

## Re: notify / notifyAll misunderstanding

**Source:** <http://coding.derkeiler.com/Archive/Java/comp.lang.java.programmer/2004-07/1562.html>

---

**From:** John C. Bollinger ([jobollin\\_at\\_indiana.edu](mailto:jobollin_at_indiana.edu))

**Date:** 07/13/04

Date: Tue, 13 Jul 2004 12:08:25 -0500

VisionSet wrote:

- > *As I understand it, notify wakes up a waiting thread that then attempts to*
- > *gain lock the object it is waiting for.*

That is correct. The thread that is awakened will return from wait() and resume execution when it re-obtains all the monitors it held before invoking wait().

- > *Where as notifyAll wakes up all*
- > *waiting threads and ONLY ONE gains the lock, the others go back to waiting.*

That is incorrect. notifyAll() wakes up all waiting threads, all initially blocked on monitor access. From each individual thread's point of view, it is no different from being the lucky thread chosen by notify(). One at a time, in no particular order, each will gain the lock and resume executing. (In your code below, most such threads will indeed ultimately resume waiting — but because they loop back around to invoke wait() again, not because of the semantics of notifyAll().)

- > *The following code does not seem to show this.*

Unsurprising, as your expected behavior is not the specified one.

- > *I have a record locking system that I have condensed to the simple example*
- > *below. Record locks are placed on Integers that are managed by a Set. When*
- > *lock is called then the record is put in the set if not already there. If*
- > *the record is already there then wait() is called.*
- > *When unlock() is called a record is removed from the set and notifyAll() is*
- > *called.*
- >
- > *In the example I have 2 threads that lock on record 5, and 8 that lock on*
- > *record 6.*
- > *So after the 10 lock threads run, I will have 1 thread locked 5, 1 thread*
- > *locked 6, 1 thread is waiting to lock 5, and 7 threads are waiting to lock*
- > *6.*
- > *Then a single thread unlocks record 5 and notifyAll is called.*
- > *So 8 waiting threads are then all notified and 1 arbitrarily gets the lock.*

comp.lang.java.programmer: Re: notify / notifyAll misunderstanding

- > *If it is the 1 waiting to lock 5 then it will succeed since 5 has just been*
- > *locked. But more likely it will be one of the 7 waiting to lock 6, which*
- > *will be unsuccessful because 6 hasn't been unlocked.*
- >
- > *SO! Why does it always succeed?*

Because all the waiting threads are awakened and eventually run, including the one waiting to lock record 5.

- > *With notifyAll() I always get:*
- >
- > *locked 5*
- > *locked 6*
- > *unlocked 5*
- > *locked 5*
- >
- > *notify() on the other hand works as expected ie:*
- >
- > *locked 5*
- > *locked 6*
- > *unlocked 5*
- >
- > *but probably for the wrong reason since my understanding is obviously*
- > *flawed.*

As an aside: isn't the behavior you actually get more what you want than the behavior you expect? I.e. if a lock for a particular record is released and one or more threads is waiting for that lock, then don't you want one of the waiting threads to reliably get the lock?

- > *import java.io.\*;*
- > *import java.util.\*;*
- >
- > *public class Test {*
- >
- > *private Set lockSet = new HashSet();*
- >
- > *public void lock(int recNo) {*
- > *Integer key = new Integer(recNo);*
- > *synchronized(lockSet) {*
- > *while(lockSet.contains(key)) {*
- > *try{*
- > *lockSet.wait();*

This method, in every thread, becomes eligible to return when lockSet.notifyAll is invoked. It will in fact return when the thread obtains lockSet's monitor, independent of what any other thread does.

- > *}*
- > *catch(InterruptedException ex) {}*
- > *}*

For each key that has been unlocked, the first thread that obtains lockSet's monitor after the unlock will drop out of the while loop here, add the key back into the lockSet, and then continue doing whatever else it is supposed to do. Other threads that obtain lockSet's monitor later will see the key in lockSet and loop around to wait() again. I admit to a bit of bemusement, because this appears to be a perfectly workable code that does what I would think you would want.

```
> lockSet.add(key);
> System.out.println("locked "+recNo);
> }
> }
>
> public void unlock(int recNo) {
>
> try {Thread.sleep(500);}
> catch(InterruptedException ex) {}
>
> Integer key = new Integer(recNo);
> synchronized(lockSet) {
> lockSet.remove(key);
> System.out.println("unlocked "+recNo);
> lockSet.notifyAll();
> }
> }
>
>
> public static void main(String[] args) throws Exception {
>
> Test test = new Test();
> test.testLocking();
>
> }
>
> private void testLocking() {
>
> lockIt(5);
>
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
> lockIt(6);
>
> lockIt(5);
>
> unlockIt(5);
>
```

```
> }  
>  
> // these just fire up threads for lock & unlock  
> private void lockIt(final int x) {  
>     new Thread(new Runnable() {  
>         public void run() {  
>             try {  
>                 lock(x);  
>             }  
>             catch(Exception e) {e.printStackTrace();}  
>         }  
>     }).start();  
> }  
> private void unlockIt(final int x) {  
>     new Thread(new Runnable() {  
>         public void run() {  
>             try {  
>                 unlock(x);  
>             }  
>             catch(Exception e) {e.printStackTrace();}  
>         }  
>     }).start();  
> }  
> }
```

If you anticipated having many threads locking on many different records then it might improve your performance to lock on the individual keys instead of on the whole lockSet. It would be a bit more complicated, but fundamentally it would work about the same. For relatively few threads, relatively few records, or relatively few expected lock contentions it's probably not worth it to introduce the extra bookkeeping.

John Bollinger  
jobollin@indiana.edu