Programming Assignment Report

Program 5

Mutual Exclusion

Abstract

The purpose of this lab was to implement Lamport's Algorithm for thread synchronization in Java, demonstrating mutual exclusion in a race scenario. The program implements multiple racers that compete for access to a critical section, managed by an Arbitrator class using a ticket system for resource allocation. The testing scenarios run through a script with different configurations that run 2-5 threads and 10-50 iterations, showing how the system handles concurrent access to shared resources while maintaining thread safety. Through this lab, the effectiveness of Lamport's Algorithm for preventing race conditions was demonstrated.

Introduction

3.1 Problem Statement

Understand how Lamport’s Algorithm works to enable multithread access to the same critical section. We were also tasked to implement a semaphore to ensure mutual exclusion.

3.2 Objective

To understand how different exclusion algorithms work to avoid collision with the critical sections.

3.3 Background Information

The concepts used within the lab relate to Mutual exclusion algorithms such as Lamport’s Algorithm and Semaphore.

Materials and Methods

4.1 Tools and Technologies

For the assignment we used several software tools to implement and test Lamport's Algorithm. For the main part of the assignment, we used a Virtual Ubuntu machine to run our Linux environment. Within the environment, we used the terminal to access the operating system's programs and compile/run our Java code. We used the nano text editor to modify the Java source files, particularly working with race.java which contains our RaceManyThreads implementation, along with the Arbitrator.java and other files. The code was compiled using javac, and we were provided with a set of starter files including GetOpt.java and Scheduler.java. Once the implementation was complete, we executed the program through the terminal using various configurations defined in our test scenarios.

4.2 Program Design  
A diagram of a process

AI-generated content may be incorrect.

4.3 Code Implementation

Using the provided files, I implemented thread synchronization using Lamport's Algorithm in the race.java file. The main components implemented were the critical section access control in the Racer class and thread creation in RaceManyThreads. The Racer class uses an Arbitrator to manage mutual exclusion when accessing the shared sum variable, while RaceManyThreads handles creating and managing multiple racing threads. When testing, I verified compilation and ran the program with various thread and iteration configurations through the run.bat script to ensure proper synchronization.Results

5.1 Test Cases

I ran each of the given scripts to test the outputs of each file. The outputs are below.

5.2 Output

Lamport’s Algorithm -M 20 -N 4

A screenshot of a computer program

AI-generated content may be incorrect.

Script part1

A screenshot of a computer program

AI-generated content may be incorrect.

Discussion

6.1 Post-Lab Questions

|  |  |
| --- | --- |
| Run # | Last Sum |
| Semaphore 1 | 8030 |
| Semaphore 2 | 12045 |
| Semaphore 3 | 234920 |
| Semaphore 4 | 293650 |
| Semaphore 5 | 10835375 |

6.2 Challenges Encountered and Improvements

A challenge I faced when implementing the code was when running the code after implementing the requirements. I only placed the critical section around the for loop not the print statements. So each thread was constantly outputting at the same time.

Conclusion

I understand that while Lamport's Bakery Algorithm ensures fair access to the critical section by assigning each thread a unique ticket number, the overhead of ticket management and synchronization can impact performance.

Source Code:

