Why does my program crash if I terminate a thread that is waiting to enter a critical section? It never got the critical section, so who cares?

devblogs.microsoft.com/oldnewthing/20191101-00

November 1, 2019

Raymond Chen

A customer had a program that used the `TerminateThread` function to terminate a thread while it was waiting for a critical section. They claim that this code worked in earlier versions of Windows, but starting in Windows 10, the program crashes when it tries to use that critical section further. Why is this happening? The documentation for `EnterCriticalSection` says

> If a thread terminates while it has ownership of a critical section, the state of the critical section is undefined.

But it doesn't say that there's anything wrong with terminating a thread that is *waiting* for a critical section.

Well, yeah, it doesn't say that because the only way you can get into that state is if you call `TerminateThread` while the `EnterCriticalSection` is still running, and terminating a thread is already a terrible idea. It's such a terrible idea that doing so puts the entire process is an undefined state. There's no point trying to go into the details of what could happen when your process is an undefined state. It's undefined!

But just for curiosity's sake, what changed in Windows 10 that made the undefined state lead to a crash?

Historically, critical sections were implemented in terms of an interlocked integer and a kernel event. The interlocked integer was used to keep track of the critical section's state: Unowned, owned with no waiting threads, or owned with at least one waiting thread. If a thread needed to wait for the critical section, it waited on the kernel event handle. When the owner of the critical section exited it, it checked if there are any waiting threads, and if so, it signaled the event, thereby allowing one of the waiting threads to attempt to enter.

This design had some flaws. For example, a process that created lots of critical sections could potentially create a lot of kernel event handles, and kernel synchronization objects consume non-paged pool. To alleviate stress on non-paged pool, critical sections created the kernel event handle only on demand, but that introduces a new failure mode if the lazy creation of the kernel event handle fails. Some programs avoided this problem by pre-creating the kernel event handle, but that just put us back where we started with high non-paged-pool usage.

In Windows 10, critical sections were rewritten in terms of `WaitOnAddress`. This removes the need for kernel events entirely, and therefore avoids entire classes of potential problems.

The `WaitOnAddress` function works by linking the list of waiting threads through their stacks. This linked list of threads is manipulated both by waiting threads (when they join the list) and by the waking thread (so it can notify a waiting thread). The code to manipulate this linked list is quite complicated because it needs to do its work in a lock-free manner.[1]

If you terminate a thread, the thread's stack is cleaned up as part of cleaning up the biggest pieces of garbage on the sidewalk. That removes a memory leak, but it also leaves dangling pointers if that thread was calling `WaitOnAddress` or any other function that links memory on the stack into a data structure visible to other threads. And it's those dangling pointers that cause any future use of the critical section to crash.

Bonus chatter: If you think about it, the original design that allocated a kernel handle per critical section was wasteful. Really, you need one kernel handle per thread, since each thread can wait on only one critical section. (There is no `WaitForMultipleCritical-Sections` function.) On the other hand, managing those kernel handles is complicated because you need to do it in a lock-free way. Once you sign up to do the complicated stuff, you may as well go all the way and do it handle-free.

[1] The slim reader-writer locks also links the list of waiting threads through their stacks. The code for slim read-writer locks is even more complicated because it needs to reorder the nodes within the linked list, while still being lock-free.

Raymond Chen

Follow