C++/WinRT envy: Bringing thread switching tasks to C# (UWP edition)

devblogs.microsoft.com/oldnewthing/20190328-00

March 28, 2019

Raymond Chen

Last time, we developed a `RunTaskAsync` method to try to make it easier to switch threads in a task, but we saw that while it simplified some operations, it was still cumbersome because of the difficulty of sharing state between the main method and the async lambdas that it kicked off to other threads.

Let's fix that by stealing an idea from C++/WinRT: Make thread-switching an awaitable operation.

In C++/WinRT, you can switch threads by awaiting a magic object where you enter on one thread and pop out the other side on a different thread. It's like Portal for threads!

```
// C++/WinRT

winrt::fire_and_forget MyPage::Button_Click()
{
  // We start on a UI thread.
  auto lifetime = get_strong();

  // Get the control's value from the UI thread.
  auto v = SomeControl().Value();

  // Move to a background thread.
  co_await winrt::resume_background();

  // Do the computation on a background thread.
  auto result1 = Compute1(v);
  auto other = co_await ContactWebServiceAsync();
  auto result2 = Compute2(result1, other);

  // Return to the UI thread to provide an interim update.
  co_await winrt::resume_foreground(Dispatcher());

  // Back on the UI thread: We can update UI elements.
  TextBlock1().Text(result1);
  TextBlock2().Text(result2);

  // Back to the background thread to do more computations.
  co_await winrt::resume_background();

  auto extra = co_await GetExtraDataAsync();
  auto result3 = Compute3(result1, result2, extra);

  // Return to the UI thread to provide a final update.
  co_await winrt::resume_foreground(Dispatcher());

  // Update the UI one last time.
  TextBlock3().Text(result3);
}
```

The thread-switching is expressed simply as an asynchronous operation. Constructors and destructors still run at the usual times, so you can use RAII types naturally. You can perform these magic `co_await` operations inside loops or conditionals, and they behave in the natural way.

```
// Move to a background thread if a condition is met.
if (condition) {
  co_await winrt::resume_background();
}

DoSomething();
```

In the above case, the `DoSomething()` occurs on a background thread if the condition is met, or it occurs on the current thread if the condition is not met. This sort of flexibility is difficult to express using our previous model of always putting off-thread actions into an asynchronous lambda.

Okay, enough hype. Let's bring `resume_foreground()` and `resume_background()` to C#!

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using Windows.System.Threading;
using Windows.UI.Core;

struct DispatcherThreadSwitcher : INotifyCompletion
{
    internal DispatcherThreadSwitcher(CoreDispatcher dispatcher) =>
        this.dispatcher = dispatcher;
    public DispatcherThreadSwitcher GetAwaiter() => this;
    public bool IsCompleted => dispatcher.HasThreadAccess;
    public void GetResult() { }
    public void OnCompleted(Action continuation) =>
        _ = dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                () => continuation());
    CoreDispatcher dispatcher;
}

struct ThreadPoolThreadSwitcher : INotifyCompletion
{
    public ThreadPoolThreadSwitcher GetAwaiter() => this;
    public bool IsCompleted =>
       SynchronizationContext.Current == null;
    public void GetResult() { }
    public void OnCompleted(Action continuation) =>
        _ = ThreadPool.RunAsync(_ => continuation());
}

class ThreadSwitcher
{
    static public DispatcherThreadSwitcher ResumeForegroundAsync(
        CoreDispatcher dispatcher) =>
        new DispatcherThreadSwitcher(dispatcher);
    static public ThreadPoolThreadSwitcher ResumeBackgroundAsync() =>
        new ThreadPoolThreadSwitcher();
}
```

We can use the methods in `ThreadSwitcher` the same way we did in C++/WinRT:

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

    // Move to a background thread.
    await ThreadSwitcher.ResumeBackgroundAsync();

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

    // Return to the UI thread to provide an interim update.
    await ThreadSwitcher.ResumeForegroundAsync(Dispatcher);

    // Back on the UI thread: We can update UI elements.
    TextBlock1.Text = result1;
    TextBlock2.Text = result2;

    // Back to the background thread to do more computations.
    await ThreadSwitcher.ResumeBackgroundAsync();

    var extra = await GetExtraDataAsync();
    var result3 = Compute3(result1, result2, extra);

    // Return to the UI thread to provide a final update.
    await ThreadSwitcher.ResumeForegroundAsync(Dispatcher);

    // Update the UI one last time.
    TextBlock3.Text = result3;
  }
}
```

This is identical to our original “all on the UI thread” code, excepty for the calls to `ThreadSwitcher` members.

