ANISH ADNANI

D17B/01

DC

(1)

EXPERIMENT 9

Aim: Write a program to demonstrate Lamport's Algorithm for Distributed mutual Exclusion.

Theory:

Lamport's Distributed mutual Exclusion Algorithm is a permission based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.

In permission based timestamp is used to order critical section requests and to resolve any conflict between requests.

In Lamport's algorithm critical section requests are executed in the increasing order of timestamp i.e a request with smaller timestamp will be given permission to execute critical section first than a request with larger timestamp

In this algorithm

- Three types of messages (REQUEST, REPLY and RELEASE) are used and communication channels are assumed to follow FIFO order
- A site send a REQUEST message to all other site to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.

- A site send a RELEASE message to all other site upon exiting the critical section.

★ To enter critical section

- When a site S_i wants to enter the critical section, it sends a request message Request(ts_i, i) to all other sites and places the request on request_queue;
- When a site S_j receives the request message REQUEST (ts_i, i) from site S_i, it returns a timestamped REPLY message to site S_i and places the request of the site S_i on request_queue;

★ To execute critical section

- A site S_i can either the critical section if it has received the message with timestamp larger than (ts_i, i) from all other sites and its own request is at the top of request_queue.

★ To release the critical section

- When a site S_i exits the critical section, it removes its own request from the top of its request Queue and sends a timestamped RELEASE message to all other sites.
- When a site S_j receives the timestamped RELEASE message from site S_i, it removes the request of S_i from its request Queue.

Message complexity

$$3(N-1)$$

$$\downarrow$$

$$\underbrace{(N-1)}_{\text{request messages}} + \underbrace{(N-1)}_{\text{reply messages}} + \underbrace{(N-1)}_{\text{release messages}}$$

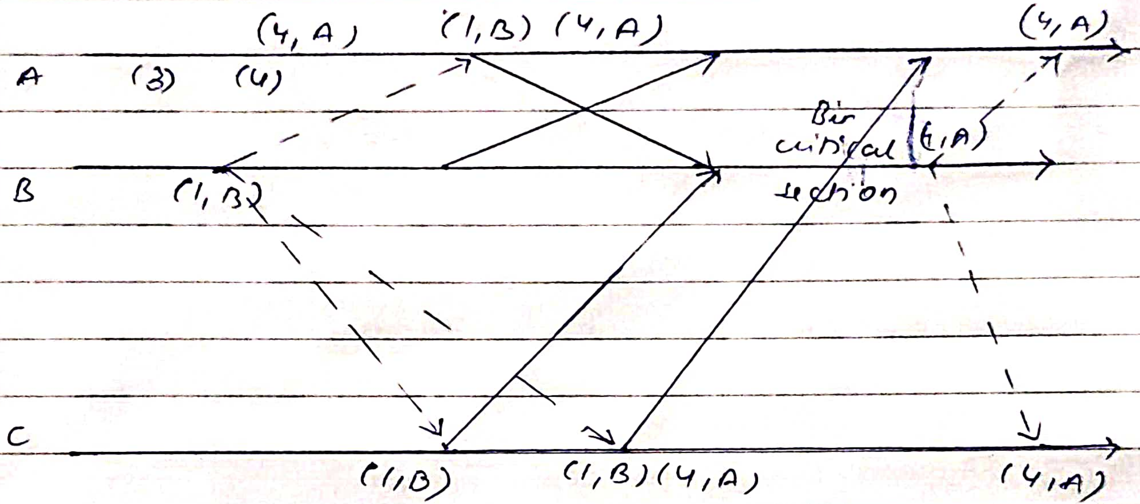
Drawbacks of Lamport's Algorithm

- Unreliable approach: failure of any one of the processes will halt the progress of entire system.
- High message complexity: Algorithm requires 3(N-1) messages per critical section invocation

Performance

- Synchronization delay is equal to maximum message transmission time
- It requires 3(N-1) messages per CS execution
- Algorithm can be optimized to 2(N-1) messages by omitting the REPLY message in some situations

