

Threading Model

📅 03/30/2017 • ⌚ 24 minutes to read • Contributors 👤👤👤👤👤 all

In this article

[Overview and the Dispatcher](#)

[Threads in Action: The Samples](#)

[Technical Details and Stumbling Points](#)

[See Also](#)

Windows Presentation Foundation (WPF) is designed to save developers from the difficulties of threading. As a result, the majority of WPF developers won't have to write an interface that uses more than one thread. Because multithreaded programs are complex and difficult to debug, they should be avoided when single-threaded solutions exist.

No matter how well architected, however, no UI framework will ever be able to provide a single-threaded solution for every sort of problem. WPF comes close, but there are still situations where multiple threads improve user interface (UI) responsiveness or application performance. After discussing some background material, this paper explores some of these situations and then concludes with a discussion of some lower-level details.

📘 Note

This topic discusses threading by using the [BeginInvoke](#) method for asynchronous calls. You can also make asynchronous calls by calling the [InvokeAsync](#) method, which take an [Action](#) or [Func<TResult>](#) as a parameter. The [InvokeAsync](#) method returns a [DispatcherOperation](#) or [DispatcherOperation<TResult>](#), which has a [Task](#) property. You can use the `await` keyword with either the [DispatcherOperation](#) or the associated [Task](#). If you need to wait synchronously for the [Task](#) that is returned by a

[DispatcherOperation](#) or [DispatcherOperation<TResult>](#), call the [DispatcherOperationWait](#) extension method. Calling [Task.Wait](#) will result in a deadlock. For more information about using a [Task](#) to perform asynchronous operations, see Task Parallelism. The [Invoke](#) method also has overloads that take an [Action](#) or [Func<TResult>](#) as a parameter. You can use the [Invoke](#) method to make synchronous calls by passing in a delegate, [Action](#) or [Func<TResult>](#).

Overview and the Dispatcher

Typically, WPF applications start with two threads: one for handling rendering and another for managing the UI. The rendering thread effectively runs hidden in the background while the UI thread receives input, handles events, paints the screen, and runs application code. Most applications use a single UI thread, although in some situations it is best to use several. We'll discuss this with an example later.

The UI thread queues work items inside an object called a [Dispatcher](#). The [Dispatcher](#) selects work items on a priority basis and runs each one to completion. Every UI thread must have at least one [Dispatcher](#), and each [Dispatcher](#) can execute work items in exactly one thread.

The trick to building responsive, user-friendly applications is to maximize the [Dispatcher](#) throughput by keeping the work items small. This way items never get stale sitting in the [Dispatcher](#) queue waiting for processing. Any perceivable delay between input and response can frustrate a user.

How then are WPF applications supposed to handle big operations? What if your code involves a large calculation or needs to query a database on some remote server? Usually, the answer is to handle the big operation in a separate thread, leaving the UI thread free to tend to items in the [Dispatcher](#) queue.

When the big operation is complete, it can report its result back to the UI thread for display.

Historically, Windows allows UI elements to be accessed only by the thread that created them. This means that a background thread in charge of some long-running task cannot update a text box when it is finished. Windows does this to ensure the integrity of UI components. A list box could look strange if its contents were updated by a background thread during painting.

WPF has a built-in mutual exclusion mechanism that enforces this coordination. Most classes in WPF derive from [DispatcherObject](#). At construction, a [DispatcherObject](#) stores a reference to the [Dispatcher](#) linked to the currently running thread. In effect, the [DispatcherObject](#) associates with the thread that creates it. During program execution, a [DispatcherObject](#) can call its public [VerifyAccess](#) method. [VerifyAccess](#) examines the [Dispatcher](#) associated with the current thread and compares it to the [Dispatcher](#) reference stored during construction. If they don't match, [VerifyAccess](#) throws an exception. [VerifyAccess](#) is intended to be called at the beginning of every method belonging to a [DispatcherObject](#).

