Quick overview of how processes exit on Windows XP

devblogs.microsoft.com/oldnewthing/20070503-00

May 3, 2007

Raymond Chen

Exiting is one of the scariest moments in the lifetime of a process. (Sort of how landing is one of the scariest moments of air travel.) Many of the details of how processes exit are left unspecified in Win32, so different Win32 implementations can follow different mechanisms. For example, Win32s, Windows 95, and Windows NT all shut down processes differently. (I wouldn't be surprised if Windows CE uses yet another different mechanism.) Therefore, bear in mind that what I write in this mini-series is implementation detail and *can change at any time without warning*. I'm writing about it because these details can highlight bugs lurking in your code. In particular, I'm going to discuss the way processes exit on Windows XP. I should say up front that I do not agree with many steps in the way processes exit on Windows XP. The purpose of this mini-series is not to justify the way processes exit but merely to fill you in on some of the behind-the-scenes activities so you are better-armed when you have to investigate into a mysterious crash or hang during exit. (Note that I just refer to it as the *way* processes exit on Windows XP rather than saying that it is how process exit is *designed*. As one of my colleagues put it, "Using the word *design* to describe this is like using the term *swimming pool* to refer to a puddle in your garden.") When your program calls `ExitProcess` a whole lot of machinery springs into action. First, all the threads in the process (except the one calling `ExitProcess`) are forcibly terminated. This dates back to the old-fashioned theory on how processes should exit: Under the old-fashioned theory, when your process decides that it's time to exit, it should already have cleaned up all its threads. The termination of threads, therefore, is just a safety net to catch the stuff you may have missed. It doesn't even wait two seconds first. Now, we're not talking happy termination like `ExitThread`; that's not possible since the thread could be in the middle of doing something. Injecting a call to `ExitThread` would result in `DLL_THREAD_DETACH` notifications being sent at times the thread was not prepared for. Nope, these threads are terminated in the style of `TerminateThread`: Just yank the rug out from under it. Buh-bye. This is an ex-thread. Well, that was a pretty drastic move, now, wasn't it. And all this after the scary warnings in MSDN that `TerminateThread` is a bad function that should be avoided! Wait, it gets worse. Some of those threads that got forcibly terminated may have owned critical sections, mutexes, home-grown synchronization primitives (such as spin-locks), all those things that the one remaining thread might need access to during its `DLL_PROCESS_DETACH` handling. Well, mutexes are sort of covered; if you try to enter that

mutex, you'll get the mysterious `WAIT_ABANDONED` return code which tells you that "Uh-oh, things are kind of messed up." What about critical sections? There is no "Uh-oh" return value for critical sections; `EnterCriticalSection` doesn't have a return value. Instead, the kernel just says "Open season on critical sections!" I get the mental image of all the gates in a parking garage just opening up and letting anybody in and out. [See correction.] As for the home-grown stuff, well, you're on your own. This means that if your code happened to have owned a critical section at the time somebody called `ExitProcess`, the data structure the critical section is protecting has a good chance of being in an inconsistent state. (Afer all, if it were consistent, you probably would have exited the critical section! Well, assuming you entered the critical section because you were updating the structure as opposed to reading it.) Your `DLL_PROCESS_DETACH` code runs, enters the critical section, and it *succeeds* because "all the gates are up". Now your `DLL_PROCESS_DETACH` code starts behaving erratically because the values in that data structure are inconsistent. Oh dear, now you have a pretty ugly mess on your hands. And if your thread was terminated while it owned a spin-lock or some other home-grown synchronization object, your `DLL_PROCESS_DETACH` will most likely simply hang indefinitely waiting patiently for that terminated thread to release the spin-lock (which it never will do). But wait, it gets worse. That critical section might have been the one that protects the process heap! If one of the threads that got terminated happened to be in the middle of a heap function like `HeapAllocate` or `LocalFree`, then the process heap may very well be inconsistent. If your `DLL_PROCESS_DETACH` tries to allocate or free memory, it may crash due to a corrupted heap. Moral of the story: If you're getting a `DLL_PROCESS_DETACH` due to process termination,† don't try anything clever. Just return without doing anything and let the normal process clean-up happen. The kernel will close all your open handles to kernel objects. Any memory you allocated will be freed automatically when the process's address space is torn down. Just let the process die a quiet death. Note that if you were a good boy and cleaned up all the threads in the process before calling `ExitThread`, then you've escaped all this craziness, since there is nothing to clean up. Note also that if you're getting a `DLL_PROCESS_DETACH` due to dynamic unloading, then you do need to clean up your kernel objects and allocated memory because the process is going to continue running. But on the other hand, in the case of dynamic unloading, no other threads should be executing code in your DLL anyway (since you're about to be unloaded), so —assuming you coded up your DLL correctly—none of your critical sections should be held and your data structures should be consistent. Hang on, this disaster isn't over yet. Even though the kernel went around terminating all but one thread in the process, that doesn't mean that the creation of new threads is blocked. If somebody calls `CreateThread` in their `DLL_PROCESS_DETACH` (as crazy as it sounds), the thread will indeed be created and start running! But remember, "all the gates are up", so your critical sections are just window dressing to make you feel good. (The ability to create threads after process termination has begun is not a mistake; it's intentional and necessary. Thread injection is how the debugger breaks into a process. If thread injection were not permitted, you wouldn't be able to debug process termination!) Next time, we'll see how the way process termination takes place on Windows XP caused not one but two problems. **Footnotes**

†Everybody reading this article should already know how to determine whether this is the case. I'm assuming you're smart. Don't disappoint me.

Raymond Chen

Follow