Removing the TerminateThread from a DLL that needs to shut down a helper thread at unload

devblogs.microsoft.com/oldnewthing/20180907-00

September 7, 2018

Raymond Chen

Once more we undertake the exercise of designing the `TerminateThread` out of some code.

The customer had an instrumentation toolchain. What you did was take your object code and sent it through an instrumentation tool, and that tool patched your object code so it could instrument things of interest. You then linked the modified object code with a special instrumentation library (statically-linked) to produce the final instrumented binary.

The static library created a worker thread, and they needed to shut down that worker thread. The object code was completely oblivious to the fact that somebody was trying to instrument it, so the static library had to somehow manage this worker thread without any help from the outside.

To clean up the worker thread, the instrumentation library used the `atexit` function to get a callback to run when the hosting DLL went through its `DLL_ PROCESS_ DETACH`. The developers didn't have way to get the worker thread to shut down cleanly, so they just called `TerminateThread` and crossed their fingers that it wouldn't come back to bite them.

Let's see if we can do better.

One idea is to put the worker thread in a helper DLL. The static library creates the thread on demand using the `FreeLibraryAndExitThread` technique to ensure that the worker thread maintains a reference to the host DLL. The `atexit` callback function calls a shutdown function in the helper DLL. Following the `FreeLibraryAndExitThread` technique, the shutdown function would signal the worker thread to exit and return immediately, allowing the worker thread to exit and free the library on its own.

There are some race conditions to be dealt with, such as the case where the host DLL is reloaded before the helper DLL's worker thread can exit. But these issues can be worked out.

The customer was reluctant to introduce a new DLL into the picture, however. For example, it means that the host's installer would have to carry the helper DLL when installing an instrumented version.

To avoid the helper DLL, the code could create a worker task in the thread pool with `CreateThreadpoolWork`, with an environment marked as `SetThreadpoolCallbackRunsLong`. Make that task do whatever the original thread was doing.

When it's time to shut down the worker thread, signal the worker task to exit using an event or some other private mechanism, and then call `WaitForThreadpoolWorkCallbacks` to wait for the exit to occur. Of course, you want to skip this if the entire process is shutting down.

This trick does assume that the worker task does not require any locks that might be held by the code running `DLL_PROCESS_DETACH` (most notably the loader lock).

The customer replied that they had found an even better third solution: They got rid of the worker thread entirely!

The purpose of the worker thread was to respond to requests for information from the instrumentation tool, and the customer realized that they could extract that information with careful use of `ReadProcessMemory`, so there was no need to have a thread dedicated to handing out that information.

(Normally, I wouldn't be a fan of using `ReadProcessMemory` as a mechanism for interprocess communication because it requires that the other process have `PROCESS_VM_READ` access to the process, which is a pretty large farm to be giving away, and it doesn't give you very useful granularity. But since this is an instrumentation tool, it's not unreasonable to require that the tool run in a security context that has full access to the process being instrumented.)

Raymond Chen

Follow