PRACTICAL - 4

**AIM** : To implement a mutual exclusion service using Lamport’s Mutual Exclusion Algorithm in a distributed system.

**Course Outcome** : CO2

**Software Used** : Java SDK

# Theory :

Mutual exclusion in distributed systems ensures that multiple processes do not enter a critical section simultaneously, which is crucial for maintaining data consistency and integrity. Lamport’s Mutual Exclusion Algorithm is a distributed solution that uses logical clocks to manage access to the critical section.

The algorithm involves the following steps:

1. Requesting the Critical Section: A process sends a request message to all other processes, including its current logical clock value.
2. Receiving Requests: Upon receiving a request, a process replies immediately if it is not in the critical section and not waiting for the critical section with a higher priority. Otherwise, it defers the reply.
3. Entering the Critical Section: A process enters the critical section when it has received replies from all other processes.
4. Releasing the Critical Section: After exiting the critical section, the process sends release messages to all processes that it had deferred replies to.

**Flowchart/Algorithm** :

1. Setup Java Environment:
   * Install JDK and set up an IDE.
   * Create a new Java project.
2. Define Process Class:
   * Implement a class Process with fields for logical clock, process ID, and state (e.g., REQUESTING, EXECUTING, RELEASED).
   * Define methods for sending request, reply, and release messages.
3. Request Handling:
   * Implement logic for handling incoming request messages. If the process is in a lower priority state, send a reply immediately; otherwise, defer the reply.
   * Maintain a queue of deferred requests.
4. Critical Section Management:
   * Implement a method enterCriticalSection() to enter the critical section after receiving all necessary replies.
   * Implement a method exitCriticalSection() to send release messages and handle deferred requests.
5. Testing:
   * Simulate multiple processes requesting access to a shared resource.
   * Observe the order of entry into the critical section and ensure mutual exclusion.

# Code:

import java.util.ArrayList; import java.util.LinkedList; import java.util.Queue; import java.util.Scanner;

class Process {

enum State { REQUESTING, EXECUTING, RELEASED }

int pid, logicalClock, replyCount; State state;

Queue<Request> deferredRequests;

public Process(int pid) { this.pid = pid; this.logicalClock = 0;

this.state = State.RELEASED; this.replyCount = 0;

this.deferredRequests = new LinkedList<>();

}

// Update logical clock

public void updateClock(int timestamp) {

logicalClock = Math.max(logicalClock, timestamp) + 1;

}

// Send a request for the critical section

public void sendRequest(ArrayList<Process> processes) { state = State.REQUESTING;

logicalClock++; replyCount = 0;

System.out.println("Process " + pid + " is requesting CS at time " + logicalClock); for (Process p : processes) {

if (p.pid != this.pid) {

p.receiveRequest(new Request(this.pid, this.logicalClock));

}

}

}

// Receive a request message from another process public void receiveRequest(Request req) {

updateClock(req.timestamp);

System.out.println("Process " + pid + " received request from Process " + req.pid + " with timestamp "

+ req.timestamp);

if (state == State.EXECUTING || (state == State.REQUESTING && (logicalClock < req.timestamp || (logicalClock == req.timestamp && pid < req.pid)))) {

// Defer the request deferredRequests.add(req);

System.out.println("Process " + pid + " defers reply to Process " + req.pid);

} else {

// Reply immediately sendReply(req.pid);

}

}

// Send a reply to a requesting process public void sendReply(int targetPid) {

System.out.println("Process " + pid + " sends reply to Process " + targetPid); replyCount++;

}

// Enter the critical section if all replies are received public void enterCriticalSection(int numProcesses) {

if (replyCount == numProcesses - 1) { state = State.EXECUTING;

System.out.println("Process " + pid + " enters the critical section");

}

}

// Exit the critical section and release resources

public void exitCriticalSection(ArrayList<Process> processes) { System.out.println("Process " + pid + " leaves the critical section"); state = State.RELEASED;

// Send release messages

for (Request r : deferredRequests) { sendReply(r.pid);

}

deferredRequests.clear();

}

}

class Request implements Comparable<Request> { int pid, timestamp;

public Request(int pid, int timestamp) { this.pid = pid;

this.timestamp = timestamp;