How does this `ThreadSwitcher` class work?

We need to understand the awaitable-awaiter pattern and how the compiler uses it. Read the linked articles for details. In summary, the line

```
result = await x;
```

compiles to something spiritually similar to the following:[1]

```
var awaiter = x.GetAwaiter();
if (!awaiter.IsCompleted) {
 awaiter.OnCompleted(() => goto resume);
 return task;
resume:;
}
result = awaiter.GetResult();
```

First, the compiler calls the `GetAwaiter` method to obtain an "awaiter". If the awaiter says that the task has not yet completed, then the compiler tells the awaiter, "Okay, well, when it's complete, let me know." Then the function returns. When the operation finally completes, execution resumes.

When the operation is complete, either because it was complete all along, or because we were resumed after a delayed completion, the result is obtained by calling the awaiter's `GetResult()` method.

You can create custom awaitable things by plugging into the above pattern.

In our case, `ThreadSwitcher.ResumeForegroundAsync()` works as follows:

- It creates a `DispatcherThreadSwitcher` with the `dispatcher` you want to use.
- The `DispatcherThreadSwitcher.GetAwaiter` method returns itself. The object serves double-duty as the awaitable object and its own awaiter.
- To determine whether the operation has already completed, the `IsCompleted` property reports whether we are already on the dispatcher's thread. If so, then the compiler won't bother scheduling a continuation; it'll just keep executing.
- If we report that the operation has not completed, the compiler will use the `OnCompleted` method to ask us to complete the operation and then call a specific delegate once it's done. We queue a work item onto the dispatcher's thread.
- The work item runs on the dispatcher's thread, and from that work item, we invoke the completion delegate. The coroutine resumes execution on the dispatcher's thread, as desired.

The `ThreadSwitcher.ResumeBackgroundAsync()` method works almost the same way, but for the thread pool rather than for a dispatcher.

- It creates a `ThreadPoolThreadSwitcher`.
- The `ThreadPoolThreadSwitcher.GetAwaiter` method returns itself. Again, the object serves double-duty as the awaitable object and its own awaiter.
- To determine whether the operation has already completed, we check the current `SynchronizationContext`. A value of `null` means that we are already on a background thread.

- If we report that the operation has not completed, the compiler will use the `OnCompleted` method to ask us to complete the operation and then call a specific delegate once it's done. We queue a work item onto the thread pool.
- The work item runs on a thread pool thread, and from that work item, we invoke the completion delegate. The coroutine resumes execution on a thread pool thread, as desired.

All the magic is done by a handful of one-line methods.

Integrating thread switching via `await` not only simplifies the code, it also opens up new usage patterns that were difficult to accomplish without it.

```
// Assume we enter on the UI thread.
using (var connection = new Connection()) {

    // Initialize on the UI thread since
    // we need information from UI objects.
    connection.Initialize(SomeParameter);

    await ThreadSwitcher.ResumeBackgroundAsync();
    // Execute on a background thread.
    connection.Execute();

} // connection is disposed here

// Process the results on a background thread.
Process(connection.GetResults());
```

Notice that we switched threads right in the middle of the `using` block, so that we exited the block on a different thread from the one we started!

When I show this trick to people, their reactions tend to fall into one of two categories.

1. This is truly embracing the concept of asynchronous operations, and it's a game-changer for code that needs to perform multiple actions on different threads. We should make this trick more widely known.
2. This is an offense against nature. C# developers have long internalized the rule that "Unless explicitly configured, `await` does not switch threads," but this class violates that rule.

Let me know in the comments which side you identify with. And if you identify with the first group, should I adopt the `ThreadSwitcher` class in the UWP samples repo?

Next time, we'll implement the `ThreadSwitcher` methods for WPF and WinForms.

[1] In reality, the compiler remembers where to resume execution in a state variable prior to the `return`, and the `goto resume` is done by resuming the state machine.

Raymond Chen

Follow