The case of the RPC_E_DISCONNECTED error thrown from await_resume

devblogs.microsoft.com/oldnewthing/20230210-00

February 10, 2023

Raymond Chen

A customer was studying some crashes with the following call stack, with function names abbreviated for readability:

```
 # Call Site
00 KERNELBASE!RaiseFailFastException+0x152
01 combase!RoFailFastWithErrorContextInternal2+0x4d9
02 Contoso!winrt::terminate+0x64
03 Contoso!std::coroutine_traits<winrt::fire_and_forget,〚…〛
>::promise_type::unhandled_exception+0x9
04
contoso!`Contoso::UserInfoViewModel::LoadUserInfoAsync$_ResumeCoro$1'::`1'::catch$128+

05 ucrtbase!_CallSettingFrame_LookupContinuationIndex+0x20
06 ucrtbase!__FrameHandler4::CxxCallCatchBlock+0x115
07 ntdll!RcFrameConsolidation+0x6
08 Contoso!Contoso::UserInfoViewModel::LoadUserInfoAsync$_ResumeCoro$1+0x502
09 Contoso!std::coroutine_handle<void>::resume+0x16
0a Contoso!std::coroutine_handle<void>::operator()+0x16
0b Contoso!winrt::impl::resume_apartment_sync+0x7f
0c Contoso!winrt::impl::resume_apartment+0xa4
0d Contoso!winrt::impl::disconnect_aware_handler<〚…〛>::operator()+0xd
0e Contoso!winrt::impl::delegate<〚…〛>::Invoke+0x15
0f Contoso!winrt::Windows::Foundation::AsyncActionCompletedHandler::operator()+0x24
10 Contoso!winrt::impl::invoke<〚…〛>+0x17
11 Contoso!winrt::impl::promise_base<〚…〛>::set_completed+0x59
12 Contoso!winrt::impl::promise_base<〚…〛>::final_suspend_awaiter::await_suspend+0x74
13
Contoso!Contoso::UserInfoViewModel::LoadUserNameAndPictureAsync$_ResumeCoro$1+0x715
14 Contoso!std::coroutine_handle<void>::resume+0x16
15 Contoso!std::coroutine_handle<void>::operator()+0x16
16 Contoso!winrt::impl::resume_apartment_sync+0x7f
17 Contoso!winrt::impl::resume_apartment+0xa4
18 Contoso!winrt::impl::disconnect_aware_handler<〚…〛>::operator()+0xd
19 Contoso!winrt::impl::delegate<〚…〛>::Invoke+0x15
1a Contoso!winrt::Windows::Foundation::AsyncOperationCompletedHandler<〚…〛
>::operator()+0x24
1b Contoso!winrt::impl::invoke<〚…〛>+0x17
1c Contoso!winrt::impl::promise_base<〚…〛>::set_completed+0x59
1d Contoso!winrt::impl::promise_base<〚…〛>::final_suspend_awaiter::await_suspend+0x74
1e Contoso!Contoso::UserInfoModel::LoadUserNameAsync$_ResumeCoro$1+0x1ea
1f Contoso!std::coroutine_handle<void>::resume+0x16
20 Contoso!std::coroutine_handle<void>::operator()+0x16
21 Contoso!winrt::impl::resume_apartment_sync+0x7f
22 Contoso!winrt::impl::resume_apartment+0xa4
23 Contoso!winrt::impl::disconnect_aware_handler<〚…〛>::operator()+0xd
24 Contoso!winrt::impl::delegate<〚…〛>::Invoke+0x15
25 usermgrproxy!Windows::Internal::Async::HardenedAsyncBase<〚…〛>::FireCompletion+0x154
26 usermgrproxy!AsyncOperationBase<〚…〛>::DoWorkStub+0xd8
27 ntdll!RtlpTpWorkCallback+0x192
28 ntdll!TppWorkerThread+0x639
29 kernel32!BaseThreadInitThunk+0x1d
2a ntdll!RtlUserThreadStart+0x28
```

They assumed that they were running on the wrong thread, so their proposed solution was to add a

```
co_await resume_foreground(Dispatcher());
```

after every `co_await` to make sure they were on the correct thread. But first, let's see what's really going on.