}

@Override

public int compareTo(Request other) {

if (this.timestamp == other.timestamp) { return Integer.compare(this.pid, other.pid);

}

return Integer.compare(this.timestamp, other.timestamp);

}

}

public class LamportsMutualExclusion { public static void main(String[] args) {

Scanner sc = new Scanner(System.in); ArrayList<Process> processes = new ArrayList<>(); System.out.println("Name: Rounak Ranjan"); System.out.println("Roll no.: 00520802721");

// Create 3 processes

for (int i = 0; i < 3; i++) { processes.add(new Process(i + 1));

}

boolean running = true; while (running) {

System.out.println("Choose an option:");

System.out.println("1. Process 1 request CS\n2. Process 2 request CS\n3. Process 3 request CS\n4.

Process 1 exit CS\n5. Process 2 exit CS\n6. Process 3 exit CS\n7. Exit"); int choice = sc.nextInt();

switch (choice) {

case 1 -> processes.get(0).sendRequest(processes); case 2 -> processes.get(1).sendRequest(processes); case 3 -> processes.get(2).sendRequest(processes);

case 4 -> processes.get(0).exitCriticalSection(processes); case 5 -> processes.get(1).exitCriticalSection(processes); case 6 -> processes.get(2).exitCriticalSection(processes); case 7 -> running = false;

default -> System.out.println("Invalid choice.");

}

// Try to enter the critical section for each process for (Process p : processes) {

p.enterCriticalSection(processes.size());

}

}

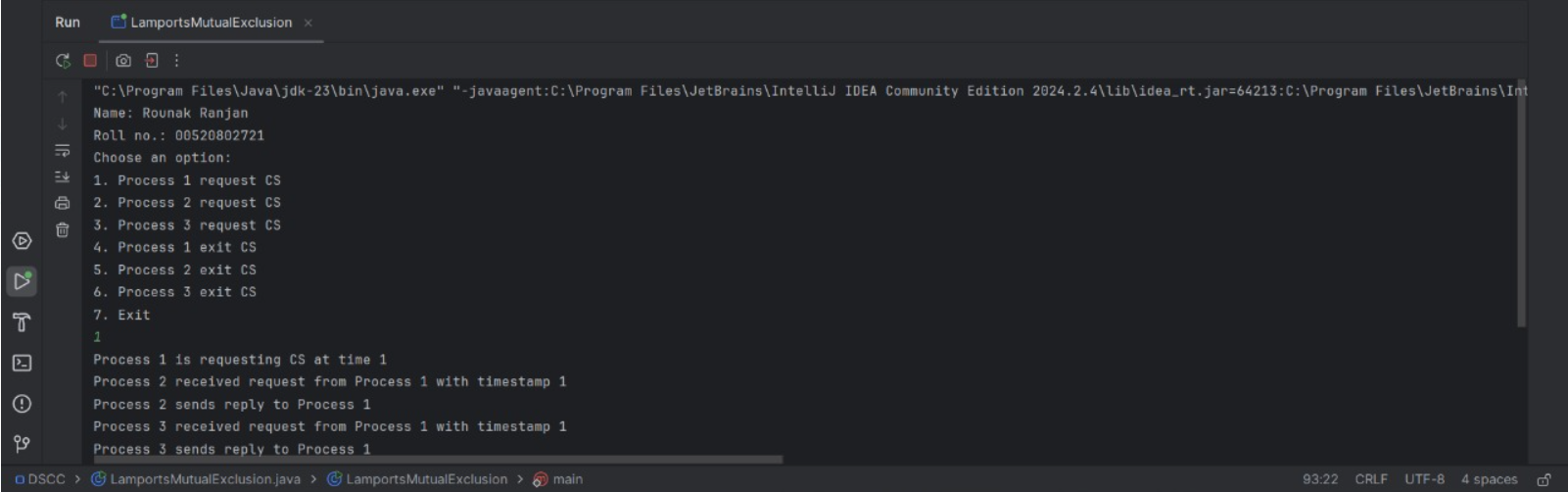
sc.close();

