Why am I getting a crash at shutdown inside the thread pool?

devblogs.microsoft.com/oldnewthing/20170203-00

February 3, 2017

Raymond Chen

A customer reported a crash in WinHTTP when their application shuts down a WebSocket. Specifically, it occurs when one of their DLL's global objects is being destructed.

The customer sent us a redacted call stack:

```
00a5e11c 7753ebbe ntdll!KiFastSystemCallRet
00a5e120 77581174 ntdll!NtAlpcSendWaitReceivePort+0xa
00a5e1d0 7758078a ntdll!SendMessageToWERService+0x14d
00a5ecc0 77580c10 ntdll!ReportExceptionInternal+0xde
00a5f118 7758085b ntdll!RtlReportExceptionEx+0x379
00a5f170 775a74dc ntdll!RtlReportException+0x9b
00a5f180 77541454 ntdll!TppRaiseInvalidParameter+0x51
00a5f194 77540ddd ntdll!_EH4_CallFilterFunc+0x12
00a5f1bc 77544d33 ntdll!_except_handler4_common+0x8d
00a5f1dc 775508d2 ntdll!_except_handler4+0x20
00a5f200 775508a4 ntdll!ExecuteHandler2+0x26
00a5f2c8 7753f477 ntdll!ExecuteHandler+0x24
00a5f2c8 775a74c2 ntdll!KiUserExceptionDispatcher+0xf
00a5f660 7755ddb0 ntdll!TppRaiseInvalidParameter+0x37
00a5f66c 774ecdd2 ntdll!TppTimerpValidateTimer+0x6e1a2
00a5f690 757ddadb ntdll!TpSetTimerEx+0x1b
00a5f6b8 757c646d WINHTTP!HTTP_THREAD_POOL::SetTimer+0x42
00a5f6f0 757c6070 WINHTTP!WEB_SOCKET_HANDLE_OBJECT::Close+0x1bb
00a5f754 69699832 WINHTTP!WinHttpWebSocketClose+0x9c
...
 global atexit call being made here
...
00a5f814 696d1f7d XXXXXX!_CRT_INIT+0xaa
00a5f874 7753cd4e XXXXXX!__DllMainCRTStartup+0x1ee
00a5f894 77505525 ntdll!LdrxCallInitRoutine+0x16
00a5f8e4 775057cb ntdll!LdrpCallInitRoutine+0x43
00a5f97c 77518e3f ntdll!LdrShutdownProcess+0x101
00a5f990 77065736 ntdll!RtlExitUserProcess+0x63
00a5f99c 77065471 msvcrt!__crtExitProcess+0x17
00a5f9e0 77065715 msvcrt!doexit+0x10a
00a5f9f4 00be2369 msvcrt!exit+0x11
00a5fa2c 7752b2dd contoso!__wmainCRTStartup+0x114
00a5fa70 7752b2a7 ntdll!__RtlUserThreadStart+0x2f
00a5fa80 00000000 ntdll!_RtlUserThreadStart+0x1b
```

The customer concluded, "We have some ideas that may work around the issue by using `WINHTTP_OPTION_WEB_SOCKET_CLOSE_TIMEOUT` to avoid the close timeout, but we'd like confirmation as to whether this will actually solve the problem."

Okay, first let's understand the problem, then we can look at possible solutions.

The customer has a DLL with a global object, and as we learned some time ago, global objects in DLLs are destructed as part of `DLL_PROCESS_DETACH`. The problem is that the thread pool has already shut down by the time this DLL gets around to destroying global objects. We know this because one of the first steps in process termination is terminating all but one of the threads. A thread pool without any threads is not really a thread pool any more.

At process termination, the thread pool is electrified. Any attempt to schedule new work on the thread pool will result in an immediate crash. In this case, the problem is that the customer's DLL is closing a WinHTTP WebSocket, and one of the things that WinHTTP does

when it closes a WebSocket is to schedule a thread pool timer so it can abort the close handshake if it takes too long.

Okay, so the chain of events goes like this: Thread pool gets electrified, then the DLL starts destructing its objects, and one of the objects tries to close a WebSocket, and closing the WebSocket creates a thread pool timer, but the thread pool is electrified, so the process crashes.

Okay, now that we understand the problem, let's look for solutions.

