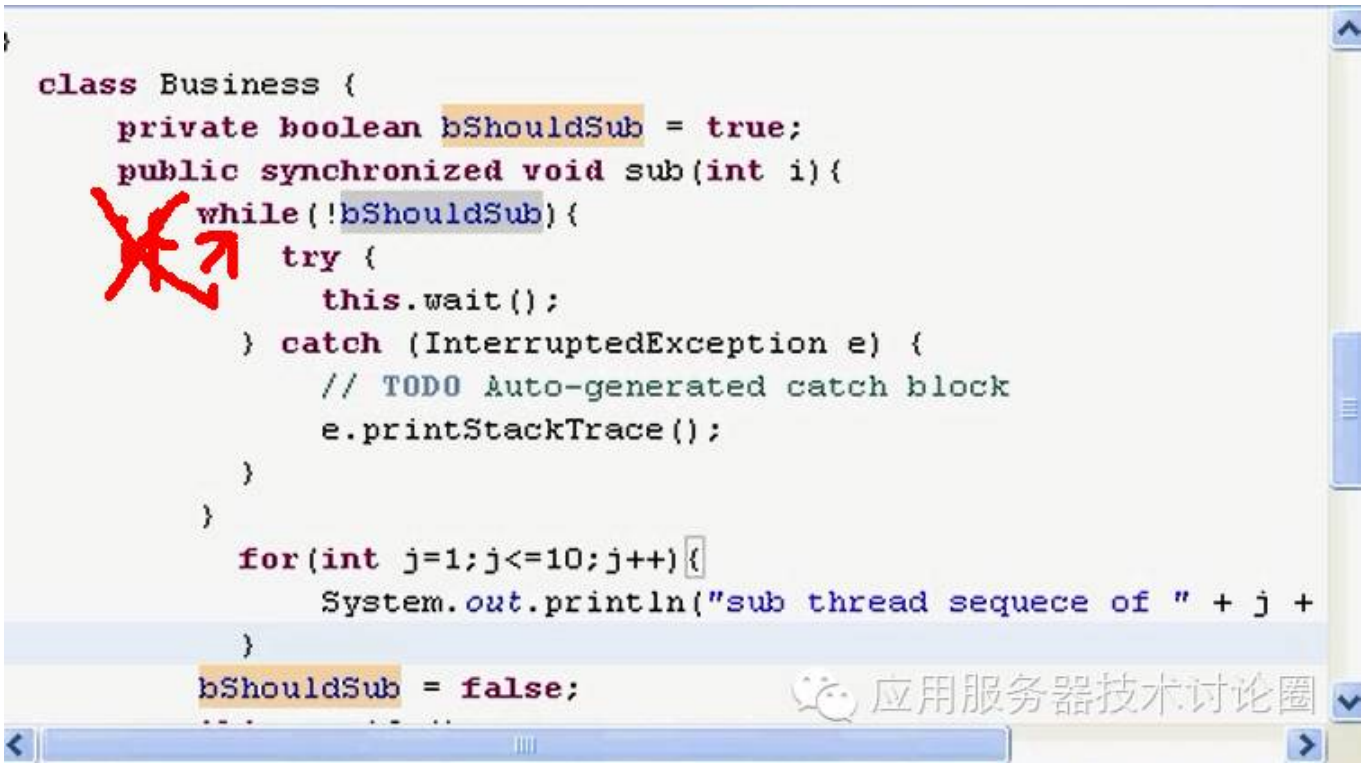
object.wait()为什么要套一个while循环?

2015-09-24 01:34 feiying 0 阅读 159

我们经常看到一段程序：



这段java程序为什么要套着一个while循环呢？

其实质的愿意是wait方法调用的是操作系统层面的pthread_cond_signal系统调用(基于Posix的线程接口，各个操作系统的实现)，

而在不同的操作系统的帮助手册中，都有关于这个虚假唤醒的定义

我们先来看看Linux中帮助中提到的，在linux的命令行中，执行man pthread_cond_signal就可以看到：

在多核处理器下，pthread_cond_signal可能会激活多于一个线程（阻塞在条件变量上的线程）。On a multi-processor, it may be impossible for an implementation of pthread_cond_signal() to avoid the unblocking of more than one thread blocked on a condition variable.