Example



---> request → acknowledge

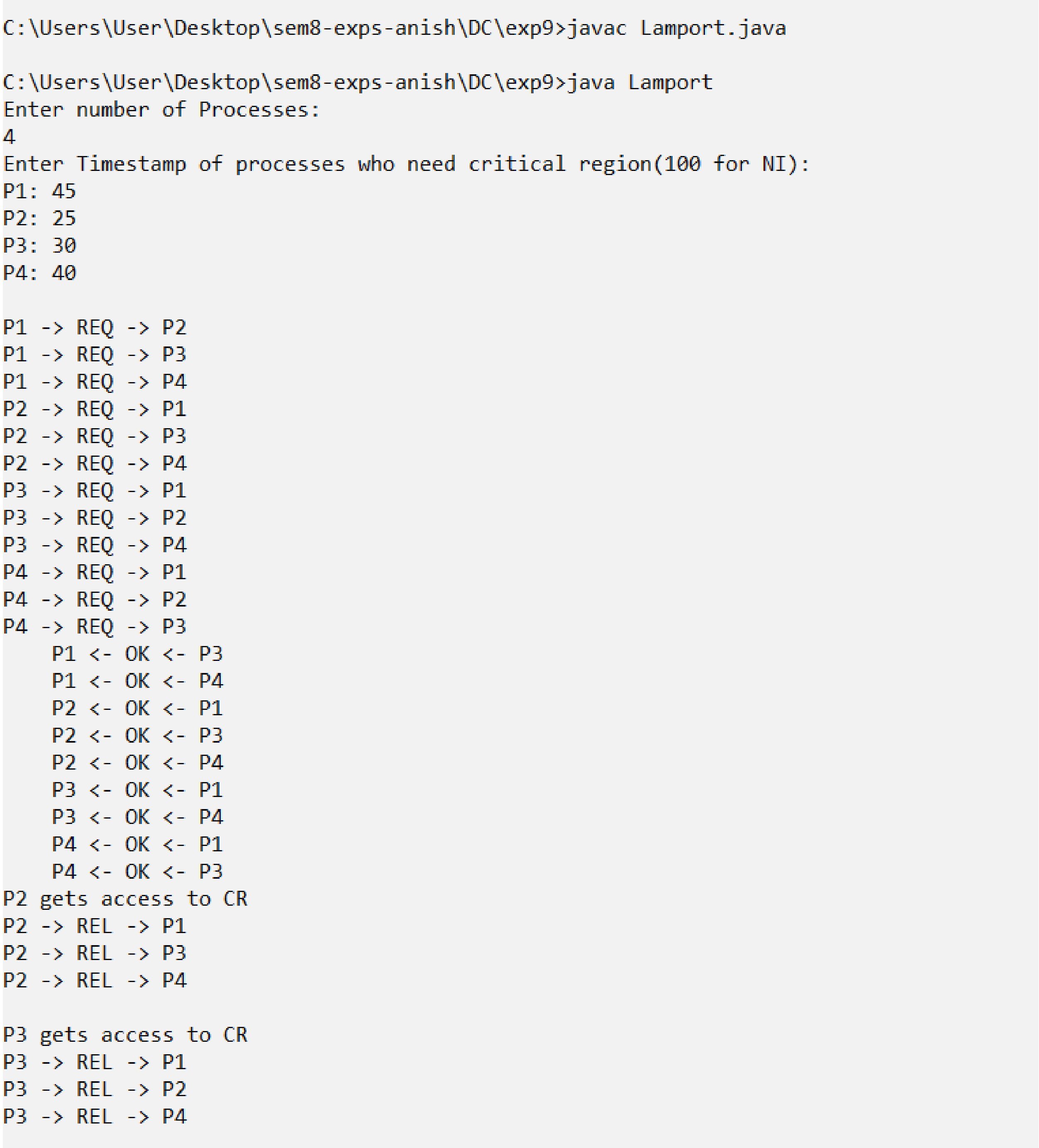
---> release

Conclusion:

