饥饿和公平

原文地址 By Jakob Jenkov 翻译 Simon-SZ 校对：方腾飞

如果一个线程因为CPU时间全部被其他线程抢走而得不到CPU运行时间，这种状态被称之为“饥饿”。而该线程被“饥饿致死”正是因为它得不到CPU运行时间的机会。解决饥饿的方案被称之为“公平性” – 即所有线程均能公平地获得运行机会。

下面是本文讨论的主题：

1. Java中导致饥饿的原因：

   高优先级线程吞噬所有的低优先级线程的CPU时间。
   线程被永久堵塞在一个等待进入同步块的状态。
   线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的wait方法)。

2. 在Java中实现公平性方案，需要:

   使用锁，而不是同步块。
   公平锁。
   注意性能方面。

Java中导致饥饿的原因

在Java中，下面三个常见的原因会导致线程饥饿：

1. 高优先级线程吞噬所有的低优先级线程的CPU时间。
2. 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
3. 线程在等待一个本身(在其上调用wait())也处于永久等待完成的对象，因为其他线程总是被持续地获得唤醒。

高优先级线程吞噬所有的低优先级线程的CPU时间

你能为每个线程设置独自的线程优先级，优先级越高的线程获得的CPU时间越多，线程优先级值设置在1到10之间，而这些优先级值所表示行为的准确解释则依赖于你的应用运行平台。对大多数应用来说，你最好是不要改变其优先级值。

线程被永久堵塞在一个等待进入同步块的状态

Java的同步代码区也是一个导致饥饿的因素。Java的同步代码区对哪个线程允许进入的次序没有任何保障。这就意味着理论上存在一个试图进入该同步区的线程处于被永久堵塞的风险，因为其他线程总是能持续地先于它获得访问，这即是“饥饿”问题，而一个线程被“饥饿致死”正是因为它得不到CPU运行时间的机会。

线程在等待一个本身(在其上调用wait())也处于永久等待完成的对象

如果多个线程处在wait()方法执行上，而对其调用notify()不会保证哪一个线程会获得唤醒，任何线程都有可能处于继续等待的状态。因此存在这样一个风险：一个等待线程从来得不到唤醒，因为其他等待线程总是能被获得唤醒。

在Java中实现公平性

虽Java不可能实现100%的公平性，我们依然可以通过同步结构在线程间实现公平性的提高。

首先来学习一段简单的同步态代码：

```
1  public class Synchronizer{
2
3      public synchronized void doSynchronized(){
4
5      //do a lot of work which takes a long time
6
7      }
8  }
```

如果有一个以上的线程调用doSynchronized()方法，在第一个获得访问的线程未完成前，其他线程将一直处于阻塞状态，而且在这种多线程被阻塞的场景下，接下来将是哪个线程获得访问是没有保障的。

使用锁方式替代同步块

为了提高等待线程的公平性，我们使用锁方式来替代同步块。

```
1  public class Synchronizer{
2      Lock lock = new Lock();
3      public void doSynchronized() throws InterruptedException{
4          this.lock.lock();
5          //critical section, do a lot of work which takes a long time
6          this.lock.unlock();
7      }
8  }
```

注意到doSynchronized()不再声明为synchronized，而是用lock.lock()和lock.unlock()来替代。

下面是用Lock类做的一个实现：

```
01 public class Lock{
02
03     private boolean isLocked      = false;
04
05     private Thread lockingThread = null;
06
07     public synchronized void lock() throws InterruptedException{
08
09     while(isLocked){
10
11         wait();
12
13     }
14
15     isLocked = true;
16
17     lockingThread = Thread.currentThread();
18
19 }
20
21 public synchronized void unlock(){
22
23     if(this.lockingThread != Thread.currentThread()){
24
25          throw new IllegalMonitorStateException(
26
27               "Calling thread has not locked this lock");
28
29          }
30
31     isLocked = false;
32
33     lockingThread = null;
34
35     notify();
36
37     }
38 }
```

注意到上面对Lock的实现，如果存在多线程并发访问lock()，这些线程将阻塞在对lock()方法的访问上。另外，如果锁已经锁上（校对注：这里指的是isLocked等于true时），这些线程将阻塞在while(isLocked)循环的wait()调用里面。要记住的是，当线程正在等待进入lock() 时，可以调用wait()释放其锁实例对应的同步锁，使得其他多个线程可以进入lock()方法，并调用wait()方法。

