CSS 430
Program 2: Operating System Scheduler

Due Date: See the syllabus

1. Purpose

This assignment implements and compares two CPU scheduling algorithms, the round-robin scheduling and the multilevel feedback-queue scheduling. The step-by-step procedure to complete this assignment is:

1. Observe the behavior of the *ThreadOS Scheduler* which uses a Java-based round-robin scheduling algorithm. Consider why it may not be working strictly in a round-robin fashion.
2. Redesign the *ThreadOS Scheduler* using *Thread.suspend( )* and *Thread.resume( )* so that it will rigidly work in a round-robin fashion.
3. Implement a multi-level feedback-queue scheduler for *ThreadOS*.
4. Compare the rigid round-robin and multi-level feedback-queue schedulers using test thread programs.

2. Java Priority Scheduling

The *ThreadOS Scheduler* (see Scheduler.java) implements a naive round-robin scheduler. Given that *ThreadOS* is a java application and *ThreadOS* applications are Java threads, the *ThreadOS* scheduler is subject to the underlying Java Virtual Machine (JVM) Scheduler. Given this, there is no guarantee that a thread with a higher priority will immediately preempt the current thread. In its default implementation *Scheduler.java* does not strictly enforce a round-robin scheduling.

We can however modify the *ThreadOS* scheduler to enforce a rigid round-robin algortihm. By using the *Thread.suspend( )* and *Thread.resume( )* methods we can force threads to block or be ready for execution. Note that these methods have been deprecated and we must use them quite carefully in order to avoid deadlocks. For more information on these methods you can reference the documentation here: Java 2 Platform, API documentation .

3. Suspend and Resume

For this assignment you will use the *Thread.suspend( )* and *Thread.resume( )* methods in the *Scheduler* class of *ThreadOS*. You should avoid using *Thread.suspend( )* and *Thread.resume( )* in all of your future *ThreadOS* programs or in other java classes in this assignment.

The *suspend()* method suspends a target thread, whereas the *resume()* method resumes (moves the thread to a ready state) a suspended thread. To implement a rigid round-robin CPU scheduling algorithm, you will modify the *ThreadOS Scheduler* to dequeue the front user thread from the ready list and resume it by invoking the *resume()* method. When the quantum has expired you will suspend the thread it with the *suspend()* method.

The *suspend()* and *resume()* methods may cause a deadlock if a suspended thread holds a lock and a runnable thread tries to acquire this lock. To avoid these deadlocks, one must pay close attention when using them with the *synchronized*, *wait( )* and *notify( )* keywords. You will notice that the Scheduler Class of *ThreadOS* uses the *synchronized* keywords for the peek method. Don't remove or put additional *synchronized* keywords in the code, otherwise *ThreadOS* may deadlock.

When you compile Java programs that use deprecated methods such as *suspend( )* and *resume()*, you must compile them with the *-deprecation* flag. You will notice that the *javac* compiler will print out some warning messages when you do this. This is expected, just ignore them in this assignment.

```
uw1-320-00% javac -deprecation Scheduler.java
           ./Scheduler.java:128: warning: resume() in java.lang.Thread has been
           currentThread.resume( );
           ^
           ./Scheduler.java:136: warning: suspend() in java.lang.Thread has bee
           currentThread.suspend( );
```

```
                ^
2 warnings
uw1-320-00%
```

4. Structure of TheadOS Scheduler

The scheduling algorithm implemented in *ThreadOS*, (i.e., Scheduler.java) is similar that the one presented in class (see lecture slides). Instead of a processor control block (PCB), the data structure used to manage each user thread is the thread control block (TCB).

Thread Control Block (TCB.java)

The implementation of the TCB includes four private data members:

1. A reference to the corresponding thread object (*thread*).
2. A thread identifier (*tid*).
3. A parent thread identifier (*pid*).
4. The *terminated* variable to indicate whether the corresponding thread has been terminated.

The *TCB* constructor simply initializes those private data members with arguments passed to it. The *TCB* class also provides four public methods to retrieve its private data members: *getThread( )*, *getTid( )*, *getPid( )*, and *getTerminated( )*. In addition, it has the *setTerminated()* method which sets *terminated* true.

Private data members of Scheduler.java

In addition to three private data members from the lecture slide example, we have two more data such as a boolean array - *tids[]* and a constant - *DEFAULT_MAX_THREADS*, both related to TCB.

Data members	Descriptions
private Vector queue;	a list of all active threads, (to be specific, TCBs).
private int timeSlice;	a time slice allocated to each user thread execution
private static final int DEFAULT_TIME_SLICE = 1000;	the unit is millisecond. Thus 1000 means 1 second.
private boolean[] tids;	Each array entry indicates that the corresponding thread ID has been used if the entry value is true.
private static final int DEFAULT_MAX_THREADS = 10000;	tids[] has 10000 elements

Methods of Scheduler.java

The following shows all the methods of *Scheduler.java*.

