How soon is too soon to report progress from a C++/WinRT coroutine that implements a Windows Runtime asynchronous operation with progress?

devblogs.microsoft.com/oldnewthing/20221117-00

November 17, 2022

Raymond Chen

A customer discovered a problem with their C++/WinRT coroutine that generates progress.

```
IAsyncOperationWithProgress<UpdateResult, bool>
    SomeClass::UpdateAsync()
{
    auto lifetime = get_strong();
    auto progress = co_await get_progress_token();

    // Tell the caller that we are preparing
    progress(false);

    Prepare();

    // Tell the caller that we have finished preparing
    progress(true);

    // ... more work ...

    co_return UpdateResult(/* something */);
}
```

This is a simple coroutine that generates progress notifications to tell the caller when we have entered and exited the “Prepare” stage. The caller looks like this:

```
auto op = someClass.UpdateAsync();
op.Progress([](auto&&, bool started)
{
    if (started) ReallyImportantFunction();
});
auto result = co_await op;
```

The expectation is that the two progress events would be received by the Progress event handler in the calling code. However, running the code in the debugger suggests that the `UpdateAsync` function runs synchronously until well past the two calls to `progress(...)`, which means that the caller has hooked up its Progress event handler too late, and it never realizes that the operation has reached the "started" state, and the `Really-ImportantFunction()` never runs.

What's going on, and what can we do about it?

The customer's analysis is correct. This is an inherent hazard of hot-start coroutines. Even if you add a `co_await resume_background()` or a `co_await resume_after(1s)` to allow the caller to regain control and hook up its Progress event handler, there's still the chance that the caller hasn't quite gotten around to doing it by the time you raise the "started" progress event.[1]

You should try to design your operations so that progress reports are just a courtesy, and missing a progress report is not fatal.

If you need guaranteed progress delivery, you can work around this behavior in a few ways.

One idea is to use the ability for the caller to obtain provisional results:

```
// idl

runtimeclass UpdateResult
{
    String NewName { get; };
    Flavor NewFlavor { get; };

    // Add a new member
    Boolean UpdateStarted { get; };
}

// implementation

IAsyncOperationWithProgress<UpdateResult, bool>
    SomeClass::UpdateAsync()
{
    auto lifetime = get_strong();
    auto progress = co_await get_progress_token();

    // Set the provisional result and notify the caller.
    UpdateResult result;
    result.Started(false);
    progress.set_result(result);
    progress(false);

    Prepare();

    // Let the caller know that we have finished preparing.
    result.Started(true);
    progress(true);

    // ... more work ...

    co_return result;
}
```

The idea here is that we report the progress in two ways. One is via the Progress event on the `IAsyncOperation`. The other is as a new property of the operation result.[2]

Windows Runtime asynchronous operations with progress can report provisional results prior to completion, and callers can obtain those provision results by calling `GetResults()` while the operation is still running. On the implementation side, C++/WinRT lets you publish provisional results by calling `set_result()` on the progress token.

The consuming side would look like this:

```
auto op = someClass.UpdateAsync();
op.Progress([](auto&&, bool started)
{
    if (started) ReallyImportantFunction();
});

// Check if the update started before we were able
// to register for the Progress event.
if (op.GetResults().UpdateStarted()) ReallyImportantFunction():

auto result = co_await op;
```

Note that there is a race condition here where the update starts after we register the Progress event handler but before we check the provisional results. In that case, we will call `ReallyImportantFunction()` twice. In our customer's case, it was harmless to call it twice, but in the more general case, you may need to add additional logic to avoid calling `ReallyImportantFunction()` twice.

But really, you shouldn't put critical information in the Progress event payload. Next time, we'll look at alternate designs for critical progress notifications.

[1] For example, the caller thread may have lost its quantum just before it was able to hook up the Progress event handler.

[2] If the original operation result was not a runtime class, you can create a new runtime class that consists of the original operation result, plus the new `UpdateStarted` property. Alternatively, you can make the operation result a Windows Runtime `struct`, but Windows Runtime `struct`s are value types, so you will have to do a `progress.set_result()` each time you update the `result` to update the copy.

Raymond Chen

Follow