Why am I getting an exception from the thread pool during process shutdown?

devblogs.microsoft.com/oldnewthing/20200130-00

January 30, 2020

Raymond Chen

A customer reported that their program was experiencing crashes that suggested that they weren't handling shutdown properly. During shutdown, the code tries to acquire a lock, which leads to an exception in `TppRaiseInvalidParameter` called from `TpAllocWait`. This appeared to be all internal implementation code from the C runtime. The closest line of code in the program was an attempt to acquire a `recursive_mutex`.

```
std::lock_guard<std::recursive_mutex> guard(item->m_lock);
```

This attempt to lock a `std::recursive_mutex` led to a crash with this call stack:

```
ntdll!TppRaiseInvalidParameter+0x48
ntdll!TpAllocWait+0x86c39
kernel32!CreateThreadpoolWait+0x14
kernel32!CreateThreadpoolWaitStub+0x1a
msvcr120!Concurrency::details::RegisterAsyncWaitAndLoadLibrary+0x12
msvcr120!Concurrency::details::ExternalContextBase::PrepareForUse+0xa1
msvcr120!Concurrency::details::ExternalContextBase::ExternalContextBase+0xa2
msvcr120!Concurrency::details::SchedulerBase::GetExternalContext+0x3e
msvcr120!Concurrency::details::SchedulerBase::AttachExternalContext+0xcf
msvcr120!Concurrency::details::SchedulerBase::CreateContextFromDefaultScheduler+0xfe
msvcr120!Concurrency::details::SchedulerBase::CurrentContext+0x26
msvcr120!Concurrency::details::LockQueueNode::{ctor}+0x24
msvcr120!Concurrency::critical_section::lock+0x2a
```

What's going on?

One of the first things that the `ExitProcess` function does is terminate all the threads except the one that called `ExitProcess`. Once this happens, critical sections become electrified, and so too is the thread pool.

And in this case, you could say that we are getting electrified *twice*.

The code is attempting to acquire a `recursive_lock`, which is the C runtime verison of a critical section. The implementation is in the form of a `Concurrency::critical_section`, the implementation of which is in the `concrt.h` header file.

From reading the code, it appears that the lock is currently held by another thread, so the `critical_section` needs to schedule a wait node to keep track of everybody who is waiting. One of the things that goes into the wait node is the current context. But there is no current context (probably because the concurrency runtime has already been shut down), so the code tries to make a new one. As part of making a context, it registers a callback with the thread pool so it can be notified when the thread exits, so it can clean up.

But since there is no thread pool, we crash.

Now, maybe the C runtime folks could get rid of the crash by not talking to the thread pool, but that's addressing the symptom and not the disease. Because even if they could wait on a critical section without talking to the thread pool, your process is already in big trouble: It's waiting on a critical section that is owned by a thread that has already been terminated. All you would do is transform a crash into a hang.

The problem is that this code is being triggered by something (not shown) in the `DLL_PROCESS_DETACH` or in the destructor of an object with static storage duration. In the case of process termination, you should just give up and exit as fast as possible. Don't play any funny games, especially any funny games that involve cross-thread synchronization.

What you need to do is bypass this work when the process is terminating. There's no point sweeping the floors when the building is being demolished.

The Windows Implementation Library contains a collection of shutdown-aware objects that you can wrap around your objects. You can specify that the object's destructor should be outright bypassed, or that an alternate function should be called if the object is destructed as part of process termination.

Bonus chatter: The Visual Studio C runtime library stopped using the concurrency runtime back in Visual Studio 2015, so you won't see this specific symptom any more. But the underlying issue remains: Doing cross-thread synchronization after threads have been terminated is a bad idea.

Raymond Chen

Follow