**Starvation and Fairness**

* [Causes of Starvation in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#starvation)
  + [Threads with high priority swallow all CPU time from threads with lower priority](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#high-priority)
  + [Threads are blocked indefinitely waiting to enter a synchronized block](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#synchronized)
  + [Threads waiting on an object (called wait() on it) remain waiting indefinitely](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#wait)
* [Implementing Fairness in Java](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairness)
  + [Using Locks Instead of Synchronized Blocks](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#locks)
  + [A Fair Lock](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#fairlock)
  + [A Note on Performance](http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html#performance)

如果一个线程没有被赋予CPU时间，因为其他线程抓住了它的全部，它被称为“饥饿”。线程被“饿死”，因为其他线程被允许CPU时间而不是它。解决饥饿的方法被称为“公平”，即所有线程都有机会执行。

**Causes of Starvation in Java**

以下三个常见的原因可能导致Java中线程的饥饿：

1. 具有高优先级的线程从较低优先级的线程吞吐了所有CPU时间。
2. 线程被无限期地阻塞以等待进入同步块，因为其他线程被允许在其之前访问。
3. 在一个对象上等待(调用该对象上的wait())的线程保持无限期地等待，因为其他线程而非该线程不断地被唤醒。

**Threads with high priority swallow all CPU time from threads with lower priority**

您可以单独设置每个线程的线程优先级。优先级越高，线程授予的CPU时间越长。可以在1到10之间设置线程的优先级。如何解释这一点取决于应用程序正在运行的操作系统。对于大多数应用程序来说，最好保持优先级不变。

**Threads are blocked indefinitely waiting to enter a synchronized block**

Java的同步代码块可能是饥饿的另一个原因。Java的同步代码块不能保证等待进入同步块的线程允许进入的顺序。这意味着理论上存在一个风险，即一个线程在试图进入该块时始终处于阻塞状态，因为其他线程总是在它之前被授予访问权限。这个问题被称为“饥饿”，即线程“饿死”是因为其他线程允许使用CPU时间而它不被允许。

**Threads waiting on an object (called wait() on it) remain waiting indefinitely**

如果调用了对象notify()上的多个线程调用wait()，notify()方法不能保证唤醒了哪个线程。它可能是等待的任何线程。因此，存在这样的风险，即等待某个对象的线程永远不会被唤醒，因为总是其他等待的线程被唤醒而不是它。

**Implementing Fairness in Java**

虽然在Java中不可能实现100%的公平性，但是我们仍然可以实现我们的同步构造来增加线程之间的公平性。

首先学习一个简单的同步代码块：

public class Synchronizer{

public synchronized void doSynchronized(){

//do a lot of work which takes a long time

}

}

如果不止一个线程调用doSynchronized()方法，那么其中一些线程将被阻塞，直到第一个被授予访问权的线程离开该方法。如果不止一个线程被阻塞等待访问，则不能保证哪个线程接下来被授予访问权限。

**Using Locks Instead of Synchronized Blocks**

为了增加等待线程的公平性，首先我们将更改由锁而不是同步块来保护代码块：

public class Synchronizer{

Lock lock = new Lock();

public void doSynchronized() throws InterruptedException{

this.lock.lock();

//critical section, do a lot of work which takes a long time

this.lock.unlock();

}

}

注意doSynchronized()方法不再被声明为同步。相反，临界区间由lock.lock()和lock.unlock()调用来保护。

Lock类的简单实现可以是这样的：

public class Lock{

private boolean isLocked = false;

private Thread lockingThread = null;

public synchronized void lock() throws InterruptedException{

while(isLocked){

wait();

}

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;

notify();

}

}

如果查看上面的Synchronizer类，并查看这个Lock实现，您将注意到，如果同时有多个线程调用lock()，那么试图访问lock()方法的线程现在被阻塞了。第二，如果锁被锁定，则线程在lock()方法中的while(isLocked)循环中的wait()调用中被阻塞。请记住，线程调用wait()将会释放Lock实例上的同步锁，因此等待进入lock()的线程现在可以进入。结果是多个线程最终可以在lock()内调用wait()。

