Do spurious wakeups in Java actually happen?

Asked 15 years, 6 months ago Modified 4 years, 11 months ago Viewed 45k times

226

Seeing various locking related question and (almost) always finding the 'loop because of spurious wakeups' terms[1] I wonder, has anyone experienced such kind of a wakeup (assuming a decent hardware/software environment for example)?

I know the term 'spurious' means no apparent reason but what can be the reasons for such kind of an event?

([1] Note: I'm not questioning the looping practice.)

Edit: A helper question (for those who like code samples):

If I have the following program, and I run it:

```
public class Spurious {
    public static void main(String[] args) {
        Lock lock = new ReentrantLock();
        Condition cond = lock.newCondition();
        lock.lock();
        try {
            try {
                cond.await();
                System.out.println("Spurious wakeup!");
            } catch (InterruptedException ex) {
                System.out.println("Just a regular interrupt.");
            }
        } finally {
            lock.unlock();
        }
    }
}
```

What can I do to wake this `await` up spuriously without waiting forever for a random event?

java multithreading locking spurious-wakeup

Share
Improve this question
Follow

edited Apr 2, 2018 at 15:26
Flow
24k ● 15 ● 103 ● 157

asked Jun 26, 2009 at 18:42

akarnokd
69.9k ● 14 ● 160 ● 195

1 For JVMs that run on POSIX systems and use `pthread_cond_wait()` the real question is "Why does pthread_cond_wait have spurious wakeups?". – Flow Apr 2, 2018 at 15:25

7 Answers

Sorted by: Highest score (default)

219

The Wikipedia article on spurious wakeups has this tidbit:

> The `pthread_cond_wait()` function in Linux is implemented using the `futex` system call. Each blocking system call on Linux returns abruptly with `EINTR` when the process receives a signal. ... `pthread_cond_wait()` can't restart the waiting because it may miss a real wakeup in the little time it was outside the `futex` system call. This race condition can only be avoided by the caller checking for an invariant. A POSIX signal will therefore generate a spurious wakeup.

Summary: If a Linux process is signaled its waiting threads will each enjoy a nice, hot *spurious wakeup*.

I buy it. That's an easier pill to swallow than the typically vague "it's for performance" reason often given.

Share Improve this answer Follow

answered Jun 27, 2009 at 0:28

John Kugelman
361k ● 69 ● 546 ● 591

15 Better explanation here: stackoverflow.com/questions/1461913/… – Gili Sep 22, 2009 at 19:05

3 This EINTR unblocking is true of all blocking system calls in Unix derived systems. This made the kernel lots simpler, but the application programmers bought the burden. – Tim Williscroft Jul 25, 2011 at 1:10

2 I thought pthread_cond_wait() and friends could not return EINTR, but return zero if spuriously woken up? From: pubs.opengroup.org/onlinepubs/7908799/xsh/… "These functions will not return an error code of [EINTR]." – gub Aug 11, 2014 at 17:57

3 @jgubby That's right. The underlying `futex()` call returns `EINTR`, but that return value isn't bubbled up to the next level. The pthread caller must therefore check for an invariant. What they're saying is that when `pthread_cond_wait()` returns you must check your loop condition (invariant) again, because the wait might have been spuriously woken up. Receiving a signal during a system call is one possible cause, but it's not the only one. – John Kugelman Aug 11, 2014 at 18:03

1 Presumably, the `pthread` library could supply its own invariant and its own checking logic so as to eliminate spurious wakeups, rather than passing that responsibility onto the user.

That would (presumably) have the claimed performance impact. – user1084944 May 16, 2015 at 16:20

25

I have a production system that exhibits this behaviour. A thread waits on a signal that there is a message in the queue. In busy periods, up to 20% of the wakeups are spurious (ie when it wakes there is nothing in the queue). This thread is the only consumer of the messages. It runs on a Linux SLES-10 8-processor box and is built with GCC 4.1.2. The messages come from an external source and are processed asynchronously because there are problems if my system does not read them fast enough.