If only one thread can modify the UI, how do background threads interact with the user? A background thread can ask the UI thread to perform an operation on its behalf. It does this by registering a work item with the [Dispatcher](#) of the UI thread. The [Dispatcher](#) class provides two methods for registering work items: [Invoke](#) and [BeginInvoke](#). Both methods schedule a delegate for execution. [Invoke](#) is a synchronous call – that is, it doesn't return until the UI thread actually finishes executing the delegate. [BeginInvoke](#) is asynchronous and returns immediately.

The [Dispatcher](#) orders the elements in its queue by priority. There are ten levels that may be specified when adding an element to the [Dispatcher](#) queue. These priorities are maintained in the [DispatcherPriority](#) enumeration. Detailed

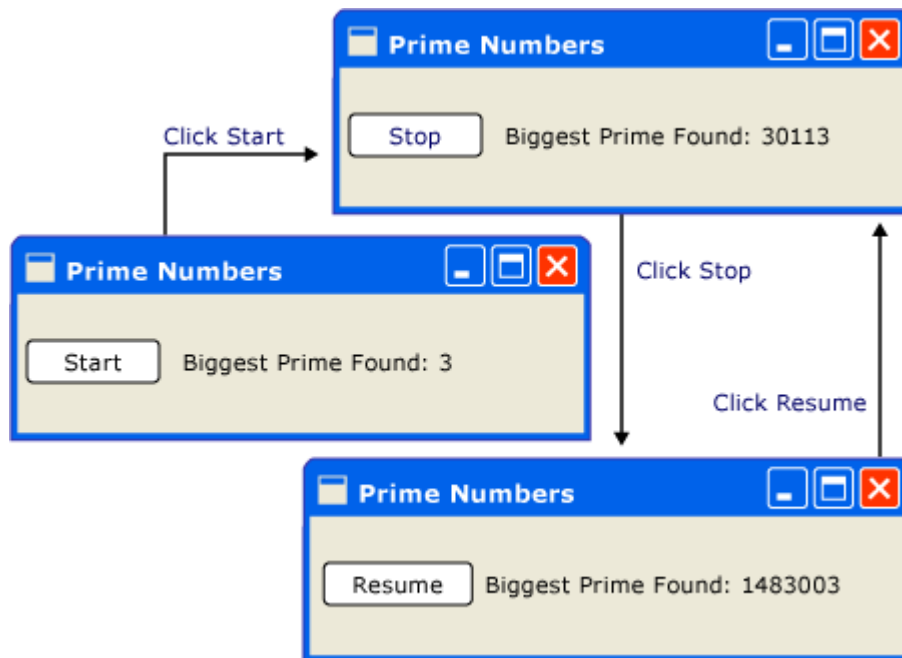
information about [DispatcherPriority](#) levels can be found in the Windows SDK documentation.

Threads in Action: The Samples

A Single-Threaded Application with a Long-Running Calculation

Most graphical user interfaces (GUIs) spend a large portion of their time idle while waiting for events that are generated in response to user interactions. With careful programming this idle time can be used constructively, without affecting the responsiveness of the UI. The WPF threading model doesn't allow input to interrupt an operation happening in the UI thread. This means you must be sure to return to the [Dispatcher](#) periodically to process pending input events before they get stale.

Consider the following example:



This simple application counts upwards from three, searching for prime numbers. When the user clicks the **Start** button, the search begins. When the

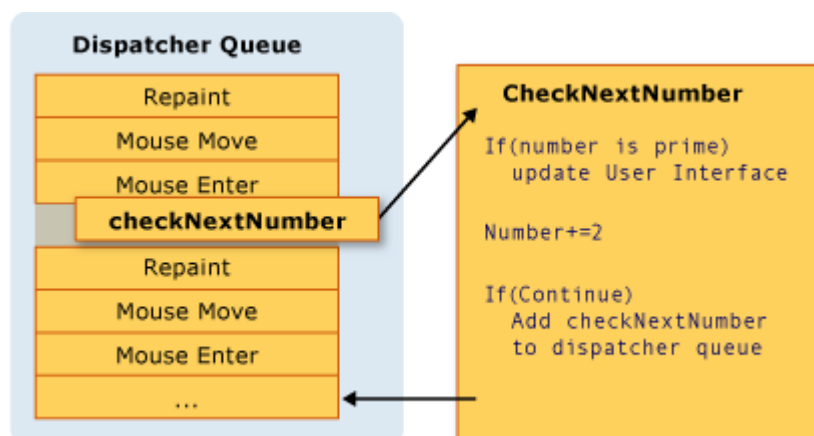
program finds a prime, it updates the user interface with its discovery. At any point, the user can stop the search.