Reading from the bottom up, we see that at stack frame 25:

```
25 usermgrproxy!Windows::Internal::Async::HardenedAsyncBase<⟦…⟧>::FireCompletion+0x154
```

The asynchronous operation completed, and therefore it calls its completion delegate:

```
24 Contoso!winrt::impl::delegate<⟦…⟧>::Invoke+0x15
```

The completion delegate is a C++/WinRT completion, so it goes to the disconnect-aware handler: which we

```
23 Contoso!winrt::impl::disconnect_aware_handler<⟦…⟧>::operator()+0xd
```

Upon completion, the C++/WinRT library tries to resume the awaiting coroutine in the original COM apartment context in which the `co_await` began:

```
20 Contoso!std::coroutine_handle<void>::operator()+0x16
21 Contoso!winrt::impl::resume_apartment_sync+0x7f
22 Contoso!winrt::impl::resume_apartment+0xa4
```

They key thing to observe here is that we are making a call from `resume_apartment_sync` directly to the awaiting coroutine. This happens only when the apartment switch has failed, and we are intentionally resuming the coroutine in the wrong apartment so we can report the error.

Let's see what the error is.

```
0:024> .frame 21
21 Contoso!winrt::impl::resume_apartment_sync+0x7f
0:024> dv
        context = <value unavailable>
         handle = struct std::experimental::coroutine_handle<void>
        failure = 0x000001c8`6f2f0f44
           args = struct winrt::impl::com_callback_args
         result = 0n-45104688
0:024> ?? *failure
int 0n-2147417848
```

The signed decimal number `-2147417848` corresponds to the error `0x80010108` which is `RPC_E_DISCONNECTED`. This means that we were unable to switch apartments because the destination apartment no longer exists.

At this point, the code starts to spiral.

The calling coroutine is

```
1e Contoso!Contoso::UserInfoModel::LoadUserNameAsync$_ResumeCoro$1+0x1ea
```

and it doesn't handle the exception. Therefore, the exception gets stowed in the promise, and then the coroutine completes and calls its completion delegate to resume the awaiting coroutine. The completion delegate is another disconnect-aware handler:

```
18 Contoso!winrt::impl::disconnect_aware_handler<〚…〛>::operator()+0xd
```

As before, the C++/WinRT library tries to resume the awaiting coroutine in the original COM apartment context in which the `co_await` began:

```
15 Contoso!std::coroutine_handle<void>::operator()+0x16
16 Contoso!winrt::impl::resume_apartment_sync+0x7f
17 Contoso!winrt::impl::resume_apartment+0xa4
```

Now, it turns out that all of these coroutines run on the UI thread, so in each case, the C++/WinRT library is trying to return to the UI thread, and the apartment switch fails, and the cycle continues: The call from `resume_apartment_sync` directly to the awaiting coroutine means that we were once again unable to switch to the original COM apartment. Again the caller doesn't handle this exception, so the exception is stowed in the promise, and then the caller completion delegate is called.

One more turn of the cycle: That completion delegate fails to switch to the original COM apartment and once again resumes the caller directly in order to report the error:

```
0a Contoso!std::coroutine_handle<void>::operator()+0x16
0b Contoso!winrt::impl::resume_apartment_sync+0x7f
0c Contoso!winrt::impl::resume_apartment+0xa4
0d Contoso!winrt::impl::disconnect_aware_handler<〚…〛>::operator()+0xd
```

The proposed fix of switching to the UI thread won't help. In fact, switching to the UI thread just repeats the problem.

The buck finally stops at `UserInfoViewModel::LoadUserInfoAsync`, because that is a `fire_and_forget` coroutine, and `fire_and_forget` coroutines treat unhandled exceptions as fatal errors, so the `fire_and_forget` promise calls `winrt::terminate` when an unhandled exception occurs, and that terminates the process.

To fix the problem, we first need to decide what we should do if the UI thread terminates while our asynchronous work is still running. In this particular case, the asynchronous work is retrieving data in order to update the UI, so if the UI no longer exists, then there's no point doing the asynchronous work, and we can just abandon it. Therefore, we can add a catch block to the outer `fire_and_forget` coroutine to catch and ignore disconnected errors.

```
winrt::fire_and_forget UserInfoViewModel::LoadUserInfoAsync() try
{
    ⟦ do asynchronous work to update the UI ⟧
}
catch (...)
{
    // Ignore disconnected errors, but anything else is fatal.
    // Disconnected errors mean that our UI thread no longer exists.
    if (winrt::to_hresult() != RPC_E_DISCONNECTED)
    {
        winrt::terminate();
    }
}
```