The case of the crash in a C++/WinRT coroutine: Unpeeling the onion

devblogs.microsoft.com/oldnewthing/20230505-00

May 5, 2023

Raymond Chen

A customer had a mysterious crash in a C++/WinRT coroutine, and they were having trouble identifying the cause. The stack of the crash looked like this:

```
Contoso!winrt::impl::consume_Contoso_IWidgetWatcher<winrt::Contoso::IWidgetWatcher>::
NotifyChange+0x3
Contoso!winrt::Contoso::implementation::WidgetController::SaveWidget+0x16
Contoso!winrt::impl::produce<winrt::Contoso::implementation::WidgetController,winrt::
Contoso::IWidgetController>::SaveWidget+0x1d
Contoso!winrt::impl::consume_Contoso_IWidgetController<winrt::Contoso::
IWidgetController>::SaveWidget+0xd
Contoso!winrt::Contoso::implementation::Widget::OnSettingsChanged$_ResumeCoro$1+0x31f
Contoso!std::experimental::coroutine_handle<void>::resume+0xc
Contoso!winrt::impl::await_adapter<winrt::Windows::Foundation::IAsyncAction>::await_
suspend+0x15e
Contoso!winrt::Contoso::implementation::Widget::OnSettingsChanged$_ResumeCoro$1+0x11a
Contoso!winrt::Contoso::implementation::Widget::OnSettingsChanged$_InitCoro$2+0x6a
Contoso!winrt::Contoso::implementation::Widget::OnSettingsChanged+0x5c
Contoso!winrt::Windows::Foundation::TypedEventHandler<winrt::Contoso::Widget,winrt::
Windows::Foundation::IInspectable>::<lambda_a7902c7784a1de3d47473a11e43d997c>::
operator()+0x5b
Contoso!winrt::impl::delegate<winrt::Windows::Foundation::TypedEventHandler<winrt::
Contoso::Widget,winrt::Windows::Foundation::IInspectable>,<lambda_
a7902c7784a1de3d47473a11e43d997c> >::Invoke+0x6d
rpcrt4!Invoke+0x73
rpcrt4!Ndr64StubWorker+0xb8a
rpcrt4!NdrStubCall3+0xd3
combase!CStdStubBuffer_Invoke+0x6f
combase!InvokeStubWithExceptionPolicyAndTracing::__l6::<lambda_
c9f3956a20c9da92a64affc24fdd69ec>::operator()+0x22
combase!ObjectMethodExceptionHandlingAction<<lambda_c9f3956a20c9da92a64affc24fdd69ec>
>+0x4d
combase!InvokeStubWithExceptionPolicyAndTracing+0xe1
combase!DefaultStubInvoke+0x268
combase!SyncServerCall::StubInvoke+0x41
combase!StubInvoke+0xf6
combase!ServerCall::ContextInvoke+0x366
combase!ComInvokeWithLockAndIPID+0x9aa
combase!ThreadInvokeReturnHresult+0x17b
combase!ThreadInvoke+0x193
rpcrt4!DispatchToStubInCNoAvrf+0x22
rpcrt4!RPC_INTERFACE::DispatchToStubWorker+0x1b4
rpcrt4!RPC_INTERFACE::DispatchToStub+0xb3
rpcrt4!RPC_INTERFACE::DispatchToStubWithObject+0x188
rpcrt4!LRPC_SBINDING::DispatchToStubWithObject+0x23
rpcrt4!LRPC_SCALL::QueueOrDispatchCall+0x253
rpcrt4!LRPC_SCALL::HandleRequest+0x996
rpcrt4!LRPC_SASSOCIATION::HandleRequest+0x2c3
rpcrt4!LRPC_ADDRESS::HandleRequest+0x17c
rpcrt4!LRPC_ADDRESS::ProcessIO+0x939
rpcrt4!LrpcIoComplete+0x109
ntdll!TppAlpcpExecuteCallback+0x157
ntdll!TppWorkerThread+0x72c
kernel32!BaseThreadInitThunk+0x1d
ntdll!RtlUserThreadStart+0x28
```

```
Contoso!winrt::impl::consume_Contoso_IWidgetWatcher<winrt::Contoso::IWidgetWatcher>::
NotifyChange+0x3:
00007ffd`ab49749d mov     rax,qword ptr [rcx] ds:00000000`00000000=????????????????
```

This is a big, ugly stack, but we can simplify it:

```
Contoso!winrt::impl::consume_IWidgetWatcher::NotifyChange+0x3
Contoso!winrt::WidgetController::SaveWidget+0x16
Contoso!winrt::impl::produce<IWidgetController>::SaveWidget+0x1d
Contoso!winrt::impl::consume_IWidgetController::SaveWidget+0xd
Contoso!winrt::Widget::OnSettingsChanged$_ResumeCoro$1+0x31f
Contoso!std::experimental::coroutine_handle<void>::resume+0xc
Contoso!winrt::impl::await_adapter::await_suspend+0x15e
Contoso!winrt::Widget::OnSettingsChanged$_ResumeCoro$1+0x11a
Contoso!winrt::Widget::OnSettingsChanged$_InitCoro$2+0x6a
Contoso!winrt::Widget::OnSettingsChanged+0x5c
Contoso!winrt::TypedEventHandler::<lambda_...>::operator()+0x5b
Contoso!winrt::impl::delegate::Invoke+0x6d
(external event machinery)
```

Reading from the bottom up, we start with a bunch of external event machinery, which led to the invocation of an event handler, evidently the `OnSettingsChanged` handler.

This handler is a coroutine, and it awaited something, and then upon resumption it called the `WidgetController::SaveWidget` method, which called `WidgetWatcher::NotifyChange`, but it crashed trying to make that call because the widget watcher is null. (We know this because `rcx` is null, and we're trying to read its vtable.)

The customer's `OnSettingsChanged` handler looked like this:

```
fire_and_forget Widget::OnSettingsChanged(
    const SettingsManager&, const IInspectable&)
{
    co_await ApplySettingsAsync();
    m_widgetController.SaveWidget(*this);
}
```

The customer saw this code and smacked their forehead. "Of course, the problem is that I forgot to hold a strong reference to the widget across the `co_await`."

```
winrt::fire_and_forget Widget::OnSettingsChanged(
    const winrt::SettingsManager&, const winrt::IInspectable&)
{
    auto lifetime = get_strong();
    co_await ApplySettingsAsync();
    m_widgetController.SaveWidget(*this);
}
```

But their fix didn't help. The crashes kept coming.

Let's look at that crash stack again:

```
Contoso!winrt::Widget::OnSettingChanged$_ResumeCoro$1+0x31f
Contoso!std::experimental::coroutine_handle<void>::resume+0xc
Contoso!winrt::impl::await_adapter::await_suspend+0x15e
Contoso!winrt::Widget::OnSettingChanged$_ResumeCoro$1+0x11a
```

Reading upward, the coroutine was executing, and then decided to `co_await` something. The coroutine machinery called `await_suspend`, and the `await_suspend` immediately resumed the coroutine, causing `OnSettingChanged$_ResumeCoro$1` to be re-entered.

So the coroutine never really suspended. Adding a strong reference wouldn't have helped here, since the strong reference's job is to keep the object alive across a suspension, which hasn't happened. (I mean, the strong reference is still required in the case where the `co_await` does suspend. I'm just saying that this particular crash didn't involve a suspension.)

Another way to observe that there was no suspension is that the stack trace leads all the way back to the external event machinery. We are still in the synchronous portion of the event handler. Nothing has suspended yet.

I looked at how the event was registered:

```
struct Widget : WidgetT<Widget>
{
    // other constructor parameters elided for expository purposes
    Widget::Widget(const WidgetController& controller)
        : m_weakController{ controller }
    {
        m_SettingChangedRevoker =
             m_settingsWatcher.SettingChanged(winrt::auto_revoke,
                { this, &Widget::OnSettingChanged });
    }

private:
    fire_and_forget OnSettingChanged(
        const SettingsManager&, const IInspectable&);

    const weak_ref<WidgetController> m_weakWidgetController;
    const SettingsWatcher m_settingsWatcher;
    SettingsWatcher::SettingChanged_revoker m_SettingChangedRevoker;

    // other members elided for expository purposes
};
```

Notice that the event handler is registered with a raw `this` pointer.

Registering with a raw `this` pointer means that you are taking full responsibility for ensuring that the object is alive at the time the event is received. This is manageable for events that are raised synchronously (such as UI events), since you can make sure to unregister the event handler from the thread that will raise the event, avoid the race condition where you unregister the event handler while the handler is running (or is irrevocably committed to running).

Thread 1	Thread 2
event triggered	
handler()	
	unregister handler
handler still running!	

We can see from the stack that the `SettingsWatcher` object raises events from a background thread, so a raw `this` capture is not safe. If the event is unregistered while a callback is in progress (or is irrevocably committed to running), the handler will have the object destructed out from under it (or possibly even have it destructed before it can execute a single instruction).

In this case, the registration of the handler should be done with a weak reference.

```
        m_SettingChangedRevoker =
             m_settingsWatcher.SettingChanged(winrt::auto_revoke,
                { get_weak(), &Widget::OnSettingChanged });
```

The C++/WinRT weak reference event handler pattern goes like this:

```
auto strong = weak.get():
if (strong) { (strong->*handler)(); }
```

C++/WinRT tries to upgrade the weak reference to a strong reference, and if successful, it calls the event handler with the strong reference active. When the handler returns, the strong reference is released.[1]

This particular race condition is between the event handler and the revocation of the event. This particular object doesn't revoke the event until it destructs, and the way the program uses the Widget, we don't expect it to be destroyed until the program shuts down, and there's no evidence that the program is shutting down.

The "object got destructed before your handler run" scenario also doesn't seem to match the crash dump:

```
0:019> ?? ((winrt::Contoso::implementation::WidgetController*) 0x00000262`1b9ce810)
struct winrt::Contoso::implementation::WidgetController * 0x00000262`1b9ce810
   +0x010 vtable           : ...
   +0x018 vtable           : ...
   +0x000 __VFN_table : 0x00007ffd`ab59d940
   +0x008 m_references     : std::atomic<unsigned __int64>
    ...
   +0x0a0 m_widgetWatcher : winrt::Contoso::WidgetWatcher

0:019> ?? ((winrt::Contoso::implementation::WidgetController*) 0x00000262`1b9ce810)-
>m_references._Storage
struct std::_Atomic_padded<unsigned __int64>
   +0x000 _Value           : 0x80000131`0e28d450

0:019> dps 0x80000131`0e28d450*2
00000262`1c51a8a0  00007ffd`ab5998d8 Contoso!winrt::impl::weak_ref<1,1>::`vftable'
00000262`1c51a8a8  00007ffd`ab5998b8 Contoso!winrt::impl::weak_source<1,1>::`vftable'
00000262`1c51a8b0  00000262`1b9ce820 // m_object - pointer to original object
(matches)
00000262`1c51a8b8  0000002e`00000006 // weak + strong references
```

The weak reference control block seems to be valid, and it shows a reasonable non-zero number of strong references, so it doesn't seem that we're in the "Object destructed while handler is running" scenario.

So we found another bug, but not the bug that caused this crash.

The crash is at the call to `m_widgetWatcher.SaveWidget()`, so let's look at that `m_widgetWatcher`.

```
0:019> ?? ((winrt::Contoso::implementation::WidgetController*) 0x00000262`1b9ce810)-
>m_widgetWatcher
struct winrt::Contoso::WidgetWatcher
   +0x000 m_ptr            : 0x00000262`1b86ec60
0:019> dps 0x00000262`1b86ec60 L2
00000262`1b86ec60  00007ffd`ab5a1728
Contoso!winrt::impl::produce<WidgetWatcher,IWidgetWatcher>::`vftable'
00000262`1b86ec68  00000000`00000000
```

Wait a second, the crash dump says that we crashed because the `m_widgetWatcher` is null, but in the dump, the `m_widgetWatcher` is non-null. This suggests to me that we encountered a race condition, where the `m_widgetWatcher` was null at the time of the crash, but in the time it took to create the crash dump, the value was updated.

This helped focus the next step of the investigation: Let's look at the `WidgetController` and how it initializes the `WidgetWatcher`.

```
struct WidgetController : WidgetControllerT<WidgetController>
{
    // constructor parameters elided for expository purposes
    WidgetController(const WidgetWatcher& watcher) :
        m_widget{ make<Widget>(*this) },
        m_widgetWatcher{ watcher }
    {
    }

    void SaveWidget(const Widget&);

private:
    const Widget m_widget;
    const WidgetWatcher m_widgetWatcher;

    // other members elided for expository purposes
};
```

Now the story comes into focus.

The non-static data members of C++ objects are constructed in order of declaration in the class.[2] In this case, we construct the `Widget` before we copy the `WidgetWatcher`.

If a setting change occurs immediately after the `Widget` is constructed, then we have a race between the event callback thread (which will call back into the `WidgetController`) and the construction of the `WidgetController`'s `m_widgetWatcher`. If the event callback thread wins the race, then it will see a not-yet-constructed `m_widgetWatcher` and crash.

That is the race condition that we hit. The event callback thread tried to use a not-yet-constructed object.

Therefore, the final step of the fix is to force the `m_widgetWatcher` to construct first:

```
struct WidgetController : WidgetControllerT<WidgetController>
{
    // constructor parameters elided for expository purposes
    WidgetController(const WidgetWatcher& watcher) :
        m_widget{ make<Widget>(*this) },
        m_widgetWatcher{ watcher }
    {
    }

    void SaveWidget(const Widget&);

private:
    // Order of declaration is important:
    // WidgetWatcher must construct before the Widget,
    // because the Widget may call into the WidgetWatcher.
    const WidgetWatcher m_widgetWatcher;
    const Widget m_widget;

    // other members elided for expository purposes
};
```

This was a rather long investigation, with two red herrings! I don't know whether you enjoyed it, but you can't get that time back now.

[1] Note that if the handler is a coroutine, the handler "returns" at the point the coroutine first suspends, not when the coroutine runs to completion. That's why we need the `get_strong()` inside the coroutine body: To keep the object alive through to completion.

[2] Note that the order in which the initializers are given in the constructor are irrelevant. It is the order of declaration in the class that controls the order of construction.

```
struct Example
{
    Example() : b{ 1 }, a{ 2 } {}

    Thing1 a;
    Thing2 b;
};
```

In this example class, the `a` member is constructed first, and then the `b` member is constructed second. You might think that `b` is constructed first because it is listed first in the constructor, but that's not the case.

If construction occurred in order of initialization, and two constructors initialized the members in different orders, then the destructor would be unable to follow the rule that objects are destructed in the same order of construction, because the destructor doesn't know which constructor was used.[3]

[3] I guess you could solve this problem by adding a hidden member variable that keeps track of which order the members were constructed, but this adds runtime costs and would likely be surprising to programmers that it's possible that adding a constructor to a class, even one that is never called, can change its size.