EXPERIMENT 2

CPU SCHEDULING

1. OBJECTIVES

After completing this lab, you will be able to:

- Understand how to schedule processes/threads using round-robin strategy with a fixed time quantum.
- Understand how to create and reset timer interrupts to implement the fixed time quantum.

2. LABORATORY

Please see the lab schedule.

3. EQUIPMENT

Pentium IV PC with Nachos 3.4.

4. MODE OF WORKING

You should be working alone. No group effort.

5. PROBLEM STATEMENT

In the previous experiment you observed that the short-term CPU scheduler in Nachos is very simple. It is non-preemptive and `scheduler.cc` in the `threads` directory implements a `FIFO` scheduling algorithm for the `readyList` ready queue. In this experiment, our aim is to implement a **preemptive round-robin CPU scheduling algorithm** for Nachos. For this purpose, we need to implement the following features:

a) Initialize a periodic timer interrupt with a **fixed quantum size of 40 time ticks** in Nachos.

b) Invoke the CPU scheduler to perform a context switch whenever the timer interrupt expires. The current running thread should be moved to the back of the `readyList` and the thread at the head of `readyList` should be scheduled for execution next.

c) Adjust the timer interrupt whenever the current running thread completes (i.e., executes `Thread→Finish();`) in the middle of a time quantum. Let us understand this using an example. If a thread completes after 10 time ticks in the current time quantum, then the next timer interrupt is currently scheduled to expire 30 time ticks in the future (total time quantum is 40). This means that the new thread that will be scheduled for execution, will only get a time quantum of 30 time ticks. We therefore must "reset" the timer interrupt so that it expires 40 time ticks after the new thread is scheduled.

6. THREADS

The thread is a light-weight process in Nachos. In the first experiment, we have learnt the basic feature of thread execution in Nachos. In this experiment, we need to use/modify the following thread operations to finish the exercises. Please read the source code under the directory `threads` (particularly for `thread.h` and `thread.cc`).

```
void Yield();                   // Relinquish the CPU if any
                                // other thread is runnable
```

A thread gives up the CPU ownership and allows another thread to execute. Its state shifts from running to ready, and it is put to the back of the ready queue (`readyList`).

```
void Finish();                  // Executed when a thread is
                                // done executing its function.
```

A thread executes this cleanup operation whenever it completes executing its function/procedure. Among other operations, it sets itself for destruction (`threadToBeDestroyed = currentThread;`) and then invokes the CPU scheduler for a context switch by calling `Sleep();`.

7. TIMERS

Timer can be used to trigger an interrupt (i.e., after a fixed number of time ticks). Please read the source code under the directories `threads` and `machine` (particularly `system.cc, timer.cc` and `timer.h`).

```
void TimerInterruptHandler(); // Interrupt handler that
                              // is called when timer expires
```

This function in `system.cc` executes whenever the timer expires and the interrupt is triggered. The timer itself is initialized in `system.cc` using the constructor for class `Timer` which is defined in `timer.cc`.

```
void TimerExpired();          // Function that executes
                              // when the timer expires
```

This function in `timer.cc` executes whenever the timer expires. It inturn invokes the interrupt handler which is defined in `system.cc` as described above.

```
int TimeOfNextInterrupt();    // Function returns the next
                              // interrupt time tick
```

This function returns the next time tick when the interrupt should be scheduled for the current `Timer` object. This function can be used to make the timer periodic as required for round-robin scheduling.

8. INTERRUPTS

The timer uses several functions from the `Interrupt` class. Pending timer interrupts in the system are maintained in a list called `pending`, comprising objects of the class `PendingInterrupt`. This list is sorted in increasing order of the time tick when the interrupt will be triggered. Please read the source code under the directory `machine` (particularly `interrupt.h` and `interrupt.cc`)

```
void Schedule();        // Function that schedules/inserts a
                        // new interrupt to the pending list.
```

This function schedules a new interrupt and inserts it into the `pending` list in sorted order. It is also used in the `Timer` class constructor to initialize a timer interrupt.

```
void OneTick();         // Function to process a single time
                        // tick. This is how Nachos keeps track
                        // of time.
```