这回看下doSynchronized()，你会注意到在lock()和unlock()之间的注释：在这两个调用之间的代码将运行很长一段时间。进一步设想，这段代码将长时间运行，和进入lock()并调用wait()来比较的话。这意味着大部分时间用在等待进入锁和进入临界区的过程是用在wait()的等待中，而不是被阻塞在试图进入lock()方法中。

在早些时候提到过，同步块不会对等待进入的多个线程谁能获得访问做任何保障，同样当调用notify()时，wait()也不会做保障一定能唤醒线程（至于为什么，请看线程通信）。因此这个版本的Lock类和doSynchronized()那个版本就保障公平性而言，没有任何区别。

但我们能改变这种情况。当前的Lock类版本调用自己的wait()方法，如果每个线程在不同的对象上调用wait()，那么只有一个线程会在该对象上调用wait()，Lock类可以决定哪个对象能对其调用notify()，因此能做到有效的选择唤醒哪个线程。

公平锁

下面来讲述将上面Lock类转变为公平锁FairLock。你会注意到新的实现和之前的Lock类中的同步和wait()/notify()稍有不同。

准确地说如何从之前的Lock类做到公平锁的设计是一个渐进设计的过程，每一步都是在解决上一步的问题而前进的：Nested Monitor Lockout, Slipped Conditions和Missed Signals。这些本身的讨论虽已超出本文的范围，但其中每一步的内容都将会专题进行讨论。重要的是，每一个调用lock()的线程都会进入一个队列，当解锁后，只有队列里的第一个线程被允许锁住Farlock实例，所有其它的线程都将处于等待状态，直到他们处于队列头部。

```
01 public class FairLock {
02     private boolean           isLocked       = false;
03     private Thread            lockingThread  = null;
04     private List<QueueObject> waitingThreads =
05             new ArrayList<QueueObject>();
06
07   public void lock() throws InterruptedException{
08     QueueObject queueObject           = new QueueObject();
09     boolean     isLockedForThisThread = true;
10     synchronized(this){
11         waitingThreads.add(queueObject);
12     }
13
14     while(isLockedForThisThread){
15       synchronized(this){
16         isLockedForThisThread =
17             isLocked || waitingThreads.get(0) != queueObject;
18         if(!isLockedForThisThread){
19           isLocked = true;
20            waitingThreads.remove(queueObject);
21            lockingThread = Thread.currentThread();
22            return;
23          }
24       }
25       try{
26         queueObject.doWait();
27       }catch(InterruptedException e){
28         synchronized(this) { waitingThreads.remove(queueObject); }
29         throw e;
30       }
31     }
32   }
33
34   public synchronized void unlock(){
35     if(this.lockingThread != Thread.currentThread()){
36       throw new IllegalMonitorStateException(
37         "Calling thread has not locked this lock");
38     }
39     isLocked      = false;
40     lockingThread = null;
```

```
41      if(waitingThreads.size() > 0){
42        waitingThreads.get(0).doNotify();
43      }
44    }
45  }
```

```
01  public class QueueObject {
02
03      private boolean isNotified = false;
04
05      public synchronized void doWait() throws InterruptedException {
06
07      while(!isNotified){
08          this.wait();
09      }
10
11      this.isNotified = false;
12
13  }
14
15  public synchronized void doNotify() {
16      this.isNotified = true;
17      this.notify();
18  }
19
20  public boolean equals(Object o) {
21      return this == o;
22  }
23
24  }
```

首先注意到lock()方法不在声明为synchronized，取而代之的是对必需同步的代码，在synchronized中进行嵌套。

FairLock新创建了一个QueueObject的实例，并对每个调用lock()的线程进行入队列。调用unlock()的线程将从队列头部获取QueueObject，并对其调用doNotify()，以唤醒在该对象上等待的线程。通过这种方式，在同一时间仅有一个等待线程获得唤醒，而不是所有的等待线程。这也是实现FairLock公平性的核心所在。

请注意，在同一个同步块中，锁状态依然被检查和设置，以避免出现滑漏条件。

还需注意到，QueueObject实际是一个semaphore。doWait()和doNotify()方法在QueueObject中保存着信号。这样做以避免一个线程在调用queueObject.doWait()之前被另一个调用unlock()并随之调用queueObject.doNotify()的线程重入，从而导致信号丢失。queueObject.doWait()调用放置在synchronized(this)块之外，以避免被monitor嵌套锁死，所以另外的线程可以解锁，只要当没有线程在lock方法的synchronized(this)块中执行即可。

