DEADLOCK IN JAVA WITH EXAMPLE

Deadlock involves a mutual interdependence between two or more threads. One simple scenario is when one thread executes some code synchronized on object1 and needs to execute code synchronized on object2 still holding the object1 lock. At the same time, a second thread executes some code synchronized to obejct2 and needs to execute code synchronized to object1. In simple terms, thread1 hold object1 lock and wait for object2 lock and thread 2 hold object 2 lock and wait for object 1 lock. Both the threads will keep on waiting for the resource held by the other without releasing the lock they hold.

Deadlock situation is usually explained using the dining philosophers problem. Consider that 5 philosophers are sitting around a round table and they need 2 chop sticks to eat. However there are only 5 chop sticks. They can all share the chop sticks such that after one person eat, he will place it back on the table and other can take. But consider the situation in which everyone get one chop stick and then wait for other. And if everyone wait without putting the chopstick back, then noon will ever get one and they will keep on waiting.

Deadlocks are dangerous and needs to be avoided using proper synchronization. Deadlocks doesn't always occur and occur only with some bad timings, which is called a race condition. And because of this, it may go undetected in initial testing. Also, deadlocks in Java are non-recoverable unlike database systems. In database systems, whenever a deadlock is detected, the system will try to abort one of the transaction so that the other can proceed with the resources. The aborted transaction can then retry again later.

According to the book Java Concurrency in Practice, there are different types of deadlocks such as Lock ordering deadlocks, Open call deadlocks and Resource deadlocks. Below example is an example for a lock ordering deadlock.

Deadlock is the most important liveness hazards in Java. Other liveness hazards include starvation, poor responsiveness and livelock. Often, starvation is confused with deadlock. They are not same. For deadlock to happen there should be some necessary conditions to happen.

Necessary conditions for deadlock (according to computer science)

A deadlock can occur if all of the following conditions hold simultaneously in a system:

1. **Mutual Exclusion**: At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.
2. **Hold and Wait** or Resource Holding: A process is currently holding at least one resource and requesting additional resources which are being held by other processes.

3. **No Preemption**: The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
4. **Circular Wait**: A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.
   These four conditions are known as the **Coffman conditions**.

Example

Let us consider an example program that deadlocks, and then see how we could have avoided that. The DeadLockExample class two objects lockObject1 and lockObject2, which we will use as locks for synchronizing. Now we will define two Runnables, which we will use for creating two threads: Runnable1 synchronizes on lockObject1 and try to get lockObject2, and Runnable2 synchronizes on lockObject2 and try to get lockObject1. Sometimes, due to some lucky timing this might not go into deadlock, for example, if Runnable1 gets lockObject1 and also gets lockObject2 before Runnable2. Therefore for our demo, we will use a Thread.sleep() after acquiring first lock. Remember that, **Thread.sleep() call does not release the lock, but keeps the lock while sleeping, unlike the wait method that will release the lock and wait.**

Program:

```
public class DeadLockExample {
  final static Object lockObject1= new Object();
  final static Object lockObject2= new Object();


  public static void main(String[] args) {
Thread t1 = new Thread(new Runnable1());
    t1.setName("Thread A");

    t1.start();

    Thread t2 = new Thread(new Runnable2());
    t2.setName("Thread B");

    t2.start();

  }


}
```

```
class Runnable1 implements Runnable
{
public void run()
  {
synchronized (DeadLockExample.lockObject1)
    {
        System.out.println(Thread.currentThread().getName()+": Got lockObject1. Trying for lockObject2");
      try {
        Thread.sleep(500);
        //DeadLockExample.lockObject1.wait(500);
      //DeadLockExample.lockObject1.wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
synchronized (DeadLockExample.lockObject2)
      {
        System.out.println(Thread.currentThread().getName()+": Got lockObject2.");
      }
    }
  }
}

class Runnable2 implements Runnable
{
public void run()
  {
synchronized (DeadLockExample.lockObject2)
    {
```

```
        System.out.println(Thread.currentThread().getName()+": Got lockObject2. Trying for lockObject1");
      try {
        Thread.sleep(500);
        //DeadLockExample.lockObject2.wait(500);
      //DeadLockExample.lockObject2.wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
synchronized (DeadLockExample.lockObject1)
      {
        System.out.println(Thread.currentThread().getName()+": Got lockObject1.");
//DeadLockExample.lockObject1.notify();
      }
    }
  }
}
```

The **output** will be:

Thread A: Got lockObject1. Trying for lockObject2

Thread B: Got lockObject2. Trying for lockObject1

Even though a program has the potential to deadlock, it may never deadlock at all, and will only happen with some bad timings. So we have used Thread.sleep() to simulate that bad timing, so that we can demonstrate deadlock.

Thread.sleep() will hold the lock while sleeping whereas, wait() method will release the lock and wait. So replace Thread.sleep(500); with **DeadLockExample.lockObject1.wait(500)**. The statement wait(500) will release the lock and wait for 500 milli seconds or until someone notifies it using a notify or notifyall method. When you call wait, you are actually releasing your lock and hence other thread

can acquire it and will then release it. After 500 milliseconds, Thread 1 will again be runnable and acquire the released lock. So there won't be any deadlock.

Now, replace **DeadLockExample.lockObject1.wait(500)** with **DeadLockExample.lockObject1.wait().** The program will again as noone is notifying the wait method to wake up. However this is not a deadlock. This is because the hold and wait condition of deadlock is not met here. You can confirm this by looking into the thread dump.

Also, in real code using Java 5 and above, you should use the higher-level concurrency utilities instead of wait and notify. According to Effective Java 2, using wait and notify directly is like programming in “concurrency assembly language,” as compared to the higher-level language provided by java.util.concurrent.

Getting Thread Dumps

You can first use jconsole, jvisualvm or jps to get the process id of your program and then run jstack <process-id> to get the thread stack dump.

You can use the same commands for a program running from an eclipse also.

Source : http://javajee.com/deadlock-in-java-with-example