COMP3511 Spring 2019 Project #2: CPU Scheduling in Nachos

(You are strongly recommended to use the servers in the Lab, the servers are csl2wk01.cse.ust.hk ~ csl2wk40.cse.ust.hk. SSH is OK for that.)

In this project you will learn how to schedule CPU for threads. You are given a simple scheduling system skeleton in Nachos and your tasks are:

1. **Compile Nachos and run the system with pre-implemented First Come First Serve CPU scheduling algorithm.**
2. **Read the code and understand how the given CPU scheduling algorithm is implemented.**
3. **Implement the Round Robin scheduling algorithm (RR), Preemptive Priority scheduling algorithm (P_Priority) and Multilevel Feedback Queue (MLFQ) in Nachos. Recompile and run the system to test your implementation.**
4. **Explain the results and answer some questions.**

Please don't be overwhelmed by the sheer amount of code provided. In fact, you don't need to worry about most of it. The parts that you need to read or modify are given in the following instructions. Please read them carefully, and follow the steps.

Task 1: Run Nachos with Pre-implemented Scheduling System Skeleton

Step 1: Download Nachos source code of this project

```
wget
http://course.cse.ust.hk/comp3511/project/project2/os2019spring_nachos_proj2.tar.g
z
```

Step 2: Extract the source code

```
tar zxvf os2019spring_nachos_proj2.tar.gz
```

Step 3: Compile the code

Enter the folder `os2019spring_nachos_proj2` and then run `make`.

Step 4: Run Nachos

This program was designed to test 4 scheduling algorithms, namely **First Come First Serve (FCFS)**, **Round Robin (RR)**, **Preemptive Priority (P_Priority)** and **Multilevel feedback queue scheduling (MLFQ)**. To cover

all the cases, we do not run the executable file `nachos` directly. Instead, we run `test0`, `test1`, `test2` and `test3` to test the 4 scheduling algorithms respectively.

For example, you can run 'test0' to test First Come First Serve scheduling algorithm.

```
./test0
```

If you succeed in running `test0`, you will see the following messages:

```
First-come first-served scheduling
Starting at Elapesd ticks: total 0
Queuing threads.
Queuing thread threadA at Time 0, priority 4, willing to burst 9 ticks
Queuing thread threadB at Time 0, priority 4, willing to burst 11 ticks
Switching from thread "main" to thread "threadA"
threadA, Starting Burst of 9 ticks. Elapesd ticks: total 0
threadA, Still 8 to go. Elapesd ticks: total 1
threadA, Still 7 to go. Elapesd ticks: total 2
threadA, Still 6 to go. Elapesd ticks: total 3
threadA, Still 5 to go. Elapesd ticks: total 4
threadA, Still 4 to go. Elapesd ticks: total 5
threadA, Still 3 to go. Elapesd ticks: total 6
threadA, Still 2 to go. Elapesd ticks: total 7
threadA, Still 1 to go. Elapesd ticks: total 8
threadA, Still 0 to go. Elapesd ticks: total 9

............(We omitted some output here.)............

threadH, Still 4 to go. Elapesd ticks: total 122
threadH, Still 3 to go. Elapesd ticks: total 123
threadH, Still 2 to go. Elapesd ticks: total 124
threadH, Still 1 to go. Elapesd ticks: total 125
threadH, Still 0 to go. Elapesd ticks: total 126
threadH, Done with burst. Elapesd ticks: total 126
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 126, idle 0, system 126, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Cleaning up...
```

To be concise, we omitted several output lines.

The following table would give very useful information to you.

Executable File	Source File	Corresponding Algorithm	Already Implemented?
test0	test.0.cc	FCFS	Yes
test1	test.1.cc	RR	No
test2	test.2.cc	P-Priority	No
test3	test.3.cc	MLFQ	No

You can run `test0` to test the pre-implemented algorithms. However, because **Round Robin** algorithm, **P-Priority** and **MLFQ** are not yet implemented, if you run `test1` or `test2` or `test3` to test the given system skeleton, there will be an error. You can view the source code of test files in `test.0.cc`, `test.1.cc`, `test.2.cc` and `test.3.cc` respectively.