Although simple enough, the prime number search could go on forever, which presents some difficulties. If we handled the entire search inside of the click event handler of the button, we would never give the UI thread a chance to handle other events. The UI would be unable to respond to input or process messages. It would never repaint and never respond to button clicks.

We could conduct the prime number search in a separate thread, but then we would need to deal with synchronization issues. With a single-threaded approach, we can directly update the label that lists the largest prime found.


If we break up the task of calculation into manageable chunks, we can periodically return to the [Dispatcher](#) and process events. We can give WPF an opportunity to repaint and process input.

The best way to split processing time between calculation and event handling is to manage calculation from the [Dispatcher](#). By using the [BeginInvoke](#) method, we can schedule prime number checks in the same queue that UI events are drawn from. In our example, we schedule only a single prime number check at a time. After the prime number check is complete, we schedule the next check immediately. This check proceeds only after pending UI events have been handled.




Microsoft Word accomplishes spell checking using this mechanism. Spell checking is done in the background using the idle time of the UI thread. Let's take a look at the code.

The following example shows the XAML that creates the user interface.

XAML	 Copy
<pre><Window x:Class="SDKSamples.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="Prime Numbers" Width="260" Height="75" > <StackPanel Orientation="Horizontal" VerticalAlignment="Center" > <Button Content="Start" Click="StartOrStop" Name="startStopButton" Margin="5,0,5,0" /> <TextBlock Margin="10,5,0,0">Biggest Prime Found:</TextBlock> <TextBlock Name="bigPrime" Margin="4,5,0,0">3</TextBlock> </StackPanel> </Window></pre>	

The following example shows the code-behind.

C#	 Copy
<pre>using System; using System.Windows; using System.Windows.Controls; using System.Windows.Threading; using System.Threading; namespace SDKSamples { public partial class Window1 : Window { public delegate void NextPrimeDelegate(); //Current number to check private long num = 3; private bool continueCalculating = false;</pre>	

```

public Window1() : base()
{
    InitializeComponent();
}

private void StartOrStop(object sender, EventArgs e)
{
    if (continueCalculating)
    {
        continueCalculating = false;
        startStopButton.Content = "Resume";
    }
    else
    {
        continueCalculating = true;
        startStopButton.Content = "Stop";
        startStopButton.Dispatcher.BeginInvoke(
            DispatcherPriority.Normal,
            new NextPrimeDelegate(CheckNextNumber));
    }
}

public void CheckNextNumber()
{
    // Reset flag.
    NotAPrime = false;

    for (long i = 3; i <= Math.Sqrt(num); i++)
    {
        if (num % i == 0)
        {
            // Set not a prime flag to true.
            NotAPrime = true;
            break;
        }
    }

    // If a prime number.
    if (!NotAPrime)
    {
        bigPrime.Text = num.ToString();
    }

    num += 2;
    if (continueCalculating)
    {
        startStopButton.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.SystemIdle,
            new NextPrimeDelegate(this.CheckNextNumber));
    }
}

```

```
        private bool NotAPrime = false;
    }
}
```

The following example shows the event handler for the [Button](#).

C#

 Copy

```
private void StartOrStop(object sender, EventArgs e)
{
    if (continueCalculating)
    {
        continueCalculating = false;
        startStopButton.Content = "Resume";
    }
    else
    {
        continueCalculating = true;
        startStopButton.Content = "Stop";
        startStopButton.Dispatcher.BeginInvoke(
            DispatcherPriority.Normal,
            new NextPrimeDelegate(CheckNextNumber));
    }
}
```

Besides updating the text on the [Button](#), this handler is responsible for scheduling the first prime number check by adding a delegate to the [Dispatcher](#) queue. Sometime after this event handler has completed its work, the [Dispatcher](#) will select this delegate for execution.

