CO3408 – Week 8 Lab Sheet

Advanced Synchronization - Semaphores and Scheduling

Learning Outcomes

By completing this lab, students will be able to:

- Differentiate pre-emptive vs non-pre-emptive scheduling.
- Use binary and counting semaphores in Java.
- Implement mutual exclusion, ordering constraints, and Demonstrate Mutual Exclusion with Race Condition and ReentrantLock

Lab Task 1 — Observe Preemptive vs Non-Preemptive Behaviour

Explain the difference between preemptive and non-preemptive scheduling in your own words.
Then run the following Java program and observe the output.
Comment on whether Java threads behave preemptively or non-preemptively.

```
public class SchedulingDemo {
    public static void main(String[] args) {
        Runnable r = () -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + " → " +
i);
                // No sleep here — one thread may dominate if non-preemptive
            }
        };

        Thread t1 = new Thread(r, "Thread-A");
        Thread t2 = new Thread(r, "Thread-B");

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

Explanation

- Preemptive scheduling: CPU can forcibly switch from one thread to another.
- Non-preemptive scheduling: Running thread keeps CPU until it yields/blocks.

Lab Task 2 — Demonstrate Mutual Exclusion with Race Condition

Create a program that increments a counter in two threads without mutual exclusion.

Observe the wrong results.

Code (Race Condition Version)

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

public class RaceConditionDemo2 {

    private static int counter = 0;
//    private static ReentrantLock lock = new ReentrantLock();

//    public static void increment() {
//        lock.lock();          // acquire lock
//        try {
//            counter++;
//        } finally {
//            lock.unlock();    // always release
//        }
//    }
    public static void increment() {
        counter++;
    }
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 10000; i++) increment();
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final counter = " + counter);
    }
}
```

Expected output (wrong):

Final counter = 18673

or

Final counter = 20000

or
varies every run → because race conditions occur.

Then fix the problem using a synchronized block or ReentrantLock.

Fixing the Race condition: Uncomment the commented code and comment out the increment() method which is currently uncommented.

Correct Output

Final counter = 20000

Mutual exclusion solves the race condition.

Task 3 — Use a Semaphore to Limit Access

Use a Semaphore(1) to allow only one thread at a time to access a "critical section." Each thread should print when it **enters** and **exits** the protected area.

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

public class SemaphoreDemo {

    private static Semaphore semaphore = new Semaphore(1);

    public static void main(String[] args) {

        Runnable task = () -> {
            String name = Thread.currentThread().getName();
            try {
                System.out.println(name + " waiting...");
                semaphore.acquire();     // request permit

                System.out.println(name + " entered critical section");
                Thread.sleep(500); // simulate work
                System.out.println(name + " leaving...");

            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                semaphore.release();     // release permit
            }
        };

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

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