Step 5: Read the code

Please read the code carefully. Try to understand how the given scheduling algorithm is implemented. You need to focus on `threadtest.cc`, `scheduler.h`, `scheduler.cc`, `list.h`, `list.cc`, `thread.h`, `thread.cc`. Here we provide you some notes about the code.

The CPU scheduling algorithms are mainly implemented in 4 functions: `ReadyToRun()`, `FindNextToRun()`, `ShouldISwitch()`, `InterruptHandler()`, in `scheduler.cc`.

1. `ReadyToRun()` decides the policy of placing a thread into ready queue or multilevel queues when the thread gets ready. For example, in FCFS we simply append the thread to the end the ready queue, while in scheduling algorithms where threads have different priority we insert the thread to the queue according to its priority.

2. `FindNextToRun()` decides the policy of picking one thread to run from the ready queue. For example, in FCFS scheduling, we fetch the first thread in ready queue to run.

3. `ShouldISwitch()` decides whether the running thread should preemptively give up to a newly forked thread. In FCFS scheduling, the running thread does not preemptively give up its CPU resources. Note that only in preemptive algorithms, it is needed to decide whether the running thread should give up or not. In other algorithms, you can simply return false.

4. `SetNumOfQueues(int level)` Set the number of queues for MLFQ - should be called only once.

5. `InterruptHandler(int dummy)` decides the policy of preempting the running thread regularly. For example, in RR scheduling, we decrement the quantum of a process and preempt the process when the quantum is 0.

Task 2: Implement three Scheduling Algorithms

In this task, you are required to implement the remaining four scheduling algorithms Round Robin, Preemptive Priority and MLFQ, and then test your implementation. To achieve this, you needn't modify any source file other than `scheduler.cc` and `test.3.cc`. You are supposed to add some code in the following functions in `scheduler.cc`.

Note: Be very careful of `cases` in `switch` block(s) in each of those functions. Make sure you put your code in the right place.

Since you have to operate one or more Lists, you could refer to `list.h` and `list.cc` to get familiar with List operations. Please make good use of appropriate List operations, and the crucial requirement of this project for you is to understand and experiment with different scheduling algorithms instead of coding itself, so the coding part is actually relatively easy.

Step 1. Implement **Round Robin** Scheduling

In this step, you are supposed to add some code with respect to Round Robin algorithm in `case SCHED_RR` in each function in `scheduler.cc`. In Round Robin scheduling algorithm, the thread should first be scheduled in a first-come-first-served manner. Each thread gets a quantum of **4 ticks**. When the quantum expires, it is preempted and added to the end of the `readyList`. Some notes are given to you:

1. A thread can be preempted by calling `Thread::Yield()`. However in an interrupt handler, directly calling it causes a context switch which switches away from the handler. `interrupt->YieldOnReturn()` should be called instead.
2. `Scheduler::InterruptHandler()` is the interrupt handler called once every tick of the scheduler timer. The scheduler timer, `Scheduler::ptimer` is set to tick every 1 CPU tick, so the interrupt handler is also called every CPU tick. Time-related schedulling actions e.g. quantum counting and expiring and preemption of process can be implemented in this handler.
3. You should run `make clean` and then `make` to recompile the code and run `test1` to check the output.

```
./test1
./test1 > project2_test1.txt
```

Step 2. Implement **Preemptive Priority** Scheduling

In this step, you are supposed to add some code with respect to P_Priority algorithm in `case SCHED_PRIO_P` in each function in `scheduler.cc`. In P_Priority algorithm, upon its arrival, the thread with the highest priority in the `readyList` will preempt the current thread and thus be scheduled immediately. If there are more than one thread with the same priority in the `readyList`, they must be scheduled in FCFS manner.

