

Program 2

Tuesday, January 26, 2016 2:15 PM

Scheduling

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 algorithm. 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 deprecated
currentThread.resume( );
^
./Scheduler.java:136: warning: suspend() in java.lang.Thread has been deprecated
currentThread.suspend( );
^
2 warnings
uw1-320-00%
```

4. Structure of ThreadOS 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)

```
1 public class TCB {
2     private Thread thread = null;
3     private int tid = 0;
4     private int pid = 0;
5     private boolean terminate = false;
6
7     // User file descriptor table:
8     // each entry pointing to a file (structure) table entry
9     public FileTableEntry[] ftEnt = null;
10
11     public TCB( Thread newThread, int myTid, int parentTid ) {
12         thread = newThread;
13         tid = myTid;
14         pid = parentTid;
15         terminated = false;
16
17         // The following code is added for the file system
18         ftEnt = new FileTableEntry[32];
19         for ( int i = 0; i < 32; i++ )
20             ftEnt[i] = null; // all entries initialized to null
21         // fd[0], fd[1], and fd[2] are kept null.
22     }
23 }
```

[view_raw_TCB.java](#) hosted with ❤ by [GitHub](#)

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 <i>tid[]</i> array with a <i>maxThreads</i> number of elements
private int getNewTid()	finds an <i>tid[]</i> array element whose value is false, and returns its index as a new thread ID.
private boolean returnTid(int tid)	sets the corresponding <i>tid[]</i> element, (i.e., <i>tid[tid]</i>) false. The return value is false if <i>tid[tid]</i> is already false, (i.e., if this <i>tid</i> has not been used), otherwise true.
public int getMaxThreads()	returns the length of the <i>tid[]</i> 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 <i>tid[]</i>). It creates an active thread queue and initializes the <i>tid[]</i> 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 <i>t</i> 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 <i>run()</i> 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 <i>sleep()</i> , 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)
-->1 Test2b
1 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 print 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 mixed up on your display, so the output may not look exactly like this example.

Next, modify this ThreadOS' Scheduler.java code using suspend() and resume(). The modification will be:

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

Part 2: implementing a multilevel feedback-queue scheduler

Modify your scheduler and implement a multilevel feedback-queue scheduler. The generic algorithm is described in the textbook. Your multilevel feedback-queue scheduler must have the following specification :

1. It has three queues: 0, 1 and 2
2. A new thread's TCB is always enqueued into queue 0
3. Your scheduler first executes all threads in queue 0. The queue 0's time quantum is timeslice/2, i.e., half of the one used in Part 1 Round-robin scheduler
4. If a thread in the queue 0 does not complete its execution for queue 0's time slice, (i.e., timeSlice / 2), the scheduler moves the corresponding TCB to queue 1.
5. If queue 0 is empty, it will execute threads in queue 1. The queue 1's time quantum is the same as the one in Part 1's round-robin scheduler, (i.e., timeSlice). However, in order to react new threads in queue 0, your scheduler should execute a thread in queue 1 for timeSlice / 2 and then check if queue 0 has new TCBs. If so, it will execute all threads in queue 0 first, and thereafter resume the execution of the same thread in queue 1 for another timeSlice / 2.
6. If a thread in queue 1 does not complete its execution for queue 1's time quantum, (i.e., timeSlice), the scheduler then moves the TCB to queue 2.
7. If both queue 0 and queue 1 is empty, it can execute threads in queue 2. The queue 2's time quantum is a double of the one in Part 1's round-robin scheduler, (i.e., timeSlice * 2). However, in order to react threads with higher priority in queue 0 and 1, your scheduler should execute a thread in queue 2 for timeSlice / 2 and then check if queue 0 and 1 have new TCBs. The rest of the behavior is the same as that for queue 1.
8. If a thread in queue 2 does not complete its execution for queue 2's time slice, (i.e., timeSlice * 2), the scheduler puts it back to the tail of queue 2. (This is different from the textbook example that executes threads in queue 2 with FCFS, see figure 1 below.)

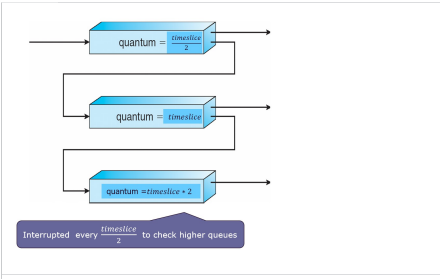


Figure 1: summary of modifications compared to the textbooks approach

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

Part 3: Conducting performance evaluations

Run Test2.java on both your round-robin and the multilevel feedback-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)
-->1 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 your multilevel feedback-queue scheduler has performed better/worse than your round-robin scheduler.


Code availability

Option 1 Linux Lab: (Preferred option)
The complete ThreadOS source code and .class files can be found in the UW1-320 Linux machines /usr/apps/CSS430/ThreadOS/

Copy all compiled class files into your directory and thereafter compile your shell.java. Do not try to compile the ThreadOS source code, some portion of which cannot be accessed. (Those are your future assignments.)

Option 2 [Right here \(not recommended, only in case of emergency\)](#)

In case you have issues accessing the lab, you can use the following, but it is highly recommended you use the files from the linux lab instead:

 P2	Inside this zip: <ul style="list-style-type: none"> The Class folder contains .class files needed for assignment 2, however, you can just reuse the .class files from the P1.zip The src folder contains the .java source files you need to edit
---	--

6. What to Turn In

- Part 1:
 - Your Scheduler.java
 - Execution output when running Test2.java
- Part 2:
 - Your Scheduler.java rename this file to Scheduler_part2.java so it doesn't conflict with Part1
 - Execution output when running Test2.java
- Report (either .pdf or .docx file)
 - Clearly explain the design and algorithm for Part 2. You may want to use flowcharts.
 - Compare the test results between Part 1 and Part 2. Discuss how and why your multilevel feedback-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 rather than Round Robin. Your discussion may focus on what happens if you run Test2.java in this FCFS-based queue 2.
 - Document your design:
 - What files you have and what they contain? (what do they contain)
 - How to compile it/ run it?
 - Assumption is you put all of them under the same folder

7. Grading Guide

[gradeguide2.txt](#) is the grade guide for the assignment 2.

8. Notes

- Check Canvas for due date
- Read all the notes and updates posted directly under the Program 2 tab.
- This is a live document that is improved based on your questions, input, and feedback.

Example of Output

