Psychic debugging: The first step in diagnosing a deadlock is a simple matter of following the money

devblogs.microsoft.com/oldnewthing/20071228-00

December 28, 2007

Raymond Chen

Somebody asked our team for help because they believed they hit a deadlock in their program's UI. (It's unclear why they asked our team, but I guess since our team uses the window manager, and their program uses the window manager, we're all in the same boat. You'd think they'd ask the window manager team for help.)

But it turns out that solving the problem required no special expertise. In fact, you probably know enough to solve it, too.

> Here are the interesting threads:
>
> ```
>   0  Id: 980.d30 Suspend: 1 Teb: 7ffdf000 Unfrozen
> ChildEBP RetAddr
> 0023dc90 7745dd8c ntdll!KiFastSystemCallRet
> 0023dc94 774619e0 ntdll!ZwWaitForSingleObject+0xc
> 0023dcf8 774618fb ntdll!RtlpWaitOnCriticalSection+0x154
> 0023dd20 00cd03f2 ntdll!RtlEnterCriticalSection+0x152
> 0023dd38 00cd0635 myapp!LogMsg+0x15
> 0023dd58 00cd0c6a myapp!LogRawIndirect+0x27
> 0023fcb8 00cb64a7 myapp!Log+0x62
> 0023fce8 00cd7598 myapp!SimpleClientConfiguration::Cleanup+0x17
> 0023fcf8 00cd8ffe myapp!MsgProc+0x1a9
> 0023fd10 00cda1a9 myapp!Close+0x43
> 0023fd24 761636d2 myapp!WndProc+0x62
> 0023fd50 7616330c USER32!InternalCallWinProc+0x23
> 0023fdc8 76164030 USER32!UserCallWinProcCheckWow+0x14b
> 0023fe2c 76164088 USER32!DispatchMessageWorker+0x322
> 0023fe3c 00cda3ba USER32!DispatchMessageW+0xf
> 0023fe9c 00cd0273 myapp!GuiMain+0xe8
> 0023feb4 00ccdeca myapp!wWinMain+0x87
> 0023ff48 7735c6fc myapp!__wmainCRTStartup+0x150
> 0023ff54 7742e33f kernel32!BaseThreadInitThunk+0xe
> 0023ff94 00000000 ntdll!_RtlUserThreadStart+0x23
>    1  Id: 980.ce8 Suspend: 1 Teb: 7ffdd000 Unfrozen
> ChildEBP RetAddr
> 00f8d550 76162f81 ntdll!KiFastSystemCallRet
> 00f8d554 76162fc4 USER32!NtUserSetWindowLong+0xc
> 00f8d578 76162fe5 USER32!_SetWindowLong+0x131
> 00f8d590 74aa5c2b USER32!SetWindowLongW+0x15
> 00f8d5a4 74aa5b65 comctl32_74a70000!ClearWindowStyle+0x23
> 00f8d5cc 74ca568f comctl32_74a70000!CCSetScrollInfo+0x103
> 00f8d618 76164ea2 uxtheme!ThemeSetScrollInfoProc+0x10e
> 00f8d660 00cdd913 USER32!SetScrollInfo+0x57
> 00f8d694 00cdf0a4 myapp!SetScrollRange+0x3b
> 00f8d6d4 00cdd777 myapp!TextOutputStringColor+0x134
> 00f8d93c 00cd04c4 myapp!TextLogMsgProc+0x3db
> 00f8d960 00cd0635 myapp!LogMsg+0xe7
> 00f8d980 00cd0c6a myapp!LogRawIndirect+0x27
> 00f8f8e0 00cd6367 myapp!Log+0x62
> 00f8faf0 7735c6fc myapp!remote_ext::ServerListenerThread+0x45c
> 00f8fafc 7742e33f kernel32!BaseThreadInitThunk+0xe
> 00f8fb3c 00000000 ntdll!_RtlUserThreadStart+0x23
> ```

The thing about debugging deadlocks is that you usually don't need to understand what's going on. The diagnosis is largely mechanical once you get your foot in the door. (Though sometimes it's hard to get your initial footing.)

Let's look at thread 0. It is waiting for a critical section. The owner of that critical section is thread 1. How do I know that? Well, I could've debugged it, or I could've used my psychic powers to say, "Gosh, that function is called `LogMsg`, and look there's another thread that is inside the function `LogMsg`. I bet that function is using a critical section to ensure that only one thread uses it at a time."

Okay, so thread 0 is waiting for thread 1. What is thread 1 doing? Well, it entered the critical section back in the `LogMsg` function, and then it did some text processing and, oh look, it's doing a `SetScrollInfo`. The `SetScrollInfo` went into `comctl32` and ultimately resulted in a `SetWindowLong`. The window that the application passed to `SetScrollInfo` is owned by thread 0. How do I know that? Well, I could've debugged it, or I could've used my psychic powers to say, "Gosh, the change in the scroll info has led to a change in window styles, and the thread is trying to notify the window of the change in style. The window clearly belongs to another thread; otherwise we wouldn't be stuck in the first place, and given that we see only two threads, there isn't much choice as to what other thread it could be!"

At this point, I think you see the deadlock. Thread 0 is waiting for thread 1 to exit the critical section, but thread 1 is waiting for thread 0 to process the style change message.

What happened here is that the program sent a message while holding a critical section. Since message handling can trigger hooks and cross-thread activity, you cannot hold any resources when you send a message because the hook or the message recipient might want to acquire that resource that you own, resulting in a deadlock.

Raymond Chen

Follow