This function processes a single time tick. Once the tick variable is updated (`stats→totalTicks`), it calls `CheckIfDue()` to process any pending interrupts

that would be triggered in the current tick. Thereafter, if the `yieldOnReturn` variable is set to true, then it triggers a context switch by invoking `Thread→Yield()`.

```
bool CheckIfDue();      // Function to process interrupts and
                        // invoke the handler.
```

This function checks if the `pendingInterrupt` at the head of `pending` list should be triggered at the current time tick. If yes, the corresponding interrupt handler is invoked.

```
void YieldOnReturn();   // Function that is called by the Timer
                        // handler.
```

This function sets the variable `yieldOnReturn` to true so that a context switch is triggered when the interrupt processing is completed. It is used in the timer interrupt handler `TimerInterruptHandler()` defined in `system.cc`.

9. EXERCISE

We will implement the fixed time-quantum round-robin CPU scheduler in 3 steps. Look for code comments starting with /* Experiment 2*/ for guidance in the code.

1. Initialize the timer interrupt with a fixed time quantum of 40 time ticks.
   a) Activate `Timer` in `system.cc`.
   b) Initialize the timer with the fixed time quantum in `timer.cc`.

2. Make the timer interrupt periodic.
   a) Modify function `TimerExpired()` in `timer.cc` to make the timer periodic. It should trigger an interrupt every 40 time ticks.
   b) At this stage, you have an initial implementation of preemptive round-robin CPU scheduling. **QUESTION:** Why don't we have to handle the context switch from one thread to another? How does it happen?
   c) Change your working directory to Experiment 2 by typing ***cd ~/nachos-exp1-2/exp2***. Compile Nachos by typing ***make***. If you see ***"ln -sf arch/intel-i386-linux/bin/nachos nachos"*** at the end of the compiling output, your compilation is successful. If you encounter any anomalies, type ***make clean*** to remove all object and executable files and then type ***make*** again for a clean compilation.
   d) Trace a run of this Nachos test program by typing ***./nachos -d > output_1.txt***. Option ***-d*** is to display Nachos debugging messages.
   e) Fill in **Table1.csv** (template is provided) whenever:
      - The timer interrupt is triggered, or
      - The current thread is changed, or
      - The current thread completes.

Tick	ready list	current thread	Timer Interrupt triggered	Thread completion	Context switch
40	child1, child2, child3	main	Timer interrupt scheduled at 80		main -> child1
50	child2, child3, main	child1		Thread 1 Completed	child1 -> child2

3. Reset the timer interrupt if a thread finishes in the middle of a time quantum.
   a) When the current thread finishes, remove the pending timer interrupt from the `pending` list, and insert a new timer interrupt with the time quantum of 40 time ticks. **Note:** For this experiment, to keep things simple, we will assume that no other interrupts are pending in the list, except the timer interrupts created by us.

b) To accomplish the above task, you would need to modify files/functions `Threads→Finish(), timer.cc, timer.h, interrupt.cc` and `interrupt.h`.

c) Compile and execute Nachos as in Step 2 above (use filename ***`output_2.txt`*** to store your results), and fill **Table2.csv** (template is provided) whenever:
- The timer interrupt is triggered, or
- The current thread is changed, or
- The current thread completes.

Tick	ready list	current thread	Timer Interrupt triggered	Thread completion	Context switch
40	child1, child2, child3	main	Timer interrupt scheduled at 80		main -> child1
50	child2, child3, main	child1	Timer interrupt reset to 90	Thread 1 Completed	child1 -> child2

10. ASSESSMENT

- Assessment of your implementation. Please leave your code, the files **output_1.txt** and **output_2.txt**, as well as files **Table1.csv** and **Table2.csv** in the **exp2** folder for TA/Supervisor to review. Deadline is 1 week after your lab session (e.g., if lab session is from 10AM-12PM on a Monday, then deadline is 9:59AM on the next Monday).
- **Lab Quiz 1**, which is an online multiple-choice quiz, will be administered through NTULearn.