结果是，当一个线程调用pthread_cond_signal()后，多个调用pthread_cond_wait()或pthread_cond_timedwait()的线程返回。这种效应成为”虚假唤醒”(spurious wakeup) [4]

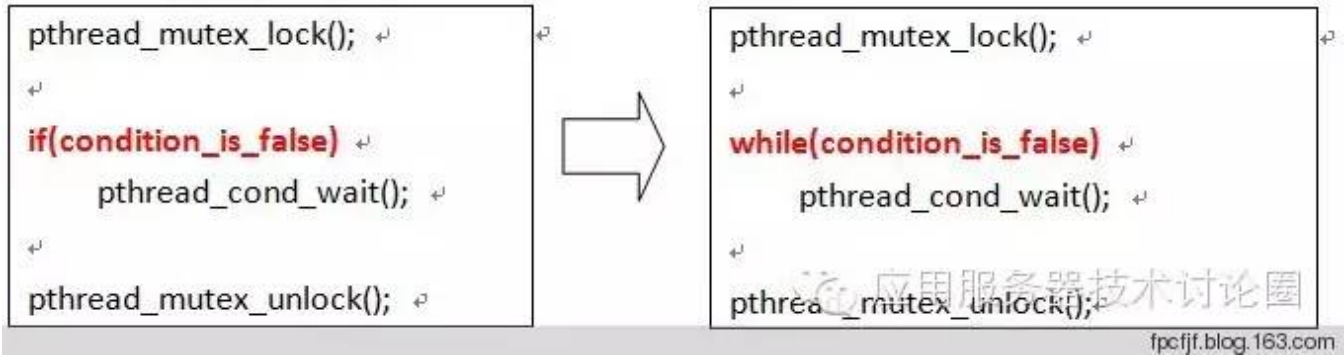
The effect is that more than one thread can return from its call to pthread_cond_wait() or pthread_cond_timedwait() as a result of one call to pthread_cond_signal(). This effect is called "spurious wakeup". Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call pthread_cond_wait() after the sequence of events above blocks.

虽然虚假唤醒在pthread_cond_wait函数中可以解决，为了发生概率很低的情况而降低边缘条件（fringe condition）效率是不值得的，纠正这个问题会降低对所有基于它的所有更高级的同步操作的并发度。所以pthread_cond_wait的实现上没有去解决它。

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

所以通常的标准解决办法是这样的：

将条件的判断从if 改为while



pthread_cond_wait中的while()不仅仅在等待条件变量前检查条件变量，实际上在等待条件变量后也检查条件变量。

这样对condition进行多做一次判断，即可避免“虚假唤醒”.

这就是为什么在pthread_cond_wait()前要加一个while循环来判断条件是否为假的原因。

有意思的是这个问题也存在几乎所有地方，包括: linux 条件等待的描述, POSIX Threads的描述, window API(condition variable), java等等。

在linux的帮助中对条件变量的描述是[4]：

添加while检查的做法被认为是增加了程序的健壮性，在IEEE Std 1003.1-2001中认为spurious wakeup是允许的。

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, IEEE Std 1003.1-2001 explicitly documents that spurious wakeups may occur.

? 在POSIX Threads中[5]:

David R. Butenhof 认为多核系统中 条件竞争（race condition [8]）导致了虚假唤醒的发生，并且认为完全消除虚假唤醒本质上会降低了条件变量的操作性能。

"..., but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare"

? 在window的条件变量中[6]:

MSDN帮助中描述为，spurious wakeups问题依然存在，条件需要重复check。

Condition variables are subject to spurious wakeups (those not associated with an explicit wake) and stolen wakeups (another thread manages to run before the woken thread). Therefore, you should recheck a predicate (typically in a while loop) after a sleep operation returns.

? 在Java中 [7]，对等待的写法如下：

```
synchronized (obj) {
 while (<condition does not hold>)
 obj.wait();

 ... // Perform action appropriate to condition
}
```

Effective java 曾经提到Item 50: Never invoke wait outside a loop.

显然，虚假唤醒是个问题,但它也是在JLS的第三版的JDK5的修订中才得以澄清。在JDK 5的Javadoc进行更新

A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops.

Apparently, the spurious wakeup is an issue (I doubt that it is a well known issue) that intermediate to expert developers know it can happen but it just has been clarified in JLS third edition which has been revised as part of JDK 5 development. The javadoc of wait method in JDK 5 has also been updated

==========》总结：

不仅仅是linux，windows也有这样的情况，因此作为操作系统平台上运行的应用程序JVM，必然也遵循这一个原则；做法就是加上一个while循环，再多执行一次判断，如果不是虚假循环的话，不会进while体中，这就仅仅损失一次判断而已，代价上确实划得来，而不至于从程序的角度上丢失了健壮性！

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

【知识宝库】十大个性数据网站。 (/html/301/201408/200554671/1.html)

作为iOS程序员，最核心的60%能力有哪些？ (/html/216/201608/2652547320/1.html)

如何当一个合格的黑客头目 (/html/278/201405/200250059/1.html)

全栈数据科学家の技能树 (/html/260/201605/2656697856/1.html)

软件架构和鸡窝 (/html/254/201607/2648945429/1.html)