

# Re: WaitForSingleObject() will not deadlock

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-07/msg00406.html>

---

- *From:* "Doug Harrison [MVP]" <dsh@xxxxxxx>
  - *Date:* Wed, 04 Jul 2007 23:44:02 -0500
- 

On Wed, 04 Jul 2007 23:33:41 -0400, Joseph M. Newcomer <newcomer@xxxxxxxxxxxx> 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.

\*\*\*\*\*

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.

## Re: WaitForSingleObject() will not deadlock

The importance of (1) should be apparent when you imagine x is a "this" pointer, a CString pointer, etc that you are passing to the thread through the CreateThread LPVOID parameter. You certainly expect the new thread to observe the same data (what's pointed to) the existing thread observed when it called CreateThread.

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, 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 mutices 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.

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.

3. Whatever memory values a thread can see when it terminates, either by cancellation, returning from its run method, or exiting, can also be seen by the thread that joins with the terminated thread by calling join on that

## Re: WaitForSingleObject() will not deadlock

thread. And, of course, data written after the thread terminates may not necessarily be seen by the thread that joins, even if the write occurs before the join.

\*\*\*\*\*

This implies that the notion of "join" exists as a fundamental concept. Most operating systems that have threads do not seem to have this concept any longer; it seems to have been more of a high-level concept when threads were implemented above the operating system.

I wonder what bizarre era of programming this specification represents.

I don't know what you mean. Windows has WaitForSingleObject(hThread) which is "join" by another name, and "join" exists by that name in .NET. I would consider any multithreading model that doesn't provide a "join" facility as fundamentally broken. To understand what (3) is saying, you can sort of reverse my example for (1). I've talked many times about another important use for "join", which is to "collect" all secondary threads, which is often necessary to perform an orderly program shutdown.

No surprises there. The MSDN link indicates this applies to other sync objects plus CRITICAL\_SECTION. It's nice for it to be stated, but for all the reasons we've talked about, it couldn't be any other way. For Windows, I expect you could add:

5. Whatever memory values a thread can see when it calls PostMessage or SendMessage can also be seen by the thread that retrieves or processes the message.

\*\*\*\*\*

Essentially, in Windows, any value in a memory location that can be seen by one thread can be seen by any other thread at any time for any reason.

But absent synchronization of some form, whether or not two threads observe the same value depends on the hardware. I'm postulating this is done in rule (5); if it weren't, object hand-off protocols wouldn't work on hardware such as IA64.

So the only issue is whether or not the COMPILER has done something that keeps some local cache somewhere.

That's a separate issue.

The conservative optimizer of Microsoft C ensures this by essentially implementing "volatile" semantics against any variable that is potentially modifiable by an arbitrary function

Re: WaitForSingleObject() will not deadlock

call. A C compiler is not required to do this, and it can still be a conforming C compiler.

Don't mix this up with "volatile". To revisit my example from a couple of messages ago, if a C compiler could look into the mutex lock/unlock operations and see that they don't modify a global variable that is intended to be protected by the critical section, allowing it to perform optimizations that are at odds with the attempt at multithreaded programming, yes, it would be a "conforming C compiler". So what? The C Standard doesn't address multithreading, and such a compiler would be useless for multithreaded programming. It would be the reincarnation of ancient compilers unsuitable for multithreaded programming you described a couple of messages ago.

—

Doug Harrison  
Visual C++ MVP

.