Lab Report: Synchronization Primitives and Deadlocks

Aim

To study the potential problems arising from the improper use of synchronization primitives, such as deadlocks, and to present solutions to avoid them.

Theory

Synchronization Primitives

In concurrent programming, synchronization primitives (such as locks, semaphores, and monitors) are used to coordinate the access of multiple threads to shared resources. However, improper usage can lead to serious issues, such as:

- Deadlocks: Occurs when two or more threads are waiting for each other to release resources, leading to a state where none of them can proceed.

- Starvation: A thread never gets the chance to acquire the necessary resources because other threads are continuously favored.

- Race Conditions: When the timing of thread execution affects the outcome of a program, leading to unpredictable results.

Deadlock Conditions

Deadlocks occur when the following four conditions hold simultaneously:

1. Mutual Exclusion: Only one thread can access a resource at a time.

2. Hold and Wait: A thread holds a resource while waiting for another resource.

3. No Preemption: A resource cannot be forcibly taken from a thread.

4. Circular Wait: A set of threads are waiting for each other in a circular chain.

Solutions to Deadlocks

1. Avoid Circular Wait: Impose a total ordering of resource acquisition to avoid circular dependencies.

2. Deadlock Detection and Recovery: Periodically check for deadlocks and recover by forcibly terminating one or more threads.

3. Resource Allocation Graphs: Model resource allocations using graphs and detect cycles that indicate deadlocks.

4. Timeouts: Use timeouts for resource acquisition attempts to prevent indefinite waiting.

Code

This Java code demonstrates how a deadlock can occur and presents a solution using the tryLock method to prevent indefinite waiting.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.TimeUnit;

public class DeadlockDemo {

    private final Lock resourceA = new ReentrantLock();
    private final Lock resourceB = new ReentrantLock();

    public void deadlockScenario() {
        Thread t1 = new Thread(() -> {
            try {
                resourceA.lock();
                System.out.println("Thread 1: Locked Resource A");
                Thread.sleep(50);
                resourceB.lock();
                System.out.println("Thread 1: Locked Resource B");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                resourceA.unlock();
                resourceB.unlock();
            }
        });

        Thread t2 = new Thread(() -> {
```

```
        try {
            resourceB.lock();
            System.out.println("Thread 2: Locked Resource B");
            Thread.sleep(50);
            resourceA.lock();
            System.out.println("Thread 2: Locked Resource A");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            resourceB.unlock();
            resourceA.unlock();
        }
    });

    t1.start();
    t2.start();
}

public void deadlockSolution() {
    Thread t1 = new Thread(() -> {
        try {
            if (resourceA.tryLock(100, TimeUnit.MILLISECONDS)) {
                System.out.println("Thread 1: Locked Resource A");
                Thread.sleep(50);
                if (resourceB.tryLock(100, TimeUnit.MILLISECONDS)) {
                    System.out.println("Thread 1: Locked Resource B");
                    resourceB.unlock();
```

```
            }
            resourceA.unlock();
        } else {
            System.out.println("Thread 1: Could not lock Resource A, avoiding deadlock");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

Thread t2 = new Thread(() -> {
    try {
        if (resourceB.tryLock(100, TimeUnit.MILLISECONDS)) {
            System.out.println("Thread 2: Locked Resource B");
            Thread.sleep(50);
            if (resourceA.tryLock(100, TimeUnit.MILLISECONDS)) {
                System.out.println("Thread 2: Locked Resource A");
                resourceA.unlock();
            }
            resourceB.unlock();
        } else {
            System.out.println("Thread 2: Could not lock Resource B, avoiding deadlock");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
```

```
        t1.start();
        t2.start();
    }

    public static void main(String[] args) {
        DeadlockDemo demo = new DeadlockDemo();

        System.out.println("Running Deadlock Scenario:");
        demo.deadlockScenario();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("\nRunning Deadlock Solution:");
        demo.deadlockSolution();
    }
}
```

Output

Deadlock Scenario Output

Thread 1: Locked Resource A

Thread 2: Locked Resource B

(Deadlock occurs here)

Deadlock Solution Output

Thread 1: Locked Resource A

Thread 1: Could not lock Resource B, avoiding deadlock

Thread 2: Locked Resource B

Thread 2: Could not lock Resource A, avoiding deadlock