Inside C++/WinRT: Apartment switching: Error reporting

devblogs.microsoft.com/oldnewthing/20230131-00

January 31, 2023

Raymond Chen

So far, we've been looking at how C++/WinRT handles apartment switching, and I noted that everything works when it works. But what if it doesn't work?

Recall that the core of the apartment-switching code is this function:

```
void resume_apartment_sync(
    com_ptr<IContextCallback> const& context,
    std::coroutine_handle<> handle)
{
    com_callback_args args{};
    args.data = handle.address();

    check_hresult(
        context->ContextCallback(resume_apartment_callback,
            &args,
            guid_of<ICallbackWithNoReentrancyToApplicationSTA>(),
            5, nullptr));
}
```

If the `ContextCallback` method fails, `check_hresult` will throw a C++/WinRT exception to whoever is calling.

In the case of `co_await`'ing an `apartment_context`, the caller is the `await_suspend()` that is running in the context of the calling coroutine, so the caller can handle (or not handle) the exception as it sees fit.

The case that doesn't work is the case where we are trying to return to the original apartment context when an awaited coroutine completes. In that case, the exception is thrown from the completion handler, which runs in the context of the completed coroutine, rather than the context of the resuming coroutine. That means that a failure to return to the original context is not catchable by the caller:

```
winrt::IAsyncAction Outer()
{
    co_await Inner();
}
```

After `Inner()` completes, we try to return to the original COM context of `Outer()`, but if that fails, the exception is thrown in the `Completed` handler that we passed to `Inner`. The `Outer` never gets to see it. The `Outer` coroutine never resumes, which manifests itself as a hung coroutine (that is also leaked).

To fix this, we need to resume the `Outer` coroutine, and then throw the exception as part of the execution of `Outer`. The `resume_apartment_sync()` function is not running in the context of the `Outer`, so it can't throw the exception yet. It has to save the error, so it can be thrown later.

```
inline void resume_apartment_sync(
    com_ptr<IContextCallback> const& context,
    coroutine_handle<> handle,
    int32_t* failure)
{
    com_callback_args args{};
    args.data = handle.address();
    auto result =
        context->ContextCallback(resume_apartment_callback,
            &args,
            guid_of<ICallbackWithNoReentrancyToApplicationSTA>(),
            5, nullptr);

    if (result < 0) {
        *failure = result;
        handle();
    }
}
```

If we are unable to resume the coroutine in the correct apartment, then we record the failure in the caller-provided location and then *resume the coroutine anyway* on the wrong thread. The expectation is that upon resumption, the coroutine will check that location and see that the apartment-switch failed and re-throw the exception, this time while inside the execution context of the `Outer`.

Exercise: Why does `resume_apartment_sync()` update `*failure` only if `Context-Callback` failed? Shouldn't we update it on success, too?

```
    *failure =
        context->ContextCallback(resume_apartment_callback,
            &args,
            guid_of<ICallbackWithNoReentrancyToApplicationSTA>(),
            5, nullptr);
    if (*failure < 0) {
        handle();
    }
```

The answer to the exercise is at the end of this article.

We now need to teach our callers to call `resume_apartment_sync` in the new way:

```
struct apartment_awaiter
{
    apartment_context const& context;
    int32_t failure = 0;

    bool await_ready() const noexcept
    {
        return false;
    }

    void await_suspend(coroutine_handle<> handle)
    {
        apartment_context extend_lifetime = context;
        resume_apartment(context.context, handle,
            &failure);
    }

    void await_resume() const // noexcept
    {
        check_hresult(failure);
    }
};
```

When the coroutine resumes, it calls `await_resume()`, and that is where we check whether the apartment switch was successful. If not, we throw an exception from `await_resume()`, which is running in the context of `Outer` and therefore can be caught and reported like any other exception that occurs in a coroutine.

We do the same thing for coroutine resumption after `co_await`'ing a Windows Runtime asynchronous operation.

```
template <typename Async>
struct await_adapter
{
    await_adapter(Async const& async) : async(async) { }

    Async const& async;
    int32_t failure = 0;

    bool await_ready() const noexcept
    {
        return false;
    }

    void await_suspend(coroutine_handle<> handle) const
    {
        auto extend_lifetime = async;
        async.Completed([
            handle,
            this,
            context = resume_apartment_context()
        ](auto&& ...)
        {
            resume_apartment(context.context, handle,
                &failure);
        });
    }

    auto await_resume() const
    {
        check_hresult(failure);
        return async.GetResults();
    }
};
```

Things are getting better, but there is still room for improvement. We'll continue our study next time.

Answer to exercise: We cannot store the answer into `*failure`, because a successful call to `ContextCallback` resumes the coroutine. When the coroutine resumes, it calls `await_resume()` and then destructs the awaiter. If we had updated `*failure` on success, we risk writing to an already-destructed object and corrupting memory.