

COP5615

Operating System Principles

Fall 2007

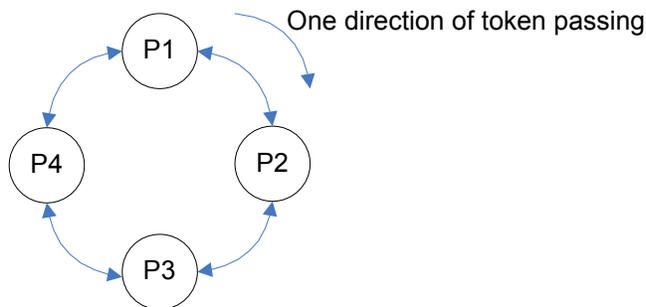
Project 4

Date Assigned: Oct. 25, 2007
Date Due: Nov. 8, 2007 (11:59 pm) for LOCAL students (Sections: 1092, 9333)
Nov. 12, 2007 (11:59 pm) for EDGE students (Sections: 5667, 8170, 8404)

Introduction:

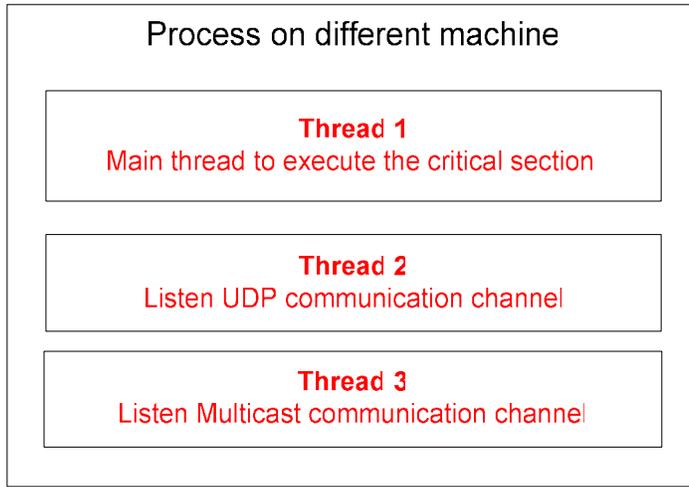
In this project, we will implement the token-based distributed mutual exclusion algorithm with a logical ring topology as discussed in class. Processes in this algorithm form a logical ring where a token circulates in the ring. A process is allowed to execute its critical section only if it possesses the token. When the process has finished the critical section (if any), it passes the token to the successor node in the ring. The logical ring structure is attractive because it is simple, fair, and deadlock-free.

The ring structure you will implement is a dynamic ring, which means that joining to the ring or leaving from the ring can happen at any time. Both multicast and unicast communications will be used to initialize the ring and circulate the token within this dynamic ring. Each process will know the IP addresses and port numbers of its own successor and predecessor, and they logically point to their successor and predecessor with the IP address and port number using the UDP protocol. That is, they will pass the token to their successor by using the IP address and the port number of the successor. They also need to point to predecessor when creating a new connection. Detailed steps for a new node to join or leave the ring are shown later in “Detailed description of process behavior” part. Only the process with the token executes the critical section and releases the token after the critical section but without the token main thread in the processes will be block to wait for the circulation of the token.



Logical connection of the ring

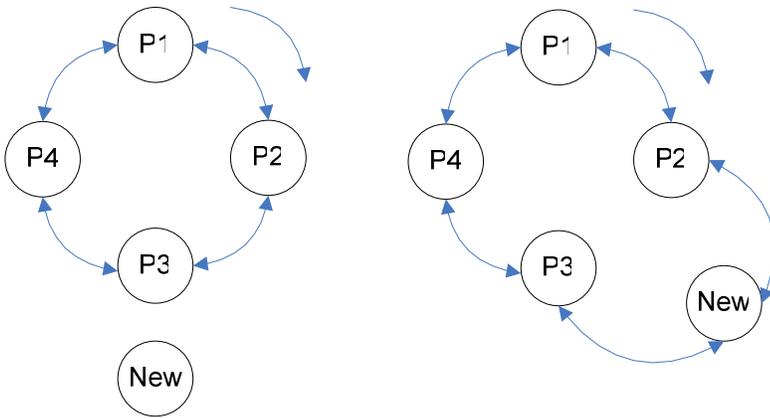
The structure of the process



Problem Details:

The processes in the ring run on different machines in the same multicast group. Each process contains three logical components represented as threads. The main thread is a loop that makes request to enter the critical section, blocks to wait for the token, enters the critical section and release the token when it is done. Thus the main thread needs to interact (through shared object) with the token passing thread, the second thread in the diagram. This thread receives the token from the predecessor and passes it down to the successor using a UDP communication channel. It is required to check if the main thread needs to execute the critical section before sending the token to the successor, and if the main thread is blocked for the token, then it wakes up the main thread to execute the critical section. Upon the completion of the critical section, the main thread informs the token passing thread to continue passing the token

In this project, the ring will be constructed dynamically by nodes joining the ring one by one asynchronously. For a new member to join the ring, it first sends a joining message through a multicast channel, and then waits replies from the existing processes in the ring. The new member picks one of the replies as its successor. The predecessor of the new member will be the predecessor of the picked process. The new member needs to know about the IP addresses and port numbers of the UDP channel of successor from the replies and predecessor to make logical connections for them. After the establishment of a new connection, the token will continue to circulate along the new ring topology eventually.



