The case of the UI thread that hung in a kernel call

devblogs.microsoft.com/oldnewthing/20250411-00/?p=111066

April 11, 2025

A customer asked for help with a longstanding but low-frequency hang that they have never been able to figure out. From what they could tell, their UI thread was calling into the kernel, and the call simply hung for no apparent reason. Unfortunately, the kernel dump couldn't show a stack from user mode because the stack had been paged out. (Which makes sense, because a hung thread isn't using its stack, so once the system is under some memory pressure, that stack gets paged out.)

```
0: kd> !thread 0xffffd18b976ec080 7
THREAD ffffd18b976ec080  Cid 79a0.7f18  Teb: 0000003d7ca28000
    Win32Thread: ffffd18b89a8f170 WAIT: (Suspended) KernelMode Non-Alertable
SuspendCount 1
    ffffd18b976ec360  NotificationEvent
Not impersonating
DeviceMap                 ffffad897944d640
Owning Process            ffffd18bcf9ec080       Image:         contoso.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      14112735       Ticks: 1235580 (0:05:21:45.937)
Context Switch Count      1442664        IdealProcessor: 2
UserTime                  00:02:46.015
KernelTime                00:01:11.515

 nt!KiSwapContext+0x76
 nt!KiSwapThread+0x928
 nt!KiCommitThreadWait+0x370
 nt!KeWaitForSingleObject+0x7a4
 nt!KiSchedulerApc+0xec
 nt!KiDeliverApc+0x5f9
 nt!KiCheckForKernelApcDelivery+0x34
 nt!MiUnlockAndDereferenceVad+0x8d
 nt!MmProtectVirtualMemory+0x312
 nt!NtProtectVirtualMemory+0x1d9
 nt!KiSystemServiceCopyEnd+0x25 (TrapFrame @ ffff8707`a9bef3a0)
 ntdll!ZwProtectVirtualMemory+0x14
 [end of stack trace]
```

Although we couldn't see what the code was doing in user mode, there was something unusual in the information that was present.

Observe that the offending thread is *Suspended*. And it appears to have been suspended for over five hours.

```
THREAD ffffd18b976ec080  Cid 79a0.7f18  Teb: 0000003d7ca28000
    Win32Thread: ffffd18b89a8f170 WAIT: (Suspended) KernelMode Non-Alertable
SuspendCount 1
    ffffd18b976ec360  NotificationEvent
Not impersonating
DeviceMap                 ffffad897944d640
Owning Process            ffffd18bcf9ec080       Image:         contoso.exe
Attached Process          N/A            Image:         N/A
Wait Start TickCount      14112735       Ticks: 1235580 (0:05:21:45.937)
```

Naturally, a suspended UI thread is going to manifest itself as a hang.

Functions like `SuspendThread` exist primarily for debuggers to use, so we asked them if they had a debugger attached to the process when they captured the kernel dump. They said that they did not.

So who suspended the thread, and why?

The customer then realized that they had a watchdog thread which monitors the UI thread for responsiveness, and every so often, it suspends the UI thread, captures a stack trace, and then resumes the UI thread. And in the dump file, they were able to observe their watchdog thread in the middle of its stack trace capturing code. But why was the stack trace capture taking five hours?

The stack of the watchdog thread looks like this:

```
ntdll!ZwWaitForAlertByThreadId(void)+0x14
ntdll!RtlpAcquireSRWLockSharedContended+0x15a
ntdll!RtlpxLookupFunctionTable+0x180
ntdll!RtlLookupFunctionEntry+0x4d
contoso!GetStackTrace+0x72
contoso!GetStackTraceOfUIThread+0x127
...
```

Okay, so we see that the watchdog thread is trying to get a stack trace of the UI thread, but it's hung inside `RtlLookupFunctionEntry` which is waiting for a lock.

You know who I bet holds the lock?

The UI thread.

Which is suspended.

The UI thread is probably trying to dispatch an exception, which means that it is walking the stack looking for an exception handler. But in the middle of this search, it got suspended by the watchdog thread. Then the watchdog thread tries to walk the stack of the UI thread, but it can't do that yet because the function table is locked by the UI thread's stack walk.

This is a practical exam for a previous discussion: Why you should never suspend a thread.

Specifically, the title should say "Why you should never suspend a thread *in your own process*." Suspending a thread in your own process runs the risk that the thread you suspended was in possession of some resource that the rest of the program needs. In particular, it might possess a resource that is needed by the code which has responsible for eventually resuming the thread. Since it is suspended, it will never get a chance to release those resources, and you end up with a deadlock between the suspended thread and the thread whose job it is to resume that thread.

If you want to suspend a thread and capture stacks from it, you'll have to do it from another process, so that you don't deadlock with the thread you suspended.[1]

Bonus chatter: In this kernel stack, you can see evidence that the `SuspendThread` operates asynchronously. When the watchdog thread calls `SuspendThread` to suspend the UI thread, the UI thread was in the kernel, in the middle of changing memory protections. The thread does not suspend immediately, but rather waits for the kernel to finish its work, and then before returning to user mode, the kernel does a `CheckForKernelApcDelivery` to see if there were any requests waiting. It picks up the request to suspend, and that is when the thread actually suspends.[2]

Bonus bonus chatter: "What if the kernel delayed suspending a thread if it held any user-mode locks? Wouldn't that avoid this problem?" First of all, how would the kernel even know whether a thread held any user-mode locks? There is no reliable signature for a user-mode lock. After all, you can make a user-mode lock out of any byte of memory by using it as a spin lock. Second, even if the kernel somehow could figure out whether a thread held a user-mode lock, you don't want that to block thread suspension, because that would let a program make itself un-suspendable! Just call `AcquireSRWLockShared(some_global_srwlock)` and never call the corresponding `Release` function. Congratulations, the thread perpetually owns the global lock in shared mode and would therefore now be immune from suspension.

[1] Of course, this also requires that the code that does the suspending does not wait on cross-process resources like semaphores, mutexes, or file locks, because those might be held by the suspended thread.

[2] The kernel doesn't suspend the thread immediately because it might be in possession of internal kernel locks, and suspending a thread while it owns a kernel lock (such as the lock that synchronizes access to the page tables) would result in the kernel itself deadlocking!