MP5: Kernel-Level Thread Scheduling

By: Mehul Varma

Implemented: *Option 1 and Option 2*

Modified: *Scheduler.C/H , Thread.C/H , eoq_timer.C/H*

To test option 2: Un-comment lines *47-48 from scheduler.C*

Part 1 Introduction

In this assignment, we were supposed to implement a scheduler for the kernel threads we have been dealing with as implemented by the thread class given to us.

For the scheduler, we implemented a FIFO implementation of the ready queue and extended this solution by enabling interrupts and implementing a round-robin preemption approach.

Part 2 Architectural and component-level design

The majority of implementation for this Machine problem was the implementation of the Scheduler class

Scheduler Constructor:

Here we do the following three things to set up the Scheduler.

1. We initialize the ready queue's Head and Tail pointers to NULL.
2. We initialize the zombie queue's Head and Tail pointers to NULL.
3. We create the EOQTimer and register the interrupt handler which is the EOQTimer's interrupt handler.

From here on we enable the EOQTimer which starts and creates interrupts every 50msec.

Scheduler Yield

The Scheduler yield function is meant to be called when the current thread wants to give up the CPU to another thread. Here the following steps are implemented for this function-

1. Disable interrupts before we do anything so that we do not end up preempting inside of a yield call.
2. Return from the function if the ready queue is empty

3. Otherwise, pop from the ready queue and dispatch to the popped thread from the ready queue.

Scheduler Resume:

This function is similar to adding to the linked list of ready threads except we also make sure to clean the zombie queue once we are done adding the current thread to the ready queue. The following steps are how we perform these actions-

1. Disable interrupts before we do anything so that we do not end up preempting inside of a resume call.
2. Add the thread given to us to the end of the linked list of the ready queue.
3. Iterate through each of the zombie queue elements and call the *delete* function on them.

Scheduler Add:

This function is similar to the resume function except we do not clean the zombie queue in this function.

Scheduler Terminate:

The main use of this function is to make sure the thread passed into the function is never preempted to. We make sure this doesn't happen by implementing this function using the following steps -

1. Disable interrupts before we do anything so that we do not end up preempting inside of a terminate call.
2. Check if the thread we have to terminate is not the current thread, in this case, we just simply remove the thread from the ready queue.
3. If the thread that is being terminated is the current thread, we add the current thread to the zombie queue and simply call yield.

EOQ Timer Handle Interrupt:

In this function, I just called resume and the yield function where in between I would signal the EOI to the master interrupt controller using the following line -

- Machine::outportb(0x20, 0x20);

Thread.C Thread Shutdown:

For the thread shutdown function I simply just call the Terminate function from the Scheduler of the thread and therefore the Scheduler takes care of yielding to the next appropriate thread.

Thread.C Thread Start:

In this function, I just enable the interrupts and before I do that I also make sure to signal the EOI to the master interrupt controller to take care of the round-robin scheme.

Part 4 Testing

Testing:

- The program was tested with the given tests where they were manipulated so that the different functionalities of the Scheduler could be tested.