COP5615 – Distributed Operating Systems Principles – Spring 2011
Project 2

Assigned: Feb 04, 2011 (Friday)
Due Dates: February 18, 2011 (Local), February 21 (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:
In this project we will develop a distributed client/server implementation of the Concurrent Readers and Exclusive Writer (CREW) problem. More specifically, we are to implement strong reader preference of the CREW as described on pages 71-72 and you can get an idea how to implement strong reader preference by referring weak reader preference solution shown in Figure 3.13 in our textbook. (also in lecture note chapter 3).

The implementation will consist of a server that maintains a shared object (an integer variable) for concurrent read and exclusive write accesses and several reader/writer clients on remote machines communicating with the server using socket on TCP/IP protocol suite. The readers and writers are distributed processes running independently on various CISE machines.

Synchronization in Java is achieved through 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 cannot continue and therefore is suspended.

In addition we need a mechanism to synchronize access to shared variables. Java provides three methods for this purpose: **wait()**, **notify()**, **notifyAll().** These are similar to the Wait and Signal operations of the monitor approach for process synchronization. Please notice that notify() wakes up only one arbitrary thread and notifyAll() does not guarantee any ordering of the thread requests.

This project requires three programs that run on different machines. These are the server and the client (reader and writer) programs. In addition to these programs we need a configuration file that provides information about the overall system.

Helpful Resources:
Socket Programming Tutorial: http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
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

System Configuration:
The system configuration shown below is in the file system.properties

```
RW.server=sand.cise.ufl.edu
RW.numberOfReaders=4
RW.reader1=lin114-01.cise.ufl.edu
RW.reader2=lin114-02.cise.ufl.edu
RW.reader3=lin114-03.cise.ufl.edu
RW.reader4=lin114-04.cise.ufl.edu
RW.numberOfWriters=4
RW.writer5=lin114-05.cise.ufl.edu
RW.writer6=lin114-06.cise.ufl.edu
RW.writer7=lin114-07.cise.ufl.edu
RW.writer8=lin114-08.cise.ufl.edu
RW.numberOfAccesses= 3
RW.reader1.opTime=400
RW.reader1.sleepTime=510
RW.reader2.opTime=400
RW.reader2.sleepTime=490
RW.reader3.opTime=400
RW.reader3.sleepTime=495
RW.reader4.opTime=400
RW.reader4.sleepTime=500
RW.writer5.opTime=500
RW.writer5.sleepTime=600
RW.writer6.opTime=500
RW.writer6.sleepTime=610
RW.writer7.opTime=500
RW.writer7.sleepTime=605
RW.writer8.opTime=500
RW.writer8.sleepTime=600
```

RW.server is the address of the host on which the server runs. RW.reader1/RW.writer1 is the address of the host on which the reader/writer with ID 1 runs, and so on. RW.numberOfReaders is the number of readers, RW.numberOfWriters is the number of writers, RW.numberOfAccesses indicates how many times a reader/writer should read/write the values in shared object. RW.opTime is an operation time that each reading and writing take. RW.sleepTime is a sleep time for readers/writers between two requests. Each reader and writer should sleep for RW.sleepTime before making a next request. Time unit for RW.opTime and RW.sleepTime is ms.

For server port number, it is not assigned in the system.properties. You should use a system assigned free port when you start server. This can be achieved by leaving the port number out while creating the Serversocket object in Java. **DO NOT USE PREASSIGNED HARDCODED PORT NUMBER** inside the server or client code. It may lead to grade penalty.

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

Server Description:

The name of the program we will run is **"start.java". Please note that the file name begins with small case 's' and NOT a captical 'S'.** This program is responsible for starting the server on RW.server specified in system.properties and the clients on remote machines. First it should create a thread, which will run the server code. Then, it should start clients on remote machines. Therefore, the server will be running in the background as a thread. The server thread accepts remote requests from clients (readers and writers) and spawns a new thread for each client. In other words, the threads are per client, not per request. These threads represent the remote clients and act on their behalves. Hence for each writer or reader, there will be a thread running at the server site until the client is done. This means our server is non-blocking and is able to service concurrent requests without waiting for any reading or writing process to be completed.
This program will read the configuration file mentioned above and start up the system accordingly.
The server thread needs to maintain the number of readers and writers served and when all the clients are done, should terminate gracefully.