最后，注意到queueObject.doWait()在try – catch块中是怎样调用的。在InterruptedException抛出的情况下，线程得以离开lock()，并需让它从队列中移除。

性能考虑

如果比较Lock和FairLock类，你会注意到在FairLock类中lock()和unlock()还有更多需要深入的地方。这些额外的代码会导致FairLock的同步机制实现比Lock要稍微慢些。究竟存在多少影响，还依赖于应用在FairLock临界区执行的时长。执行时长越大，FairLock带来的负担影响就越小，当然这也和代码执行的频繁度相关。

匿名
2013/04/02 2:23下午

第三段代码是不是会死锁啊，等待的线程lock方法synchnized锁住了当前对象，拥有锁的对象不是不能unlock了？

匿名
2013/08/26 10:48下午

我也感觉这个段代码有问题：

public synchronized void lock() throws InterruptedException{

08

09 while(isLocked){

10

11 wait();

12

13 }

14

15 isLocked = true;

16

17 lockingThread = Thread.currentThread();

18 }

第一个线程进入lock放之后，将isLocked = true,如果这个时候第二个线程进入lock，会被阻塞，这个时候，第一个线程退出lock方法，释放锁，并且执行其他操作，第二个线程获取到锁进入lock方法，这个时候由于isLocked = tru,所以进入循环，执行wait,这个时候它也释放锁了，并等待，如果第一个完成自己操作执行unlock方法 ，获取锁，执行isLocked = false,执行notify唤醒第二线程，这个等退出unlock的时候，释放锁，这个时候第二个线程才能获取锁，退出阻塞，继续执行。如果是三个线程或者更多会不会有问题？？？？

heipacker

2013/09/24 7:58下午

好像是死锁了

niannian
2013/10/16 12:20下午

等待的线程执行了wait就会释放锁了吧，这样其他线程就可以unlock了。

zhoulin
2017/10/05 12:40下午

这不会导致死锁 首先他保证了lock和unlock的单线程执行顺序 最多会发生活锁

匿名
2013/06/04 10:54上午

你网站上不一样有广告吗？环保吗？

方 腾飞
2013/06/04 10:49下午

的确不环保。抵制的百度把广告放在内容里。

moondust
2014/10/20 5:30下午

广告是市场经济，广告也是该站更好发展的一个保障吧，觉得无可厚非，哈哈

滑行姿态
2013/07/05 7:02下午

读完“公平锁 ”的实现代码 深深被其实现的精妙打动

但 “使用锁方式替代同步块” 中 “要记住的是，当线程正在等待进入lock() 时，可以调用wait()释放其锁实例对应的同步锁，使得其他多个线程可以进入lock()方法，并调用wait()方法。” 这句话不太明白 等待进入lock()

不就是还没得到 “锁实例对应的同步锁”吗 怎么还要 调用wait（ ）释放它呢

Silence

2014/03/24 9:35下午

比如第一个线程得到锁进入lock()方法，并将isLocked设置为true，然后进入临界区，所以第一个线程此时释放了Lock上的锁，其他线程可以继续抢占锁从而进入lock()方法，然后发现isLocked为true，则调用wait()方法释放锁，这就是“可以调用wait()释放其锁实例对应的同步锁”这句话的意思，当释放了锁后，其他线程是不是可以进入lock()方法并调用wait()方法嘛？我是这样理解的，不知道正确不。

maleking
2019/06/06 5:29下午

```
public synchronized void lock() throws InterruptedException{

while(isLocked){

wait();

}

isLocked = true;

lockingThread = Thread.currentThread();

}
```

第一个线程执行完这段代码后就释放了锁，其他线程可以抢占锁并进入lock方法，但是此时会发现isLocked是true从而进入while语句块执行了wait方法释放掉自己持有的锁并进入等待区，其他线程依次按照这个步骤进入等待区，直到isLocked为false，将会有线程再次把isLocked设置为true。所以，所谓的释放锁，就是后续线程抢占了lock方法同步锁却被标志限制住的时候，需要进入等待同时释放掉持有的锁。

匿名
2013/08/26 10:54下午

Second, if the lock is locked, the threads are blocked in the wait() call inside the while(isLocked) loop in the lock() method.

