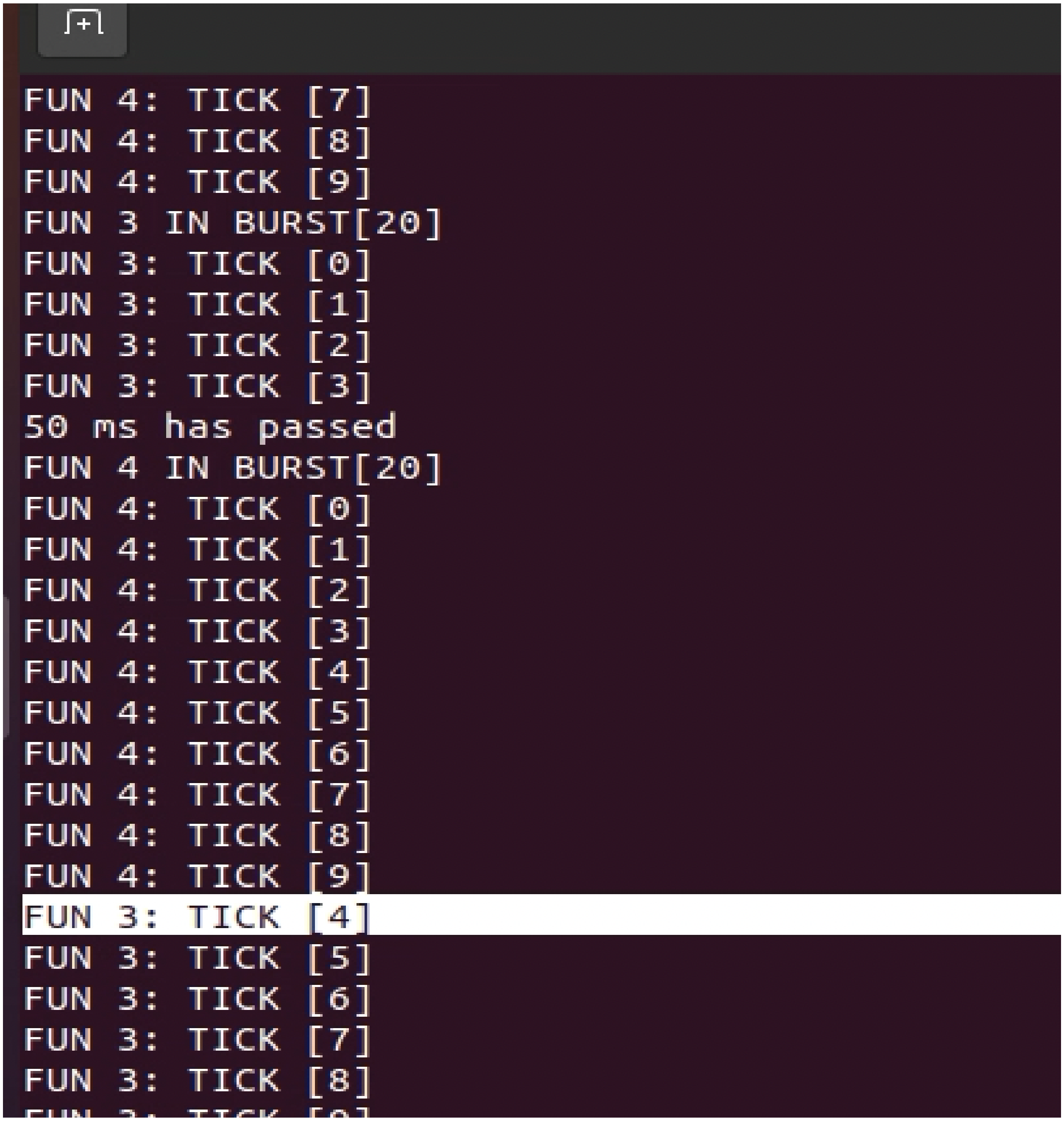
The output when interrupts where handled is as follows.



Option 2: Round-Robin Scheduling

Here the simple_timer.C is modified in its handle_interrupt method to implement the round robin scheduler. We check if the 50ms has passed and if it has we resume the thread by adding it to the end of the queue and then calling yield to dispatch the next thread. We also add Machine::outportb(0x20, 0x20) to the scheduler.C yield method to let the interrupt controller know that the interrupt has been handled. This instruction is essential to reset the ticks and ensure the proper functioning of the round robin scheduler.

The output when round robin scheduler is implemented shows that thread 3 is preempted after 50ms has passed and it later resumes its sequence right where it left of.



```
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[20]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
50 ms has passed
FUN 4 IN BURST[20]
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: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
```

Files modified:
scheduler.C
scheduler.H
simple_timer.C
thread.C
kernel.C