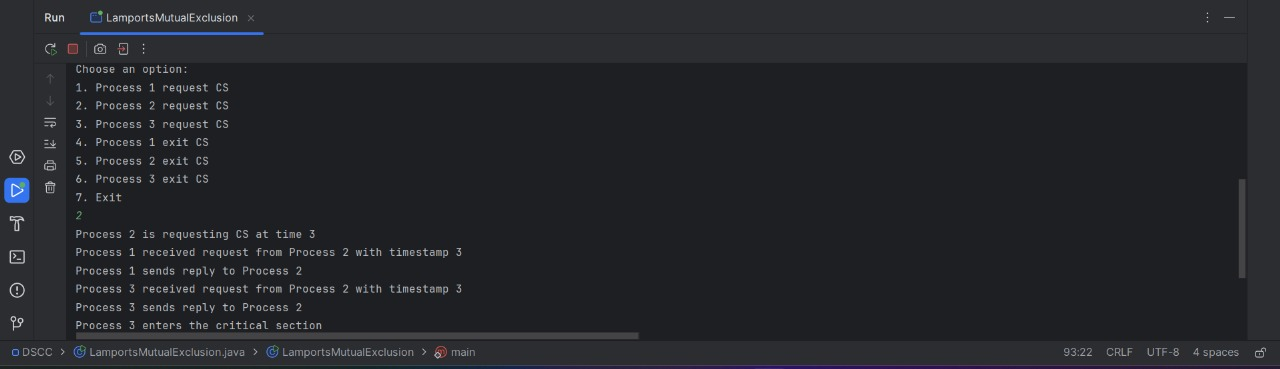
}

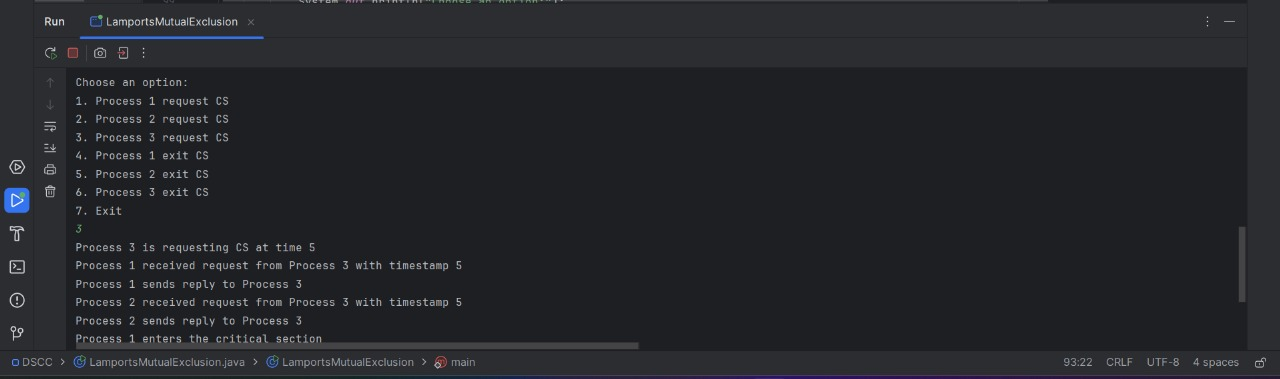
}

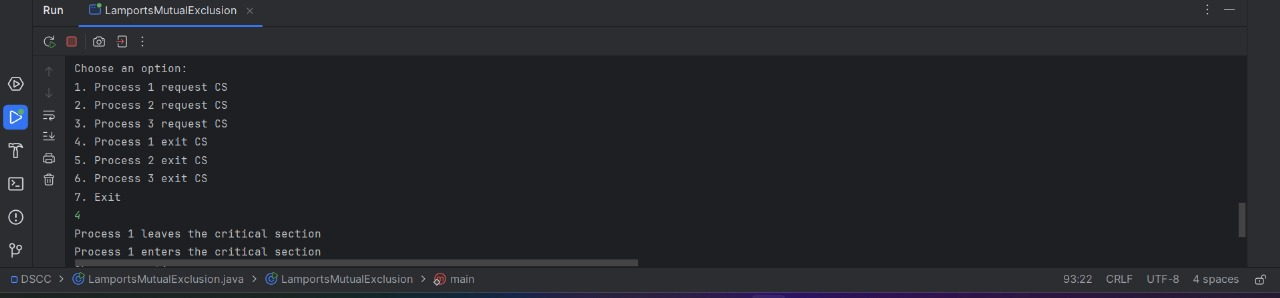
# Results:

