Dinesh Pamu

B2- 30

ASSIGNMENT NO: 5

Title:

Token ring based mutual exclusion algorithm.

Problem Statement:

Implement token ring based mutual exclusion algorithm.

Problem Statement:

In distributed systems, multiple processes may require access to a shared resource. The challenge is to ensure that only one process can access the resource at any given time. The token ring based mutual exclusion algorithm provides a solution to this problem. The objective of this assignment is to implement the token ring based mutual exclusion algorithm in Java and demonstrate its functionality.

Tools/Environment:

Java programming language and Eclipse IDE will be used for implementing the algorithm. The Eclipse IDE provides a convenient environment for writing, testing, and debugging Java code.

Theory:

The token ring based mutual exclusion algorithm is a distributed algorithm that ensures only one process can access a shared resource at a time. The algorithm uses a logical ring of processes to coordinate access to the resource. A token is passed around the ring, and the process holding the token can access the resource.

The algorithm works as follows:

Each process in the system is assigned a unique ID.

A logical ring is created by linking the processes in the order of their IDs.

The token is initially held by a designated process.

When a process requires access to the shared resource, it sends a request message to the next process in the ring.

The process receiving the request message checks if it currently holds the token. If it does, it grants access to the requesting process by passing the token to it.

If the process receiving the request message does not hold the token, it forwards the request message to the next process in the ring.

When a process finishes accessing the shared resource, it releases the token by passing it to the next process in the ring.

Implementation:

The implementation of the token ring based mutual exclusion algorithm involves the following steps:

Step 1: Create a Process class that represents a process in the system. The Process class should have the following properties:

id: the unique ID of the process

nextProcess: the next process in the ring

hasToken: a boolean flag indicating whether the process currently holds the token

Step 2: Create a Token class that represents the token. The Token class should have no properties.

Step 3: Create a Ring class that represents the logical ring of processes. The Ring class should have the following properties:

processes: an array of Process objects representing the processes in the ring

currentProcess: the process currently holding the token

Step 4: Implement the run() method in the Process class. The run() method should contain the logic for sending and receiving request messages and passing the token.

Step 5: Create a main() method in the Ring class. The main() method should create a Ring object, initialize the processes in the ring, and start the processes.

Step 6: Test the implementation by simulating multiple processes accessing the shared resource.

Conclusion:

In conclusion, the token ring based mutual exclusion algorithm provides a simple yet effective solution to the problem of coordinating access to a shared resource in distributed systems. The implementation of the algorithm in Java involves creating a logical ring of processes, passing a token around the ring, and granting access to the shared resource to the process holding the token.

Code:

Server Side Code (RingMutex):

```
import java.io.*;
import java.net.*;

public class TokenRingMutex {

   static int port = 8090;
   static String host = "localhost";
   static int numProcesses = 3;
   static int processId;
   static int tokenValue = 0;
   static boolean hasToken = false;
   static Socket socket;
   static BufferedReader in;
   static PrintWriter out;

   public static void main(String[] args) throws Exception {
      if (args.length == 1) {
         processId = Integer.parseInt(args[0]);
      } else {
```

```
            System.out.println("Usage: java TokenRingMutex <processId>");
            System.exit(1);
        }

        // Connect to next process in the ring
        int nextProcessId = (processId + 1) % numProcesses;
        socket = new Socket(host, port + nextProcessId);
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);

        while (true) {
            // Wait for token
            while (!hasToken) {
                String message = in.readLine();
                if (message != null) {
                    int token = Integer.parseInt(message);
                    if (token == tokenValue) {
                        hasToken = true;
                        System.out.println("Process " + processId + " has the token");
                    }
                }
            }

            // Critical section
            System.out.println("Process " + processId + " is in the critical section");

            // Release token
            hasToken = false;
            tokenValue = (tokenValue + 1) % numProcesses;
            out.println(tokenValue);
            System.out.println("Process " + processId + " released the token");
        }
    }
}
```

Client Side Code :

```
import java.io.*;
import java.net.*;

public class TokenRingClient {

    static int port = 8000;
    static String host = "localhost";

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket(host, port);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        // Send request to enter critical section
        out.println("request");
        String response = in.readLine();
        if (response.equals("grant")) {
            // Entered critical section
            System.out.println("Entered critical section");
            // Do some work here
            // ...
            // Release critical section
            out.println("release");
        } else {
```

```
            // Failed to enter critical section
            System.out.println("Failed to enter critical section");
        }

        // Clean up
        in.close();
        out.close();
        socket.close();
    }
}
```

Output: