Creating a manual-start C++/WinRT coroutine from an eager-start one, part 1

devblogs.microsoft.com/oldnewthing/20220913-00

September 13, 2022

Raymond Chen

C++/WinRT coroutines are eager-start, meaning that they start running as soon as they are created, rather than waiting for you to call a method like `Start` to get them started. We saw last time that we sometimes want to be able to start a coroutine lazily. We can port the C# solution to C++/WinRT:

```
template<typename Make>
auto MakeLazy(Make make) -> decltype(make())
{
    auto start = winrt::handle(winrt::check_pointer(
        CreateEvent(nullptr, false, false, nullptr)));
    auto startHandle = start.get();
    auto currentTask = [](auto start, auto make)
        -> decltype(make()) {
        winrt::apartment_context context;
        co_await winrt::resume_on_signal(start.get());
        co_await context;
        co_return co_await make();
    }(std::move(start), std::move(make));

    // Resume the coroutine
    SetEvent(startHandle);
    return currentTask;
}
```

We create a kernel event, which is a rather convenient awaitable object built into C++/WinRT, save its handle, and transfer ownership into the lambda. We also transfer the maker into the lambda so it can make the eager-started task.

After creating and starting the lambda task (which has no captures, because capturing into a coroutine lambda is a bad idea), the lambda task captures its current context (we'll see why later) and then waits for the kernel event. This has the effect of a lazy-start coroutine, since it pauses before doing any work.

Back in the main function, after everything is all settled, we set the event handle, which wakes up the `resume_on_signal`, and then we return the coroutine that we just started.

After `resume_on_signal` resumes, the lambda coroutine awaits the original context in order to resume execution in the same context in which it had started. Whereas `co_await`ing an `IAsyncAction` or `IAsyncOperation` resume in the same COM context in which they started, the `resume_on_signal` does not offer the same guarantee. We need to `co_await context` to get back into the original context.

Once resumed, we can ask the maker to produce the eager-started coroutine, which we then await and propagate.

Of course, there's not much point here to creating a lazy-start wrapper around an eager-start coroutine, only to start it immediately. But you can imagine splitting the two steps:

```
template<typename Make>
auto MakeLazy(Make make) -> std::pair<HANDLE, decltype(make())>
{
    auto start = winrt::handle(winrt::check_pointer(
        CreateEvent(nullptr, false, false, nullptr)));
    auto startHandle = start.get();
    auto currentTask = [](auto start, auto make)
        -> decltype(make()) {
        winrt::apartment_context context;
        co_await winrt::resume_on_signal(start.get());
        co_await context;
        co_return co_await make();
    }(std::move(start), std::move(make));
    return { startHandle, currentTask };
}
```

This gives you a kernel handle, which you can signal to start the task, and it also gives you a task you can `co_await` to get things started.

This is a fairly straightforward translation of the C# lazy-start wrapper, but it turns out that we can do something more efficient if we are willing to roll up our sleeves and work with the C++ coroutine infrastructure. We'll look at that next time.

Raymond Chen

Follow