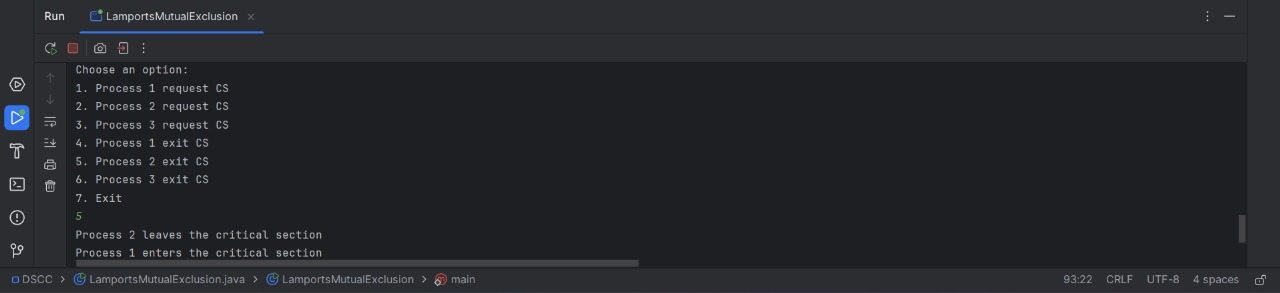
Demonstration of mutual exclusion using Lamport’s algorithm, ensuring that only one process accesses the critical section at a time.

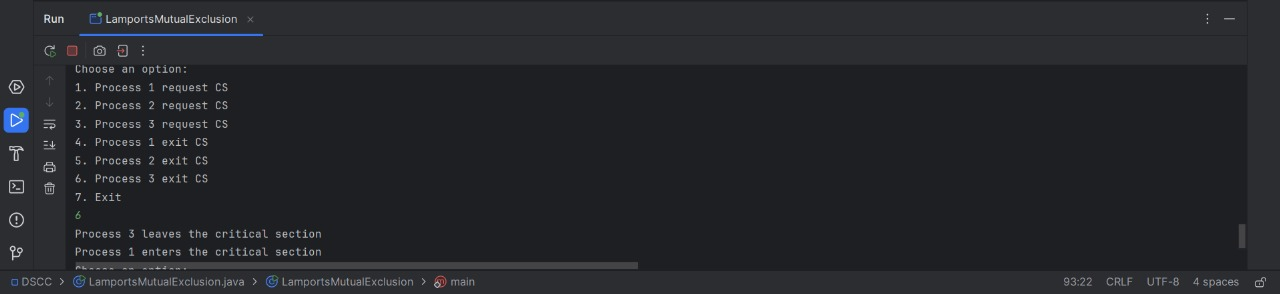


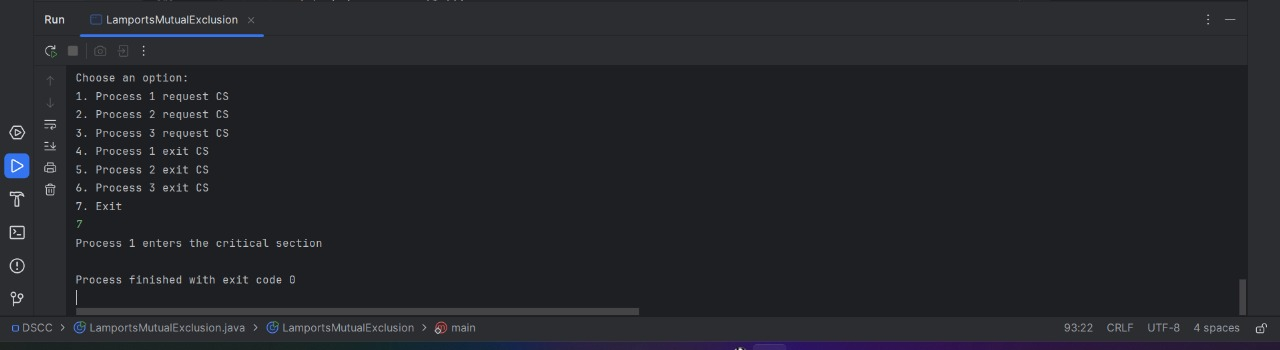












**Expected Outcome attained:** Yes