As we mentioned earlier, [BeginInvoke](#) is the [Dispatcher](#) member used to schedule a delegate for execution. In this case, we choose the [SystemIdle](#) priority. The [Dispatcher](#) will execute this delegate only when there are no important events to process. UI responsiveness is more important than number checking. We also pass a new delegate representing the number-checking routine.

C#

 Copy


```

public void CheckNextNumber()
{
    // Reset flag.
    NotAPrime = false;

    for (long i = 3; i <= Math.Sqrt(num); i++)
    {
        if (num % i == 0)
        {
            // Set not a prime flag to true.
            NotAPrime = true;
            break;
        }
    }

    // If a prime number.
    if (!NotAPrime)
    {
        bigPrime.Text = num.ToString();
    }

    num += 2;
    if (continueCalculating)
    {
        startStopButton.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.SystemIdle,
            new NextPrimeDelegate(this.CheckNextNumber));
    }
}

private bool NotAPrime = false;

```

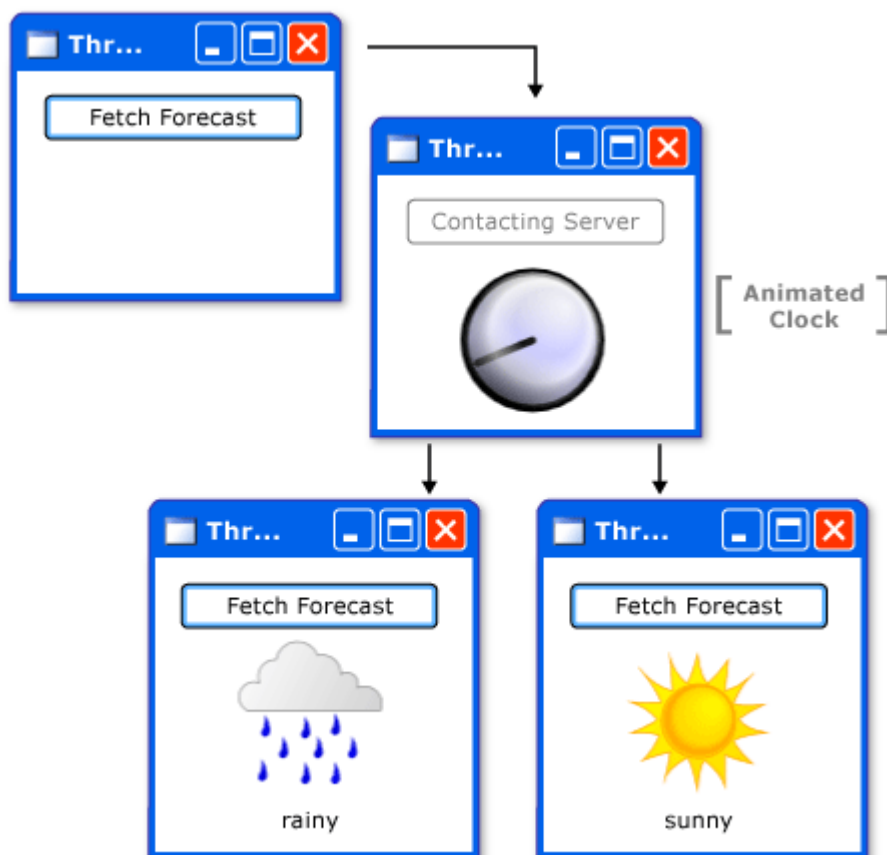
This method checks if the next odd number is prime. If it is prime, the method directly updates the `bigPrime` [TextBlock](#) to reflect its discovery. We can do this because the calculation is occurring in the same thread that was used to create the component. Had we chosen to use a separate thread for the calculation, we would have to use a more complicated synchronization mechanism and execute the update in the UI thread. We'll demonstrate this situation next.

