Understanding a mysterious RPC_E_WRONGTHREAD exception when we're on the right thread

devblogs.microsoft.com/oldnewthing/20230224-37

February 24, 2023

Raymond Chen

Last time, we were looking a customer whose code tried to catch all exceptions, but accidentally raised an exception from the code that handled all exceptions. (You had one job!)

The code in question looked like this:

```
winrt::fire_and_forget MyClass::DoSomethingAsync()
{
    auto lifetime = get_strong();
    try {
        auto name = co_await m_user.GetNameAsync();
        m_label.Text(name);
    } catch (...) {
        m_label.Text(L"unknown");
    }
}
```

The obvious exception that this code was trying to catch was an exception from `GetName-Async`, and it handled it by just reporting the name as "unknown" if the name could not be obtained.

But the problem was that its recovery code threw an exception too!

In this particular case, the exception was `RPC_E_WRONGTHREAD`. The customer noted that the crash occurred if the user closed their XAML window while the `GetNameAsync` was still running.

Okay, with that clue, it becomes possible to develop some theories as to what happened.

The `m_label` here is a XAML text block, and XAML objects have thread affinity. It's apparent that the call starts on the correct thread, because in the cases where the user doesn't close the XAML window, the code executes successfully without an exception.

The other case that you get a `RPC_E_WRONGTHREAD` exception from XAML is if you try to use XAML from a thread on which XAML is not initialized. And that fits the scenario here: The user closed the XAML window, so the app cleans up XAML for the thread that was hosting the window, and then the `GetNameAsync` operation completes, and the code resumes executing on a thread that has had XAML uninitialized out from under it.

It then attempts to set the name into the `m_label` that fails with `RPC_E_WRONGTHREAD`: You can't do XAML things on this thread any more.

The exception is caught, and the exception handler recovers by… trying to set the name into the `m_label` again! The anthropomorphized XAML runtime says, "C'mon, man, like I just told you to stop doing that," and throws `RPC_E_WRONGTHREAD` a second time.

Another possible scenario is that the thread that hosted the XAML window not only shut down XAML on that thread, but in fact managed to exit entirely. The coroutine machinery in C++/WinRT follows the same policy as C# and PPL and resumes Windows Runtime operations on the same COM apartment on which they started. But in this case, the original apartment is gone. In that case, the attempt to get back to the original context fails with `RPC_E_DISCONNECTED`, because all of the references to the original apartment got disconnected as part of apartment rundown. We looked at this problem a little while ago.

In this customer's case, they can just abandon the label-setting work if the attempt to set the fallback label fails, and the solution from last time is appropriate: Add a second catch block to catch and ignore exceptions from the exception handler.

Bonus chatter: From the crash stack, you can observe which of the two cases we are in:

```
contoso!std::experimental::coroutine_handle<void>::operator()+0x5
contoso!winrt::impl::resume_apartment_callback+0x9
...
```

The coroutine was resumed from `resume_apartment_callback`, which is the function that C++/WinRT passes to `IContextCallback::ContextCallback`. This means that we did successfully get back to the original apartment, and the problem was that the coroutine is trying to access XAML from a thread that has already uninitialized XAML.

If the problem was that C++/WinRT could not get back to the correct apartment, then the coroutine would have resumed from `resume_apartment_sync`.

Bonus bonus chatter: In the case where C++/WinRT cannot get back to the correct apartment, it will throw the `RPC_E_DISCONNECTED` from whatever thread it happens to be on when it gets stuck. In that case, the exception handler may find itself running on the wrong apartment.

You can try to detect this case by checking what apartment you are running in, but one of the design principles of the Windows Runtime is that recoverable errors should not be reported via exceptions, but rather by reporting an error in some in-API manner, like a status code or a `Succeeded` property. If you are catching an exception, then something terrible has probably happened and you may just want to fail fast and stop before things get out of control.

Bonus bonus bonus chatter: Some of the older Windows Runtime classes still use exceptions to report recoverable errors, but we are slowly updating them to provide non-exceptional versions, like `HttpClient.TryGetAsync` as a non-exceptional alternative to `HttpClient.GetAsync`.