How to (not) deadlock your UI using async-await

This article explains the causes of UI deadlock when using `async-await`, and how to avoid it.

Introduction

With the introduction of asynchronous programming capabilities in C# 5.0 it becomes easier to write asynchronous code in a sequential manner.

When the caller awaits on a `Task`, it will be semantically blocked until its completion. "Semantically blocked" means that the flow of execution of the calling method will be interrupted and resumed when the asynchronous operation is completed.

Normally this means that the executing thread will be blocked. However UI threads like the Windows Phone UI need their message pump to keep working in order to keep the UI responsive. In this case the thread is not blocked, but instead the flow of execution is returned to the message pump. When the asynchronous operation ends a message for its continuation is posted on the message pump and execution resumes on the next instruction.

The test app

The test app is composed of just a text block, a progress bar (to show UI activity while doing "work" in the background) and two buttons (one that deadlocks the UI and another that doesn't).

Deadlock test app

```
<phone:PhoneApplicationPage
    x:Class="PhoneApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="Portrait" Orientation="Portrait"
    shell:SystemTray.IsVisible="True">

    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
            <TextBlock Text="ASYNC-AWAIT BLOCKING" Style="{StaticResource PhoneTextNormalStyle}"/>
            <TextBlock x:Name="TextBlock1" TextWrapping="Wrap" VerticalAlignment="Top" Style="{StaticResource PhoneTextTitle1Style}"
            <ProgressBar x:Name="ProgressBar1" Height="10" IsIndeterminate="True" Visibility="Collapsed"/>
        </StackPanel>

        <StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Button x:Name="Button1" Content="deadlock" Click="Button1_Click" />
            <Button x:Name="Button2" Content="don't deadlock" Click="Button2_Click" />
        </StackPanel>
    </Grid>

</phone:PhoneApplicationPage>
```

The test method

For the purpose of this article, our asynchronous method that does "work" in the background will just return the current date/time after a five second delay:

```
private async Task<DateTime> GetValueAsync()
```


(C) Copyright Nokia 2013. All rights reserved.

```
{
    await Task.Delay(5000);
    return DateTime.Now;
}
```

Deadlocking the UI

A major cause of UI deadlock is to forget to wait (`await`) on the asynchronous method - instead accessing the `Task`'s `Result` property directly:

```
private void Button1_Click(object sender, RoutedEventArgs e)
{
    // ...
    var datetime = GetValueAsync().Result;
    this.TextBlock1.Text = datetime.ToLongTimeString();
    // ...
}
```

The UI is blocked because without the `await` C# compiler (or VB compiler) does not generate the code needed to return the flow of execution to the message pump or post the continuation when the operation completes. As a result the message pump thread is blocked and the UI is deadlocked.

Not deadlocking the UI

The easiest way to not block the UI is to `await` on the returned `Task` instead of directly accessing its `Result` property:

```
private async void Button2_Click(object sender, RoutedEventArgs e)
{
    // ...
    var datetime = await GetValueAsync();
    TextBlock1.Text = datetime.ToLongTimeString();
    // ...
}
```

This way, the UI thread will continue to handle messages from the message pump and the continuation will be scheduled when the task completes.

Conclusion

Once you decide to use `Tasks` in the UI you should always use `async-await` because the C# compiler (and the Visual Basic compiler) will generate a state machine that will schedule the execution to the right context without blocking the UI.

Sample projects

Download a solution with sample projects for Windows Phone 7.5 and Windows Phone 8 from here: File:How to (not) deadlock your UI using async-await sample solution.zip

Resources

- Asynchronous Programming with Async and Await (C# and Visual Basic)
- Task-based Asynchronous Pattern (TAP)
- Async for .NET Framework 4, Silverlight 4 and 5, and Windows Phone 7.5 and 8 (needed for using *async/await* in Windows Phone 7.5 and 8)
- Three Essential Tips For Async - Introduction video by Lucian Wischik
- async by Eric Lippert
- Eduasync by Jon Skeet
- An Async Primer

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.