COP5615 – Distributed Operating Systems Principles – Spring 2012
Project 1

Assigned: Jan. 27, 2012 (Friday)
Due Dates: Feb 10, 2012 (Local), Feb13 (EDGE), 11:55 PM EDT
Programming Language Allowed: Java (only)
Platforms Allowed: sand, rain (SunOS Machines) and/or lin113-01 (Linux Machines)
Java versions available: OpenJDK Runtime Environment (build 1.6.0_0-b11) on lin113-01
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_10-b03) on sand/rain

Project Overview:
The purpose of the project is to get you familiar with multithreading programming. In addition, you will have a chance to review some of the process synchronization concepts that you have learned from your previous operating system class. Specifically, we will implement the exercise problem, Ex. 3-9, in Chapter 3 on page 95 of the textbook. We will implement a mutual exclusion algorithm for *n* processes (threads in our project). We consider a binary tree with *n* leaves, one for each process. *n* is a power of 2. Each internal node (non-leaf) of the tree has an associated semaphore. A process desiring to enter into its critical section performs P operations on successive semaphores from the first non-leaf to the root of the tree, executes its critical section, and then performs V operations in the reverse order back down the tree. Each process repeats this request for critical section *m* times and then exits. The program terminates gracefully when all *n* processes have completed m times of mutually exclusive executions. We will assume *n*=8 and *m*=3 for our testing. However, your program should not be hard-coded for this choice of *n* and *m*.

In Java, semaphores can be defined as a class with methods, P and V.
Synchronization can be achieved through the *synchronized* keyword. This keyword can be used for any Java object as well as for any method. The mechanism is very similar to the *monitor* abstraction. If an object is defined as synchronized, that object can be accessed by only one thread at a time. Similarly a Java object can have synchronized methods and hence becomes a monitor. Only one thread can execute an instance of the synchronized method at a time. Entry to the method is protected by a monitor lock around it. If there is any other thread that calls one of the synchronized methods on the same object simultaneously, it will be blocked (suspended).

The implementation of P and V requires some mechanisms for *wait* and *signal* of processes (like those in the monitor approach for process synchronization). Java provides three methods for this purpose: **wait()**, **notify()**, **notifyAll().** Please notice that notify() wakes up only one arbitrary thread and notifyAll() does not guarantee any ordering of the thread requests.

Helpful Resources:
Multi-threading Tutorial: http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html
Java Synchronization Tutorial: http://www.java-samples.com/showtutorial.php?tutorialid=306

In addition, we need a configuration file that provides information about the overall system.

System Configuration:
An example of the system configuration shown below is specified in the file, *system.properties*.

```
n=8
m=3
P1.opTime=400
P1.sleepTime=200
P2.opTime=400
P2.sleepTime=250
P3.opTime=400
P3.sleepTime=300
P4.opTime=400
P4.sleepTime=350
P5.opTime=500
P5.sleepTime=400
P6.opTime=500
P6.sleepTime=450
```

P7.opTime=500
P7.sleepTime=500
P8.opTime=500
P8.sleepTime=550

Note that this is only a sample configuration file. We will use different configurations to grade your programs, so you need to do the same for testing.

Each process begins with some *sleepTime* and then tries to acquire the semaphores with some P operations. Once it is successful in reaching and obtaining the root semaphore, it enters into the critical section and executes with a simulated *opTime*. The process then releases the semaphore using V operations. The process terminates itself after *m* successful executions of the critical section. The pseudo code describing this process is:

for i=1 to m {sleep(sleepTime); P operations on the corresponding semaphores; sleep (opTime); V operations in reverse}.

Output Format:
The name of the program we will run should be named **"Start.java". Please note that the file name begins with capital 'S' and NOT a small case 's'.** The "Start.java" contains the Main function.

Print the status of all semaphores and processes only when a process finished the critical section and just before releases the semaphore it occupied. This means the output file only contains 8*3 status outputs if n=8,m=3. Reaching the root semaphore means that the process already obtains all the semaphores to enter into the critical section.
The actual output format is describes as follows and please follow it exactly.

