

Re: WaitForSingleObject() will not deadlock

Re: WaitForSingleObject() will not deadlock

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-07/msg00604.html>

- *From:* "Doug Harrison [MVP]" <dsh@xxxxxxx>
 - *Date:* Sat, 07 Jul 2007 14:05:46 -0500
-

On Sat, 07 Jul 2007 13:21:49 -0400, Joseph M. Newcomer <newcomer@xxxxxxxxxxxx> wrote:

On Thu, 05 Jul 2007 22:20:56 -0500, "Doug Harrison [MVP]" <dsh@xxxxxxx> wrote:

On Thu, 05 Jul 2007 17:02:44 -0400, Joseph M. Newcomer <newcomer@xxxxxxxxxxxx> wrote:

So the question is, what compromises are made to allow C to work in a multithreaded environment? One is to hijack the semantics of volatile to disable compiler optimizations that would not be thread-safe, and otherwise let the compiler to aggressive optimization.

That's an incredibly undesirable approach for a number of reasons. AFAIK, no one approaches the problem in that way.

Did you see the C++ Standards Committee discussion I pointed out?

No, I missed it. Please post it again, cite the relevant sections, and tell me what you think they mean.

They are in fact considering that.

No, they are not, at least not in the way you think.

We were certainly doing this 35 years ago, so the technology is well-understood. I worked with optimizing compilers between 1969 and 1983, and these were

Re: WaitForSingleObject() will not deadlock

issues we were constantly discussing and worrying about. We designed languages that had explicit specifications of concurrency, and although the word 'volatile' was not one we used, the issues raised by volatile seem indistinguishable from the issues we were addressing.

Why is this undesirable?

It's undesirable to require volatile in addition to mutex lock/unlock, which let's be clear, is what you're proposing, for at least the following reasons:

1. The compiler would not be free to ignore volatile inside a critical section; therefore, requiring volatile would slow down access inside the critical section, but the vast majority of the time, there would be no reason to respect volatile inside the critical section. Instead, the compiler should recognize the operations that define the critical section, e.g. mutex lock/unlock, and then you don't have to "hijack volatile". You don't have to use it at all. That, in effect, is what POSIX did.
2. Classes such as `std::vector` are written to be used in a single-threaded environment. We can use them in a thread-safe way without touching their implementation by using mutexes to protect their operations that are not thread-safe. If it were required to also declare vector objects volatile, then:
 - 2a. You could not call a member function without casting volatile away, because none of the member functions are declared volatile. Besides being ridiculous, this would be undefined for an object originally declared volatile.
 - 2b. Declaring an object `x` volatile doesn't make pointer and reference members of `x` refer to volatile objects.

The net effect of 2a and 2b is to invalidate current practice and require classes to come in two versions, one that uses volatile internally and one that does not. Among other things, this is what one of the guys (Hans J. Boehm) working on the multithreaded memory model for C++ says here:

A Memory Model for C++: FAQ

http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/faq.html

<q>

Why cannot the compiler optimization issues just be side-stepped by declaring the relevant shared variables volatile?

....

this turns out to be completely impractical. It is very common to "wrap" single-threaded code in a lock to make it usable in a multithreaded application. This would not be possible if all variables/fields in the single-threaded code now had to be declared volatile. Thread-safe versions of the C++ standard library, effectively rely on this approach, which we believe to be the only viable one. It is also similar to the one adopted by

Re: WaitForSingleObject() will not deadlock

more recent Java container libraries, for example.

Requiring volatile declarations for lock-protected variables would effectively require most libraries to come in two versions: A standard version, and one in which all internal static variables were declared volatile.

....

</q>

FWIW, this is stuff I was telling you years ago. If you just can't accept it, I implore you, drop by [comp.programming.threads](#), [comp.lang.c++.moderated](#), and [comp.std.c++](#) and tell everyone how they should be doing things. Maybe they can explain these things better than me.

I haven't read the remainder of your post, and after skimming it, I may or may not get to it. I really think our time would be better spent if there were a wider audience to respond to your viewpoints.

--

Doug Harrison
Visual C++ MVP

.