嵌套管程锁死

[原文链接](#)　**作者：**Jakob Jenkov

译者：余绍亮　**校对：**丁一

嵌套管程锁死类似于死锁，下面是一个嵌套管程锁死的场景：

> 线程1获得A对象的锁。
>
> 线程1获得对象B的锁（同时持有对象A的锁）。
>
> 线程1决定等待另一个线程的信号再继续。
>
> 线程1调用B.wait()，从而释放了B对象上的锁，但仍然持有对象A的锁。
>
> 线程2需要同时持有对象A和对象B的锁，才能向线程1发信号。
>
> 线程2无法获得对象A上的锁，因为对象A上的锁当前正被线程1持有。
>
> 线程2一直被阻塞，等待线程1释放对象A上的锁。
>
> 线程1一直阻塞，等待线程2的信号，因此，不会释放对象A上的锁，
>
> 而线程2需要对象A上的锁才能给线程1发信号……

你可以能会说，这是个空想的场景，好吧，让我们来看看下面这个比较挫的Lock实现：

```
01 //lock implementation with nested monitor lockout problem
02 public class Lock{
03     protected MonitorObject monitorObject = new MonitorObject();
04     protected boolean isLocked = false;
05
06     public void lock() throws InterruptedException{
07         synchronized(this){
08             while(isLocked){
09                 synchronized(this.monitorObject){
10                     this.monitorObject.wait();
11                 }
12             }
13             isLocked = true;
14         }
15     }
16
17     public void unlock(){
18         synchronized(this){
19             this.isLocked = false;
```

```
            synchronized(this.monitorObject){
                this.monitorObject.notify();
            }
        }
    }
}
```

可以看到，lock()方法首先在”this”上同步，然后在monitorObject上同步。如果isLocked等于false，因为线程不会继续调用monitorObject.wait()，那么一切都没有问题 。但是如果isLocked等于true，调用lock()方法的线程会在monitorObject.wait()上阻塞。

这里的问题在于，调用monitorObject.wait()方法只释放了monitorObject上的管程对象，而与”this“关联的管程对象并没有释放。换句话说，这个刚被阻塞的线程仍然持有”this”上的锁。

（*校对注：如果一个线程持有这种Lock的时候另一个线程执行了lock操作*）当一个已经持有这种Lock的线程想调用unlock(),就会在unlock()方法进入synchronized(this)块时阻塞。这会一直阻塞到在lock()方法中等待的线程离开synchronized(this)块。但是，在unlock中isLocked变为false，monitorObject.notify()被执行之后，lock()中等待的线程才会离开synchronized(this)块。

简而言之，在lock方法中等待的线程需要其它线程成功调用unlock方法来退出lock方法，但是，在lock()方法离开外层同步块之前，没有线程能成功执行unlock()。

结果就是，任何调用lock方法或unlock方法的线程都会一直阻塞。这就是嵌套管程锁死。

一个更现实的例子

你可能会说，这么挫的实现方式我怎么可能会做呢？你或许不会在里层的管程对象上调用wait或notify方法，但完全有可能会在外层的this上调。

有很多类似上面例子的情况。例如，如果你准备实现一个公平锁。你可能希望每个线程在它们各自的QueueObject上调用wait()，这样就可以每次唤醒一个线程。

下面是一个比较挫的公平锁实现方式：

```
//Fair Lock implementation with nested monitor lockout problem
public class FairLock {
    private boolean isLocked = false;
    private Thread lockingThread = null;
    private List waitingThreads =
        new ArrayList();

    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 {}
```

乍看之下，嗯，很好，但是请注意lock方法是怎么调用queueObject.wait()的，在方法内部有两个synchronized块，一个锁定this，一个嵌在上一个synchronized块内部，它锁定的是局部变量queueObject。当一个线程调用queueObject.wait()方法的时候，它仅仅释放的是在queueObject对象实例的锁，并没有释放”this”上面的锁。

现在我们还有一个地方需要特别注意， unlock方法被声明成了synchronized，这就相当于一个synchronized（this）块。这就意味着，如果一个线程在lock()中等待，该线程将持有与this关联的管程对象。所有调用unlock()的线程将会一直保持阻塞，等待着前面那个已经获得this锁的线程释放this锁，但这永远也发生不了，因为只有某个线程成功地给lock()中等待的线程发送了信号，this上的锁才会释放，但只有执行unlock()方法才会发送这个信号。

因此，上面的公平锁的实现会导致嵌套管程锁死。更好的公平锁实现方式可以参考Starvation and Fairness。

嵌套管程锁死 VS 死锁

嵌套管程锁死与死锁很像：都是线程最后被一直阻塞着互相等待。

但是两者又不完全相同。在死锁中我们已经对死锁有了个大概的解释，死锁通常是因为两个线程获取锁的顺序不一致造成的，线程1锁住A，等待获取B，线程2已经获取了B，再等待获取A。如死锁避免中所说的，死锁可以通过总是以相同的顺序获取锁来避免。

但是发生嵌套管程锁死时锁获取的顺序是一致的。线程1获得A和B，然后释放B，等待线程2的信号。线程2需要同时获得A和B，才能向线程1发送信号。所以，一个线程在等待唤醒，另一个线程在等待想要的锁被释放。

不同点归纳如下：

> 死锁中，二个线程都在等待对方释放锁。
>
> 嵌套管程锁死中，线程1持有锁A，同时等待从线程2发来的信号，线程2需要锁A来发信号给线程1。

余绍亮
2013/03/14 11:22下午

额，看到校对过的文章，真是有种惭愧的感觉

方 腾飞
2013/03/15 12:01上午

翻译文章，走出第一步是关键，继续加油！

wh
2018/08/06 9:14上午

大佬棒棒哒

夕水溪下
2013/03/15 1:34下午

最后一段总结的真好！

Itachi
2014/07/17 9:38下午

其实，在第一个嵌套管程锁死的例子中，只要人为的手动将isLocked 设为false就可以解决锁死。

aoao
2015/04/12 3:41上午

嵌套管程锁死时正在等待信号呢。。不是在忙等啊，兄弟！！