

Re: Is the following code MT-Safe?

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2004-07/0126.html>

From: Ken Durden (*creepiecrawlies_at_hotmail.com*)

Date: 07/01/04

Date: 1 Jul 2004 05:50:37 -0700

Joseph M. Newcomer <newcomer@flounder.com> wrote in message
news:<3jv5e0h0hb1m2ma6cr5ddbq3hfk4dfqd@4ax.com>...

```
>
>>
>> void start()
>> {
>> CSingleLock( &crit, TRUE );
>>
>> AfxBeginThread(ThreadFunc, (LPVOID) this, ... );
> ****
> There is no reason to lock the critical section before starting the thread. I see nothing
> here that accesses the variables of the thread within the scope of the critical section.
> ****
>> }
```

The critical section prevents clients in multiple threads from calling start(), stop(), and test() from stepping over each other's feet.

Here's a more complete implementation of start() is re-entrantly safe to allow for multiple clients to call start() w/o any race conditions to allow unintended creation of multiple internal threads.

```
void start()
{
    CSingleLock( &crit, TRUE );

    if( !_bRunning )
    {
        _startedEvent.ResetEvent();

        AfxBeginThread(ThreadFunc, (LPVOID) this, ... );
        ::WaitForSingleObject( _startedEvent, INFINITE );
    }
}

>>
>> void stop()
>> {
```

microsoft.public.vc.mfc: Re: Is the following code MT-Safe?

```
> > CSingleLock( &crit, TRUE );
> >
> > _stoppedEvent.Reset();
> > _stopEvent.Set();
> >
> > ::WaitForSingleObject( _stoppedEvent, INFINITE );
> *****
> This is actually dangerous. There is no reason to set the critical section, since nothing
> in here manipulates shared variables. And the lock won't be released until this exits.
> But the bug is deeper; see the comment below.
> *****
```

Again, my mistake I suppose. I was trying to have a simplified demo version of this problem. Here's a more complete "production" version of stop(). I think this version makes it clear why the lock is needed; to avoid issues where calls to stop() from multiple threads cause deadlocks when both threads get inside the if() conditional and attempt to wait on the thread.

```
void stop()
{
    CSingleLock( &crit, TRUE );

    if( _bRunning )
    {
        _stoppedEvent.Reset();
        _stopEvent.Set();

        ::WaitForSingleObject( _stoppedEvent, INFINITE );
    }
}

> > }
> >
> > void test()
> > {
> > CSingleLock( &crit, TRUE );
> >
> > if( _bRunning )
> > stop();
> >
> > assert( !_bRunning );
> > }
> *****
> There is no need to set the critical section here, since it makes it IMPOSSIBLE to stop
> the thread. As long as the thread is running, the critical section is locked, so the
> stop() function will block (if called from another thread), and not be able to stop the
> thread. However, on the call to stop(), it attempts to re-acquire the critical section
> that the main thread already locked, so it passes. This is risky, because it means you
> don't actually have a guarantee about lockout, and this leads to potential disasters.
> *****
```

Re: Is the following code MT-Safe?

microsoft.public.vc.mfc: Re: Is the following code MT-Safe?

The thread function does not lock the critical section at all. When stop() is called from inside of test(), the lock will be guaranteed to be obtained because test() already got it. This is recursive locking, woo hoo.

```
> >
> >private:
> > static UINT ThreadFunc( LPVOID pParam )
> > {
> > A * a = (A*) pParam;
> >
> > a->_bRunning = true;
> > ::WaitForSingleObject( a->_stopEvent, INFINITE );
> > a->_bRunning = false;
> >
> > a->_stoppedEvent.Set();
> > }
> *****
> I presume there is intended to be more to this thread than what I see here. All that
> happens is that this thread blocks until someone executes stop(), Then it sets an event
> that says "I have successfully stopped". Which you are also waiting on. I believe that
> following this logic, the assert (which is a fundamental design error: NEVER use 'assert'
> in a program, because if the assertion fails, the program exits. A program should exit
> only on user command, never spontaneously) will not be taken.
```