Methods	Descriptions
private void initTid(int maxThreads)	allocates the *tid[]* array with a *maxThreads* number of elements
private int getNewTid( )	finds an *tid[]* array element whose value is false, and returns its index as a new thread ID.
private boolean returnTid(int tid)	sets the corresponding *tid[]* element, (i.e., *tid[tid]*) false. The return value is false if *tid[tid]* is already false, (i.e., if this *tid* has not been used), otherwise true.
public int getMaxThreads( )	returns the length of the *tid[]* array, (i.e., the available number of threads).
public TCB getMyTcb( )	finds the current thread's TCB from the active thread queue and returns it
public Scheduler(int quantum, int maxThreads)	receives two arguments: (1) the time slice allocated to each thread execution and (2) the maximal number of threads to be spawned, (namely the length of tid[]). It creates an active thread queue and initializes the *tid[]* array
private void schedulerSleep( )	puts the Scheduler to sleep for a given time quantum
public TCB addThread(Thread t)	allocates a new TCB to this thread *t* and adds the TCB to the active thread queue. This new TCB receives the calling thread's id as its parent id.
public boolean deleteThread( )	finds the current thread's TCB from the active thread queue and marks its TCB as terminated. The actual deletion of a terminated TCB is performed inside the run( ) method, (in order to prevent race conditions).

public void sleepThread(int milliseconds)	puts the calling thread to sleep for a given time quantum.
public void run( )	This is the heart of Scheduler. The difference from the lecture slide includes: (1) retrieving a next available TCB rather than a thread from the active thread list, (2) deleting it if it has been marked as "terminated", and (3) starting the thread if it has not yet been started. Other than this difference, the Scheduler repeats retrieving a next available TCB from the list, raising up the corresponding thread's priority, yielding CPU to this thread with *sleep( )*, and lowering the thread's priority.

The scheduler itself is started by *ThreadOS Kernel*. It creates a thread queue that maintains all user threads invoked by the *SysLib.exec(String args[])* system call. Upon receiving this system call, *ThreadOS Kernel* instantiates a user thread and calls the scheduler's *addThread(Thread t)* method. A new *TCB* is allocated to this thread and enqueued in the scheduler's thread list. The scheduler repeats an infinite *while* loop in its *run* method. It picks up a next available *TCB* from the list. If the thread in this *TCB* has not yet been activated (but instantiated), the scheduler starts it first. It thereafter raises up the thread's priority to execute for a given time slice.

When a user thread calls *SysLib.exit( )* to terminate itself, the *Kernel* calls the scheduler's *deleteThread( )* in order to mark this thread's *TCB* as terminated. When the scheduler dequeues this *TCB* from the circular queue and finds out that it has been marked as terminated, it deletes this *TCB*.

5. Statement of Work

Part 1: Modifying Scheduler.java with suspend( ) and resume( )

To begin with, run the *Test2b* thread on our *ThreadOS*:

```
$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2b
l Test2b
threadOS: a new thread (thread=Thread[Thread-6,2,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-8,2,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-10,2,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-12,2,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-14,2,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-16,2,main] tid=6 pid=1)
Thread[a] is running
....
```

Test2b spawns five child threads from *TestThread2b*, each named *Thread[a]*, *Thread[b]*, *Thread[c]*, *Thread[d]*, and *Thread[e]*. They prints out "*Thread[name]* is running" every 0.1 second. If the round-robin schedule is rigidly enforced to give a 1-second time quantum to each thread, you should see each thread printing out the same message about 10 times consecutively:

```
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[a] is running
Thread[b] is running
Thread[b] is running
....
....
```

However, messages will be interleaved on your terminal. Now, modify the *ThreadOS* ***Scheduler.java*** code using *suspend()* and *resume()* in order to force Round Robin behavior. The modifications will be the following:

1. to remove *setPriority(2)* (line 96) from the *addThread()* method,
2. to remove *setPriority(6)* (line 127) from the *run( )*method,
3. to replace *current.setPriority(4)* (line 143) with *current.resume( )*,
4. to remove *current.setPriority(4)* (line 148) from the *run( )* method, and finally
5. to repalce *current.setPriority(2)* (line 157) with *current.suspend( )*.

Compile Scheduler.java with javac, and thereafter test with Test2b.java if your Scheduler has implemented a rigid round-robin scheduling algorithm. If the Scheduler is working correctly, each *TestThread2b* thread should print out the same message 10 times consecutively.

