

Re: Share .cpp and .h along projects

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2007-08/msg00741.html>

- *From:* "Doug Harrison [MVP]" <dsh@xxxxxxx>
 - *Date:* Tue, 21 Aug 2007 11:48:07 -0500
-

On Mon, 20 Aug 2007 17:16:11 -0500, "Ben Voigt [C++ MVP]" <rbv@xxxxxxxxxxxx> wrote:

Common sense dictates using the highest-level abstraction available unless there's a good, specific reason to do otherwise. If, when someone says "mutex", as I repeatedly did, you think "InterlockedXXX", well, it's just hard to understand why.

I agree. You asked me to show how a volatile pointer could be used to protect a C++ class object and I did so. I didn't say that it was the best, most readable, or anything like that.

To be fair, you introduced it into the conversation. My purpose in asking you to expand on what you meant by:

any larger object can be controlled in a threadsafe manner using a volatile pointer (which is word-sized) and memory barriers.

was to learn more about what you were claiming for "volatile". (That's why I also said to imagine uniprocessor or SMP x86 and forget about MBs.) I was surprised by the InterlockedXXX reply, because no one would use that method as an alternative to mutexes (except perhaps on an exceedingly rare basis), which is what I had been referring to for a couple of messages. I was expecting an answer in terms of mutexes.

BTW, why `_exactly_` did you use volatile in your declaration of `g_sharedVector`? (Based on the declaration of `InterlockedExchangePointerAcquire`, it shouldn't even compile.)

Re: Share .cpp and .h along projects

No answer? I really would like to hear what you think volatile accomplishes here.

Well, there is no "InterlockedExchangePointerAcquire".

Well, it is what you used in your example. :)

It's also documented in MSDN:

<http://msdn2.microsoft.com/en-us/library/ms683611.aspx>

There's an "InterlockedExchangePointer"

That's fine, too. The signatures are the same, so it doesn't affect my question.

which does declare the parameter as a pointer to a volatile (and yes, a void*, so g_sharedVector could either be made a volatile void* or a cast could be used).

No, if your pointer p has the type:

volatile void*

then based on the declaration of InterlockedExchangePointer, &p will require a cast, too. I was hoping that hinting at the declaration problem would get you to talk about what you think volatile actually does in the example you posted WRT to what you said earlier:

any larger object can be controlled in a threadsafe manner using a volatile pointer (which is word-sized) and memory barriers.

(Again, scratch "memory barriers" and talk about uniprocessor or x86 SMP.)

If f2 is not dllexport, and its address is not taken (and it isn't reachable from any function that has an address taken — this is going to save you because it is reachable from some ThreadProc which has an address taken and passed to CreateThread, ... unless of course f2 happens to be called only from the main thread).

Re: Share .cpp and .h along projects

Note that it isn't enough to recognize just `CreateThread`. It has to recognize `MyCreateThread` in my opaque DLL, but I have no way to tell it `MyCreateThread` is a "thread creation" function. And that's just the tip of the iceberg.

So, what if `f2` is `WinMain` (and has appropriate `WinMain` behavior above and below the mutex access)? I'm pretty sure that if we work at this long enough, the compiler is going to determine through aliasing analysis that neither `lock` nor `unlock` nor `g` change `x`, and that it is safe to optimize away the second access in `f1`.

In a nutshell, the compiler would have to find all the functions that modify `x` and generate their call graphs. Then it would have to recognize all the ways for functions appearing in those graphs to become callable directly or indirectly from `lock/unlock`. Any of those functions whose address is passed to some opaque function kills the optimization, because the opaque function could potentially save the pointer for later use by `lock/unlock`. So for example, if `f2` is called directly or indirectly by a Windows message handler, the optimization must be killed since `f2` is reachable from the `WndProc`, whose address is passed to an opaque Windows API. This is all very time-consuming and will usually result in a dead-end. It's a lot easier to assume globals are reachable from opaque functions, and AFAIK, that's what the compiler does. It gives the desired semantics for multithreaded programming with synchronization, and it's what I've observed by constructing the simplest possible example for it to try to optimize. It doesn't optimize it.

Sorry, the correct declaration of `g_sharedVector` is:
`volatile void* g_sharedVector;`
or else a cast is needed to make it compatible with `InterlockedCompareExchangePointer`.

Again, that doesn't help at all. You would still need a cast when you say `&g_sharedVector`, because you put the `volatile` in the wrong place. So what is `volatile` supposed to accomplish in your example? How does your example demonstrate what you said:

any larger object can be controlled in a threadsafe manner using a volatile pointer (which is word-sized) and memory barriers.

What do you think your use of `volatile` accomplishes in your example?

Re: Share .cpp and .h along projects

Actually you need a cast either on either the parameter or the return value, but that's normal for InterlockedCompareExchangePointer. Typically I would write a templated wrapper to hide the cast and ensure that the input and return value have matching type.

Why exactly did I put volatile in the declaration, when the compiler allows it to be implicitly added? For the same reason that I declare immutable variables with const -- to prevent them from being used in an inappropriate context.

But you asserted that using volatile is necessary to prevent the compiler from performing unsafe optimizations. I'd like you to explain how it does that in your example. About the "inappropriate context", see below.

Anyway, in your example, what do you think would be the performance hit of declaring x as volatile if, as you seem to think, the compiler cannot optimize access to x because of the presence of function calls.

Not just "function calls", but "opaque function calls". Using the pre-VC2005 volatile means the compiler will never cache the value, so it goes to memory for every access. The VC2005 and later volatile adds memory barrier overhead to this, which is even worse. Both represent unnecessary overhead inside a critical section, where access to that variable is single-threaded and should be subject to the usual optimizations. In addition, make x a volatile std::vector or just about any class, and you have to cast volatile away to use it, because no one declares their member functions volatile. Besides being a pain, casting volatile away from an object originally declared volatile and referring to the object through the non-volatile lvalue is undefined.

Then there's this:

```
struct X
{
int* const p;
};

Mutex mx;
volatile X x = { new int(0) };

void f()
{
int* p = x.p; // Fine

lock(mx);
// use *p
unlock(mx);
```

Re: Share .cpp and .h along projects

```
lock(mx)
// use *p
unlock(mx);
}
```

```
void f2()
{
lock(mx)
++*x.p;
unlock(mx);
}
```

Making x volatile doesn't help, because volatile doesn't penetrate very far at all. That's the sort of thing I was getting at several messages ago. The compiler won't detect this "usage in an inappropriate context", namely initializing p with x.p, which is another reason it's a bad idea to try to hijack volatile for multithreading purposes.

--
Doug Harrison
Visual C++ MVP

.