其次，如果the lock is locked，线程将会被阻塞在lock方法的while循环的wait方法调用上。

你们的翻译：

这些线程将阻塞在while(isLocked)循环的wait()调用里面

匿名
2013/08/26 11:08下午

Remember that a thread calling wait() releases the synchronization lock on the Lock instance, so threads waiting to enter lock() can now do so.

记住一个调用wait线程将会释放在Lock实例上的同步锁，所以等待进入lock方法的线程也可以这么干。就是说在进入lock方法前，直接调用lock实例前调用wait，用来释放这个锁？？？？能这么干吗？？？

niannian
2013/10/22 3:52下午

这个公平锁不是可重入的对吧？

krystaljake
2013/11/15 5:11下午

我觉得是，那块会导致错乱？

唐猎果子
2017/02/07 9:13上午

是不可重入的

Wayne
2013/12/05 8:07下午

我有点不明白 希望小编能给予指导下

在公平锁最后端代码里 通过一个Queue来实现公平

```
public class QueueObject {

private boolean isNotified = false;

public synchronized void doWait() throws InterruptedException {

while(!isNotified){

this.wait();

}

this.isNotified = false;

}
```

假设有个线程a 第一次调用了lock（ ），然后一切都执行的很顺利直到

```
try{

queueObject.doWait();
```

}

调用了queueObject.doWait();

但是我觉得第一次调用这个方法就有可能被阻塞了

while(!isNotified){

this.wait();

}

因为isNotified = false 一开始便是定义成false

我觉得一开始不是应该为true ?

方 腾飞
2013/12/09 1:46下午

问题是什么呢 ?

Wayne
2013/12/10 9:38上午

不好意思，回去吧代码 仔细看了下 跑了遍

自己看错了

Thanks

randy
2014/10/21 5:17下午

但我们能改变这种情况。当前的Lock类版本调用自己的wait()方法，如果每个线程在不同的对象上调用wait()，那么只有一个线程会在该对象上调用wait()，Lock类可以决定哪个对象能对其调用notify()，因此能做到有效的选择唤醒哪个线程。

这段话，看的不甚明白。能详细解释下吗 ?

无为
2014/11/13 10:03下午

不错 收益很大

jzh0535
2016/06/21 9:47下午

mark

zhili
2018/06/05 3:12下午

有个单词少个i

Farlock

13x
2019/03/08 5:03下午

这里我有个疑问，文章前面说了，饥饿发生的一个可能的原因是“Java的同步代码区也是一个导致饥饿的因素。Java的同步代码区对哪个线程允许进入的次序没有任何保障。线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。”；
那么后面的公平锁，在将线程加入waitingThreads这个List的时候，不也要先进入一个同步块吗？这里就不会出现一个线程一直处于等待进入同步块的状态吗？
很困惑，有没有人能指点下？

NeverWell
2019/07/06 10:47下午

嗯 文章说了不能实现完全的公平，所以只是提高。第一个方法，加了synchronized关键字的doSynchronized()，在第一个获得访问的线程未完成前，其他线程将一直处于阻塞状态。这个粒度显然太粗了，doSynchronized()的执行时间越长，等待的阻塞线程就越多，更加容易出现不公平的情况。 第三个方法只用队列，竞争仅仅只是产生在将线程加入waitingThreads这个List的时候，这个add的操作粒度更加细，出现不公平的可能性降低了。

LittleClayBoy
2019/04/30 11:23上午

“但我们能改变这种情况。当前的Lock类版本调用自己的wait()方法，如果每个线程在不同的对象上调用wait()，那么只有一个线程会在该对象上调用wait()，Lock类可以决定哪个对象能对其调用notify()”

改为

“但我们能改变这种情况。当前的Lock类版本调用自己（一个对象）的wait()方法，如果每个线程在不同的对象上调用wait()，那样每个对象上只有一个线程调用wait，Lock类可以决定（这些对象中）哪个对象能对其调用notify()”，
会不会好理解些~

NeverWell
2019/07/06 9:01下午

第一个例子中的isLocked不用加volatile关键字么？？？

zx1254755805
2020/11/24 8:32下午

isLocked作为局部变量，它在代码里只在synchronized(当前对象)块中访问，就没有必要额外加可见性了