如果回顾doSynchronized()方法，您将注意到lock()和unlock()之间的注释指出，这两个调用之间的代码花费“长”时间来执行。让我们进一步假设，与输入lock()方法并调用wait()相比，执行此代码需要较长的时间，因为锁被锁定了。这意味着，为了能够锁定锁并进入临界区间而等待的大部分时间，都花在lock()方法内部的wait()调用中，而不是在试图进入lock()方法时被阻塞。

如前所述，如果不止一个线程正在等待输入，那么同步块不会保证授予哪个线程访问权限。wait()也不保证在调用notify()时唤醒什么线程。因此，Lock类的当前版本在保证公平性方面与doSynchronized()的同步版本没有什么不同。但我们可以改变。

Lock类的当前版本调用它自己的wait()方法。如果取而代之每个线程在分开的对象上调用wait()，那么在每个对象上只有一个线程调用了wait()方法，Lock类可以决定在这些对象中的哪个上面调用notify()，因此有效地精确选择唤醒什么线程。

**A Fair Lock**

下面显示了前面的锁类变成了一个名为FairLock的公平锁。您将注意到，与前面显示的Lock类相比，实现在同步和wait()/notify()方面稍微有些变化。

我是如何从上一个Lock类开始达到这个设计的，这是一个较长的故事，涉及几个增量式设计步骤，每个步骤都解决了前一个步骤的问题：嵌套监视器锁定([**Nested Monitor Lockout**](http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html))、滑动条件([**Slipped Conditions**](http://tutorials.jenkov.com/java-concurrency/slipped-conditions.html))和遗漏信号([**Missed Signals**](http://tutorials.jenkov.com/java-concurrency/thread-signaling.html#missedsignals))。为了简短起见，本文中省略了上述讨论，但是每个步骤都在有关该主题的适当文本（参见上面的链接）中进行讨论。重要的是，现在对每个调用lock()的线程进行排队，并且只有队列中的第一个线程被允许锁定FairLock实例(如果它是解锁的)。所有其他线程都停机等待，直到它们到达队列的顶部。

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();

boolean isLockedForThisThread = true;

synchronized(this){

waitingThreads.add(queueObject);

}

while(isLockedForThisThread){

synchronized(this){

isLockedForThisThread =

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

if(!isLockedForThisThread){

isLocked = true;

waitingThreads.remove(queueObject);

lockingThread = Thread.currentThread();

return;

}

}

try{

queueObject.doWait();

}catch(InterruptedException e){

synchronized(this) { waitingThreads.remove(queueObject); }

throw e;

}

}

}

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){

waitingThreads.get(0).doNotify();

}

}

}

public class QueueObject {

private boolean isNotified = false;

public synchronized void doWait() throws InterruptedException {

while(!isNotified){

this.wait();

}

this.isNotified = false;

}

public synchronized void doNotify() {

this.isNotified = true;

this.notify();

}

public boolean equals(Object o) {

return this == o;

}

}

首先你可能注意到lock()方法不再被声明为synchronized。相反只有必须同步的块儿嵌套在synchronized块儿中。

FairLock创建一个新的QueueObject实例，然后对每个调用lock()的线程入队。调用unlock()的线程将会拿取队列顶部的QueueObject然后在它上面调用doNotify()，来唤醒等待在那个对象上的线程。这样一次只有一个等待线程被唤醒，而不是全部等待线程。这部分就是支配FairLock公平的因素。

请注意，锁的状态仍旧是在相同的同步块中测试和设置，以避免滑动条件。

同样注意QueueObject 实际是一个信号量。doWait()和doNotify()方法将信号存储在QueueObject内部。这一步是为了避免由线程在调用queueObject.doWait()前被另一个调用unlock()和queueObject.doNotify()的线程取代，导致丢失信号。

最后，注意queueObject.doWait()在try-catch块中被调用。如果InterruptedException异常抛出，线程离开lock()方法，我们必须将其出列。

**A Note on Performance**

如果你比较Lock和FairLock类，你将会注意到FairLock类中的lock()和unlock()内部有更多内容。这个额外的代码将导致FairLock是一个比Lock慢一点的同步机制。这对应用程序的影响程度取决于FairLock所保护的临界区间中的代码需要多长时间执行。执行所需的时间越长，同步器的附加开销就越不显著。它当然也取决于这个代码被调用的频率。