Windows started picking up the really big pieces of TerminateThread garbage on the sidewalk, but it’s still garbage on the sidewalk

devblogs.microsoft.com/oldnewthing/20150814-00

August 14, 2015

Raymond Chen

Ah, `TerminateThread` . There are still people who think that there are valid scenarios for calling `TerminateThread` .

> Can you explain how `ExitThread` works?
>
> We are interested because we have a class called `ThreadClass` . We call the `Start()` method , and then the `Stop()` method, and then the `WaitUntilStopped()` method, and then the process hangs with this call stack:
>
> ```
> ntdll!ZwWaitForSingleObject
> ntdll!RtlpWaitOnCriticalSection
> ntdll!RtlEnterCriticalSection
> ntdll!LdrShutdownThread
> ntdll!RtlExitUserThread
> kernel32!BaseThreadInitThunk
> ntdll!RtlUserThreadStart
> ```
>
> Can you help us figure out what’s going on?

From the stack trace, it is clear that the thread is shutting down, and the loader (`Ldr`) is waiting on a critical section. The critical section the loader is most famous for needing is the so-called *loader lock* which is used for various things, most notably to make sure that all DLL thread notification are serialized.

I guessed that the call to `WaitUntilStopped()` was happening inside `DllMain` , which created a deadlock because the thread cannot exit until it delivers its `DllMain` notifications, but it can’t do that until the calling thread exits `DllMain` .

The customer did some more debugging:

> The debugger reports the critical section as
>
> ```
> CritSec ntdll!LdrpLoaderLock+0 at 77724300
> WaiterWoken        No
> LockCount          3
> RecursionCount     1
> OwningThread       a80
> EntryCount         0
> ContentionCount    3
> *** Locked
> ```
>
> The critical section claims that it is owned by thread `0xa80`, but there is no such active thread in the process. In the kernel debugger, a search for that thread says
>
> ```
> Looking for thread Cid = a80 ...
> THREAD 8579e1c0  Cid 0b58.0a80  Teb: 00000000 Win32Thread: 00000000 TERMINATED
> Not impersonating
> DeviceMap                 862f8a98
> Owning Process            0       Image:         <Unknown>
> Attached Process          84386d90       Image:         Contoso.exe
> Wait Start TickCount      12938474       Ticks: 114780 (0:00:29:50.579)
> Context Switch Count      8
> UserTime                  00:00:00.000
> KernelTime                00:00:00.000
> Win32 Start Address 0x011167c0
> Stack Init 0 Current bae35be0 Base bae36000 Limit bae33000 Call 0
> Priority 10 BasePriority 8 PriorityDecrement 2 IoPriority 2 PagePriority 5
> ```
>
> `Contoso.exe` is our process.

Okay, we’re getting somewhere now. The thread `0xa80` terminated while it held the loader lock. When you run the program under a debugger, do you see any exceptions that might suggest that the thread terminated abnormally?

> We found the cause of the problem. We use `TerminateThread` in the other place. That causes the thread to continue to hold the loader lock after it has terminated.

It’s not clear what the customer meant by “the other place”, but no matter. The cause of the problem was found: They were using `TerminateThread`.

At this point, Larry Osterman was inspired to write a poem.

> How many times does
> it have to be said: Never
> call TerminateThread.

In the ensuing discussion, somebody suggested,

> One case where it is okay to use `TerminateThread` is if the thread was created suspended and has never been resumed. I believe it is perfectly legal to terminate it, at least in Windows Vista and later.

No, it is not "perfectly legal," for certain values of "perfectly legal."

What happened is that Windows Vista added some code to try to limit the impact of a bad idea. Specifically, it added code to free the thread's stack when the thread was terminated, so that each terminated thread didn't leak a megabyte of memory. In the parlance of earlier discussion, I referred to this as stop throwing garbage on the sidewalk.

In this case, it's like saying, "It's okay to run this red light because the city added a delayed green to the cross traffic." The city added a delayed green to the cross traffic because people were running the light and the city didn't want people to die. That doesn't mean that it's okay to run the light now.

Unfortunately, the guidance that says "Sometimes it's okay to call `TerminateThread`" has seeped into our own Best Practices documents. The Dynamic-Link Library Best Practices under *Best Practices for Synchronization* describes a synchronization model which actually involves calling `TerminateThread`.

Do not do this.

It's particularly sad because the downloadable version of the document references both Larry and me telling people to stop doing crazy things in `DllMain`, and terminating threads is definitely a crazy thing.

(The solution to the problem described in the whitepaper is not to use `TerminateThread`. It's to use the `FreeLibraryAndExitThread` pattern.)

Now the history.

Originally, there was no `TerminateThread` function. The original designers felt strongly that no such function should exist because there was no safe way to terminate a thread, and there's no point having a function that cannot be called safely. But people screamed that they needed the `TerminateThread` function, even though it wasn't safe, so the operating system designers caved and added the function because people demanded it. Of course, those people who insisted that they needed `TerminateThread` now regret having been given it.

It's one of those "Be careful what you wish for" things.

Raymond Chen

Follow