CSCE 611: MP5: DESIGN DOCUMENT
AUTHOR: Pranav Anand Taukari
UIN: 433003781

PROBLEM STATEMENT:
To perform scheduling of multiple Kernel-level threads.

1) To implement a FIFO scheduler to schedule non-terminating threads
2) To modify the FIFO scheduler mechanism to schedule and terminate the Terminating threads
3) [OPTION 1: BONUS] Disable and Enable Interrupts

Part 1: Implementation of a FIFO Scheduler:

To implement this we design a ready queue in which threads are waiting for the CPU. When running threads call yield, the thread which is next in the queue is given to the CPU. The yielded thread is then added to the tail of the queue.

Implementation of the ready queue:

We design the ready queue using a linked list which is implemented in a class Queue in the scheduler.H file. It has public functions called enQ and deQ which are used to populate and remove threads to the queue respectively. We maintain a node for linked list which has the thread and next pointer. The queue is maintained with private variables front and rear which maintain the queue structure.

List of functions:

1. **enQ(Thread *t) :** This function is responsible for adding the thread to the queue. It adds the incoming elements to the last in the queue.
2. **deQ() :** This function is responsible for removing the threads from the queue. Since both the queue and FIFO scheduler follow a FIFO operation, the thread at the beginning is always removed from the queue and dispatched. If empty queue, it returns NULL.

FIFO SCHEDULER IMPLEMENTATION :
The Scheduler class consists of a ready Queue of class type Queue, which maintains a queue of the incoming threads and dispatches them in a FIFO fashion.

List of variables:

```
Queue   readyQ;
int     Qsize;
```

List of functions:

1. **Scheduler::yield():** This function is responsible for yielding the threads in the queue. It removes the thread from the queue and sends it to the thread dispatcher for execution.
2. **Scheduler::resume(Thread * _thread):** When a thread that is dispatched is ready to be executed again, it is added back to the queue at the end, similar to the add function.
3. **Scheduler::add(Thread * _thread):** It is used to add a thread to the queue.

Part 2: To implement terminating threads:

Terminating threads involves removing the thread from the ready queue, freeing memory allocated to it. The Thread_shutdown() function is called whenever a thread returns from the thread function. The scheduler is the one who calls the termination. In this function, we first dequeue the first thread and check if its id matches the target thread. If it matches we delete the thread and call yield(), else we enqueue the thread.

We check this by terminating threads 1 and 2 after 10 iterations, post which threads 3 and 4 run infinitely.

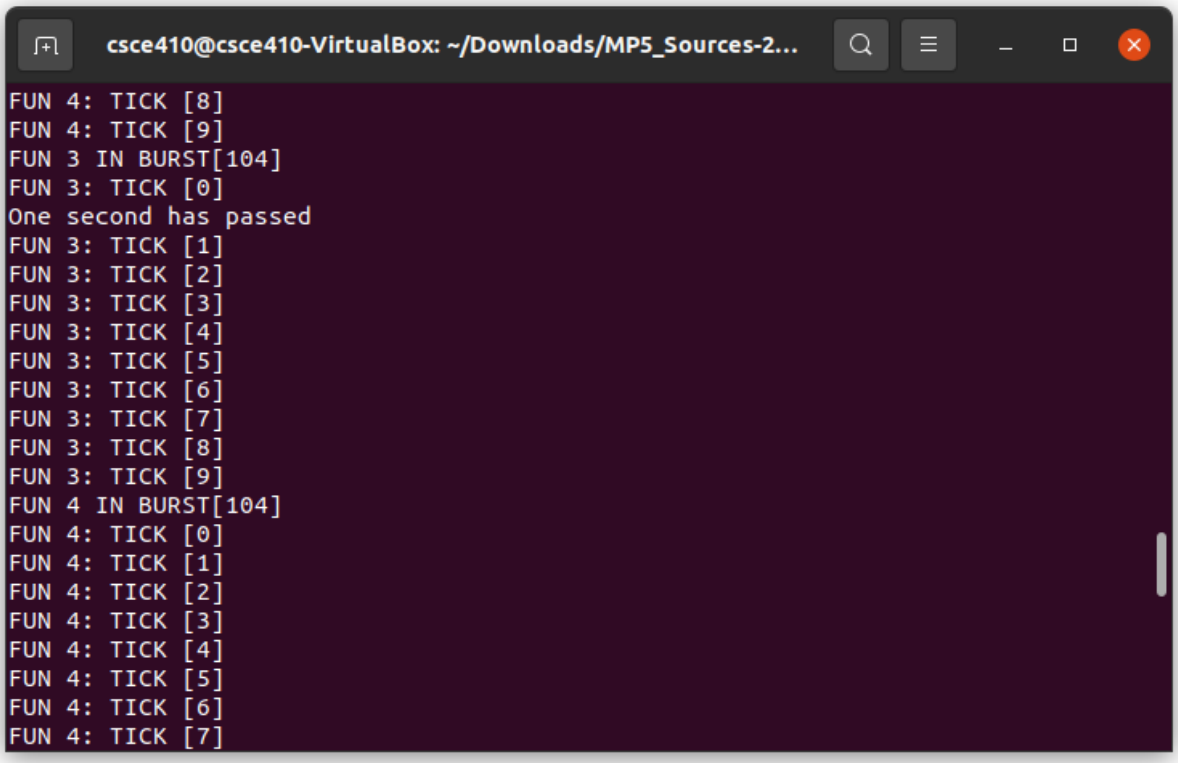
Part3: BONUS: INTERRUPT HANDLING:

For this part, we changed scheduler.C and thread.C where we enable and disable the interrupts. It is necessary to enable the interrupts at the start of the thread and disable it when there are any ongoing operations in the Queue.

Changes made for interrupt handling:
In thread.C, in thread_start() , Machine::enable_interrupts(); is used to enable the interrupts at the start of each thread.

In scheduler.C we disable the interrupts at the beginning of yield(), add(), resume(), terminate(), and enable them post the queue operations. This way we ensure mutual exclusion.

Sample Output:
Enabling and disabling the interrupts would ensure we see the message “One second has passed” in the console. The screenshot is as below,


csce410@csce410-VirtualBox: ~/Downloads/MP5_Sources-2...
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 3 IN BURST[104]
FUN 3: TICK [0]
One second has passed
FUN 3: TICK [1]
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[104]
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]