Ahsanullah University of Science & Technology

Department of Computer Science & Engineering

<u>Report on</u>

"Dining Philosopher"

<u>Prepared for:</u>

Mr. Mohammad Moinul Hoque
Associate Professor, CSE

<u>Course Number & Name:</u>

CSE-3213 : Operating System

<u>Prepared by:</u>

Parvez Ahammed
ID: 202001040129
Section: C

Date: 12.07.23

Problem Explanation:

The dining philosopher's problem is a well-known synchronization problem. It involves five philosophers sitting around a circular table with a bowl of noodles at the center and five forks, one placed between each pair of philosophers. In order to eat, a philosopher needs both forks. If a philosopher doesn't have both forks, they have to wait until one becomes available. However, if all five philosophers try to pick up their forks simultaneously, a deadlock can occur. To illustrate, let's consider the scenario where the first philosopher picks up the left fork but gets preempted before picking up the right fork. Then the second philosopher enters and picks up their left fork but also gets preempted. If this sequence continues until the last philosopher, all the philosophers will have only the left fork, making it impossible for them to eat as they lack the right fork. This situation ultimately leads to a deadlock.

Solution:

The solution to the Dining Philosophers problem using binary semaphores involves using a semaphore for each fork on the table. The semaphores can be in one of two states: available (value of 1) or unavailable (value of 0). Each philosopher is represented by a process or thread. Each philosopher is assigned a unique identifier ranging from 0 to 4 (for five philosophers). Each fork is represented by a binary semaphore. Initially, all semaphores are available (set to 1). To avoid deadlock, a philosopher must follow a simple protocol to acquire the forks:

a. When a philosopher wants to eat, they first check if both forks on their sides are available. They do this by attempting to acquire the left fork first and then the right fork.

b. If both forks are available, the philosopher proceeds to pick them up and starts eating.

c. If any of the forks is unavailable (semaphore value of 0), the philosopher cannot pick it up and must wait until both forks become available.

d. When the philosopher finishes eating, they release both forks back to the table, incrementing the semaphore values to make them available for other philosophers.

To prevent deadlock in the Dining Philosophers problem, a modification can be made where the last philosopher changes the order of acquiring forks. Instead of starting with the left fork, they first try to acquire the right fork and then the left fork.

Scenarios:

Scenario-1:

Consider a scenario where four philosophers, P1, P2, P3, and P4, are already eating, and only P0 is waiting to eat. In this scenario, P0 follows the protocol to avoid deadlock P0 thinks and then tries to acquire the left fork. If the left fork is available, P0 picks it up. However, in this case, P0 finds that the left fork is already held by P4, so P0 is unable to acquire it immediately. P0 enters a waiting state until the left fork held by P4 becomes available. P0 patiently waits until P4 finishes eating and releases the left fork. Once the left fork is available, P0 seizes the opportunity and quickly acquires it. Now, P0 attempts to acquire the right fork to complete the pair of forks needed for eating. However, P0 discovers that the right fork is currently held by P1, who is still eating. Therefore, P0 patiently waits for P1 to finish eating and release the right fork. After some time, P1 finishes eating and puts down the right fork, making it available for others. P0 seized this opportunity and swiftly picked up the right fork. With both the left and right forks in its possession, P0 finally starts eating its meal, enjoying the acquired resources. Upon finishing its meal, P0 releases both forks back onto the table, making them available for other philosophers to use.

In this scenario, even though all other philosophers are already eating, P0 avoids deadlock by patiently waiting for the necessary forks to become available. By following the protocol and acquiring forks in the correct order, P0 successfully avoids a situation where all philosophers are stuck waiting for a fork, ensuring a deadlock-free execution.

Scenario-2:

In a scenario where all philosophers except one (P2) are already eating, and P2 wants to eat, the following steps demonstrate how deadlock is avoided:

P2 checks if the left fork is available. If it is, P2 acquires the left fork.If the left fork is unavailable (held by another philosopher), P2 waits until it becomes available.Once the left fork becomes available, P2 proceeds to try acquiring the right fork.Since P3 is currently using the right fork, P2 cannot acquire it immediately. Instead, P2 waits for P3 to finish eating and releases the right fork.Once P3 finishes eating and releases the right fork, it becomes available for P2 to acquire.P2 successfully acquires both the left and right forks and starts eating. After P2 finishes eating, they release both forks, making them available for other philosophers to use. By following this protocol, which includes waiting for forks to become available and acquiring them in a specific order, deadlock is avoided

Scenario-3:

In a situation where all philosophers except P2 are currently eating, P2 desires to eat. P2 first checks if the left fork is available and if so, takes hold of it. If the left fork is already in use, P2 patiently waits for it to become available. Then, P2 attempts to acquire the right fork, but since it is currently being used by P3, P2 waits for P3 to finish eating and release the right fork. Once the right fork is free, P2 acquires it and starts enjoying the meal. After finishing, P2 puts down both forks, allowing other philosophers to utilize them.

Appendix:

```
 diningphilosopher;


import java.util.concurrent.Semaphore;


 DiningPhilosopher {
   private static final int NUM_PHILOSOPHERS = 5;
   private static final int NUM_FORKS = 5;


   private static Semaphore[] forks;
   private static Philosopher[] philosophers;


   public static void main(String[] args) {
     forks = new Semaphore[NUM_FORKS]; philosophers = new
     Philosopher[NUM_PHILOSOPHERS];


     // Initialize forks for (int i =
     0; i < NUM_FORKS; i++) {
       forks[i] = new Semaphore(1); // Semaphore with initial permit of 1
     }


     // Create philosophers and start their
     threads for (int i = 0; i < NUM_PHILOSOPHERS;
     i++) {
       forks[i] = new Semaphore(1); // Semaphore with initial permit of 1
     }


     // Create philosophers and start their
     threads for (int i = 0; i < NUM_PHILOSOPHERS;
     i++) {
       Semaphore leftFork = forks[i];
       Semaphore rightFork = forks[(i + 1) %
       NUM_FORKS]; // Adjusting priority for deadlock
       avoidance if (i == NUM_PHILOSOPHERS - 1) {
  philosophers[i] = new Philosopher(i, rightFork, leftFork); // Last
 philosopher picks up right fork first
       } else { philosophers[i] = new Philosopher(i,
         leftFork, rightFork);
       } new
       Thread(philosophers[i]).start();
     }
   }
   public static class Philosopher implements Runnable {
     private int id; private
     Semaphore firstFork; private
     Semaphore secondFork;


     public Philosopher(int id, Semaphore firstFork, Semaphore secondFork) {
       this.id = id; this.firstFork
       = firstFork; this.secondFork
       = secondFork;
     }


     @Override public
     void run() { try
     {
         while (true) {
           // Thinking
           System.out.println("Philosopher " + id + " is thinking.");
           Thread.sleep((long) (Math.random() * 1000));


           // Pick up forks
           pickUpForks();
           // Eating
           System.out.println("Philosopher " + id + " is eating.");
           Thread.sleep((long) (Math.random() * 1000));

           // Put down forks
           putDownForks();
         }
       } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
       }
     }

     private void pickUpForks() throws
       InterruptedException { // Acquire the forks in the
       proper order if (id % 2 == 0) {
         firstFork.acquire();
         secondFork.acquire();
       } else {
         secondFork.acquire();
         firstFork.acquire();
       }
     }

     private void putDownForks() {
       // Release the forks
       firstFork.release();
       secondFork.release();
     }
   }
 }
```