How do I wait for the completion of the delegate I passed to CoreDispatcher.RunAsync and ThreadPool.Run-Async?

devblogs.microsoft.com/oldnewthing/20190327-00

March 27, 2019

Raymond Chen

As we saw last time, The the `CoreDispatcher::RunAsync` and `ThreadPool::RunAsync` methods complete when the delegate returns, which is not the same as when the delegate *completes*. How can you wait until the delegate completes?

We'll have to track the delegate completion ourselves.

One way is to signal the completion with a custom `TaskCompletionSource`. There's a task snippet that demonstrates this. Here's a simplified version:

```
using System;
using System.Threading.Tasks;
using Windows.UI.Core;

public static class DispatcherTaskExtensions
{
    public static async Task<T> RunTaskAsync<T>(
        this CoreDispatcher dispatcher,
        Func<Task<T>> func)
    {
        var tcs = new TaskCompletionSource<T>();
        await dispatcher.RunAsync(CoreDispatcherPriority.Normal,
            async () =>
            {
                tcs.SetResult(await func());
            });
        return await tcs.Task;
    }

    public static async Task RunTaskAsync(this CoreDispatcher dispatcher,
        Func<Task> func) =>
        await RunTaskAsync(dispatcher,
                           async () => { await func(); return false; });
}
```

The idea is that you pass an async lambda, and the `RunTaskAsync` extension method wraps it inside another lambda that awaits the async lambda, thereby waiting until the task completes. Upon completion, it uses a `TaskCompletionSource` to indicate to the caller that everything is all finished.

As a courtesy, the returned task matches the return type of the original async lambda, so that if the async lambda completes with a value, that is also the completed value of the returned task.

An alternative approach is to make the wrapper lambda synchronous, so that the outer `await` of `RunAsync` completes when the wrapper lambda is done. We can then await the async lambda's result from the calling thread.

```
public static class DispatcherTaskExtensions
{
    static public async Task<T> RunTaskAsync<T>(
        this CoreDispatcher dispatcher,
        Func<Task<T>> func)
    {
        Task<T> result = null;
        await dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                  () => result = func());
        return await result;
    }

    static public async Task RunTaskAsync<T>(
        this CoreDispatcher dispatcher,
        Func<Task> func)
    {
        Task result = null;
        await dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                  () => result = func());
        await result;
    }
}
```

The idea behind this second version is that we invoke the async lambda on the dispatcher thread, and then wait on the task from the calling thread.

Both of these versions elide exception-handling for expository simplicity.

You can imagine for yourself analogous versions for running an async lambda on a background thread: You use `ThreadPool.` `RunAsync` instead of `CoreDispatcher.` `Run-Async`, but the rest is basically the same.

Suppose you have a function that is called on the UI thread, and it performs long computations as part of its processing, and you want to update the UI partway through the computation. The single-threaded version looks like this:

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    // Get the control's value.
    var v = SomeControl.Value;

    // Do the computation.
    var result1 = Compute1(v);
    var other = await ContactWebServiceAsync();
    var result2 = Compute2(result1, other);

    // Provide an interim update.
    TextBlock1.Text = result1;
    TextBlock2.Text = result2;

    // Back to more computations.
    var extra = await GetExtraDataAsync();
    var result3 = Compute3(result1, result2, extra);

    // Show final results.
    TextBlock3.Text = result3;
  }
}
```

Unfortunately, this code performs the long computations on the UI thread, so your app stops responding while the computations are taking place. You really want to do the computations on a background thread, and return to the UI thread only to provide UI updates.

The original code without `RunTaskAsync` consumes a level of nesting each time you want to switch threads.

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    // Get the control's value from the UI thread.
    var v = SomeControl.Value;

    // Do the computation on a background thread.
    await ThreadPool.RunAsync(async (item) =>
    {
      var result1 = Compute1(v);
      var other = await ContactWebServiceAsync();
      var result2 = Compute2(result1, other);

      // Back to the UI thread to provide an interim update.
      await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                async () =>
      {
        TextBlock1.Text = result1;
        TextBlock2.Text = result2;

        // Back to the background thread to do more computations.
        await ThreadPool.RunAsync(async (item) =>
        {
          var extra = await GetExtraDataAsync();
          var result3 = Compute3(result1, result2, extra);

          // And back to the UI thread one last time.
          await Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                    async () =>
          {
            TextBlock3.Text = result3;
          };
        });
      });
    });
  }
}
```

This is painful enough for straight-line code, but you can imagine how much more complicated it gets if you have loops or other nonlinear control flow.

To avoid the nesting, you can bounce back to the originating thread.

