|  |  |
| --- | --- |
| *Sneha Singh (UIN: 131005031)* | *CSCE-611: Operating System | MP5* |

**Design Idea:**

**Kernel-Level Thread Scheduling – MP5:**

I Have implemented the following:

* FIFO Scheduler
* Terminating of threads
* Option 1: Interrupt management
* Option 2: Round-Robin Scheduling using End-of-Quantum Timer

**FIFO Scheduler:**

A pointer-based list with the front and tail pointers is used to implement the ready queue. The queue is initially empty with its front and tail pointer pointing to null. A variable thread\_count is used to maintain the number of threads in the system.

The Scheduler provides the following functionality:

* add(thread) – Adds the thread to the end of the ready queue after the tail node. If the front and tail pointer are null, they now point to the new thread added to the queue.
* resume(thread) – Adds thread to the end of the queue using add(thread) functionality.
* yield() – Scheduler passes the control to the thread at the front of the ready queue. It throws and assertion error if the queue is empty. If the ready queue only contains one thread, the front and tail nodes are made to point to null.
* terminate(thread) – Terminates the thread by removing it from the ready queue and yields to the next thread in queue.

**Termination of threads:**

Threads are terminated by thread\_shutdown api in the thread class. The threads may be terminated after task completion or on suiciding. To handle this, the thread is removed from the ready queue by the scheduler before deleting the thread.

**Test Screenshots / Output – FIFO Scheduler with terminating threads:**

As seen below, every thread completes 10 ticks in a burst before passing the CPU to the next thread in queue. Also, threads 0 and 1 run for bursts 0 to 9 each, while the threads 2 and 3 run infinitely, unless the operating system is switched off at bursts 51 and 50 respectively.

Text

Description automatically generated Text

Description automatically generated

Text

Description automatically generated Text

Description automatically generated

Text

Description automatically generated Text

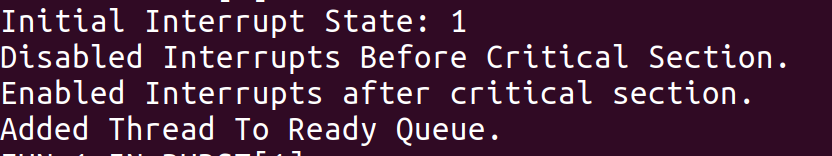
Description automatically generated

**Option 1 - Interrupt management:**

* Interrupt is enabled on the starting a thread. This is also written to the EFLAGS while populating the thread stack by checking if the interrupt is enabled. (0=interrupt disabled, 1=interrupt enabled).
* Further, to ensure mutual exclusion, the interrupt is disabled and enabled before and after critical sections of manipulating the ready queue.

**Test Screenshots / Output:**

The below log shows clearly that the interrupt is enabled at thread start and disabled in the critical section. Once the critical section is over, the thread is enabled again.

****

**Option 2 – Round-Robin Scheduling using End-of-Quantum Timer:**

* End-of-Quantum Timer (EOQTimer) is implemented as a derived class of the SimpleTimer class. Upon completion of the quantum time (50ms / 5 clock ticks), the scheduler adds the current thread to the tail of the ready queue using the resume function. It also, yields the control from the current thread to the next thread at the front of the queue.
* When dispatching an interrupt, to let the interrupt controller (PIC) know that the interrupt has been handled, an EOI signal is sent and then the interrupt is handled by yielding to the next thread in ready queue.
* To implement Round Robin scheduling, the kernel is updated to use the EOQTimer instead of the SimpleTimer. Also, interrupt is enabled after the start of the first thread, instead of at the start of the operating system.
* Further the makefile is updated to compile the new EOQTimer files created.

**Test Screenshots / Output:**

As seen in the logs below:

* Every thread runs for the set quantum, before yielding to the next thread. Hence, instead of completing a burst (10 ticks) before giving up the CPU, they perform different number of ticks in each quantum. Thread 0, 1, 2 and 3 perform 3, 4, 2, and 3 ticks of burst 0 in their first time-quantum and then give up the CPU.
* In the next time quantum, these threads resume from where they left off, i.e., ticks 4, 5, 3, and 4 of bursts 0 respectively.
* Threads 0 and 1 terminate after completing 10 bursts at different time-quantums.
* Threads 2 and 3 are non-terminating threads and continue until the system is switched off.
* The non-terminating threads have completed different amounts of bursts, when the system was turned off. Specifically, threads 2 and 3 are in bursts 34 and 37 respectively.

**A picture containing text

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text

Description automatically generatedA picture containing text

Description automatically generated**

**Documents changed:**

I have changed the following files:

* **eoq\_timer.C (new file)** – as per instructions in Option 2
* **eoq\_timer.H (new file)** – as per instructions in Option 2
* **interrupts.C** – as per instructions in Option 2
* **kernel.C** – as per instructions in the handout for the compulsory parts, options 1 & 2
* **makefile** – updated to compile eoq\_timer
* **scheduler.C** – as per instructions in the handout for the compulsory parts & option 1
* **scheduler.H** – as per instructions in the handout for the compulsory parts & option 1
* **thread.C** – as per instructions in the handout for the compulsory parts, options 1 & 2