LAB # 8

Deadlock in concurrency:

OBJECTIVE:

Implementing multiple thread blocked resources with help of lock and deadlock conditions.

Lock:

A lock is a thread synchronization mechanism like synchronized blocks except locks can be more sophisticated than Java's synchronized blocks. Locks (and other more advanced synchronization mechanisms) are created using synchronized blocks, so it is not like we can get totally rid of the synchronized keyword.

Deadlock:

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

Let’s consider a deadlock case:

Bank: Deposit money -> Account open

Person: Account open -> deposit money

Here is an example:

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
```

~~Software Engineering Department, SSUET~~

```
   }

   private static class ThreadDemo1 extends Thread {
      public void run() {
         synchronized (Lock1) {
            System.out.println("Thread 1: Holding lock 1...");

            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 1: Waiting for lock 2...");

            synchronized (Lock2) {
               System.out.println("Thread 1: Holding lock 1 & 2...");
            }
         }
      }
   }
   private static class ThreadDemo2 extends Thread {
      public void run() {
         synchronized (Lock2) {
            System.out.println("Thread 2: Holding lock 2...");

            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            System.out.println("Thread 2: Waiting for lock 1...");

            synchronized (Lock1) {
               System.out.println("Thread 2: Holding lock 1 & 2...");
            }
         }
      }
   }
}
```

When you compile and execute the above program, you find a deadlock situation and following is the output produced by the program –

Output

```
Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...

Thread 2: Waiting for lock 1...
```

The above program will hang forever because neither of the threads in position to proceed and waiting for each other to release the lock, so you can come out of the program:

Deadlock Solution:

Let's change the order of the lock and run of the same program to see if both the threads still wait for each other –

Example

```
public class TestThread {
   public static Object Lock1 = new Object();
   public static Object Lock2 = new Object();

   public static void main(String args[]) {
      ThreadDemo1 T1 = new ThreadDemo1();
      ThreadDemo2 T2 = new ThreadDemo2();
      T1.start();
      T2.start();
   }
```

```
    private static class ThreadDemo1 extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Thread 1: Holding lock 1...");

                try { Thread.sleep(10); }
                catch (InterruptedException e) {}
                System.out.println("Thread 1: Waiting for lock 2...");

                synchronized (Lock2) {
                    System.out.println("Thread 1: Holding lock 1 & 2...");
                }
            }
        }
    }
    private static class ThreadDemo2 extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Thread 2: Holding lock 2...");

                try { Thread.sleep(10); }
                catch (InterruptedException e) {}
                System.out.println("Thread 2: Waiting for lock 1...");

                synchronized (Lock2) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }
            }
        }
    }
}
```

So just changing the order of the locks prevent the program in going into a deadlock situation and completes with the following result −

Output

```
Thread 1: Holding lock 1...

Thread 1: Waiting for lock 2...

Thread 1: Holding lock 1 & 2...

Thread 2: Holding lock 1...

Thread 2: Waiting for lock 2...

Thread 2: Holding lock 1 & 2...
```

Lab Task:

Create three threads by implementing thread synchronization block through 3 locks. (Hint: Apply un-sequenced lock to analyze deadlock and solve it through provided solution: