The case of the orphaned critical section despite being managed by an RAII type

devblogs.microsoft.com/oldnewthing/20181228-00

December 28, 2018

Raymond Chen

Some time ago, I was enlisted to help debug an elusive deadlock. Studying a sampling of process memory dumps led to the conclusion that a critical section had been orphaned. Sometimes, the thread that owned the critical section had already exited, but sometimes the thread was still running, but it was running code completely unrelated to the critical section. It was as if the code that acquired the critical section had simply forgotten to release it before returning.

The thing is, all attempts to acquire the critical section were managed by an RAII type, so there should be no way that the critical section could have been forgotten. And yet it was.

When would the destructor for an RAII object by bypassed? One possibility is that somebody did an `ExitThread` or (horrors) `TerminateThread`. But this doesn't match the evidence, because as noted above, in some of the crash dumps, the critical section owner is still alive and running, but unaware that it owns the critical section.

On all platforms other than x86, exception unwind information is kept in tables in a rarely-used portion of the image, so that we don't waste memory on exception unwind information until an exception actually occurs: When an exception occurs, the system pages in the unwind tables and does a table lookup to see which unwind handler should run. But on x86, the exception unwind state is maintained manually in the code. This is a bad thing for x86 performance, but a good thing for getting inside the head of the compiler.

> Bonus reading: Unwinding the Stack: Exploring How C++ Exceptions Work on Windows.
> — James McNellis, CppCon 2018

The unwind checkpoint is a 32-bit value, usually stored at `[ebp-4]`. The compiler uses it to keep track of what needs to get unwound if an exception occurs. If the compiler can deduce that no exception can occur between two checkpoints, then it can optimize out the first checkpoint.

There are four functions that enter the critical section in question. The code that does so looks like this:

```
{
  auto guard = SystemChangeListenerCS.Lock();
  ... some code ...
} // guard destructor releases the lock
```

Finding the exact point where the guards are created is made easier with the assistance of the `#` debugger command, which means "Disassemble until you see this string in the disassembly."

```
0:000> #SystemChangeListenerCS SystemChangeListenerThreadProc
SystemChangeListenerThreadProc+0x7c:
1003319c mov     ecx,offset SystemChangeListenerCS (100b861c)
0:000>
```

Okay, so the debugger found a line of assembly that mentions `SystemChangeListenerCS`. Let's look to see whether there is an unwind checkpoint after the lock is taken.

```
0:000> u 1003319c
ChangeMonitorThreadProc+0x7c:
1003319c mov     ecx,offset contoso!SystemChangeListenerCS (100b861c)
100331a1 push    eax
100331a2 call    Microsoft::WRL::Wrappers::CriticalSection::Lock (1002a863)
100331a7 mov     byte ptr [ebp-4],5
```

We see that immediately after acquiring the lock, the code updates `[ebp-4]` to remember that it needs to destruct the lock guard in case an exception occurs.

Exercise: I said that the unwind state is recorded in a 32-bit value stored at `[ebp-4]`, but the code here updates only a byte. Why only a byte?

The lock is acquired again later in that same function, so we'll search some more. If you leave off the second parameter to the `#` command, it continues searching where the previous search left off.

```
0:000> #SystemChangeListenerCS
SystemChangeListenerThreadProc+0x487:
100335a7 mov     ecx,offset contoso!SystemChangeListenerCS (100b861c)
0:000> u 100335a7
contoso!SystemChangeListenerThreadProc+0x487:
100335a7 mov     ecx,offset contoso!SystemChangeListenerCS (100b861c)
100335ac push    eax
100335ad call    Microsoft::WRL::Wrappers::CriticalSection::Lock (1002a863)
100335b2 mov     byte ptr [ebp-4],0Dh
```

Okay, so this lock guard is also marked for unwinding.

The next function that uses the critical section is `ResetWidgets`.

```
0:000> #SystemChangeListenerCS ResetWidgets
ResetWidgets+0x133:
10033fcc mov     ecx,offset SystemChangeListenerCS (100b861c)
0:000> u 10033fcc l4
ResetWidgets+0x133:
10033fcc mov     ecx,offset SystemChangeListenerCS (100b861c)
10033fd1 push    eax
10033fd2 call    Microsoft::WRL::Wrappers::CriticalSection::Lock (1002a863)
10033fd7 call    Microsoft::WRL::ComPtr<IStream>::Reset (10039932)
10033fdc call    Microsoft::WRL::ComPtr<Widget>::Reset (10039142)
10033fe1 cmp     dword ptr [ebp-4Ch],0
10033fe5 je      ResetWidgets+0x157 (10033ff0)
10033fe7 push    dword ptr [ebp-4Ch]
```

Hm, this function doesn't create an unwind checkpoint after taking the lock. This means that the compiler believes that no exception can occur between the point the guard is created and the next thing that would require updating the unwind checkpoint (in our case, that would be the point the lock is destructed).

We repeat this analysis with the other two functions. One of them creates an unwind checkpoint; the other doesn't.

Why does the compiler believe that no exceptions can occur in the guarded block? Well, inside the block it calls `ComPtr::Reset` twice, and it does some other stuff. The `Reset` method is declared like this:

```
template<typename T>
class ComPtr {
unsigned long Reset() { return InternalRelease(); }
unsigned long InternalRelease() throw() { ... }
...
};
```

Observe that the `InternalRelease` method uses the deprecated `throw()` specifier, which says that the method never throws an exception. The compiler then inferred that the `Reset` method also never throws an exception, since it does nothing that could result in an exception.

This code was compiled before the Microsoft C++ compiler added the `/std:C++17` switch, so it uses the old rules for the `throw()` specifier, which for the Microsoft C++ compiler boils down to “I'm trusting you never to throw an exception.”

My theory is that the `Reset` actually did throw an exception. Since the compiler didn't create an unwind checkpoint, the lock guard did not get unwound. The exception was caught higher up the stack, so the process didn't crash.

Digging into the two objects wrapped inside the `ComPtr` revealed that the first one was a `WidgetMonitor` object.

Exercise: The first was really an `IWidgetMonitor` interface, so why did it disassemble as `ComPtr<IStream>` ?

The `WidgetMonitor` 's destructor went like this:

```
WidgetMonitor::~WidgetMonitor()
{
 Uninitialize();
}

void WidgetMonitor::Uninitialize()
{
 blah blah;
 ThrowIfFailed(m_monitor.Deactivate());
 blah blah;
 ThrowIfFailed(m_monitor.Disconnect());
 blah blah;
}
```

Now you see the problem. If the `Uninitialize` method throws an exception, the exception will propagate out of the destructor. (This code is so old that it predates C++11's rule that destructors are `noexcept` by default where possible.) And then it will propagate out of `ComPtr:: InternalRelease` , and then out of `ComPtr:: Reset` , and then out of `ResetWidgets` . And unwinding out of `ResetWidgets` will not run the lock guard's destructor because the compiler assumed that no exception could be thrown, thanks to the `throw()` specifier on the `ComPtr:: InternalRelease` method.

As is often the case, it's usually a lot easier to find something once you know what you're looking for. The team dug into its telemetry to see that, yes indeed, the systems that encountered the problem had also thrown an exception from `Widget-Monitor:: Uninitialize` , thus confirming the theory.

Now they could work on fixing the problem: Fix the destructor so it doesn't throw any exceptions. In this specific case, the exception was thrown because they were deactivating an object that hadn't been fully activated. Since cleanup functions cannot fail, the best you can do is to just soldier on and clean up as much as you can.

Raymond Chen

Follow