The customer's proposed workaround is to use `WINHTTP_OPTION_WEB_SOCKET_CLOSE_TIMEOUT` to set the timeout to `INFINITE`. This tells WinHTTP to let the close operation take as long as it wants, which means that it doesn't bother creating a thread pool timer to abort a close operation that is taking too long (because you said that there's no such thing as "too long").

That solves the proximate problem, but really this is just playing whack-a-mole. You may be able to get rid of this crash caused by closing a WinHTTP WebSocket, but this may merely expose some other object that is also using the thread pool at destruction, and you're going to have to go through all this analysis again and look for a way to get that other object to avoid the thread pool at process termination.

The best solution is to try to get rid of the global variables in the first place. If you can't do that, then you at least want to avoid running the destructors at process termination. There are a few ways of accomplishing this:

- Clean up the global variables explicitly prior to process termination. The destructors will run at `DLL_PROCESS_DETACH`, but since you already released the resources, the destructors won't do anything.
- Neuter the global variables in `DLL_PROCESS_DETACH` if the reason for the notification is that the process is terminating. That way, when their destructors run, they won't do anything.
- A special case of the previous item is to set a flag in `DLL_PROCESS_DETACH` if the reason for the notification is that the process is terminating. Have the destructors check the flag and do nothing if the flag is set.

The point is that you don't want to do any cleanup at process termination, because the process has already stopped providing services, and lots of things may be electrified. You just want to let the process terminate and stay out of its way.

Exercise: By a startling coincidence, the day I wrote this blog entry, this question arrived from another customer. Use what you know to diagnose the customer's problem. (In particular, why is the problem sporadic?)

> We are using a C++ wrapper around Win32 timers. During object destruction, we deactivate the timer by following the recommended pattern: `::SetThreadpoolTimer(this->GetHandle(), nullptr, 0, 0);` This works fine, but in some rare scenarios, we encounter this crash.
>
> ```
> ntdll!ZwWaitForMultipleObjects+0xa
> ntdll!RtlReportExceptionEx+0x452
> ntdll!RtlReportException+0xbc
> ntdll!TppReportExceptionFilter+0x16
> ntdll!TppRaiseInvalidParameter$filt$0+0xe
> ntdll!__C_specific_handler+0x96
> ntdll!__GSHandlerCheck_SEH+0x76
> ntdll!RtlpExecuteHandlerForException+0xd
> ntdll!RtlDispatchException+0x197
> ntdll!RtlRaiseException+0x18d
> ntdll!TppRaiseInvalidParameter+0x48
> ntdll!TppTimerpValidateTimer+0x6eb93
> ntdll!TpSetTimerEx+0x33
> contoso!WinAPI::ThreadPool::Timer<...>::Reset+0x12
> contoso!WinAPI::ThreadPool::Timer<...>::{dtor}+0x12
> contoso!std::default_delete<WinAPI::ThreadPool::Timer<...>>::operator()+0x12
> contoso!std::unique_ptr<WinAPI::ThreadPool::Timer<...>, ...>::reset+0x23
> contoso!Contoso::SharedMemoryCache::~SharedMemoryCache+0x57
> contoso!Contoso::SharedMemoryCache::`scalar deleting destructor'+0x14
> contoso!std::_Ref_count_base::_Decref+0x17
> contoso!std::_Ptr_base<...>::_Decref+0x20
> contoso!std::shared_ptr<...>::{dtor}+0x20
> contoso!std::tuple<...>::~tuple<...>+0x49
> contoso!`dynamic atexit destructor for 'Extension::s_extension''+0x23
> ucrtbase!<lambda_275893d493268fdec8709772e3fcec0e>::operator()+0xb7
> ucrtbase!__crt_seh_guarded_call<int>::operator()<...>+0x3b
> ucrtbase!__acrt_lock_and_call+0x1e
> ucrtbase!_execute_onexit_table+0x31
> contoso!dllmain_crt_process_detach+0x4e
> contoso!dllmain_dispatch+0xd3
> ntdll!LdrpCallInitRoutine+0x4c
> ntdll!LdrShutdownProcess+0x142
> ntdll!RtlExitUserProcess+0x98
> kernel32!ExitProcessImplementation+0xa
> contososerver!ControlSignalHandler::HandleControlSignal+0x68
> KERNELBASE!CtrlRoutine+0xb3
> kernel32!BaseThreadInitThunk+0x22
> ntdll!RtlUserThreadStart+0x34
> ```
>
> Any pointers would be appreciated.

Raymond Chen

Follow