COMP3511 Spring 2015 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 Shortest Job First scheduling algorithm (SJF) and Non-preemptive Priority scheduling algorithm (NP_Priority) 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

```
wget http://course.cse.ust.hk/comp3511/project/project2/os2015spring_nachos_proj2.tar.gz
```

Step 1: Download Nachos source code of this project

Step 2: Extract the source code

Step 3: Compile the code

```
tar zxvf os2015spring_nachos_proj2.tar.gz
```

Enter the folder "os2015spring_nachos_ proj2" and then run "make".

Step 4: Run Nachos

This program was designed to test 3 scheduling algorithms, namely First Come First Serve (FCFS), Shortest Job First (SJF) and Non-preemptive Priority (NP_Priority). To cover all the cases, we do not run the executable file 'nachos' directly. Instead, we run 'test0', 'test1' and 'test2' to test the 3 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 5, willing to burst 20 ticks
Switching from thread "main" to thread "threadA"
threadA, Starting Burst of 20 ticks. Elapesd ticks: total 0
threadA, Still 19 to go. Elapesd ticks: total 1
threadA, Still 18 to go. Elapesd ticks: total 2
threadA, Still 17 to go. Elapesd ticks: total 3
threadA, Still 16 to go. Elapesd ticks: total 4
threadA, Still 15 to go. Elapesd ticks: total 5
threadA, Still 14 to go. Elapesd ticks: total 6

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

threadE, Still 0 to go. Elapesd ticks: total 57
threadE, Done with burst. Elapesd ticks: total 57
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 57, idle 0, system 57, 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	SJF	No
test2	test.2.cc	NP_Priority	No

You can run test0 to test the pre-implemented algorithms. However, because SJF algorithm and NP_Priority are not yet implemented, if you run test1 or test2 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 and test.2.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`*. Here we provide you some notes about the code.

The CPU scheduling algorithms are mainly implemented in 3 functions: `ReadyToRun()`, `FindNextToRun()`, `ShouldISwitch()`, in scheduler.cc.

1) *`ReadyToRun()`* decides the policy of placing a thread into ready queue (or multilevel queues, which will not be included in this project) when the thread gets ready. For example, in FCFS we simply append the thread to the end the ready queue, while in scheduling algorithm 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 new 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.

Task 2: Implement two Scheduling Algorithms

In this task, you are required to implement the remaining two scheduling algorithms Shortest Job First and Non-preemptive Priority, and then test your implementation. To achieve this, you needn't modify any source file other than *`scheduler.cc`*. You are supposed to add some code in the following three 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 Shortest Job First Scheduling

In this step, you are supposed to add some code with respect to SJF algorithm in **`case SCHED_SJF`** in each function in *scheduler.cc*. In SJF algorithm, the thread with the shortest burst time in the ReadyList should be scheduled for running after the current thread is done with burst. If there are more than one thread with the same shortest burst time in the ReadyList, they must be scheduled in FCFS manner.

Some notes are given to you:

1. The burst time of a thread is an integer greater than 0. The burst time of a thread can be obtained by the function getBurstTime() defined in the class *thread*.
2. Do NOT use the function setBurstTime() to change the burst time of the thread dynamically in your own code.
3. You can insert the thread to ReadyList according to its burst time when a thread gets ready. Therefore, it can be guaranteed that the first thread in ReadyList is the thread with the shortest burst time.

Then you should run "make clean" and then "make" to recompile the code and run test1 to check the output. (The first command is for you to view and the second is to record the result in the file project2_test1.txt.)

```
./test1

./test1 > project2_test1.txt
```

Step 2. Implement Non-preemptive Priority Scheduling

In this step, you are supposed to add some code with respect to NP_Priority algorithm in **`case SCHED_PRIO_NP`** in each function in *scheduler.cc*. In NP_Priority algorithm, the thread with the highest priority in the ReadyList should be scheduled for running after the current thread is done with burst. If there are more than one thread with the same priority in the ReadyList, they must be scheduled in FCFS manner.

Some notes are given to you:

1. The priority of a thread is an integer between 0 and 20. The two threshold (MAX_PRIORITY and MIN_PRIORITY) and defined in *thread.h* . The priority of a thread can be obtained by the function getPriority() defined in the class *thread.h*.
2. You can insert the thread to ReadyList according to its priority when a thread gets ready. Please be noted that **the thread with the highest priority value in the ReadyList would be chosen instead of the lowest**. Therefore, it can be guaranteed that the first thread in ReadyList is the thread with the highest priority.

Then you should run “make clean” and then “make” to recompile the code and run test2 to check the output. Do not forget to record the output.

```
./test2

./test2 > project2_test2.txt
```

Task 3: Explain the Results

1. Understand the output of test0 (FCFS scheduling), test1 (SJF scheduling) and test2 (NP_Priority scheduling). Then calculate the following performance of all the three scheduling algorithms:

1) Average waiting time;

2) Response time;

3) Turn-around time.

2. Compare the performance of the two scheduling algorithms FCFS and SJF 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.)

Please write your answers in project2_report.txt

After Finishing These Tasks

1) Please generate a single file using ZIP and submit it through CASS.

2) The name of the ZIP file should be "proj2_********.zip", using your student ID to replace the star symbols.

3) The following files should be included inside the ZIP 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_report.txt	The answer to the questions in Task 3