COMP3511 Fall2018 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), Non-preemptive Priority scheduling algorithm (NP_Priority), Preemptive Priority scheduling algorithm (P_Priority) and Multilevel Queue (MLQ) 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/os2018fall_nachos_proj2.tar.gz
```

Step 2: Extract the source code

```
tar zxvf os2018fall_nachos_proj2.tar.gz
```

Step 3: Compile the code

Enter the folder "os2018fall_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), Shortest Job First (SJF) and Non-preemptive Priority (NP_Priority), Preemptive Priority (P_Priority) and Multilevel queue scheduling (MLQ). To cover

all the cases, we do not run the executable file 'nachos' directly. Instead, we run 'test0',

```
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...
```

'test1', 'test2', 'test3', and 'test4' to test the 5 scheduling algorithms respectively.

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

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

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
test3	test.3.cc	P_Priority	No
test4	test.4.cc	MLQ	No

You can run test0 to test the pre-implemented algorithms. However, because SJF algorithm, NP_Priority, P_Priority and MLQ are not yet implemented, if you run test1 or test2 or test3 or test4 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, test.3.cc and test4.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.

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

Task 2: Implement four Scheduling Algorithms

In this task, you are required to implement the remaining four scheduling algorithms Shortest Job First, Non-preemptive Priority, Preemptive Priority and MLQ, and then test your implementation. To achieve this, you needn't modify any source file other than `scheduler.cc and test.4.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 non-preemptive Shortest Job First Scheduling

In this step, you are supposed to add some code with respect to non-preemptive 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 thresholds (MIN_PRIORITY and MAX_PRIORITY) are defined in *thread.h* . The priority of a thread can be obtained by the function getPriority(), which is also defined in *thread.h*. Note that a larger priority value means a higher priority.
2. You can insert the thread to ReadyList according to its priority when a thread gets ready. You should guarantee that **the thread with the highest priority in the ReadyList would be scheduled first**. (Please take care of the order: in SJF the thread with the shortest burst time shall be scheduled first; while in Priority scheduling the thread with the largest priority value shall be scheduler first.)

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
```

Step 3. 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 test3 to check the output.

```
./test3

./test3 > project2_test3.txt
```

Step 4. Implement Multilevel Queue Scheduling

In this step, you are supposed to add some code with respect to MLQ algorithm in **`case SCHED_MLQ`** in each function in ***scheduler.cc*** **and** ***test.4.cc***. In MLQ algorithm, upon its arrival, each thread will be put at the tail to a specific queue depending on its priority (MultiLevelList). If there are more than one thread with the same, they will be scheduled in the same queue.

Some notes are given to you:

1. **Multi-level Queue** scheduling algorithm is used in scenarios where the threads can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. One general classification of the threads is foreground threads and background threads. In a multi-level queue scheduling algorithm, there will be '**n**' number of queues, where 'n' is the number of groups the threads are classified into. Each queue will be assigned a priority and has its own scheduling. In this project, each single queue consists of threads with the **same thread priority** (*thread->getPriority()*), this priority is also the priority of this queue. Within each queue, the threads follow **FIFO** scheduling.
2. In this MLQ scheduling, it is a kind of **Non-preemptive** scheduling.
3. Function *Scheduler::SetNumOfQueues(int level)* should be called in **and** ***test.4.cc***, in order to initialize the MLQ, using the variable *NumOfLevel*
4. In ReadyToRun (Thread *thread), you should add codes in **`case SCHED_MLQ.`** `Each thread should be put into its corresponding queue in the queue List of` *`MultiLevelList[]`* `instead of the queue of` *`readyList.`*
5. In Scheduler::FindNextToRun (), you should add codes in **`case SCHED_MLQ.`** `The thread with higher priority will be put in the tail of the queue with higher index in the List` *`MultiLevelList[].`*

Then you should run “make clean” and then “make” to recompile the code and run test4 to check the output.

```
./test4
./test4 > project2_test4_1.txt
```

Next, you are supposed to change two code lines in ***test.4.cc***. In particular, in Line 27 and Line 29, change the startTime[] to {13, 0, 20, 18,10 ,7,12,6} and the priority[] to { 3, 3, 3, 3, 6 , 6, 6, 6 }.
Then you should run “make clean” and then “make” to recompile the code and run test4 to check the output.

```
./test4

./test4 > project2_test4_2.txt
```

Task 3: Explain the Results

1. Understand the output of test0 (FCFS scheduling), test1 (SJF scheduling), test2 (NP_Priority scheduling), test3 (Priority scheduling) and test4_1 and test_2 (MLQ). Then calculate the following performance of all the ~~three~~ five scheduling algorithms:

    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 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.)

3. Compare the performance of the two scheduling algorithms NP_Priority and Priority in the aspects mentioned in question 1, then discuss the pros and cons of each of the two scheduling algorithms. Please also explain the difference of the two results in detail.

4. Compare the results in project2_test4_1.txt and project2_test4_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 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_test3.txt	Output of test3
project2_test4_1.txt	Output of test4
project2_test4_2.txt	Output of test4
project2_report.txt	The answer to the questions in Task 3
test.4.cc	Source file