Reader and Writer Client Description:

Clients are started on remote machines as processes. Therefore they will run in the background. Readers send their request type to the server as read and receive a packet that contains the value of the shared object. Writers send their request type as write and also the value (its ID) to be written to the shared object. The initial value of the shared object is assumed to be -1.

Output Format:

The server should print out its actions in the following format:

```
Read Requests:
Service Sequence     Object   Value      Read by          Num of Readers
------------------    ---------------     ---------         --------------
        1                   -1              R4                  1
        2                   -1              R1                  2
        3                   -1              R3                  3
        4                   -1              R2                  4
        5                   -1              R1                  3
        6                   -1              R1                  3
        7                   -1              R4                  2
        8                   -1              R2                  2
        9                   -1              R3                  2
        15                   8              R4                  1
        16                   8              R3                  2
        17                   8              R2                  3

Write Requests:
Service Sequence       Object  Value           Written by
-------------------     ---------------         -----------
        10                 8                    W8
        11                 6                    W6
        12                 5                    W5
        13                 5                    W5
        14                 8                    W8
        18                 6                    W6
        19                 7                    W7
        20                 5                    W5
        21                 8                    W8
        22                 6                    W6
        23                 7                    W7
        24                 5                    W5
```

where *Service Sequence* is the sequence number that reader/writer accesses the shared object, *Object Value* is the value now saved in the object, *Read by* is the ID of the readers, *Num of Readers* is the number of readers currently accessing the news. *Written by* is the ID of the writer updating the shared object.

Each reader client should print out its actions in the following format:

```
Client type: Reader
Client Name: 1

Request Sequence    Service Sequence    Object Value
----------------    ----------------    ------------
2                        2                 -1
9                        5                 -1
10                       6                 -1
```

The *Request Sequence* is the sequence number of the reader trying to access the shared object. Both request sequence and service sequence are generated by the server for the purpose of showing that the requirements for the strong CREW problem are followed.

The name of the file that will be written by the clients must be **log** concatenated with the **ID of the reader** (i.e., R1.log, R2.log).

Each writer client should print out its actions in the following format:

```
Client type: Writer
Client Name: 7

Request Sequence           Service Sequence
----------------          ------------
       5                       10
      11                       14
      21                       21
```

The *Request Sequence* is the sequence number of the writer trying to access the news.

The name of the file written by the clients must be **log** concatenated with the **ID of the writer** (i.e. W5.log, W6.log).

How to start processes on remote machines

To start a process on a remote machine we use the remote login facility ssh in Unix systems. It accepts the host (computer) name as a parameter and commands to execute separated by semicolons.

To execute the ssh command from a Java program, you should use the exec() method of the Runtime class.

First, we need to get the current directory of the user (all the files necessary for this project should be in the same directory, and Start.java will be run in this directory), as follows:

```
String path = System.getProperty("user.dir");    // get current directory of the user
```

Second, we invoke exec() method as in the following:

```
Runtime.getRuntime().exec("ssh " + host + " cd " + path + " ; java Site " + ... arguments ... );
```

Here ssh is the command to start a process that will run on the host giving by the program variable host. Immediately after the host name you need to specify the cd command to make the current working directory of the new remote process be the same as the starting program. Hence the output of the remote processes will all be directed to the same directory where we originally started the system. The path is another program variable, which specifies the full path name of the directory where we invoked the Start.java. Note that you should use this path as exactly returned by the Java method. You need not append anything to this string.

After the semicolon you should specify the name of the program to run on the remote machine, which is specified as Site.java in the above example. Following the program name, of course, you need to specify the necessary arguments to the program.

Additional instructions:

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 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 project2.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 proj12.tar
tar xvf proj2.tar; rm *.class
rm *.log
javac *.java; java start


----------------------------------------

#!/bin/sh
cat R*.log
cat W*.log
cat report.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.**

Note on Runaway Processes:
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`

Grading Criteria:

Correct Implementation/Graceful Termination:	80%
Elegant Report:	10%
Nonexistence of runaway processes upon termination:	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.