Implementation of Kernel-Level Thread Scheduling

Vishruth Raghuraj Gollahalli

Attempted Bonus: Option 1 (Interrupt Handling)

Part 1: Implementing scheduler functionalities

First, a queue data structure was implemented in the queue.H file which included standard queue operations such as enquing and dequing.

The scheduler functionalities within scheduler.C were implemented in the following manner:

1. Constructor: The Scheduler() constructor implementation included just setting the ready queue size as 0.

2. yield(): To implement this, first, we verify that the ready queue has threads within it. If not we panic. If threads are present on the ready queue, the top thread (using first-in first-out principle), is dequeued and given to the CPU to be run. The size parameter is updated by reducing it by 1. Thread is given to CPU by calling the dispatch_to function.

3. resume(): For a thread to resume, it is simply added to the ready queue at the end (enqueued) and so works as expected of a FIFO scheduling.

4. add(): This function is implemented the same way as resume. Only difference is the circumstance in which this function is called (when thread is newly created, rather than when resuming after being moved out of the ready queue)

5. terminate(): Termination of a thread involved looking for a thread with the same thread ID as the requested thread to be terminated. Then, the obtained thread was "terminated" by simply not being added to the ready queue while all other thread were added back into the ready queue.

Once the functionalities were realized, the macro "_USES_SCHEDULER_" was uncommented to enable FIFO scheduling. The modification in scheduler.H file for this was to add private class members of class Scheduler. These were the initialization of the ready queue and the queueSize variable.

Part 2: Adding support for terminating threads:

This step required modifications to the thread_shutdown() method in the thread.C file. First the thread was terminated using the terminate function as implemented in Part 1. The current thread was deleted and the CPU was given to the next thread in the ready queue.
After this change, when the code was tested by uncommenting the macro "_TERMINATING_FUNCTIONS" in kernel.C file, it was observed that after 10 bursts, threads 1

and 2 were terminated and only 3 and 4 ran in an interleaving fashion for 10 ticks before one yielding to the other thread.

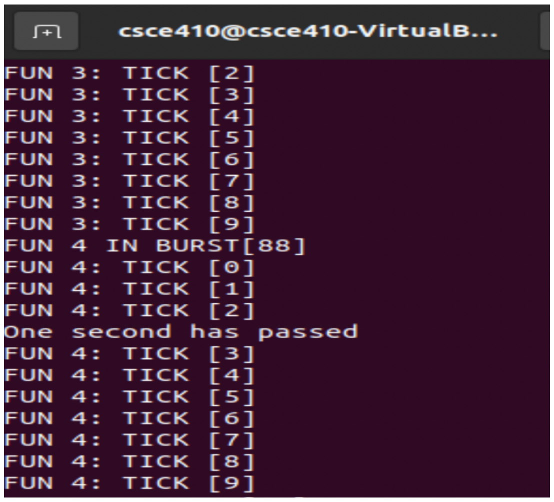
<u>Bonus points attempt: Option 1 - Correct handling of interrupts:</u>

To handle interrupts the following modifications were made:

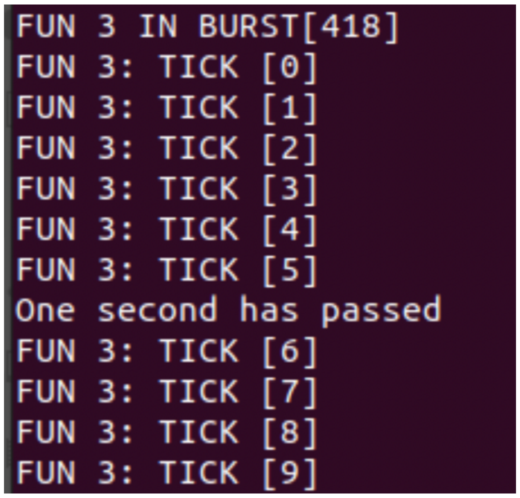
1. In the thread_start() function in thread.C file, interrupts were enabled by calling the enable_interrupts() function of class Machine (present in machine.H file).

2. Within each of the member functions in class scheduler, if interrupts were initially enabled, they were disabled. Once the function was completed, then the interrupts were enabled again.

3. This would mean that every second the interrupt handler would fire and the message “One second has passed” message would be seen without disrupting the processing of the threads. This was indeed seen. Screenshots of this is shown below.



```
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[88]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
One second has passed
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[418]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
One second has passed
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
```

This showed that the interrupt handling were implemented correctly.