

COP5615

Operating System Principles

Fall 2007

Project 2

Date Assigned: September 20, 2007

Date Due: October 4, 2007 (11:59 pm) for LOCAL students (Sections: 1092, 9333)

October 8, 2007 (11:59 pm) for EDGE students (Sections: 5667, 8170, 8404)

Introduction:

In this project we will develop a distributed client/server implementation of the *Concurrent Readers and Exclusive Writer (CREW)*. The implementation will consist of a server that maintains a shared object (an integer variable) for read and 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.

Recall that the definition of the *writer preference* CREW is: An arriving reader must wait if there is an active writer or waiting writer(s). When a writer client completes its writing and there are both waiting readers and writers, we always choose a writer client next, and it is in the FCFS order among the waiting writers. If there is no waiting writer, all waiting readers would be allowed to join the access concurrently. Please notice that `notify()` wakes up only one arbitrary thread and `notifyAll()` does not guarantee any ordering of the thread requests.

The definition of the socket is: A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent and the socket API allows application programs to communicate remotely over the Internet using the concept of ports in the transport layer as service access points. A socket is analogous to a file (an example of transparency) and is used very much like a file. Its send/receive primitives correspond to the write/read operations in a file system. In this project, we will use sockets over TCP/IP.

In TCP/IP protocol suite there are two transport layer protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP provides connection-oriented, reliable, and ordered delivery of packets. UDP is a best-effort connectionless protocol, which requires no connection establishment between communicating hosts. The distinction also exists in the higher layer (socket abstraction), which uses these protocols. We will use TCP in this project.

Problem Details:

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

More about the socket communication:

<http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/socket.html>

http://en.wikipedia.org/wiki/Berkeley_sockets

System Configuration

The system configuration (with a sample port number, 49053) shown below is in the [*system.properties*](#) file to be read by your program.

```
RW.server=sand.cise.ufl.edu
RW.server.port=49053
RW.numberOfWorkers= 4
RW.reader1=sun114-11.cise.ufl.edu
RW.reader2=sun114-12.cise.ufl.edu
RW.reader3=sun114-13.cise.ufl.edu
RW.reader4=sun114-14.cise.ufl.edu
RW.numberOfWriters=4
RW.writer5=sun114-21.cise.ufl.edu
RW.writer6=sun114-22.cise.ufl.edu
RW.writer7=sun114-23.cise.ufl.edu
RW.writer8=sun114-25.cise.ufl.edu
RW.numberOfAccesses= 3
```

RW.server is the address of the host on which the server runs. RW.server.port is the well-known port number on which the server socket will be listening. RW.reader X and RW.writer Y are the addresses of the hosts on which the reader/writer with ID X and Y runs respectively. RW.numberOfWorkers is the number of readers, RW.numberOfWriters is the number of writers, RW.numberOfAccesses indicates how many times a reader/writer will read/write the values in the shared object before it exits.

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 Specification

The main program should be named start.java. This program is responsible for starting up the server in the local host and the clients on remote machines. First it should create a thread, which will run the server code. Then, it should start the clients (readers and writers) on remote machines. Therefore, the server will be running in the background as a thread. The server thread accepts remote requests from the clients and spawns a new thread for each request. In other words, the threads are per request, not per client. Hence for each read/write request, a thread is created dynamically at the server site and terminated when the request is done. The server can 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' requests served and when all the clients' requests are done, should terminate gracefully.

Reader and Writer Client Specification

Clients are started on remote machines as processes and 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.

To simulate a real situation, each reading and writing operation takes a random amount of time (500 to 1000ms). A reader/writer must sleep for a random time in the range (250, 500ms) before the first request and between two requests.

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 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.

Execution Requirements:

1. Your java file containing main() should be named start.java
2. Your program must compile and execute properly on CISE sand and rain (Solaris) machines.
Your projects will **not** be tested on any other machines.
3. Execution must begin with command: java start
4. All your code files must reside at the root level of the project directory and it should 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 code containing no other subdirectory.
6. We will use this script to test your project:
#!/bin/sh
tar szvf proj1.tar; rm*class
javac *.java; java start

Reminder

Please remember that Project Overview and Requirements page is valid for all projects. The name of the program to run the system must be start.java. Output files must be named like log1, log2, etc. These are plain text files. All file names must be exactly as specified in this document and in the general project requirements document.

Readers should be given names 1,2,3,... x
Writers should be given names x+1,x+2....

Additional Details:

Once the clients 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 readers or a writer lock the resource on the server, it must wait for a random time (500-1000 millisecond) before releasing all locks. Also after releasing all the locks, the clients must again sleep for random amount of time before becoming active again to access the shared resource (250-500 milliseconds).

Your code must follow the sample output format very closely

Sample Partial Output:

Two numbering sequences are maintained by the server: the *Request Sequence* is the sequence number for the incoming requests and the *Service Sequence* is the sequence number of the request being serviced. They are different due to the access policy that we are implementing. *Object Value* is the value now saved in the object and *Written by* is the ID of the writer updating the shared object. The server should print out its actions in the following format:

Read Requests:

<i>Service Sequence</i>	<i>Request Sequence</i>	<i>Object Value</i>
----- 4	----- 4	----- 8

5	6	8
7	3	7
8	8	7
10	10	5
11	11	5
12	12	5
14	14	6
16	16	8
19	19	6
21	21	7
23	23	8

Write Requests:

<i>Service Sequence</i>	<i>Request Sequence</i>	<i>Object Value</i>	<i>Written by</i>
-----	-----	-----	-----
1	1	6	6
2	2	5	5
3	5	8	8
6	7	7	7
9	9	5	5
13	13	6	6
15	15	8	8
17	17	5	5
18	18	6	6
20	20	7	7
22	22	8	8
24	24	7	7

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

Client type: Reader
Client Name: 1

<i>Request Sequence</i>	<i>Service Sequence</i>	<i>Object Value</i>
-----	-----	-----
4	4	8
10	10	5
19	19	6

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, and returned as messages to the readers and writers.

The name of the file that will be written by the clients must be log concatenated with the ID of the reader (i.e., log1, log2).

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

Client type: Writer
Client Name: 7

<i>Request Sequence</i>	<i>Service Sequence</i>
-----	-----
8	6
20	20
24	24

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. log5, log6).

Report:

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 and 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 section 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.

Submission Guidelines:

1. Only submit your Java code files, do not include .class files in your submissions.
2. Use **tar cvf proj2.tar <file list>** to tar your submission.
3. Include a 1 – 3 pag(s) project documentation in .txt format. Name it **report.txt**

Grading Criteria:

Correct Implementation/Execution/Graceful Termination/Nonexistence of Run-away proceses	75%
Report	20%
Sytlish Code/Readability/Comments/Structure and Readability of Output	5%
Total:	100%

Java Resources:

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.

Note on Run-away 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:

```
kill java
```

To check runaway processes on remote hosts:
and to clean:

```
ssh <host-name> ps -u <your-username>
```

```
ssh <host-name> kill java
```