Design and Analysis of Algorithms

Assignment - 6

Implement Concurrent Dining Philosophers Problem

Aim

To implement concurrent Dinning philosopher problem

Objective

The objective of the Dining Philosopher Problem is to find a solution that allows a fixed number of philosophers to dine together, each with a bowl of rice, and a chopstick on either side of them. The problem is to find a way to allow each philosopher to eat without causing a deadlock or a starvation situation, where a philosopher is left waiting indefinitely for the chopstick they need to become available.

Theory

Five philosophers are sitting on a round table caught in deep thought. Besides thinking, they have to eat every now and then or they will starve. Between each philosopher is one chopstick. In the middle of the table is a bowl of spaghetti. Two chopsticks are required in order to eat the entangled spaghetti. If a philosopher (task) wants to eat (run), he has to acquire the chopstick (resources) of both of his neighbours. He can only get the chopsticks if none of his neighbours are using them. This implies that no neighbouring philosophers can eat at the same time and that at most two

philosophers can eat at a time. A solution to the problem has to guarantee mutually exclusive access to the resources. Absence of deadlock and absence of starvation are important as well, however, we don't want to address starvation at the moment.

- The monitor dining controls the distribution of the chopsticks.
- Each philosopher, before starting to eat, invoke the operation pickup. After the successful completion of the operation, the philosopher may eat. The philosopher invokes the putdown operation, and may start to think. Thus, philosopher i invokes the operations pickup and putdown.
- We count the number of times that each philosopher gets to eat to see if starvation happens. The Dining Philosopher Problem presents a classic concurrency problem that can lead to deadlock, starvation, and unfairness.

There are several strategies that can be used to address these issues:

1. **Fairness:** One way to ensure fairness is to implement a solution where each philosopher gets an equal amount of time to eat. This can be achieved by implementing a round-robin scheduling approach where each philosopher gets a turn to pick up their chopsticks and eat. Additionally, we can also enforce that a philosopher only picks up the chopstick on their left first, and then the one on their right. This way, no two adjacent philosophers will have the same chopstick, preventing a philosopher from being indefinitely blocked from eating.

2. **Deadlock:** Deadlock can occur when each philosopher picks up their left chopstick, but then they are all waiting indefinitely for their right chopstick to become available. To avoid this, we can implement a solution where a philosopher must pick up both chopsticks within a certain time frame, or else they must put down their left chopstick and start over again.

3. **Starvation:** Starvation can occur when one or more philosophers are blocked from eating indefinitely, while others are constantly picking up the chopsticks. One way to avoid this is to implement a priority-based solution where philosophers with higher priority (e.g., those who have waited longer to eat) get to eat first. Additionally, we can also implement a timeout mechanism where philosophers who have been waiting for a certain amount of time get to jump the queue and eat.

In summary, to ensure fairness, prevent deadlock, and avoid starvation in the Dining Philosopher Problem, we need to implement a combination of scheduling strategies, priority mechanisms, and timeout mechanisms that ensure that each philosopher gets a turn to eat, no two adjacent philosophers have the same chopstick, and no philosopher is blocked from eating indefinitely.

Code

```java
import java.util.concurrent.Semaphore;

public class DiningPhilosopher {

    private static final int N = 5;
    private static final int THINKING = 2;
    private static final int HUNGRY = 1;
    private static final int EATING = 0;
    private static final int[] phil = { 0, 1, 2, 3, 4 };
    private static final Semaphore mutex = new Semaphore(1);
    private static final Semaphore[] S = new Semaphore[N];

    static {
        for (int i = 0; i < N; i++) {
            S[i] = new Semaphore(0);
        }
    }

    private static int left(int i) {
        return (i + 4) % N;
    }
```

```
    private static int right(int i) {
        return (i + 1) % N;
    }

    private static void test(int i) throws
InterruptedException {
        if (state[i] == HUNGRY && state[left(i)] != EATING
&& state[right(i)] != EATING) {
            state[i] = EATING;
            Thread.sleep(2000);
            System.out.println("Philosopher " + (i + 1) + "
takes fork " + (left(i) + 1) + " and " + (i + 1));
            System.out.println("Philosopher " + (i + 1) + "
is Eating");
            S[i].release();
        }
    }

    private static void takeFork(int i) throws
InterruptedException {
        mutex.acquire();
        state[i] = HUNGRY;
        System.out.println("Philosopher " + (i + 1) + " is
hungry");
        test(i);
        mutex.release();
        S[i].acquire();
        Thread.sleep(1000);
    }

    private static void putFork(int i) throws
InterruptedException {
        mutex.acquire();
        System.out.println("Philosopher " + (i + 1) + "
putting fork " + (left(i) + 1) + " and " + (i + 1) + "
down");
        System.out.println("Philosopher " + (i + 1) + " is
thinking");
        state[i] = THINKING;
        test(left(i));
        test(right(i));
        mutex.release();
    }

    private static int[] state = new int[N];

    public static void main(String[] args) throws
InterruptedException {
```

```
        for (int i = 0; i < N; i++) {
            state[i] = THINKING;
        }

        Thread[] thread = new Thread[N];
        for (int i = 0; i < N; i++) {
            final int index = i;
            thread[i] = new Thread(() -> {
                while (true) {
                    try {
                        Thread.sleep(1000);
                        takeFork(index);
                        putFork(index);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            thread[i].start();
            System.out.println("Philosopher " + (i + 1) + "
is thinking");
        }

        for (int i = 0; i < N; i++) {
            thread[i].join();
        }
    }
}
```

Output

```
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 5 is hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 is hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is hungry
Philosopher 3 is hungry
Philosopher 1 is hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
```

Conclusion

In conclusion, the Dining Philosopher Problem is a classic concurrency problem that presents challenges such as fairness, deadlock, and starvation. Through the use of various strategies such as round-robin scheduling, chopstick acquisition timeouts, priority mechanisms, and timeout mechanisms, it is possible to implement a solution that ensures that all philosophers get a chance to eat, no philosopher is blocked indefinitely, and no two adjacent philosophers have the same chopstick.