When new member picks P3 as a successor

Similarly, in the case of leaving the ring, a process needs to notify its successor and predecessor to establish new connection between them.

Example format of the multicast message:

JOIN	Process ID	IP address	Port number
------	------------	------------	-------------	-----	-----

Example format of the unicast message:

TOKEN
LEAVE

The token *should not be lost* at any moment of joining or leaving of any processes.

Two phases of ring topology:

Initialization phase: Additional joining of processes into the ring, the size of the ring will grow.

Termination phase: Processes are leaving out of the ring, so the size of the ring will shrink.

In this project, termination phase will start with the pressing key of ‘Enter’ key from user at any moment. That is, after detecting input of the key, all processes in the ring should leave from the ring one by one until no process left behind and then the program will terminate gracefully. *Not all they will leave from the ring at the same time.*

Detailed description of process behavior:

Whenever a process is executed for the first time, it must pick a random port number for its own UDP unicast server process. Continue the random port number selection until your program finds a free port to listen to. On this port number your UDP server would listen to any incoming unicast UDP messages. After successful selection of random port number, this is the flow of behavior for each group member:

1. New member sends a RJOIN message to the multicast channel. This message must also contain the new process' IP address and port number of the UDP channel to listen the replies.
2. After step 1, the process waits for a max of 5 seconds for any replies through the UDP channel. There could be one of two possibilities:
 - If no message is received within 5 seconds, the process must assume that the ring does not exist and it should point itself to form a ring of size one.
 - If there are some replies containing the IP address and port number of an existing process, the new member picks one of them as its successor from the replies and then makes logical connections between them to join into the ring as described earlier.
3. Whenever a process in the ring receives a joining message from the new member through the multicast channel, it must reply back to the new member with its own IP address and port number and its predecessor through UDP channel.
4. After joining into the ring, the process participates in the token passing using the UDP channel and concurrently enters its loop of requesting the critical section. The critical section is simulated to be a random sleep time between 500ms and 1000ms. We would also like to introduce a little delay in token passing. The token passing thread needs to hold the token for 100ms before passing it to the successor.
5. Before execution of critical section, the program must write to the log file what the process has done and also before passing the token, the process must write to the log its process ID. The log file is for the purpose testing and grading.
6. In the termination phase, each process should leave out of the ring after notifying its predecessor and successor about the new logical connection between them.

NOTE: *The log file is a shared file named "log.txt" in the same directory. Only the process with token can write its action to the log file before passing the token to its successor.*

Manifest constants relevant to this project:

New Member Join Request Multicast Channel: 239.x.x.x : xxxx

Project termination trigger:

You can hard code the termination triggering process in your project implementation. The trigger will be pressing of 'Enter' key. After detecting this key, all processes enter the termination phase. Each process sends an exit message to leave the ring. Your program must ensure that the token continues to circulate correctly and all processes terminate gracefully. After all the creation of processes on remote sites, your start.java will wait for the Enter key. When the program detects the input of Enter key, then start.java will send LEAVE message to each process on remote sites one by one with some delay(500ms) between the consecutive LEAVE message to finish the program.

System Configuration

The system configuration shown below is in the system.properties file to be read by your program.

P1=sun114-11.cise.ufl.edu
P2=sun114-12.cise.ufl.edu
P3=sun114-13.cise.ufl.edu
P4=sun114-14.cise.ufl.edu

PX are the IDs of each process and the address of the machine for each process to run on.

For multicast channel, you should use only the machines in lab 114, or multicast communication might not work properly.

NOTE: Make sure there is 5 seconds delay between execution of successive remote processes for the first time. 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.

Additional Details:

Once the process executes the critical section, it must wait for a random time of 500-1000 millisecond before leaving the critical section. Also after coming out of the critical section, the process must again sleep for a random amount of time (3000-5000millisecond) before becoming active again to access the critical section.

Sample Partial Output:

P1: RECV and SENT Token:
P1: RECV and SENT Token:
P1: RECV, ENTERED CRITICAL SECTION:1 and SENT Token:
P2: RECV and SENT Token:
P1: RECV, ENTERED CRITICAL SECTION:2 and SENT Token:
P2: RECV and SENT Token
P1: RECV and SENT Token:
P2: RECV, ENTERED CRITICAL SECTION:1 and SENT Token:
P1: RECV and SENT Token:
P2: RECV and SENT Token:
P3: RECV and SENT Token:
.
.
.

NOTE: This is what your log file "log.txt" must look like. Your code must follow the sample output format very closely

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 lab 114 (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
```

Note: There will be no exception to testing your code on sand / rain machines under any circumstance.

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.

Process should be given names 1,2,3,... x

Your code must follow the sample output format very closely

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:

7. Only submit your Java code files, do not include .class files in your submissions.
8. Use **tar cvf proj2.tar <file list>** to tar your submission.
9. 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%

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:

```
skill java
```

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

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