For the purpose of easy illustration, we assume we only have 4 processes in the example although we will have 8 processes in our testing case. We use a quadruple, (semaphore number, value, holding process, waiting process), to show the state of each semaphore. The following shows the state of root semaphore and the two next level semaphores.

(s1, 0, p[1,1], p[3,1])
(s2, 0, p[1,1], p[2,1]) (s3, 0, p[3,1], p[4,1])
Unix time=500

- si is the name of the semaphore ranging from s1 to sn.
- value: equals to 1 if it is free, 0 if it has been acquired.
- holding process: the process which has acquired that semaphore. The format of the process is p[x,y]. x (from 1 to n) is the process number, and y (from 1 to m) is the y_{th} request to enter into the critical section. If no process is holding that semaphore, the p[x,y] should be p[0,0].
- waiting process: the process which is waiting for that semaphore. Its format is similar to the holding process. It should be p[0,0] if no process is waiting.
- UnixTime: UnixTime=CurrentUnixTime-BeginUnixTime(in millisecond). For example, your program is launched at Unix time 1327260100599(2:22 PM, 2012-1-22). The current event happens at Unix time 1327260101099. So the Unix time for the current event is 500.

Note: Nodes in the same tree level are separated by one TAB (eight empty spaces) in the same line. Example: (s2, 0, p[1,1], p[2,1]), one TAB separated, (s3, 0, p[3,1], p[4,1]).

The output file should be something like this.
(s1, 0, p[1,1], p[3,1])
(s2, 0, p[1,1], p[2,1]) (s3, 0, p[3,1], p[4,1])
UnixTime=200
(s1, 0, p[3,1], p[2,1])
(s2, 0, p[2,1], p[0,0]) (s3, 0, p[3,1], p[4,1])
UnixTime=600
(s1, 0, p[2,1], p[4,1])
(s2, 0, p[2,1], p[1,2]) (s3, 0, p[4,1], p[0,0])
UnixTime=1000
(s1, 0, p[4,1], p[1,2])
(s2, 0, p[1,2], p[0,0]) (s3, 0, p[4,1], p[3,2])

…
sequence:(this line is included in the output file)

p[1-1]:200(UnixTime)
p[3-1]:UnixTime
p[2-1]:UnixTime
p[4-1]:UnixTime
p[1-2]:UnixTime
p[3-2]:UnixTime
…
sequence is the sequence of events of all n*m entries to the critical section in the ascending order of their UnixTime. These event times should be the same as the UnixTime printed with the semaphore state.

The output file (log file) should be named 'event.log'.
You are required to submit a 1 page TXT report. Name it **report.txt**. Note the small case 'r' and **NOT** capital 'R'. Your report must contain a brief description telling us how you developed your code, what difficulties you faced and what you have learned. Remember this has to be a simple text file and not a MSWORD or PDF file. It is recommended that you use wordpad.exe in windows or vi or gedit or pico/nano applications in UNIX/Linux to develop your report. Avoid using notepad.exe.

You should tar all you java code files including the txt report in a single tar file using this command on Linux/SunOS:

```
tar cvf project1.tar <file list to compress and add to archive>
```

We will use the following scripts to test and execute your project:

```
#!/bin/sh
mv ?*.tar proj1.tar
tar xvf proj1.tar; rm *.class
rm *.log
javac *.java; java Start
```

```
#!/bin/sh
cat report.txt
cat event.txt
```

Please test your tar file using these scripts before you submit on line. **If your code fails to execute with these scripts, we will initially assign a score of 0 pending resolution.**

Grading Criteria:

Correct Implementation/Graceful Termination:	90%
Elegant Report:	10%
Total	100%

Late Submission Policy:
Late submissions are not encouraged but we rather have you submit a working version than submit a version that is broken or worse not submit at all.

Every late submission will be penalized 15% for each day late for up to a maximum of 5 days from the due date.
Please follow strictly the university policy on plagiarism. We will run the Moss or Jplag system to check the similarity among projects.