C++ coroutines: Awaiting an IAsyncAction without preserving thread context

devblogs.microsoft.com/oldnewthing/20191212-00

December 12, 2019

Raymond Chen

The C++/WinRT library provides an awaiter for Windows Runtime asynchronous activities. Those asynchronous activities are represented by `IAsyncAction`, `IAsyncOperation`, and progress versions of the above. The C++/WinRT-provided awaiter resumes execution of the caller in the same COM apartment that awaited the activity.

Here's a refresher on COM apartments. If you don't want to read it, then a simplified version is to say that it resumes execution in the same UI context. If you perform the `co_await` on a UI thread, then when the asynchronous activity completes, the caller resumes execution on the same UI thread. If you perform the `co_await` on a background thread, then the caller resumes execution on a background thread (possibly not the exact same thread that initiated the operation).

But maybe you don't need to resume in the same apartment. Your code is fine with running in any apartment, How can you `co_await` a Windows Runtime asynchronous activity and resume execution on any thread?

With a custom awaiter, of course.

```
template<typename Async>
[[nodiscard]] auto resume_in_any_apartment(Async async)
{
  struct awaiter : std::experimental::suspend_always
  {
    awaiter(Async async_) : async(std::move(async_)) { }

    void await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      async.Completed([handle](auto&&...) { handle(); });
    }

    auto await_resume()
    {
        return async.GetResults();
    }
    Async async;
  };
  return awaiter{ std::move(async) };
}
```

Note that we use the function pattern for generating the awaiter because that makes it easier to generate a different awaiter for the four different kinds of Windows Runtime asynchronous activities: We can templatize the function and propagate the type into the custom awaiter. (Alternatively, we could use CTAD.)

Our custom awaiter has a simple constructor that moves its parameter, and the `resume_in_any_apartment` constructs the object by moving its own parameter into the awaiter. This moves the original parameter to the `resume_in_any_apartment` function all the way into the awaiter.

When the caller performs the `co_await` of this custom awaiter, we schedule the handle for completion by hooking it up to the `Completed` handler. We use the magic `auto&&...` parameter list to say that the lambda accepts any number of arbitrary parameters.

When the asynchronous activity completes, the lambda is invoked, and the lambda throws away the parameters and simply invokes the `handle`, which resumes the coroutine.

When the coroutine resumes, the compiler will call `await_resume` to find out what the result of the `co_await` is. We call the asynchronous activity's `GetResults` and propagate that as our result using the `auto` return type. (If the asynchronous activity failed with an exception, the `GetResult()` method will re-raise the exception.)

Since we did no work in the `Completed` handler to get onto any particular thread, the resumption of the coroutine will occur on whatever thread called the `Completed` handler.

Here's an example of how you could use it:

```
winrt::fire_and_forget DoSomething()
{
  co_await FirstStep();
  co_await resume_in_any_apartment(SecondStep());
  co_await ThirdStep();
}
```

Assuming that all of the steps return `IAsyncAction`, the first and third `co_await`s resume execution in the same apartment, but the second one can resume in any apartment.

Now to add style points:

```
template<typename Async,
         typename = std::enable_if_t<
             std::is_convertible_v<
                 Async,
                 winrt::Windows::Foundation::IAsyncInfo>>>
[[nodiscard]] auto resume_in_any_apartment(Async async)
{
  struct awaiter : std::experimental::suspend_always
  {
    // awaiter(Async async_) : async(std::move(async_)) { }

    void await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      async.Completed([handle](auto&&amp...) { handle(); });
    }

    auto await_resume()
    {
        return async.GetResults();
    }
    Async async;
  };
  return awaiter{ {}, std::move(async) };
};
```

We tweak the template parameters so that the overload is eligible only if the `Async` is convertible to `IAsyncInfo`, which is an interface common to all of the Windows Runtime asynchronous activities. That way, if you try to use this with the wrong type, you get a more helpful error message saying that no suitable overload of `resume_ in_ any_ apartment` was found, rather than issuing a weird error message about a missing `Completed` method.

We also remove the constructor of the custom awaiter and instead construct it via aggregate construction. The empty braces initialize the `suspend_ always` base class, and the `std::move(async)` initializes the awaiter's `async` member.

Next time, we'll look at a feature of custom awaiters that is useful to avoid runaway stack consumption.

Raymond Chen

Follow