What is the point of FreeLibraryAndExitThread?

devblogs.microsoft.com/oldnewthing/20131105-00

November 5, 2013

Raymond Chen

The `FreeLibraryAndExitThread` function seems pointless. I mean, all the function does is

```
DECLSPEC_NORETURN
void WINAPI FreeLibraryAndExitThread(
    HMODULE hLibModule,
    DWORD dwExitCode)
{
    FreeLibrary(hLibModule);
    ExitThread(dwExitCode);
}
```

Who needs such a trivial function? If I wanted to do that, I could just write it myself.

```
DWORD CALLBACK MyThreadProc(void *lpParameter)
{
    ... blah blah blah ...
    // ~~FreeLibraryAndExitThread(g_hinstSelf, 0);~~
    FreeLibrary(g_hinstSelf);
    ExitThread(0);
}
```

And then you discover that occasionally your program crashes. What’s going on?

Let’s rewind and look at the original problem.

Originally, you had code that did something like this:

```
DWORD CALLBACK SomethingThreadProc(void *lpParameter)
{
 ... do something ...
 return 0;
}
void DoSomethingInTheBackground()
{
 DWORD dwThreadId;
 HANDLE hThread = CreateThread(nullptr, 0, SomethingThreadProc,
                  nullptr, 0, &dwThreadId);
 if (hThread) CloseHandle(hThread);
}
```

This worked great, until somebody did this to your DLL:

```
HMODULE hmodDll = LoadLibrary(TEXT("awesome.dll"));
if (hmodDll) {
 auto pfn = reinterpret_cast<decltype(DoSomethingInTheBackground)*>
            (GetProcAddress(hmodDll, "DoSomethingInTheBackground"));
 if (pfn) pfn();
 FreeLibrary(hmodDll);
}
```

This code fragment calls your `DoSomethingInTheBackground` function and then immediately unloads the DLL, presumably because all they wanted to do was call that one function.

Now you have a problem: That `FreeLibrary` call frees your DLL, while your `Something-ThreadProc` is still running! Result: A crash at an address where there is no code. Older debuggers reported this as a crash in ⟨unknown⟩; newer ones can dig into the recently-unloaded modules list and report it as a crash in `awesome_unloaded`.

This is a very common class of error. When I helped out the application compatibility team by looking at crashes in third-party code, the majority of the crashes I looked at in Internet Explorer were of this sort, where a plug-in got unloaded while it still had a running thread.

How do you prevent your DLL from being unloaded while you still have code running (or have registered callbacks)? You perform a bonus `LoadLibrary` on yourself, thereby bumping your DLL reference count by one.

If you don't need to support Windows 2000, you can use the new `GetModuleHandleEx` function, which is much more convenient and probably a lot faster, too.

```
BOOL IncrementDLLReferenceCount(HINSTANCE hinst)
{
 HMODULE hmod;
 return GetModuleHandleEx(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
                          reinterpret_cast<LPCTSTR>(hinst),
                          &hmod);
}
```

Bumping the DLL reference count means that when the original person who called `LoadLibrary` finally calls `FreeLibrary`, your DLL will still remain in memory because the reference count has not yet dropped all the way to zero because you have taken a reference to the DLL yourself.

When you unregister your callback or your background thread finishes, you call `FreeLibrary` to release your reference to the DLL, and if that's the last reference, then the DLL will be unloaded.

But wait, now we have a problem. When you call `FreeLibrary` to release your reference to the DLL, that call might end up unloading the code that is making the call. When the call returns, there is no more code there. This most commonly happens when you are calling `FreeLibrary` on yourself and that was the last reference. In rarer circumstances, it happens indirectly through a chain of final references.

Let's walk through that scenario again, since understanding it is central to solving the problem.

1. Some application calls `LoadLibrary` on your DLL. The reference count on your DLL is now 1.
2. The application calls a function in your DLL that uses a background thread.
3. Your DLL prepares for the background thread by doing a `GetModuleHandleEx` on itself, to avoid a premature unload. The reference count on your DLL is now 2.
4. Your DLL starts the background thread.
5. The application decides that it doesn't need your DLL any more, so it calls `FreeLibrary`. The reference count on your DLL is now 1.
6. Your DLL background thread finishes its main work. The thread procedure ends with the lines

    ```
    FreeLibrary(g_hinstSelf);
    return 0;
    ```

7. The thread procedure calls `FreeLibrary(g_hinstSelf)` to drop its reference count.
8. The `FreeLibrary` function frees your DLL.
9. The `FreeLibrary` function returns to its caller, namely your thread procedure.
10. Crash, because your thread procedure was unloaded!

This is why you need `FreeLibraryAndExitThread`: So that the return address of the `FreeLibrary` is not in code that's being unloaded by the `FreeLibrary` itself.

Change the last two lines of the thread procedure to `FreeLibraryAndExitThread(g_hinstSelf, 0);` and watch what happens. The first five steps are the same, and then we take a turn:

6. Your DLL background thread finishes its main work. The thread procedure ends with a call to

    ```
    FreeLibraryAndExitThread(g_hinstSelf, 0);
    ```

7. The `FreeLibraryAndExitThread` function calls `FreeLibrary(g_hinstSelf)`.
8. The `FreeLibrary` function frees your DLL.
9. The `FreeLibrary` function returns to its caller, which is not your thread procedure but rather the `FreeLibraryAndExitThread` function, which was not unloaded.
10. The `FreeLibraryAndExitThread` function calls `ExitThread(0)`.
11. The thread exits and no further code is executed.

That's why the `FreeLibraryAndExitThread` function exists: So you don't pull the rug out from underneath yourself. Instead, you have somebody else pull the rug for you.

This issue of keeping your DLL from unloading prematurely rears its head in several ways. We'll look at some of them in the next few days.

Bonus chatter: The thread pool version of `FreeLibraryAndExitThread` is `FreeLibraryWhenCallbackReturns`.

Raymond Chen

Follow