We have successfully implemented and studied lamports mutual exclusion algorithm.

```java
import static java.lang.Thread.sleep;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

public class Lamport {
    public static void main(String[] args) throws InterruptedException {
         Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of Processes: ");
        int n = sc.nextInt();
        List<Integer> sort = new ArrayList<Integer>();
        System.out.println("Enter Timestamp of processes who need critical region(100 for NI):");
        for (int i = 0; i < n; i++) {
            System.out.print("P" + (i + 1) + ": ");
            sort.add(sc.nextInt());
        }
        System.out.println("");
        int iterator = 0;
        while (Collections.min(sort) != 100) {
            int min = Collections.min(sort);
            int min_index = sort.indexOf(Collections.min(sort));
            if (iterator == 0) {
                for (int i = 0; i < n; i++) {
                    if (sort.get(i) != 100) { //process who are interested in entering CR
                        for (int j = 0; j < n; j++) {
                            if (i != j) { //so that p1 does not send req to p1
                                System.out.println("P" + (i + 1) + " -> REQ -> P" + (j + 1));
                            }
                        }
                    }
                }
                for (int i = 0; i < n; i++) {
                    for (int j = 0; j < n; j++) {
                        if (j != min_index && i != j && sort.get(i) != 100) {
                            System.out.println("    P" + (i + 1) + " <- OK <- P" + (j + 1));
                        }
                    }
                }
            }
            System.out.println("P" + (min_index + 1) + " gets access to CR");
            sleep(5000);
            for (int i = 0; i < n; i++) {
                if (i != min_index) {
                    System.out.println("P" + (min_index + 1) + " -> REL -> P" + (i + 1));
                }
            }
            System.out.println("");
            sort.set(min_index, 100);
            iterator += 1;
        }
    }
}
```

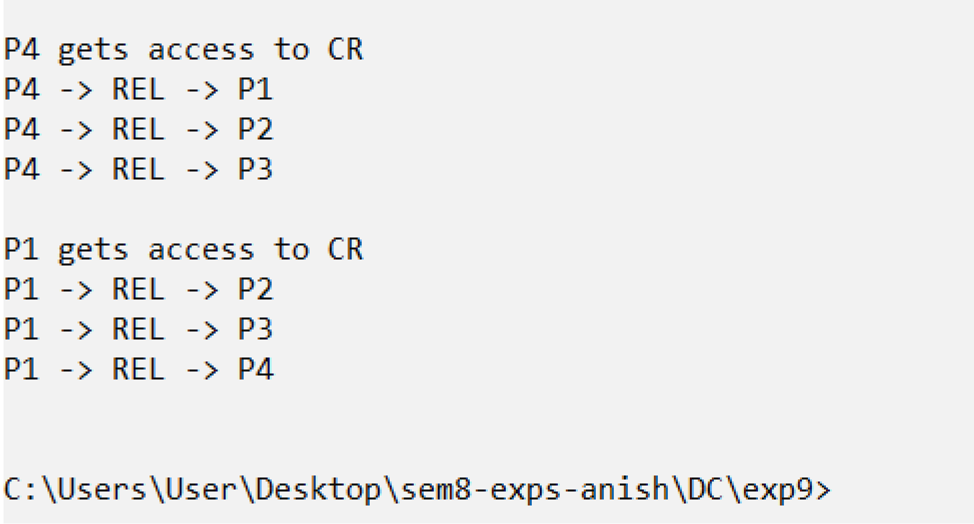


```
C:\Users\User\Desktop\sem8-exps-anish\DC\exp9>javac Lamport.java

C:\Users\User\Desktop\sem8-exps-anish\DC\exp9>java Lamport
Enter number of Processes:
4
Enter Timestamp of processes who need critical region(100 for NI):
P1: 45
P2: 25
P3: 30
P4: 40

P1 -> REQ -> P2
P1 -> REQ -> P3
P1 -> REQ -> P4
P2 -> REQ -> P1
P2 -> REQ -> P3
P2 -> REQ -> P4
P3 -> REQ -> P1
P3 -> REQ -> P2
P3 -> REQ -> P4
P4 -> REQ -> P1
P4 -> REQ -> P2
P4 -> REQ -> P3
    P1 <- OK <- P3
    P1 <- OK <- P4
    P2 <- OK <- P1
    P2 <- OK <- P3
    P2 <- OK <- P4
    P3 <- OK <- P1
    P3 <- OK <- P4
    P4 <- OK <- P1
    P4 <- OK <- P3
P2 gets access to CR
P2 -> REL -> P1
P2 -> REL -> P3
P2 -> REL -> P4

P3 gets access to CR
P3 -> REL -> P1
P3 -> REL -> P2
P3 -> REL -> P4
```



```
P4 gets access to CR
P4 -> REL -> P1
P4 -> REL -> P2
P4 -> REL -> P3

P1 gets access to CR
P1 -> REL -> P2
P1 -> REL -> P3
P1 -> REL -> P4


C:\Users\User\Desktop\sem8-exps-anish\DC\exp9>
```