Does creating a thread from DllMain deadlock or doesn't it?

devblogs.microsoft.com/oldnewthing/20070904-00

September 4, 2007

Raymond Chen

Let me get this out of the way up front: Creating a thread from `DllMain` is not recommended. The discussion here has to do with explaining the behavior you may observe if you violate this advice. Commenter Pete points out that "according to Usenet" creating a thread in `DllMain` is supposed to deadlock, but that's not what he saw. All he saw was that the thread entry procedure was not called. I'm going to set aside what "according to Usenet" means. Recall how a thread starts up. When you call `CreateThread` , a kernel thread object is created and scheduled. Once the thread gets a chance to run, the kernel calls all the `DllMain` functions with the `DLL_THREAD_ATTACH` code. Once that's done, the thread's entry point is called. The issue with deadlocks is that all `DllMain` functions are serialized. At most one `DllMain` can be running at a time. Suppose a `DllMain` function is running and it creates a thread. As we noted above, a kernel thread object is created and scheduled, and the first thing the thread does is notify all the DLLs with `DLL_THREAD_ATTACH` . Since `DllMain` functions are serialized, the attempt to send out the `DLL_THREAD_ATTACH` notifications must wait until the current `DllMain` function returns. That's why you observe that the new thread's entry point doesn't get called until after you return from `DllMain` . The new thread hasn't even made it that far; it's still working on the `DLL_THREAD_ATTACH` notifications. On the other hand, there is no actual deadlock here. The new thread will get itself off the ground once everybody else has finished doing their `DllMain` work. So what is this deadlock that Usenet talks about? If you've been following along, you should spot it easily enough. If your `DllMain` function creates a thread and then waits for the thread to do something (e.g., waits for the thread to signal an event that says that it has finished initializing, then you've created a deadlock. The `DLL_PROCESS_ATTACH` notification handler inside `DllMain` is waiting for the new thread to run, but the new thread can't run until the `DllMain` function returns so that it can send a new `DLL_THREAD_ATTACH` notification. This deadlock is much more commonly seen in `DLL_PROCESS_DETACH` , where a DLL wants to shut down its worker threads and wait for them to clean up before it unloads itself. You can't wait for a thread inside `DLL_PROCESS_DETACH` because that thread needs to send out the `DLL_THREAD_DETACH` notifications before it exits, which it can't do until your `DLL_PROCESS_DETACH` handler returns.

(It is for this thread cleanup case that the function `FreeLibraryAndExitThread` was created.)

Raymond Chen

Follow