Race.Java Lamport’s Algorithm

|  |  |
| --- | --- |
| */\*---------------------------------------------------------------------*  *Name: Kristopher Adams*  *Course: CS 3230, Section 1, Spring 2025*  *Purpose: This program demonstrates a simple multi-threaded environment*  *where multiple "Racers" perform computations. The program ensures mutual*  *exclusion using the "Arbitrator" class to manage access to a shared resource.*  *Input: Command line arguments -M (number of computations) and -N (number of racers).*  *Output: Displays the progress and results of each Racer thread.*  *---------------------------------------------------------------------\*/*  class Racer extends Thread {  */\*\**  *\* Represents a racing thread that performs computations in a controlled manner.*  *\* This class handles mutual exclusion using an Arbitrator to safely access shared resources.*  *\*/*     private String name;     private int M = 0;     private static volatile long sum = 0;     private int id;     private Arbitrator arbitrator;  */\*\**  *\* Constructor for creating a new Racer thread.*  *\* @param name String - The name identifier for this racer thread*  *\* @param M int - The number of computations this racer will perform*  *\* @param id int - Unique identifier for this racer, used for arbitration*  *\* @param arbitrator Arbitrator - The arbitrator instance managing mutual exclusion*  *\*/*     public Racer(String name, int M, int id, Arbitrator arbitrator) {        this.name = name;        this.M = M;        this.id = id;        this.arbitrator = arbitrator;        System.out.println("age()=" + Scheduler.age() + ", "           + name + " is alive, M=" + M);     }  */\*\**  *\* Performs a mathematical computation on the given values.*  *\* @param j long - The initial value to start computations from*  *\* @param k int - The number of iterations to perform calculations*  *\* @return long - The computed sum after k iterations*  *\*/*     private long fn(long j, int k) {        long total = j;        for (int i = 1;  i <= k; i++) total += (2 \* i - 1) \* (2 \* i - 1);        return total;     }  */\*\**  *\* Main execution method for the racer thread.*  *\* Performs M iterations of computations while ensuring mutual exclusion*  *\* through the arbitrator.*  *\*/*     @Override     public void run() {          System.out.println("\n");       System.out.println("age()=" + Scheduler.age() + ", "           + name + " is running ");       System.out.println("Name:"         + Thread.currentThread().getName());       System.out.println("ID:"         + Thread.currentThread().getId());         arbitrator.wantToEnterCS(id);        for (int m = 1; m <= M; m++)        {           arbitrator.wantToEnterCS(id);           sum = fn(sum, m);       arbitrator.finishedInCS(id);        }          System.out.println("age()=" + Scheduler.age() + ", "           + name + " is done, sum = " + sum);       System.out.println("\n");       }  }  class RaceManyThreads {  */\*\**  *\* Main class that manages multiple racing threads.*  *\* Handles command line arguments and coordinates the execution of racer threads.*  *\*/*     private static int M = 100;     private static int numRacers = 1;     private static Arbitrator arbitrator;  */\*\**  *\* Main entry point for the program.*  *\* Processes command line arguments and creates/manages racer threads.*  *\* @param args String[] - Command line arguments:*  *\*                       -M: number of computations per racer*  *\*                       -N: number of racer threads*  *\*                       -U: display usage information*  *\*/*     public static void main(String[] args) {        GetOpt go = new GetOpt(args, "UtN:M:");        go.optErr = true;        String usage = "Usage: -t -M m -N numracers";        int ch = -1;  *//boolean timeSlicingEnsured = false;*        while ((ch = go.getopt()) != go.optEOF) {           if      ((char)ch == 'U') {              System.out.println(usage);  System.exit(0);           }           else if ((char)ch == 'M')              M = go.processArg(go.optArgGet(), M);           else if ((char)ch == 'N')              numRacers = go.processArg(go.optArgGet(), numRacers);           else {              System.err.println(usage);  System.exit(1);           }        }        System.out.println("RaceManyThreads: M=" + M + ", N=" + numRacers  *//+ ", timeSlicingEnsured=" + timeSlicingEnsured*        );  *// Racer racerObject = new Racer("RacerObject", M);*        arbitrator = new Arbitrator(numRacers);        Racer[] racer = new Racer[numRacers];  *// Thread[] racer = new Thread[numRacers];*        for (int i = 0; i < numRacers; i++)  *// racer[i] = new Thread(racerObject, "RacerThread" + i);*           racer[i] = new Racer("RacerThread" + i, M, i, arbitrator);        for (int i = 0; i < numRacers; i++) {  *// racer[i].start();*          racer[i].start();        }        System.out.println("age()=" + Scheduler.age() +                         ", all Racer threads started");        try {              for (int i = 0; i < numRacers; i++) racer[i].join();           } catch (InterruptedException e) {              System.err.println("interrupted out of join");           }        System.out.println("RaceManyThreads done");        System.exit(0);     }  } |  |