Share Improve this answer Follow

answered Sep 29, 2010 at 0:43

Mr.Dirty.Birdy
271 ● 3 ● 4

19

To answer the question in the titile - **Yes!** it does happen.Though the Wiki article mentions a good deal about spurious wakeups a nice explanation for the same that I came across is as follows -

> Just think of it... like any code, thread scheduler may experience temporary blackout due to something abnormal happening in underlying hardware / software. Of course, care should be taken for this to happen as rare as possible, but since there's no such thing as 100% robust software it is reasonable to assume this can happen and take care on the graceful recovery in case if scheduler detects this (eg by observing missing heartbeats).
>
> Now, how could scheduler recover, taking into account that during blackout it could miss some signals intended to notify waiting threads? If scheduler does nothing, mentioned "unlucky" threads will just hang, waiting forever - to avoid this, scheduler would simply send a signal to all the waiting threads.
>
> This makes it necessary to establish a "contract" that waiting thread can be notified without a reason. To be precise, there would be a reason - scheduler blackout - but since thread is designed (for a good reason) to be oblivious to scheduler internal implementation details, this reason is likely better to present as "spurious".

I was reading this answer from Source and found it reasonable enough. Also read

Spurious wakeups in Java and how to avoid them.

PS: Above link is to my personal blog that has additional details about spurious wakeups.

Share
Improve this answer
Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 • 1

answered Aug 2, 2014 at 15:56

Aniket Thakur
68.8k • 40 • 287 • 296

8

Just to add this. Yes it happens and I spent three days searching for the cause of a multi-threading problem on a 24 core machine (JDK 6). 4 of 10 executions experienced that without any pattern. This never happened on 2 core or 8 cores.

Studied some online material and this is not a Java problem but a general rare but expected behavior.

Share Improve this answer Follow

answered May 5, 2011 at 12:42

ReneS
3,535 • 2 • 28 • 35

Hello ReneS, are(were) you developing the app running there? Does(did) it have wait() method calling in while loop checking external condition as it is suggested in java doc docs.oracle.com/javase/6/docs/api/java/lang/... ? – humkins Mar 18, 2015 at 21:45

I wrote about it and yes the solution is a while loop with a condition check. My mistake was the missing loop... but so I learnt about these wakeups... never on two cores, often on 24cores blog.xceptance.com/2011/05/06/spurious-wakeup-the-rare-event – ReneS Mar 18, 2015 at 22:14

I had similar experiences when I ran an application on a 40+ core unix server. It had an extreme amount of spurious wakeups. - So, it does seem like the amount of spurious wakeups is directly proportional to the amount of processor cores of the system. – bvdb Feb 11, 2019 at 9:24

8

Cameron Purdy wrote a blog post a while back about being hit by the spurious wakeup problem. So yes, it happens

I'm guessing it's in the spec (as a possibility) because of limitations of some of the platforms which Java gets deployed on? although I may be wrong!

Share
Improve this answer
Follow

edited Jul 10, 2014 at 14:30

Nathan Hughes
96.3k • 20 • 191 • 282

answered Jun 26, 2009 at 19:03

oxbow_lakes
134k • 56 • 320 • 450

I read the post and gave me an idea about having unit tests for testing one application's conformance to the looping-wait paradigm by waking it up randomly/deterministically. Or is it already available somewhere? – akarnokd Jun 26, 2009 at 19:17

It's another question on SO: "Is there a *strict* VM that can be used for testing?". I'd love to see one with strict thread-local memory - I don't think they exist yet – oxbow_lakes Jun 26, 2009 at 20:01

1 as of 2022, those links redirect me to a casino – julaine Dec 15, 2022 at 14:12

3

Answering the OP's question

> What can I do to wake this await up spuriously without waiting forever for a random event?

