Liveness Hazards

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

April 8, 2018
Deadlock
  Deadlock graph
  Lock-ordering deadlocks
  Open calls
  Resource deadlocks

Avoiding and diagnosing deadlocks
  Timed lock attempts
  Deadlock analysis with thread dumps

Other liveness hazards
  Starvation
  Poor responsiveness
  Livelock

Summary
There is no Count Deadlock

But there is a deadlock graph. See next slide.
Resource allocation graph

Object owned by process

Object \rightarrow Process \rightarrow Object

Process requires object

A very simple graphical presentation. (Not UML). A resource allocation graph (from Tanenbaum, Modern Operating Systems).
Example of a resource graph

One process has all resources and can continue, the other waits.

The dreaded deadlock graph, a.k.a. the deadly embrace.
Four conditions of deadlock

The following conditions, which when ALL present, lead to a deadlock.

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

If you take out one, deadlock will not occur. We will concentrate on avoiding circular wait.
The first three are given e.g. by the intrinsic locks.
Liveness Hazards

Deadlock
Deadlock graph
Lock-ordering deadlocks
Open calls
Resource deadlocks
Avoiding and diagnosing deadlocks
Timed lock attempts
Deadlock analysis with thread dumps
Other liveness hazards
Starvation
Poor responsiveness
Liveloop

Summary

Time sequence of LeftRightDeadlock

lock left → try to lock right → wait forever

lock right → try to lock left → wait forever
Lock ordering deadlocks

- A lock ordering deadlock can occur if the resources needed by multiple threads are locked in different orders.
- A program will be free of lock-ordering deadlocks if all threads acquire the locks in a **fixed global** order.
- One way of doing this is using some kind of unique and `Comparable` identifier, like the `System.identityHashCode`.
- Sometimes there is even a better way, if the lock related objects have unique ID’s (example: bank accounts).
- In both cases: use the same order for locking like lower before higher.
- Deadlocks can also occur if object intentionally cooperate, but do not maintain a proper locking order.
- The trigger to spot a problem is a method call on another object with a lock held.
Open calls

A call with a lock held is like putting your head in one of these.

Prefer open calls

- Calling a method outside the current class (object) with a lock held is a risky business. You might run into a snare or noose.
- An open call is a call with no locks held.
- Using open calls is to deadlock what encapsulation is to thread safety.

Open calls

A call with a lock held is like putting your head in one of these.
Resource deadlocks

Just as threads can deadlock when they are each waiting for a lock that the other holds and will not release, they can also deadlock when waiting for resources. The classical example is *the Dining Philosophers*. Let’s demonstrate.
Lijntrekker Demo

Lijntrekken: treuzelen
Deutsch : Trödeln, Bummeln.
English: dawdle, linger, tarry
Avoiding and diagnosing deadlocks

If you need only one lock at the time, your code never deadlocks.

So that is a first strategy.

If you need more locks at the same time then always check your code for deadlock freedom in two steps:

- Identify the places in the code where multiple locks can be acquired. Make this the smallest set possible.
- Then for all those spots, make sure the locking-order is consistent. Using open calls will make finding these spots a lot easier.
Timed lock attempts

Intrinsic locks have no time-out but **Locks** have. With the the `tryLock()` method you can make an attempt for a lock and poll until you succeed.

When a timed lock fails 
(`public boolean tryLock(long time, TimeUnit unit)`), you do not always know the reason of failure. Now at least you have a chance to record the failure.

This can be combined with backing off for some time and try again.

**Note** This only works with acquiring the locks in one place. You cannot release the outer lock in a nested call. Another reason to use **open calls** only.
Using the JVM to detect deadlocks

Demo with book code.

1. Add a small main program to Start LeftRightDeadLock.
2. Start the application.
3. Start Jconsole and attach to VM of point 2.
4. Show Threads-tab and detect deadlock button.
**Other liveness hazards**

Deadlocks are not the only thing that can go wrong, liveness wise. Other hazards are:

- Starvation
- Missed signals (see chapter 14)
- Livelock
Starvation

Starvation occurs when a thread is continuously denied access to a resource. The most common starvation is a lack of CPU cycles which itself is often caused by inappropriate use of thread priorities.

Avoid tweeking with thread priorities

Avoid the temptation to use thread priorities, since they increase platform dependence and can cause liveness problems. Most concurrent applications can use the default priority for all threads.
Poor responsiveness

Background threads may take so much CPU power, that the foreground GUI thread suffers, with a noticeable effect on the responsiveness of the GUI. This is one example where setting the background thread priority to a lower value can have a positive effect on the perceived responsiveness of the foreground gui tasks.
Livelock

We call a liveness failure a *livelock*, when a thread, although not blocked, cannot make progress and keeps retrying.

Another example is the *after you after you* politeness protocol.

In situations where the livelock potential is inherent to the protocol (think networks with collision detection) backing of random and maybe even (exponentially) growing times can help you out of this problem.
Summary

- The most common liveness hazard is caused by lock-ordering deadlock.
- You can use a graphing technique to detect these deadlocks. This is what the JVM does when detecting deadlocks (but not preventing them).
- Avoiding this hazard is a design issue, not just a coding issue.
- If you need multiple locks, make sure you acquire these locks in a consistent (as in always the same) order.
- Also preferably always use open calls, so that you do not have to follow all execution paths (especially those you do not have the code for) to detect any multiple locks held.
Summary of summary

The three simple rules

1. Always use open calls
2. acquire multiple locks in one place (method)
3. always acquire multiple locks in the same relative order.