Version: 1.0 Author: PH

!"# $% September 06, 2007

& % (11:59 pm) for LOCAL students (Sections: 1093, 9333)

(11:59 pm) for EDGE students (Sections: 5667, 8170, 8404)

$& !#%

When you write a program in which several concurrent threads are competing for resources, you must take precautions to avoid and . Starvation occurs when one or more threads in your program are blocked indefinitely from gaining access to a resource and, as a result, cannot make progress. Deadlock, the ultimate form of starvation, occurs when two or more threads are waiting on a condition that cannot be satisfied. Deadlock most often occurs when two (or more) threads are waiting for the other(s) resulting in a cyclic waiting that no process can continue.

For this project you will use Java as the programming medium. This project requires you to implement a multi-threaded application and further achieve synchronization among threads.

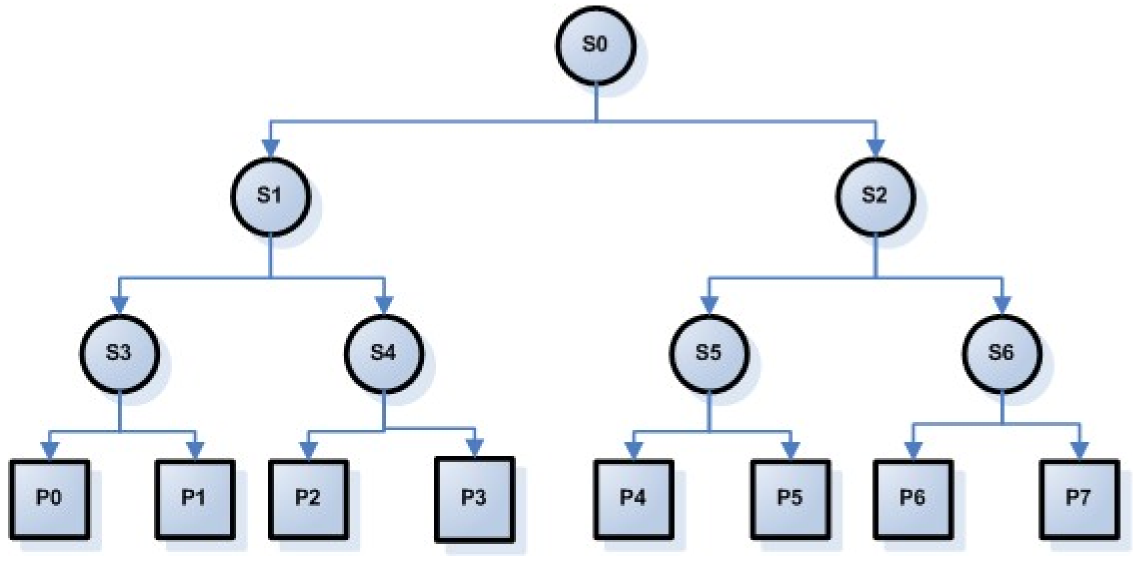
There are two ways to implement threads in Java. The first one is implementing the interface and the second is to subclass the class and overriding the method. Feel free to choose either way that is most convenient to you and suitable to the application requirements.
Synchronization in Java is achieved through the ' # (#!) $ 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. At any time, only one synchronized method can be executed by a thread on the object. If there is any other thread that calls one of the synchronized methods on the same object simultaneously, it cannot continue and therefore is suspended.

In addition we need a mechanism to block and unblock a process. Java provides two methods for this purpose: * ! +, and !- ' +, . These are similar, but not quite the same, to the and operations of the monitor approach for process synchronization.

. / ! %

Your Java application must take ' ' as command line parameter to the program. We will assume that there is a top level "shared resource" which all threads will be competing to access. The mutual exclusion will be achieved using a number of semaphores which will be assumed to be logically arranged in a full binary tree hierarchy. These semaphores will be logically treated as internal nodes of the binary tree and the threads will be treated as leaf nodes of this logical binary tree.

For example suppose your application is to start 8 threads, then your program must calculate that it would need 7 semaphores to construct the logical semaphore binary tree. A illustrative diagram is given below for further explanation:



According to this diagram process P0 (or thread with id P0) in order to get access to the shared resource must do a down operations and up operations on the semaphores in this order:

```
down(s3)
down(s1)
down(s0)
        access the critical section / shared resource
up(s0)
up(s1)
up(s3)
```

Your program must also compute automatically given a process id say p0, p1, p2 etc. which semaphores it must do up and down operations on and in which order.

Remember that the 7 semaphores in the diagram above are in fact global in their scope, the tree representation is just a logical arrangement in order to achieve mutual exclusion. You must use Java's wait() and notifyAll() in order to achieve similar functionality of semaphore operations up and down.