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    // Get the control's value from the UI thread.
    var v = SomeControl.Value;

    // Do the computation on a background thread.
    await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      var result1 = Compute1(v);
      var other = await ContactWebServiceAsync();
      var result2 = Compute2(result1, other);
    });

    // Provide an interim update.
    TextBlock1.Text = result1;
    TextBlock2.Text = result2;

    // Back to the background thread to do more computations.
    await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      var extra = await GetExtraDataAsync();
      var result3 = Compute3(result1, result2, extra);
    });

    // Provide a final update.
    TextBlock3.Text = result3;
  }
}
```

Let’s ignore for the moment that this code doesn’t compile.

The thread which started the entire sequence ends up orchestrating the subsequent thread switches. Each `await`ed `RunTaskAsync` returns back to that thread, which can do some work before kicking off another `RunTaskAsync`.

This gets rid of the nesting, but it still isn’t great. Control keeps returning to the original thread, even if the intent was to go directly from the second thread to a third. For example, perhaps you wanted to display the interim update in a different thread’s window.

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    ...

    // Do the computation on a background thread.
    await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      var result1 = Compute1(v);
      var other = await ContactWebServiceAsync();
      var result2 = Compute2(result1, other);
    });

    // Provide an interim update in a secondary window.
    await otherDispatcher.RunAsync(CoreDispatcherPriority.Normal,
                               () =>
    {
        TextBlock1.Text = result1;
        TextBlock2.Text = result2;
    });

    // Back to the background thread to do more computations.
    ...
  }
}
```

We returned to the original thread even though all it's going to do is ask another thread to do some more work. That's two thread switches when we wanted needed only one.

The next problem is addressing the fact that the code doesn't compile because the `result1`, `result2`, and `result3` variables all belong to the lambda and are not visible to the outer function.

One option is to hoist the variables out of the async lambda so that they can be shared with the calling method. Furthermore, the compiler requires you to initialize the variables because it cannot prove that the async lambda will definitely assign values to the variables prior to use. (The compiler doesn't know what `RunTaskAsync` does. For all the compiler knows, the method ignores its lambda parameter!)

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    // Get the control's value from the UI thread.
    var v = SomeControl.Value;

    string result1 = null;
    string result2 = null;

    // Do the computation on a background thread.
    await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      result1 = Compute1(v);
      var other = await ContactWebServiceAsync();
      result2 = Compute2(result1, other);
    });

    // Provide an interim update.
    TextBlock1.Text = result1;
    TextBlock2.Text = result2;

    string result3 = null;

    // Back to the background thread to do more calculations.
    await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      var extra = await GetExtraDataAsync();
      result3 = Compute3(result1, result2, extra);
    });

    // Provide a final update.
    TextBlock3.Text = result3;
  }
}
```

Hoisting and preinitializing the shared variables is awkward at best, and impossible in the general case, because the variables might be anonymous types, which therefore cannot be declared with an explicit type.

We can take advantage of the fact that `RunTaskAsync` forwards the inner task's result as its own result, so we can return the things we want to make available to the caller. In the case where it's only one thing (as with `result3`), we can return it directly. Otherwise, we'll have to wrap it in another object, like an anonymous type, and then extract the values from the wrapper on the receiving side.

```
public sealed partial class MyPage : Page
{
  void Button_Click()
  {
    // Get the control's value from the UI thread.
    var v = SomeControl.Value;

    // Do the computation on a background thread.
    var (result1, result2) =
      await ThreadPoolHelper.RunTaskAsync(async () =>
      {
        var innerResult1 = Compute1(v);
        var other = await ContactWebServiceAsync();
        var innerResult2 = Compute2(innerResult1, other);

        return (result1, result2);
      });

    // Provide an interim update.
    TextBlock1.Text = result1;
    TextBlock2.Text = result2;

    // Back to the background thread to do more calculations.
    var result3 = await ThreadPoolHelper.RunTaskAsync(async () =>
    {
      var extra = await GetExtraDataAsync();
      return Compute3(result1, result2, extra);
    });

    // Provide a final update.
    TextBlock3.Text = result3;
  }
}
```

This is cumbersome and error-prone because of the manual packing and unpacking, especially since the code that unpacks the values is far away from the code that packs them. Woe unto you if you get the tuple elements backward! Matching them up is made even more frustrating because you can’t name the variables inside the lambda the same as the ones outside the lambda. Variables in a lambda are not allowed to shadow variables outside it.

And of course neither option really helps if you want to put the variables into a `using` statement.

Fortunately, all is not lost. Next time, we’ll develop an alternative that allows you to write code in a much more natural way.

Raymond Chen

Follow