Then you should run `make clean` and then `make` to recompile the code and run test2 to check the output.

```
./test2
./test2 > project2_test2.txt
```

Step 3. Implement **Multilevel Feedback Queue** Scheduling

In this step, you are supposed to add some code with respect to MLFQ algorithm in `case SCHED_MLFQ` in each function in `scheduler.cc` and `test.3.cc`. In MLFQ algorithm, upon its arrival, each thread will be put at the tail of the first level of `MultiLevelList`. The MLFQ should have **3 levels**, with the first level having a **quantum of 4**, second level having a **quantum of 8**, and the third level is **FCFS**.

Some notes are given to you:

1. In this MLFQ scheduling, it is a kind of **preemptive** scheduling.
2. Function `SetNumOfQueues(int level)` should be called in `test.3.cc`.
3. In `ReadyToRun(Thread *thread)`, you should add codes in `case SCHED_MLFQ`. Each thread should be put into its corresponding queue in the queue List of `MultiLevelList[]` instead of the queue of `readyList`.
4. In `FindNextToRun()`, you should add codes in `case SCHED_MLFQ`. You should search for ready thread from the first level queue in `MultiLevelList[]` first, then in the second level, then in the third level.
5. In `ShouldISwitch()`, you should add codes in `case SCHED_MLFQ`. Upon a thread's arrival, the thread with the highest priority will preempt the current thread and thus be scheduled immediately. The preempted thread should be prepended to its own level with the remaining quantum. If there are more than one thread with the same priority, they must be scheduled in FCFS manner.
6. In `InterruptHandler()`, you should add codes in `case SCHED_MLFQ`. A thread should be counted for its quantum and be preempted and moved to the next level when it used up the quantum.

Then you should run `make clean` and then `make` to recompile the code and run `test3` to check the output.

```
./test3
./test3 > project2_test3_1.txt
```

Next, you are supposed to change two code lines in `test.3.cc`. In particular, in **Line 24 to Line 26**, change the `startTime[]` to `{13, 0, 20, 18, 10, 7, 12, 6}`, `burstTime[]` to `{15, 9, 3, 18, 32, 2, 14, 19}` and the `priority[]` to `{2, 2, 2, 2, 2, 2, 2, 2}`.

Then you should run `make clean` and then `make` to recompile the code and run `test3` to check the output.

```
./test3
./test3 > project2_test3_2.txt
```

Task 3: Explain the Results

1. Understand the output of `test0` (FCFS scheduling), `test1` (RR scheduling), `test2` (P_Priority scheduling), and `test3_1` and `test3_2` (MLFQ). Then calculate the following performance of **all the five settings**:
    1. Average waiting time;
    2. Average response time;
    3. Average turn-around time.

    Notice: By definition, **response time** means *the amount of time it takes from when a request was submitted until the first response is produced*, and **waiting time** means *the sum of the periods that a process spent waiting in the ready queue*.

2. Compare the performance of the two scheduling algorithms **FCFS and RR** in the aspects mentioned in question 1, then discuss the pros and cons of each of the two scheduling algorithms. (Note: you are strongly encouraged to change the input threads in `test.0.cc` and `test.1.cc` in order to make your discussion more convincing. However, when submitting the outputs of `test0` and `test1`, please do submit the outputs with the original input threads.)

3. Compare the results in `project2_test3_1.txt` and `project2_test3_2.txt`, and explain the differences in details.

Please write your answers in project2_report.txt

After Finishing These Tasks

1. Please generate a single file using **TAR GZIP** and submit it through **CASS**.

2. The name of the TAR GZIP file should be `proj2_********.tar.gz`, using **your student ID** to replace the star symbols.

3. The following files should be included inside the TAR GZIP file:

File Name	Description
scheduler.cc	Source code you have accomplished by the end of Task2
project2_test1.txt	Output of test1
project2_test2.txt	Output of test2
project2_test3_1.txt	Output of test3
project2_test3_2.txt	Output of test3
project2_report.txt	The answer to the questions in Task 3
test.3.cc	Source file