, **no any spurious wakeup** could wake up this awaiting thread!

Regardless of whether spurious wakeups can or cannot happen on a particular platform, in a case of the OP's snippet it is positively **impossible** for `Condition.await()` to return and to see the line "Spurious wakeup!" in the output stream.

Unless you are using very exotic Java Class Library

This is because standard, OpenJDK's `ReentrantLock`'s method `newCondition()` returns the `AbstractQueuedSynchronizer`'s implementation of `Condition` interface, nested `ConditionObject` (by the way, it is the only implementation of `Condition` interface in this class library), and the `ConditionObject`'s method `await()` itself checks whether the condition does not holds and no any spurious wakeup could force this method to mistakenly return.

By the the way, you could check it yourself as it is pretty easy to emulate spurious wakeup once the `AbstractQueuedSynchronizer`-based implementation is involved. `AbstractQueuedSynchronizer` uses low-level `LockSupport`'s `park` and `unpark` methods, and if you invoke `LockSupport.unpark` on a thread awaiting on `Condition`, this action cannot be distinguished from a spurious wakeup.

Slightly refactoring the OP's snippet,

```
public class Spurious {

    private static class AwaitingThread extends Thread {

        @Override
        public void run() {
            Lock lock = new ReentrantLock();
            Condition cond = lock.newCondition();
            lock.lock();
            try {
                try {
                    cond.await();
```

```
                    System.out.println("Spurious wakeup!");
                } catch (InterruptedException ex) {
                    System.out.println("Just a regular interrupt.");
                }
            } finally {
                lock.unlock();
            }
        }
    }

    private static final int AMOUNT_OF_SPURIOUS_WAKEUPS = 10;

    public static void main(String[] args) throws InterruptedException {
        Thread awaitingThread = new AwaitingThread();
        awaitingThread.start();
        Thread.sleep(10000);
        for(int i =0 ; i < AMOUNT_OF_SPURIOUS_WAKEUPS; i++)
            LockSupport.unpark(awaitingThread);
        Thread.sleep(10000);
        if (awaitingThread.isAlive())
            System.out.println("Even after " + AMOUNT_OF_SPURIOUS_WAKEUPS + "
\"spurious wakeups\" the Condition is stil awaiting");
        else
            System.out.println("You are using very unusual implementation of
java.util.concurrent.locks.Condition");
    }
}
```

, and no matter how hard the unparking(main) thread would try to awake the awaiting thread, the `Condition.await()` method will never return in this case.

The spurious wakeups on `Condition`'s awaiting methods are discussed in the javadoc of `Condition` interface . Although it does say that,

> when waiting upon a Condition, a spurious wakeup is permitted to occur

and that

> it is recommended that applications programmers always assume that they can occur and so always wait in a loop.

but it later adds that

> An implementation is free to remove the possibility of spurious wakeups

and `AbstractQueuedSynchronizer`'s implementation of `Condition` interface does exactly that - **removes any possibility of spurious wakeups**.

This surely holds true for other `ConditionObject`'s awaiting methods.

So, the **conclusion** is :

we should always call `Condition.await` in the loop and check if the condition does not hold, but with standard, OpenJDK, Java Class Library is **can never happen**. Unless, again, you use very unusual Java Class Library (which must be very very unusual, because another well-known non-OpenJDK Java Class Libraries, currently almost extinct GNU Classpath and Apache Harmony, seems to have identical to standard implementation of `Condition` interface)

Share Improve this answer Follow

answered Jan 23, 2020 at 22:30

igor.zh
2,223 ● 22 ● 24

0

https://stackoverflow.com/a/1461956/14731 contains an excellent explanation of why you need to guard against of spurious wakeups even if the underlying operating system does not trigger them. It is interesting to note that this explanation applies across multiple programming languages, including Java.

Share Improve this answer Follow

answered Jul 3, 2018 at 17:52

Gili
89.7k ● 104 ● 410 ● 720

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.