**Nested Monitor Lockout**

* [How Nested Monitor Lockout Occurs](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html#how-nested-monitor-lockout-occurs)
* [A More Realistic Example](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html#example)
* [Nested Monitor Lockout vs. Deadlock](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html#vsdeadlock)

**How Nested Monitor Lockout Occurs**

嵌套监视器锁定是一个类似死锁的问题。嵌套监视器锁定发生如下：

Thread 1 synchronizes on A

Thread 1 synchronizes on B (while synchronized on A)

Thread 1 decides to wait for a signal from another thread before continuing

Thread 1 calls B.wait() thereby releasing the lock on B, but not A.

Thread 2 needs to lock both A and B (in that sequence)

to send Thread 1 the signal.

Thread 2 cannot lock A, since Thread 1 still holds the lock on A.

Thread 2 remain blocked indefinately waiting for Thread1

to release the lock on A

Thread 1 remain blocked indefinately waiting for the signal from

Thread 2, thereby

never releasing the lock on A, that must be released to make

it possible for Thread 2 to send the signal to Thread 1, etc.

这听起来可能是一个相当理论化的情况，但是看看下面的单纯的Lock实现：

//lock implementation with nested monitor lockout problem

public class Lock{

protected MonitorObject monitorObject = new MonitorObject();

protected boolean isLocked = false;

public void lock() throws InterruptedException{

**synchronized(this){**

while(isLocked){

**synchronized(this.monitorObject){**

this.monitorObject.wait();

**}**

}

isLocked = true;

**}**

}

public void unlock(){

**synchronized(this){**

this.isLocked = false;

**synchronized(this.monitorObject){**

this.monitorObject.notify();

**}**

**}**

}

}

注意lock()方法首先在“this”上同步，然后在monitorObject成员上同步。如果isLocked是false，那就没问题。线程不会调用monitorObject.wait()。然而如果isLocked是true，线程调用lock()被停放在monitorObject.wait()调用中等待。

这样做的问题是，对monitorObject.wait()的调用只释放monitorObject成员上的同步监视器，而不释放与“this”相关联的同步监视器。

当一开始锁定Lock的线程试图通过调用unlock()解锁它时，它将在试图进入unlock()方法中的synchronized(this)块时被阻塞。它将一直保持阻塞，直到lock()中等待的线程离开synchronized(this)块。但是，在lock()方法中等待的线程不会离开该块，直到isLocked设置为false，并执行monitorObject.notify()为止，就像在unlock()中发生的那样。

简而言之，在lock()中等待的线程需要一个unlock()调用才能成功执行，以便它退出lock()及其内部的同步块。但是，没有线程实际上可以执行unlock()，直到等待在lock()中的线程离开外部同步块。

这个结果是，任何调用lock()或unlock()的线程都会被无限期地阻塞。这被称为嵌套监视器锁定。

**A More Realistic Example**

您可能会声明，永远不会像前面所示的那样实现锁。您不会在内部监视器对象上调用wait()和notify()，而在this上则可能会。但是有些情况下，像上面那样的设计可能会出现。例如，如果要在锁中实现公平([**fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html))。当这样做时，您希望每个线程对每个它们自己的队列对象调用wait()，以便您可以一次通知一个线程。

看看这个公平锁的简单实现：

//Fair Lock implementation with nested monitor lockout problem

public class FairLock {

private boolean isLocked = false;

private Thread lockingThread = null;

private List<QueueObject> waitingThreads =

new ArrayList<QueueObject>();

public void lock() throws InterruptedException{

QueueObject queueObject = new QueueObject();

**synchronized(this){**

waitingThreads.add(queueObject);

while(isLocked || waitingThreads.get(0) != queueObject){

**synchronized(queueObject){**

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

**}**

}

waitingThreads.remove(queueObject);

isLocked = true;

lockingThread = Thread.currentThread();

**}**

}

public **synchronized** void unlock(){

if(this.lockingThread != Thread.currentThread()){

throw new IllegalMonitorStateException(

"Calling thread has not locked this lock");

}

isLocked = false;

lockingThread = null;

if(waitingThreads.size() > 0){

QueueObject queueObject = waitingThreads.get(0);

**synchronized(queueObject){**

queueObject.notify();

**}**

}

}

}

public class QueueObject {}

乍一看，这个实现可能看起来不错，但是请注意lock()方法从两个同步块内部调用queueObject.wait()。一个块在“this”上同步，嵌套在其中的一个块在queueObject本地变量上同步。当线程调用queueObject.wait()时，它释放QueueObject实例上的锁，而不是与“this”关联的锁。

注意，unlock()方法被声明为同步的，它等于一个synchronized(this)块。这意味着，如果线程在lock()内等待，则与“this”相关联的监视器对象将被等待线程锁定。所有调用unlock()的线程将被无限期地阻塞，等待等待线程释放“this”上的锁。但这永远不会发生，因为只有在线程成功地向等待线程发送信号时才会发生，并且这只能通过执行unlock()方法来发送。

因此，上面实现的FairLock可能会导致嵌套监视器锁定。在文本饥饿和公平性([**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html))中描述了公平锁的更好实现。

**Nested Monitor Lockout vs. Deadlock**

嵌套监视器锁定和死锁的结果几乎相同：所涉及的线程最终被阻塞，永远等待彼此。

不过这两种情况是不相同的。正如在死锁([**Deadlock**](http://tutorials.jenkov.com/java-concurrency/deadlock.html))文本中所解释的，当两个线程以不同的顺序获得锁时会发生死锁。线程1锁定A，等待B。线程2已锁定B，现在等待A。如关于死锁预防死锁([**Deadlock Prevention**](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html))文本中所解释的，可以通过始终以相同的顺序锁定锁来避免死锁（锁排序）。但是，嵌套的监视器锁定完全由两个线程以**相同的顺序**进行。线程1锁定A和B，然后释放B并等待来自线程2的信号。线程2既需要A又需要B来发送线程1的信号。因此，一个线程正在等待一个信号，另一个线程正在释放一个锁。

不同点总结如下：

In deadlock, two threads are waiting for each other to release locks.

In nested monitor lockout, Thread 1 is holding a lock A, and waits

for a signal from Thread 2. Thread 2 needs the lock A to send the

signal to Thread 1.