```
threadOS ver 1.0:
Type ? for help
threadOS: a new thread (thread=Thread[Thread-3,5,main] tid=0 pid=-1)
-->! Test2
1. Test2
threadOS: a new thread (thread=Thread[Thread-5,5,main] tid=1 pid=0)
threadOS: a new thread (thread=Thread[Thread-7,5,main] tid=2 pid=1)
threadOS: a new thread (thread=Thread[Thread-9,5,main] tid=3 pid=1)
threadOS: a new thread (thread=Thread[Thread-11,5,main] tid=4 pid=1)
threadOS: a new thread (thread=Thread[Thread-13,5,main] tid=5 pid=1)
threadOS: a new thread (thread=Thread[Thread-15,5,main] tid=6 pid=1)
Thread[e]: response time = 5999 turnaround time = 6500 execution time = 501
Thread[b]: response time = 2999 turnaround time = 10001 execution time = 7002
Thread[c]: response time = 3998 turnaround time = 21001 execution time = 17003
Thread[a]: response time = 1999 turnaround time = 29003 execution time = 27004
Thread[d]: response time = 4999 turnaround time = 33002 execution time = 28003
-->q
q
Superblock synchronized
```

9. FAQ

This website could answer your questions: click [here](#) and checking out the [Notes](#) before emailing the professor :-)