Racer.java Semaphore Algorithm

|  |
| --- |
| */\*---------------------------------------------------------------------*  *Name: Kristopher Adams*  *Course: CS 3230, Section 1, Spring 2025*  *Purpose: This program demonstrates a simple multi-threaded environment*  *where multiple "Racers" perform computations. The program ensures mutual*  *exclusion using the "Arbitrator" class to manage access to a shared resource.*  *Input: Command line arguments -M (number of computations) and -N (number of racers).*  *Output: Displays the progress and results of each Racer thread.*  *---------------------------------------------------------------------\*/*  class Racer extends Thread {  */\*\**  *\* Represents a racing thread that performs computations in a controlled manner.*  *\* This class handles mutual exclusion using an Arbitrator to safely access shared resources.*  *\*/*     private String name;     private int M = 0;     private static volatile long sum = 0;     private int id;     private Arbitrator arbitrator;  */\*\**  *\* Constructor for creating a new Racer thread.*  *\* @param name String - The name identifier for this racer thread*  *\* @param M int - The number of computations this racer will perform*  *\* @param id int - Unique identifier for this racer, used for arbitration*  *\* @param arbitrator Arbitrator - The arbitrator instance managing mutual exclusion*  *\*/*     public Racer(String name, int M, int id, Arbitrator arbitrator) {        this.name = name;        this.M = M;        this.id = id;        this.arbitrator = arbitrator;        System.out.println("age()=" + Scheduler.age() + ", "           + name + " is alive, M=" + M);     }  */\*\**  *\* Performs a mathematical computation on the given values.*  *\* @param j long - The initial value to start computations from*  *\* @param k int - The number of iterations to perform calculations*  *\* @return long - The computed sum after k iterations*  *\*/*     private long fn(long j, int k) {        long total = j;        for (int i = 1;  i <= k; i++) total += (2 \* i - 1) \* (2 \* i - 1);        return total;     }  */\*\**  *\* Main execution method for the racer thread.*  *\* Performs M iterations of computations while ensuring mutual exclusion*  *\* through the arbitrator.*  *\*/*     @Override     public void run() {        arbitrator.wantToEnterCS(id);        System.out.println("\n");       System.out.println("age()=" + Scheduler.age() + ", "           + name + " is running ");       System.out.println("Name:"         + Thread.currentThread().getName());       System.out.println("ID:"         + Thread.currentThread().getId());            for (int m = 1; m <= M; m++)        {             sum = fn(sum, m);          }          System.out.println("age()=" + Scheduler.age() + ", "           + name + " is done, sum = " + sum);       System.out.println("\n");       arbitrator.finishedInCS(id);       }  }  class RaceManyThreads {  */\*\**  *\* Main class that manages multiple racing threads.*  *\* Handles command line arguments and coordinates the execution of racer threads.*  *\*/*     private static int M = 100;     private static int numRacers = 1;     private static Arbitrator arbitrator;  */\*\**  *\* Main entry point for the program.*  *\* Processes command line arguments and creates/manages racer threads.*  *\* @param args String[] - Command line arguments:*  *\*                       -M: number of computations per racer*  *\*                       -N: number of racer threads*  *\*                       -U: display usage information*  *\*/*     public static void main(String[] args) {        GetOpt go = new GetOpt(args, "UtN:M:");        go.optErr = true;        String usage = "Usage: -t -M m -N numracers";        int ch = -1;  *//boolean timeSlicingEnsured = false;*        while ((ch = go.getopt()) != go.optEOF) {           if      ((char)ch == 'U') {              System.out.println(usage);  System.exit(0);           }           else if ((char)ch == 'M')              M = go.processArg(go.optArgGet(), M);           else if ((char)ch == 'N')              numRacers = go.processArg(go.optArgGet(), numRacers);           else {              System.err.println(usage);  System.exit(1);           }        }        System.out.println("RaceManyThreads: M=" + M + ", N=" + numRacers  *//+ ", timeSlicingEnsured=" + timeSlicingEnsured*        );  *// Racer racerObject = new Racer("RacerObject", M);*        arbitrator = new Arbitrator(numRacers);        Racer[] racer = new Racer[numRacers];  *// Thread[] racer = new Thread[numRacers];*        for (int i = 0; i < numRacers; i++)  *// racer[i] = new Thread(racerObject, "RacerThread" + i);*           racer[i] = new Racer("RacerThread" + i, M, i, arbitrator);        for (int i = 0; i < numRacers; i++) {  *// racer[i].start();*          racer[i].start();        }        System.out.println("age()=" + Scheduler.age() +                         ", all Racer threads started");        try {              for (int i = 0; i < numRacers; i++) racer[i].join();           } catch (InterruptedException e) {              System.err.println("interrupted out of join");           }        System.out.println("RaceManyThreads done");        System.exit(0);     }  } |

Script01

|  |
| --- |
|  |

Script02

|  |
| --- |
|  |

Appendix

n/a