

Re: WaitForSingleObject() will not deadlock

## Re: WaitForSingleObject() will not deadlock

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-07/msg00511.html>

---

- *From:* Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)>
  - *Date:* Thu, 05 Jul 2007 23:24:32 -0400
- 

See below...

On Thu, 05 Jul 2007 17:13:25 -0500, "Doug Harrison [MVP]" <[dsh@xxxxxxxxx](mailto:dsh@xxxxxxxxx)> wrote:

On Thu, 05 Jul 2007 17:02:44 -0400, Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)> wrote:

But see below...

Yes. Please read carefully. *\*Very\** carefully. <g>

On Wed, 04 Jul 2007 23:44:02 -0500, "Doug Harrison [MVP]" <[dsh@xxxxxxxxx](mailto:dsh@xxxxxxxxx)> wrote:

On Wed, 04 Jul 2007 23:33:41 -0400, Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)> wrote:

"Programming with POSIX  
Threads", page 89

<q>

1. Whatever memory values  
a thread can see when it  
creates a new thread can  
also be seen by the new  
thread once it starts. Any  
data written to memory  
after the new thread is  
created may not necessarily  
be seen by the new  
thread, even if the write  
occurs before the thread  
starts.

\*\*\*\*\*

## Re: WaitForSingleObject() will not deadlock

Seems an odd specification. Essentially, it means that threads cannot share variables when they are running.

\*\*\*\*\*

It makes sense. Here's an illustration of what it's saying:

```
int x = 0; // global
```

```
***** Thread 1:
```

```
x = 2;  
CreateThread;  
// x = 3;
```

```
***** Thread 2:
```

```
if (x == 2)  
puts("x == 2");
```

Rule (1) guarantees that when started, thread 2 observes `x == 2`, since

thread 1 set it to 2 before creating the second thread.

However, if you

were to uncomment the line that sets `x` to 3, thread 2 might not see that value, even if it were executed before thread 2 got around to testing it.

That is, thread 2 might still observe the value 2, even though from thread

1's perspective, `x` contains 3, at the moment thread 2 tested it.

\*\*\*\*

Since this code is erroneous, there seems to be little point to worrying about whether

thread 2 sees `x==2` or `x==3`. There is a race condition here, and there is no way the value of `x` is guaranteed.

Come on, Joe. Did you fail to notice the third line in this sequence is a comment?

\*\*\*\*\*

Yes, but you also said that there was an issue if the line was uncommented. Since with the line commented out, and assuming there is no other gratuitous assignment to `x`, then it is completely deterministic

\*\*\*\*\*



## Re: WaitForSingleObject() will not deadlock

which is the counterpart to my line:

```
// x = 3;
```

IOW, you're talking about what I talked about after I said, "However, if you were to uncomment the line that sets x to 3."

\*\*\*\*\*

Yes, but the point is that if there is no change in the data after the thread is started, there is no reason to expect the thread would see anything other than the assignment before the thread started. If there is an assignment after the thread starts, the program is essentially erroneous, and locking will not make it correct

\*\*\*\*\*

No amount of synchronization is going to change the fact that this program is erroneous.

No rule about "memory visibility" is going to change the fact that this program is erroneous.

No rules about cache coherency are going to change the fact that this program is erroneous

I know it's erroneous. That was the point of my sentence that began, "However, if you were to uncomment the line that sets x to 3," which goes on to explain what happens if you were to uncomment the line.

In summary, I presented an example that demonstrated what Rule (1) guarantees and what it doesn't guarantee. I hope it's clear now.

2. Whatever memory values a thread can see when it unlocks a mutex (leaves a synchronized method or block in Java), either directly or by waiting on a condition variable (calling wait in Java), can also be seen by any thread that later locks the same mutex. Again, data written after the mutex is unlocked may not necessarily be seen by the thread that locks the mutex,

Re: WaitForSingleObject() will not deadlock

even if the write occurs  
before the lock.

\*\*\*\*\*

I find this truly unbelievable. How can a mutex know what values were accessed during the thread, so that it can ensure the values are going to be consistent if that same mutex is locked? This strikes me as requiring immensley complicated bookkeeping on the part of the mutex implementation. It seems so much easier to follow the semantics of most hardware and just make sure that locking guarantees all pipes and caches are coherent across all processors.

It makes sense. It's a high-level, abstract description of something you're thinking about at the hardware level and taking very literally. How else would you describe this in terms of mutexes? You wouldn't expect things to work right if you used two mutexes pell-mell to protect the same piece of data. No, you have to lock the same mutex, and (2) reflects that.

(The stuff about Java doesn't appear in the book. I guess the guy who wrote the message I copied the excerpt from added it.)

While I first find it hard to imagine how it is possible to create a situation of this nature on any closely-coupled MIMD architecture, I can imagine how it can exist in a distributed MIMD architecture, but in that case, the mutice have to keep track of every variable that is accessed within their scope, and ensure that an attempt to lock a mutex can ensure that all remotely-cached data is sent back and all locally-cached data is distributed out. An awesome task.

Re: WaitForSingleObject() will not deadlock

Of course that's not what it's saying. The requirement (2) can be fulfilled in much simpler though coarser ways than you're thinking. Also keep in mind that Butenhof was an architect of pthreads, a high-level, portable, standardized multithreading library designed to run efficiently on existing hardware, and he's not going to write about impossible requirements.

\*\*\*\*\*

Then why doesn't he write what he means, instead of writing something so clumsy that it is subject to misinterpretation

Because as I explained, he's writing at a certain high level of abstraction at this point, and as Martha would say, "It's a Good Thing". I'm sure you'd be happy to learn that p. 91 starts off:

<q>

"Warning! We are now descending below the Pthreads API into details of hardware memory architecture that you may prefer not to know. You may want to skip this explanation for now and come back later."

</q>

Then he talks on pages 91–95 about cache coherency, memory barriers, and all that yucky stuff you don't need to know to use mutexes correctly, as long as you follow the memory visibility rules he presented on page 89. And with that, I'm gonna have to call it a day here.

\*\*\*\*\*

Lacking the ability to see the pages, it was not obvious what was on them.

joe

\*\*\*\*\*

Joseph M. Newcomer [MVP]

email: [newcomer@xxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxx)

Web: <http://www.flounder.com>

MVP Tips: [http://www.flounder.com/mvp\\_tips.htm](http://www.flounder.com/mvp_tips.htm)

.