C++/WinRT envy: Bringing thread switching tasks to C# (WPF and WinForms edition)

devblogs.microsoft.com/oldnewthing/20190329-00

March 29, 2019

Raymond Chen

Last time, we brought `ThreadSwitcher.ResumeForegroundAsync` and `ThreadSwitcher.ResumeBackgroundAsync` to C# for UWP. Today, we'll do the same for WPF and Windows Forms.

It'll be easier the second and third times through because we already learned how to structure the implementation. It's just the minor details that need to be tweaked.

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;          // For ThreadPool
using System.Windows.Forms;      // For Windows Forms
using System.Windows.Threading;  // For WPF

// For WPF
struct DispatcherThreadSwitcher : INotifyCompletion
{
    internal DispatcherThreadSwitcher(Dispatcher dispatcher) =>
        this.dispatcher = dispatcher;
    public DispatcherThreadSwitcher GetAwaiter() => this;
    public bool IsCompleted => dispatcher.CheckAccess();
    public void GetResult() { }
    public void OnCompleted(Action continuation) =>
        dispatcher.BeginInvoke(continuation);
    Dispatcher dispatcher;
}

// For Windows Forms
struct ControlThreadSwitcher : INotifyCompletion
{
    internal ControlThreadSwitcher(Control control) =>
        this.control = control;
    public ControlThreadSwitcher GetAwaiter() => this;
    public bool IsCompleted => !control.InvokeRequired;
    public void GetResult() { }
    public void OnCompleted(Action continuation) =>
        control.BeginInvoke(continuation);
    Control control;
}

// For both WPF and Windows Forms
struct ThreadPoolThreadSwitcher : INotifyCompletion
{
    public ThreadPoolThreadSwitcher GetAwaiter() => this;
    public bool IsCompleted =>
        SynchronizationContext.Current == null;
    public void GetResult() { }
    public void OnCompleted(Action continuation) =>
        ThreadPool.QueueUserWorkItem(_ => continuation());
}

class ThreadSwitcher
{
    // For WPF
    static public DispatcherThreadSwitcher ResumeForegroundAsync(
        Dispatcher dispatcher) =>
        new DispatcherThreadSwitcher(dispatcher);

    // For Windows Forms
    static public ControlThreadSwitcher ResumeForegroundAsync(
        Control control) =>
        new ControlThreadSwitcher(control);

    // For both WPF and Windows Forms
    static public ThreadPoolThreadSwitcher ResumeBackgroundAsync() =>
         new ThreadPoolThreadSwitcher();
}
```

The principles for these helper classes are the same as for their UWP counterparts. They are merely adapting to a different control pattern.

WPF uses the `System.Threading.Dispatcher` class to control access to the UI thread. The way to check if you are on the dispatcher's thread is to call `CheckAccess()` and see if it grants you access. If so, then you are already on the dispatcher thread. Otherwise, you are on the wrong thread, and the way to get to the dispatcher thread is to use the `BeginInvoke` method.

In Windows Forms, controls incorporate their own dispatcher. To determine if you're on the control's thread, you check the `InvokeRequired` property. If it tells you that you need to invoke, then you call `BeginInvoke` to get to the correct thread.

Both WPF and Windows Forms use the CLR thread pool. As before, we check the `SynchronizationContext` to determine whether we are on a background thread already. If not, then we use `QueueUserWorkItem` to get onto the thread pool.

So there we have it, C++/WinRT-style thread switching for three major C# user interface frameworks. If you feel inspired, you can do the same for Silverlight, Xamarin, or any other C# UI framework I may have forgotten.

Raymond Chen

Follow