Partially eliminating the need for SetThreadpoolCallbackLibrary and reducing the cost of FreeLibraryAndExitThread

devblogs.microsoft.com/oldnewthing/20131107-00

November 7, 2013

Raymond Chen

Update: Daniel points out that there is still a race condition here, so this trick won't work. Rats.

The documentation for the `SetThreadpoolCallbackLibrary` says

> This prevents a deadlock from occurring when one thread in DllMain is waiting for the callback to end, and another thread that is executing the callback attempts to acquire the loader lock.
>
> If the DLL containing the callback might be unloaded, the cleanup code in DllMain must cancel outstanding callbacks before releasing the object.
>
> Managing callbacks created with a `TP_CALLBACK_ENVIRON` that specifies a callback library is somewhat processor-intensive. You should consider other options for ensuring that the library is not unloaded while callbacks are executing, or to guarantee that callbacks which may be executing do not acquire the loader lock.

I'm not going to help you with the DllMain cleanup issues. (My plan is to simply avoid the issue by preventing the DLL from unloading while a callback is still pending. That way, you never have to cancel the callback from DllMain.) But I am going to help with the "consider other options for ensuring that the library is not unloaded while callbacks are executing."

The first-pass solution is to use the same trick we use when creating worker threads: We bump the DLL reference count when queueing the work item and use `FreeLibraryWhenCallbackReturns` to decrement the reference count after the callback finishes. (We can't use `FreeLibraryAndExitThread`, of course, since we're running on a thread on loan to us from the thread pool. Exiting the thread from a thread pool callback is like demolishing the house you're renting.)

The second-pass solution is to manage the DLL reference count manually. (Don't go down this route unless your profiling suggests that DLL reference count management is a performance bottleneck.) The rule is still that the DLL reference count is prevented from dropping to zero while a callback is pending, but instead of incrementing the reference count each time we scheduled a callback, we'll increment it only when the number of callbacks goes from zero to nonzero. Conversely, we decrement the reference count only when the number of callbacks drops from nonzero to zero.

You can think of this as proxying the reference count, similar to how COM creates proxies that collapse `AddRef` and `Release` calls and signal the remote object only when the reference count transitions from zero to nonzero or vice versa.

This optimization works for `FreeLibraryAndExitThread`, too, so let's fold that in while we're there.

```
LONG g_lProxyRefCount = 0;
BOOL ProxyAddRefThisDll()
{
 if (InterlockedIncrement(&g_lProxyRefCount) == 1) {
  HMODULE hmod;
  return GetModuleHandleEx(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
                           reinterpret_cast<LPCTSTR>(g_hinstSelf),
                           &hmod);
 }
 return TRUE;
}
DECLSPEC_NORETURN
void ProxyFreeLibraryAndExitThread(DWORD dwExitCode)
{
 if (InterlockedDecrement(&g_lProxyRefCount) == 0) {
  FreeLibraryAndExitThread(g_hinstSelf, dwExitCode);
 } else {
  ExitThread(dwExitCode);
 }
}
void ProxyFreeLibraryWhenCallbackReturns(PTP_CALLBACK_INSTANCE pci)
{
 if (InterlockedDecrement(&g_lProxyRefCount) == 0) {
  FreeLibraryWhenCallbackReturns(pci, g_hinstSelf);
 }
}
```

Raymond Chen

Follow