A process shutdown puzzle: Answers

devblogs.microsoft.com/oldnewthing/20090206-00

February 6, 2009

Raymond Chen

Last week, I posed a process shutdown puzzle in honor of National Puzzle Day. Let's see how we did.

Part One asked us to explain why the `ThreadFunction` thread no longer exists. That's easy. One of the things that happen inside `ExitProcess` is that all threads (other than the one calling `ExitProcess`) are forcibly terminated in the nastiest way possible. This happens before the `DLL_PROCESS_DETACH` notification is sent. Therefore, the code in `StopWorkerThread` that waits for the thread completion event waits forever because the `ThreadFunction` is no longer running. There is nobody around to see the shutdown event and respond by setting the completion event.

Okay, that was the easy part. Part Two asked us to criticize the replacement solution which replaced the completion event with a call to `FreeLibraryAndExitThread` and changed the `StopWorkerThread` function to wait for the thread handle to become signaled. This solution is also flawed.

Consider the case that the DLL is receiving its `DLL_PROCESS_DETACH` notification because the DLL is being unloaded by a call to `FreeLibrary`, rather than due to process termination. In that case, `StopWorkerThread` sets the shutdown event, and the `ThreadFunction` proceeds to clean up and call `FreeLibraryAndExitThread`. But one of the steps in thread shutdown is sending `DLL_THREAD_DETACH` notifications, which will not happen until the `DLL_PROCESS_DETACH` notifications are complete. The `WaitForSingleObject` waits indefinitely because it won't complete until the thread exits, but the thread won't exit until `StopWorkerThread` returns. Deadlock.

Finally, Part Three asks us to explain why the code doesn't cause a problem in practice even though the code is flawed. The call to `FreeLibraryAndExitThread` implies that the code follows the "Worker thread retains its own reference on the DLL" model. After all, that's why the last thing the thread does is free the library. But if that's the case, then a call to `FreeLibrary` coming from the application won't actually unload the DLL, because the DLL

reference count is still nonzero: There is one reference still being held by the worker thread. Therefore, the DLL will never actually unload outside of process termination. All the flaws in the dynamic unload case are masked by the fact that the code never executes.

Led astray: Some of us mentioned that waiting on `ThreadHandle` returned immediately because the handle to a thread is automatically closed when the thread exits. This is wrong. Handles do not self-close. You have to call `CloseHandle` to close them. This is "obvious" if you apply the "imagine if the world actually worked this way" rule: Suppose thread handles were invalidated (and eligible for re-use) when a thread exited. Then how could you use a thread handle *at all*? Any time you use a thread handle, there would be an unavoidable race condition where the thread might have exited just before you used the handle. And it would be impossible to call `GetExitCodeThread` at all! (Since it only does anything interesting if you pass the handle to a thread that has exited.)

A handle to a thread remains valid until you close it. If the thread has exited, then a wait on the thread handle completes, but the handle is still valid *because if it went invalid, programming would become impossible*.

Raymond Chen

Follow