On Windows Runtime asynchronous operations with critical progress reports

devblogs.microsoft.com/oldnewthing/20221118-00

November 18, 2022

Raymond Chen

Last time, we looked at the inherent race condition in hot-start coroutine progress notifications. In general, progress notifications should be treated as advisory and not critical to proper operation, because the initial progress notifications could be lost due to the race.

As a refresher, we have this old and busted code:

```
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 */);
}
```

with this old and busted consumer:

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

The problem is that there is a race condition between the coroutine generating progress reports and the consumer registering its Progress event handler. If the consumer registers too late, it may miss some progress notifications.

If your scenario requires that all progress notifications be delivered in order for things to work properly, then maybe a progress notification isn't the right pattern.

Instead, what you can do is pass the progress callback directly to the `UpdateAsync` function, so that the coroutine can call it directly.

```
// This version is incomplete, so don't use it yet.
IAsyncOperation<UpdateResult>
    SomeClass::UpdateAsync(
        delegate<bool> progressCallback)
{
    auto lifetime = get_strong();

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

    Prepare();

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

    // ... more work ...

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

By being given the progress callback up front, we can call it directly, avoiding the race condition where we raise the Progress event before the caller can register the callback.

But this version is not yet ready for prime time. For one thing, if the body of `UpdateAsync` performs apartment-switching, such as a `co_await resume_background()`, then the progress callback may be invoked from a COM apartment different from the original one, which is a violation of COM threading rules.[1] So the above code is correct only if you ensure that the `progressCallback` is invoked only on the original apartment.

To allow the progress callback to be invoked from any apartment, you'll have to wrap it in an agile reference:

```
IAsyncOperation<UpdateResult>
    SomeClass::UpdateAsync(
        delegate<bool> progressCallback)
{
    auto lifetime = get_strong();
    auto agileCallback = agile_ref(progressCallback);

    // Tell the caller that we are preparing
    if (agileCallback) agileCallback.get()(false);

    Prepare();

    // Tell the caller that we have finished preparing
    if (agileCallback) agileCallback.get()(true);

    // ... more work ...

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

We immediately wrap the `progressCallback` in an `agile_ref`, which makes it possible to use the callback from any apartment. When it's time to make the callback, we call `get()` on the agile reference to import it into the current apartment, and then we use the `()` operator to invoke it.

Another gotcha is the case of the callback that throws an exception, say, because it is a method on a weak reference to an object that has already been destroyed. If we took no special steps, those exceptions would cause `UpdateAsync` to stop executing and propagate the exception, which is probably going to be rather confusing.

We can use a `winrt::event` to solve both the "agile wrapper" problem and the "exception in the callback" problem.

```
IAsyncOperation<UpdateResult>
    SomeClass::UpdateAsync(
        delegate<bool> progressCallback)
{
    event<delegate<bool>> progress;
    progress.add(progressCallback);

    auto lifetime = get_strong();

    // 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 */);
}
```

The `winrt::event` internally optimizes the case where the `progressCallback` is already agile and skips the agile wrapper. It also deals with exceptions in the callback: If a disconnection exception occurs, the event removes the delegate. Any other exceptions are swallowed. This makes `UpdateAsync` resilient to exceptions in the callback.

Mind you, if the callback hangs, then `UpdateAsync` will also hang. We could try to fix this by making progress callback asynchronously, but then you have to worry about multiple progress callbacks racing against each other. Most implementations don't try to deal with this case. If the progress callback wants to hang the operation, then let it hang the operation.

[1] If the callback is written in C# or C++/WinRT, then the callback will be agile and support being called from arbitrary threads. On the other hand, JavaScript is a single-threaded execution environment, and the callback relies on the COM thread rules which specify that you must invoke it only from the originating apartment.

Raymond Chen

Follow