CO3408 – Week 9 Lab Sheet

Lecture: Deadlock

Learning Outcomes

By completing this lab, students will be able to:

- Explain what deadlock is and why it occurs.
- Identify the four necessary conditions for deadlock.
- Demonstrate how resource ordering prevents deadlock.
- Analyse Java code for potential deadlock risks.
- Describe the Reader–Writer problem and why simple locking is not enough.

Lab Task: Identify and Reproduce a Deadlock (Simple Java Example)

The following program may deadlock.
Your task is to run it and observe when the deadlock happens.

```
public class DeadlockDemo {
    private static final Object A = new Object();
    private static final Object B = new Object();
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (A) {
                sleep(100);
                synchronized (B) {
                    System.out.println("t1 acquired A and B");
                }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized (B) {
                sleep(100);
                synchronized (A) {
                    System.out.println("t2 acquired B and A");
                }
            }
        });

        t1.start();
        t2.start();
    }
    private static void sleep(long ms) {
        try { Thread.sleep(ms); } catch (InterruptedException e) {}
    }
}
```

1. Run the program several times.
2. Note that sometimes it prints nothing and hangs → deadlock.
3. Explain which of the four deadlock conditions occur.

Deadlock happens because all four conditions hold.

Lab Task 2 - Fix the Deadlock Using Resource Ordering

Rewrite the previous program so **both threads lock resources in the same order**.

Use the rule: Always lock **A → B** (alphabetical order).

```
public class OrderedLocking {
    private static final Object A = new Object();
    private static final Object B = new Object();
    public static void acquireLocks() {
        synchronized (A) {
            synchronized (B) {
                System.out.println(Thread.currentThread().getName() + "
acquired A and B");
            }
        }
    }
    public static void main(String[] args) {
        Thread t1 = new Thread(OrderedLocking::acquireLocks);
        Thread t2 = new Thread(OrderedLocking::acquireLocks);

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

Program should never deadlock, no matter how many times you run it.

Task 3 - Simple Reader–Writer Problem Using ReadWriteLock

Implement a program where:

- Multiple readers can read simultaneously
- Only one writer can update the value
- No writer can write while a reader is reading

Use Java's packages/libraries:

`java.util.concurrent.locks.ReadWriteLock`

`java.util.concurrent.locks.ReentrantReadWriteLock`

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

```
public class ReaderWriterDemo {
    private static int data = 0;
    private static final ReadWriteLock lock = new ReentrantReadWriteLock();
    public static void main(String[] args) {
        Runnable reader = () -> {
            lock.readLock().lock();
            try {
                System.out.println(Thread.currentThread().getName() + " read:
" + data);
            } finally {
                lock.readLock().unlock();
            }
        };

        Runnable writer = () -> {
            lock.writeLock().lock();
            try {
                data++;
                System.out.println(Thread.currentThread().getName() + "
wrote: " + data);
            } finally {
                lock.writeLock().unlock();
            }
        };

        // Start readers and writers
        new Thread(reader, "Reader-1").start();
        new Thread(reader, "Reader-2").start();
        new Thread(writer, "Writer-1").start();
        new Thread(reader, "Reader-3").start();
    }
}
```

Run the program and observe:

- Readers run together
- Writer runs alone

Add more reader/writer threads.

Confirm no deadlock occurs.