Deadlocks!

Green points (P1,P2) can be changed by green truck driver. Red points (P3,P4) are unchanged by truck movement. Blue points (P5,P6) can be changed by blue truck driver.

Player 1 is green.
Player 2 is blue.
The entire black line is repainted based on the spline from time to time.
Green points (P1,P2) can be changed by green truck driver. Red points (P3,P4) are unchanged by truck movement. Blue points (P5,P6) can be changed by blue truck driver.

Player 1 is green.  
Player 2 is blue.  
The entire black line is repainted based on the spline from time to time.

What does player 1 need to lock each time its truck moves?  
What does player 2 need to lock each time its truck moves?  
What does the repainter need to lock?

---

**Very Pseudo-code**

**Player 1**

```java
greenTruckMoved:
    synchronized (p1) {
        synchronized (p2) {
            ...do stuff to the points...
        }
    }
```

**Player 2**

```java
blueTruckMoved:
    synchronized (p5) {
        synchronized (p6) {
            ...do stuff to the points...
        }
    }
```

**Repainter**

```java
repaIntLine:
    synchronized (p1) {
        synchronized (p2) {
            synchronized (p3) {
                synchronized (p4) {
                    synchronized (p5) {
                        synchronized (p6) {
                            ...repaint the line...
                        }
                    }
                }
            }
        }
    }
```
When good locks go bad! What could happen?
Consider the previous example, with some slight changes:

```
Player 1
greenTruckMoved:
synchronized (p1) {
  synchronized (p2) {
    ...do stuff to the points...
  }
}

Repainter
repaintLine:
synchronized (p6) {
  synchronized (p5) {
    synchronized (p4) {
      synchronized (p3) {
        synchronized (p2) {
          synchronized (p1) {
            ...repaint the line...
          }
        }
      }
    }
  }
}

Player 2
blueTruckMoved:
synchronized (p5) {
  synchronized (p6) {
    ...do stuff to the points...
  }
}
```

They might **DEADLOCK**
Consider this sequence
- The **Player 1** thread acquires lock **P1**
- The **Repainter** thread acquires locks **P6, P5, P4, P3, P2**
- The **Player 1** thread tries to acquire **P2**
- The **Repainter** thread tries to acquire **P1**
- The **Player 2** thread tries to acquire **P5**

None of the threads can acquire the lock it needs to make progress.

All threads are blocked.

The application “freezes” for both players and the screen is no longer able to update.
**Defining Deadlock**

A set of processing contexts (e.g., threads) is **deadlocked** if each of these is waiting for an action by another in the set (such as releasing a resource like a lock that is being held).

Since they are all waiting on each other in some fashion, none can ever complete.

---

**Our Deadlock**

In the preceding example, the sequence of events leads to three different threads entering a deadlock condition.

- The **Player 1** thread is waiting for a lock on **P2** which is held by the **Repainter** thread.
- The **Repainter** thread is waiting for a lock on **P1** which is held by the **Player 1** thread.
- The **Player 2** is waiting for a lock on **P5** which is held by the **Repainter** thread.
**A major challenge of Deadlock**

A system can have a bug in that it has the potential to deadlock even if it rarely occurs (or has never occurred).

In our example, as long as the repainter thread never happens to be attempting to get its locks while a player is in the middle of getting theirs, all runs fine.

Many bugs exist of this sort; they might happen only under a rare condition, but when that condition arises, the system locks.

---

**Automatically Detecting Deadlocks**

Difficult!

- When processing contexts are in a state of deadlock, “nothing is happening” within those specific contexts, but how do you know they aren’t supposed to be waiting?
- When the entire system is deadlocked “nothing is happening” but then again when none of the threads are currently scheduled that’s also true.
- How can you tell the difference?
**Detecting Deadlock**

At compile time this might be near-impossible.

At run-time perhaps we could create a dependency graph and then apply it to currently-held locks to look for deadlock risks. (*This would be a lot of overhead so I need to look into things and see if this has been tried.*)

Of course, even if you do detect it has happened, what one would do *when* a deadlock scenario was reached? Dump everything that’s locked and throw an exception and crash the thread?

---

**Conditions Necessary for Deadlock**

**Mutual exclusion**

*There is at least one non-sharable resource (e.g. lock).*

**Hold-and-wait**

*Threads already holding resources may request other resources held by other threads.*

**Non-preemptability**

*No resource held by a thread may be forcibly removed from its control.*

**Circular waiting**

*There is a circular chain of dependencies consisting of one thread waiting for a resource held by another thread.*
Circular Waiting and Waits-For Graphs

Circular waiting can be depicted using graphs (i.e. diagrams).
- Circles: threads
- Boxes: locks

There is an arrow from a lock to a thread if the thread holds the lock.

There is an arrow from a thread to a lock if the thread is waiting for the lock.

Waits-For Graph: Cycle = Deadlock!

Thread AB has lockA

Thread AB is waiting for lockB

Thread BA has lockB

Thread BA is waiting for lock A
Preventing Deadlock

Impose your own logical order on the resources (sometimes called *ranked locks*).

When a processing context such as a thread needs to acquire multiple locks, you write your code so that it attempts to acquire them following the imposed ordering.

Our players and repainter.
- The obvious order would be $P_1 < P_2 < P_3 < P_4 < P_5 < P_6$.
- The repainter in the modified example is requesting them in the “wrong” order.
- If we went back to our original version where they were acquired in the “correct” order, there is no chance of deadlock between the three threads.

A disjoint set of chains...

Let’s add another set of resources, a 3-channel color stored as a Red value, a Green value, and a Blue value that could be altered on an individual basis (imagine a rule where you have three players and each color channel represents the strength of one of those three and the merged color is being used to visualize the overall state of the game).

We should once again create a ranking for the locks, perhaps $\text{Red} < \text{Green} < \text{Blue}$.

We would still have the $P_1 < P_2 < P_3 < P_4 < P_5 < P_6$ ranking.

However, there is no need for the rankings to form a single chain!
Copyright © 2012-2013 : Evan Golub