**Slipped Conditions**

* [What is Slipped Conditions?](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html#what-is-slipped-conditions)
* [A More Realistic Example](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html#example)
  + [Removing the Slipped Conditions Problem](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html#removing-slipped-conditions)

**What is Slipped Conditions?**

滑动条件是指，从线程检查某个条件直到它作用于该条件为止，该条件已经被另一个线程更改，因此第一个线程的操作是错误的。这里有一个简单的例子：

public class Lock {

private boolean isLocked = true;

public void lock(){

synchronized(this){

while(isLocked){

try{

this.wait();

} catch(InterruptedException e){

//do nothing, keep waiting

}

}

}

synchronized(this){

isLocked = true;

}

}

public synchronized void unlock(){

isLocked = false;

this.notify();

}

}

记得lock()方法包含两个同步块。第一个块等待直到isLocked为false。第二个块设置isLocked为true，来锁定Lock实例对于其他线程。

假设isLocked为false，两个线程同时调用lock()。如果进入第一个同步块的第一个线程在第一个同步块之后被抢占，那么这个线程将检查isLocked，并标记它为false。如果第二个线程现在允许执行，因此进入第一个同步块，这个线程也将看到isLocked为false。现在两个线程都读取条件为false。然后两个线程都将进入第二个同步块，设置isLocked为true，然后继续。

该场景是滑动条件的一个例子。两个线程都检测这个条件，然后退出同步块，因此允许其他的线程测试该条件，在起初的两个线程改变对于子序列线程的条件之前。换句话说，条件已经从条件被检查的时刻滑掉了，直到线程为了子序列线程修改它。

为了避免滑动条件，条件的测试和设置都必须通过线程原子性的完成。意思是没有其他线程能够在第一个线程的测试和设置之间检查条件。

以上例子中的解决很简单。将isLocked = true;向上移动到第一个同步块中，正好在while循环后面。这就是它的样子：

public class Lock {

private boolean isLocked = true;

public void lock(){

synchronized(this){

while(isLocked){

try{

this.wait();

} catch(InterruptedException e){

//do nothing, keep waiting

}

}

isLocked = true;

}

}

public synchronized void unlock(){

isLocked = false;

this.notify();

}

}

现在isLocked条件的测试和设置在同一个同步块内部原子完成。

**A More Realistic Example**

你可能正当地争辩你可能不会像本文展示的第一个实现那样实现一个Lock，因此声明滑动条件更接近为一个理论问题。但是第一个例子保持的尽量简单以更好地传达滑动条件的概念。

一个更现实的例子可能是公平锁的实施过程，正如[**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)中讨论的那样。如果我们观察[**Nested Monitor Lockout**](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html)一文中的简单实现，然后尝试移除嵌套监视器锁问题，很容易到达这个遭受滑动条件的实现。首先我将会展示来自嵌套监视器锁定文章中的这个例子：

//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 = waitingThread.get(0);

**synchronized(queueObject){**

queueObject.notify();

**}**

}

}

}

public class QueueObject {}

注意synchronized(queueObject)的queueObject.wait()调用嵌套在synchronized(this)块内部，导致嵌套监视器锁定问题。为了避免这个问题，synchronized(queueObject)块必须被移动到synchronized(this) 外面，这是那样的样子：

//Fair Lock implementation with slipped conditions 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);

}

boolean mustWait = true;

while(mustWait){

synchronized(this){

mustWait = isLocked || waitingThreads.get(0) != queueObject;

}

synchronized(queueObject){

if(mustWait){

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

}

}

}

synchronized(this){

waitingThreads.remove(queueObject);

isLocked = true;

lockingThread = Thread.currentThread();

}

}

}

注意：只有lock()方法展示出了，因为那是我唯一修改的方法。

注意lock()方法现在包含3个同步块。

第一个synchronized(this)通过设置mustWait = isLocked || waitingThreads.get(0) != queueObject.检查条件。

第二个synchronized(queueObject)检查是否线程将会等待。在这一刻另一个线程可能已经解锁了锁，但让我们暂时忘记那个。让我们假设锁是解开的，因此线程立刻退出synchronized(queueObject)块。

第三个synchronized(this)块只有当mustWait = false才执行。这会设置条件isLocked返回为true等并离开lock()方法。

想像如果两个线程当锁被解开时同时调用lock()方法会发生什么。首先线程1会检查isLocked条件看到它为false。然后线程2将做同样的事。然后它们都不会等待，都将设置isLocked为true。这是一个滑动条件的基本例子。

**Removing the Slipped Conditions Problem**

为了从上述例子中移除滑动条件问题，最后一个synchronized(this)块的内容必须被移进第一个块里。代码将会自然的修改一些，以适应这个移动。这是它的样子：

//Fair Lock implementation without nested monitor lockout problem,

//but with missed signals 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);

}

boolean mustWait = true;

while(mustWait){

**synchronized(this){**

**mustWait = isLocked || waitingThreads.get(0) != queueObject;**

**if(!mustWait){**

**waitingThreads.remove(queueObject);**

**isLocked = true;**

**lockingThread = Thread.currentThread();**

**return;**

**}**

**}**

synchronized(queueObject){

if(mustWait){

try{

queueObject.wait();

}catch(InterruptedException e){

waitingThreads.remove(queueObject);

throw e;

}

}

}

}

}

}

记得现在局部变量mustWait被测试和设置在同一个同步代码块内。同样注意，即使mustWait局部变量也在synchronized(this)外部、while(mustWait)分句内检查，mustWait变量的值从未在synchronized(this)外部改变过。一个评估mustWait为false 的线程也将原子地设置内部条件(isLocked)以便另外其他的检查该条件的线程将会评估它为true。

在synchronized(this)中的return;语句不是必须的。它只是个小优化。如果线程一定不能等待(mustWait == false)，那么没有理由进入synchronized(queueObject)块和执行if(mustWait)语句。

有观察力的读者将会注意到以上的公平锁的实现仍旧遭受丢失信号问题。想像FairLock实例在一个线程调用lock()时被锁定。在第一个synchronized(this)块之后mustWait为true。然后想像调用lock()的线程被抢占，锁定锁的线程调用unlock()。如果你观察之前展示的unlock()实现，你将注意到它调用了queueObject.notify()。但是，因为等待在lock()中的线程还没有调用queueObject.wait()，对queueObject.notify()的调用被遗忘。信号消失了。当调用lock()的线程刚好在调用queueObject.wait()之后，它将会保持阻塞直到某个其他的线程调用unlock()，这也可能不会发生。

丢失信号问题是[**Starvation and Fairness**](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html)文中展示的FairLock实现转变QueueObject类为带有两个方法：doWait()和doNotify()的信号量的原因。这些方法在QueueObject内部存储和反应信号。那样信号未丢失，即使doNotify()在doWait()之前调用。