- a.) The assert is part of an `_example_`, to demonstrate that I am insisting that the state of this variable is known at this point.
- b.) Lots of people like assertions, live and let live,

Here's an improvement to the ThreadFunc, if it makes you feel better about the work its doing:

```
static UINT ThreadFunc( LPVOID pParam )
{
    A * a = (A*) pParam;

    a->_bRunning = true;
    a->_startedEvent.Set();

    while( WAIT_TIMEOUT == ::WaitForSingleObject( a->_stopEvent, 100 ) )
    {
        DoStuff();
    }

    a->_bRunning = false;
    a->_stoppedEvent.Set();
}
```

```
> >
> >In this simplistic example, assume the only two threads you see are
> >the only two which are relevant. In this simple example, am I
```

Re: Is the following code MT-Safe?

microsoft.public.vc.mfc: Re: Is the following code MT-Safe?

> > *guaranteed that the assert in the test function will not fire?*
> *****
> *There is no guarantee. Follow the logic. You have _stopped set FALSE. You start the thread
> in a.start(). You then call a.test(), and the value is still FALSE. So test does not call
> stop(). It then falls into the next line, which asserts that !m_bRunning. Fine, except
> that right before that test was executed, the timeslice of the thread that called test()
> ran out, and then, and ONLY then, is the thread permitted to run. It starts up, sets the
> running flag to true, then blocks on the stop event (which will never be set). This allows
> the first thread to resume, which then takes the assert.*

Good, that's an error in the implementation of start(). I missed it in my initial post. Here's a fix to the start, ThreadFunc functions.

```
void start()
{
    CSingleLock( &crit, TRUE );

    if( !_bRunning )
    {
        _startedEvent.ResetEvent();

        AfxBeginThread(ThreadFunc, (LPVOID) this, ... );
        ::WaitForSingleObject( _startedEvent, INFINITE );
    }
}
```

```
static UINT ThreadFunc( LPVOID pParam )
{
    A * a = (A*) pParam;

    a->_bRunning = true;
    a->_startedEvent.Set();

    ::WaitForSingleObject( a->_stopEvent, INFINITE );

    a->_bRunning = false;
    a->_stoppedEvent.Set();
}
```

```
CEvent _startedEvent;
```

>
> *Anything this complex to reason about is almost certainly wrong, and there is far too much
> unnecessary synchronization going on here. Get rid of all of it, and rethink how you would
> handle the notifications. My own preference is to use I/O Completion Ports as event queues
> (when I don't have a GUI, and often when I do). You are trying to force threads into
> lockstep behavior, which is always risky and error-prone, and in this case you have a
> serious defect. Fall back to a message-driven model and most of these complexities
> evaporate. This sort of make-my-parallel-problem-look-single-threaded solution is a common
> error, and almost always leads to either major synchronization failures (as in this case)
> or deadlocks. I try to avoid it these days.*

Re: Is the following code MT-Safe?

> *****

I disagree there's anything too complex to reason about here. We have a class which internally contains a thread function to do some work, it contains a start() and stop() function, and a test() function which needs to call stop to do its work.

I also disagree that any of the synchronization is unnecessary, the initial versions didn't have enough functionality to make this required, but I've beefed up the implementation of these functions to hopefully allieve your concerns.

>

> >

> > *I'm worried that the main thread will read the value of `_bRunning` in the `test()` function, and then when it comes time to do the assert, simply re-use the existing value it has already read, rather than going back to main memory to get the new value which was set by the other thread. Is this a valid concern?*

> *****

> *Well, to make sure this doesn't happen, I would add `volatile` to the declaration, but that would not solve the fundamental error in logic here. The error is so deep that worrying about memory access is not even on the radar.*

Agreed, the start() function had a significant hole in it. Be proud you're the only one who reviewed this code that noticed it. There's currently a humongous discussion in `comp.programming.threads` about this issue, with fierce insistence that `volatile` isn't needed at all.

So, assuming reasonable changes to start(), is `volatile` needed, or just "might" make it better?

-ken