0 & ! #   1&! / #   %

1. Your Java file containing main() should be named start.java.
2. Your program must compile and execute properly on CISE # $ and !# (Solaris) machines. Your projects will #__ be tested on any other machines.
3. Execution must begin with command: 2   3number of threads4
4. All your code files must reside at the root level of the project directory and it is advised not to use java package primitive at the start of your code.
5. You will assume all your code files will be expanded in a directory called $ containing no other subdirectory.

6. We will use this script to test your project:

```
#!/bin/sh
tar xvf proj1.tar; rm *.class
javac *.java; java start 8
```

$ $! ! # ! %

Once the threads are created they sleep for a random duration (between 100 – 1000 millisecond) before become active. This is to create randomness in the program execution. Once the top level semaphore lock operation by a process (thread) succeeds, it must wait for a random time (500 – 1000 millisecond) before releasing all locks. Also after releasing all the locks the thread must again sleep for random amount of time before becoming active again to access the shared resource (250 – 500 millisecond).
& $ / & - 5 (/ 6 & 6& - / 2 ' ' 7 ($ - / " + - & ,
5! # . / #$ 5! 2 ' . 5 # $!-- # 0 & ! # +. & - # $ /# , .& (
& 6& / # - / #$ ' / & . 6 2 $7

/ 6 ! & 6& %

```
Starting thread id: p0
Starting thread id: p1
Starting thread id: p2
Starting thread id: p3
Starting thread id: p4
Starting thread id: p5
Starting thread id: p6
Starting thread id: p7
Process p7 trying to lock semaphore s6 : success
Process p0 trying to lock semaphore s3 : success
Process p6 trying to lock semaphore s6 : failure [waiting]
Process p7 trying to lock semaphore s2 : success
Process p2 trying to lock semaphore s4 : success
Process p7 trying to lock semaphore s0 : success
Process p7 in critical section now. Sleeping for 847 milliseconds ..
Process p3 trying to lock semaphore s4 : failure [waiting]
Process p2 trying to lock semaphore s1 : success
Process p5 trying to lock semaphore s5 : success
Process p2 trying to lock semaphore s0 : failure [waiting]
Process p7 releasing lock on semaphore s0
Process p7 releasing lock on semaphore s2
Process p7 releasing lock on semaphore s6
....
```

6 %

You are to provide a file named report.txt that should include:

- Problem definition, proposed solution and how it works
- How you designed and implemented the system
- What are the results you got (in plain English, discuss the output of your system, do not copy & paste the program output)
- All the bugs or problems known, any missing items and limitations of your implementation - IF ANY. (honesty deserves additional points)
- Any additional sections you see necessary

Please note that your reports MUST consist of your own sentences, if you have to copy anything from anywhere you have to quote it and provide reference.

Also a perfect program does not necessarily deserve full points if it is not complemented with a good report. A good report is a brief one that help its reader to understand the system thoroughly from the problem definition through the limitations.

&. / ! ! # &!$!# %

1. Only submit your Java code files, do not include .class files in your submissions
2. Use 2- 6 7 3file list4 to tar your submission.
3. Include a 1 - 3 page(s) project documentation in .txt format. Name it 6 7 0
4. You must submit the tar file after logging into your UF-IBA account. Select Project from the drop down menu and select the appropriate tar file on your local computer before submitting.
5. UF-IBA allows file uploads up to 1 Mb in size, make sure your tar file does not exceed the size.
6. Please preserve the submission confirmation generated by UF-IBA after successful upload.

$!#" ! ! %

Correct Implementation/Execution/Graceful Termination/Nonexistence of Run-away processes	! "
Report	"
Stylish Code/Readability/Comments/Structure and Readability of Output	! "
TOTAL:	*100%*

8 2 & :

You may find tutorials and sample code on general Java programming, Threads and TCP/IP programming in Java by following the link http://java.sun.com and searching for specific information you need. Newer thread synchronization primitives are made available since Java version 5.0. You can read more about them in this on line article: http://www.onjava.com/pub/a/onjava/excerpt/jthreads3_ch6/index1.html You are free to implement this project using these newer constructs if you prefer.

&#9 5 '6 %

Your threads should terminate gracefully. While testing your programs run-away processes might exist. However these should be killed frequently. Since the department has a policy on this matter, your access to the department machines might be restricted if you do not clean these processes.

To check your processes running on a Unix:	`ps -u <your-username>`
To kill all Java processes easily in Unix, type:	`skill java`
To check runaway processes on remote hosts:	`ssh <host-name> ps -u <your-username>`
and to clean:	`ssh <host-name> skill java`