Part 2: implementing a multilevel feedback queue (MFQS) scheduler

Modify Scheduler.java to implement a MFQS scheduler. The generic algorithm, for MFQS, is described in the textbook (section 5.3.6). The multilevel feedback queue scheduler operates according to the following specification:

1. It has three queues, numbered from 0 to 2.
2. A new thread's TCB is always enqueued into $\mathbf{Queue_0}$.
3. MFQS scheduler first executes all threads in $\mathbf{Queue_0}$ whose time quantum $\mathbf{Q_0}$ is half of the time quantum in Part 1's round-robin scheduler, (i.e. $\mathbf{Q_0}$**=500ms**).
4. If a thread in the $\mathbf{Queue_0}$ does not complete its execution for $Queue_0$'s time quantum, ($\mathbf{Q_0}$), then scheduler moves the corresponding TCB to $\mathbf{Queue_1}$.
5. If $\mathbf{Queue_0}$ is empty, it will execute threads in $\mathbf{Queue_1}$ whose time quantum is the same as the one in Part 1's round-robin scheduler, (i.e. $\mathbf{Q_1}$**=1000ms**). However, in order to react new threads in $\mathbf{Queue_0}$, your MFQS scheduler should execute a thread in queue 1 for only $\mathbf{Q_0}$**=500ms** one-half the $\mathbf{Queue_1}$ time quantum and then check if $\mathbf{Queue_0}$ has new TCBs pending. If so, it will execute all threads in $\mathbf{Queue_0}$ first. The interrrupted thread will be resumed when all of the $\mathbf{Queue_0}$ threads are handled. The resumed thread will only be given the remaining part of its time quantum.
6. If a thread in $\mathbf{Queue_1}$ does not complete its execution and was given a full $\mathbf{Queue_1}$'s time quantum, (i.e., $\mathbf{Q_1}$), the scheduler then moves the TCB to $\mathbf{Queue_2}$.
7. If both $\mathbf{Queue_0}$ and $\mathbf{Queue_1}$ are empty, the MFQS schedulers will execute threads in $\mathbf{Queue_2}$ whose time quantum is a double of the one in Part 1's round-robin scheduler, (i.e., $\mathbf{Q_2}$**=2000ms**). However, in order to react to threads with higher priority in $\mathbf{Queue_0}$ and $\mathbf{Queue_1}$, your scheduler should execute a thread in $\mathbf{Queue_2}$ for $\mathbf{Q_0}$ increments and check if $\mathbf{Queue_0}$ and $\mathbf{Queue_1}$ have new TCBs. The rest of the behavior is the same as that for $\mathbf{Queue_1}$.
8. If a thread in $\mathbf{Queue_2}$ does not complete its execution for $\mathbf{Queue_2}$'s time slice, (i.e., $\mathbf{Q_2}$), the scheduler puts it back to the tail of $\mathbf{Queue_2}$.

Again, compile your Scheduler.java and test with Test2b.java to assure that your Scheduler has implemented a multilevel feed back-queue scheduling algorithm.

Part 3: Conducting performance evaluations

Run *Test2.java* on both your round-robin and the multilevel feed back-queue scheduler.

```
$ java boot
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,2,main] tid=0 pid=-1)
-->l Test2
```

Similar to *Test2b*, *Test2* spawns five child threads from *TestThread2b*, each named *Thread[a]*, *Thread[b]*, *Thread[c]*, *Thread[d]*, and *Thread[e]*. They prints out nothing but their performance data upon their termination:

```
thread[b]: response time = 2012 turnaround time = 3111 execution time = 1099
thread[e]: response time = 5035 turnaround time = 5585 execution time = 550
....
```

The following table shows their CPU burst time:

Thread name	CPU burst (in milliseconds)
Thread[a]	5000
Thread[b]	1000
Thread[c]	3000
Thread[d]	6000
Thread[e]	500

Compare test performance results between Part 1 and Part 2. Discuss how and why the multilevel feed back-queue (MFQS) scheduler has performed better/worse than the round-robin scheduler.

6. What to Turn In

- Part 1:
  - Your Scheduler.java (i.e. version with suspend/resume round robin)
  - Execution output when running Test2b.java
- Part 2:
  - Your Scheduler.java (i.e. version with MFQS implemented)
  - Execution output when running Test2b.java
- Report (*.pdf or *.docx file and *.xlsx for spreadsheet Gantt charts)
  - Clearly explain the design and algorigthm for Part 2. You may want to use flowcharts or other software diagrams tools (e.g. DrawIO).
  - Compare the test results between Part 1 and Part 2 **using Test2.java**. Discuss how and why your multilevel feed back-queue scheduler has performed better or worse than your round- robin scheduler. **This is an important part of the assignment.**
  - Consider what would happen if you were to implement the part 2 based on FCFS without pre-emption rather than Round Robin. Your discussion may focus on what happens if you run **Test2.java** in this FCFS-based **$\mathbf{Queue_2}$** (e.g. quantum=2000ms). **Note:** It is recommended that you build 3 different *Gantt charts* so that your execution performance numbers can clearly be visualized for your discussion.

7. Grading Guide

Grading rubric is here.

8. FAQ

This website could answer your questions. Please click here before emailing the professor :-).