For the complete source code for this sample, see the [Single-Threaded Application with Long-Running Calculation Sample](#)

Handling a Blocking Operation with a Background Thread

Handling blocking operations in a graphical application can be difficult. We don't want to call blocking methods from event handlers because the application will appear to freeze up. We can use a separate thread to handle these operations, but when we're done, we have to synchronize with the UI thread because we can't directly modify the GUI from our worker thread. We can use [Invoke](#) or [BeginInvoke](#) to insert delegates into the [Dispatcher](#) of the UI thread. Eventually, these delegates will be executed with permission to modify UI elements.

In this example, we mimic a remote procedure call that retrieves a weather forecast. We use a separate worker thread to execute this call, and we schedule an update method in the [Dispatcher](#) of the UI thread when we're finished.



C#

Copy

```
using System;  
using System.Windows;  
using System.Windows.Controls;
```

```

using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        // Delegates to be used in placking jobs onto the Dispatcher.
        private delegate void NoArgDelegate();
        private delegate void OneArgDelegate(String arg);

        // Storyboards for the animations.
        private Storyboard showClockFaceStoryboard;
        private Storyboard hideClockFaceStoryboard;
        private Storyboard showWeatherImageStoryboard;
        private Storyboard hideWeatherImageStoryboard;

        public Window1(): base()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // Load the storyboard resources.
            showClockFaceStoryboard =
                (Storyboard)this.Resources["ShowClockFaceStoryboard"];
            hideClockFaceStoryboard =
                (Storyboard)this.Resources["HideClockFaceStoryboard"];
            showWeatherImageStoryboard =
                (Storyboard)this.Resources["ShowWeatherImageStoryboard"];
            hideWeatherImageStoryboard =
                (Storyboard)this.Resources["HideWeatherImageStoryboard"];
        }

        private void ForecastButtonHandler(object sender, RoutedEventArgs
        {
            // Change the status image and start the rotation animation.
            fetchButton.IsEnabled = false;
            fetchButton.Content = "Contacting Server";
            weatherText.Text = "";
            hideWeatherImageStoryboard.Begin(this);

            // Start fetching the weather forecast asynchronously.
            NoArgDelegate fetcher = new NoArgDelegate(
                this.FetchWeatherFromServer);
        }
    }
}

```

```

        fetcher.BeginInvoke(null, null);
    }

    private void FetchWeatherFromServer()
    {
        // Simulate the delay from network access.
        Thread.Sleep(4000);

        // Tried and true method for weather forecasting - random number
        Random rand = new Random();
        String weather;

        if (rand.Next(2) == 0)
        {
            weather = "rainy";
        }
        else
        {
            weather = "sunny";
        }

        // Schedule the update function in the UI thread.
        tomorrowWeather.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.Normal,
            new OneArgDelegate(UpdateUserInterface),
            weather);
    }

    private void UpdateUserInterface(String weather)
    {
        //Set the weather image
        if (weather == "sunny")
        {
            weatherIndicatorImage.Source = (ImageSource)this.Resources
                ["SunnyImageSource"];
        }
        else if (weather == "rainy")
        {
            weatherIndicatorImage.Source = (ImageSource)this.Resources
                ["RainingImageSource"];
        }

        //Stop clock animation
        showClockFaceStoryboard.Stop(this);
        hideClockFaceStoryboard.Begin(this);

        //Update UI text
        fetchButton.IsEnabled = true;
        fetchButton.Content = "Fetch Forecast";
        weatherText.Text = weather;
    }

```

```

        private void HideClockFaceStoryboard_Completed(object sender,
            EventArgs args)
        {
            showWeatherImageStoryboard.Begin(this);
        }

        private void HideWeatherImageStoryboard_Completed(object sender,
            EventArgs args)
        {
            showClockFaceStoryboard.Begin(this, true);
        }
    }
}

```

The following are some of the details to be noted.

- Creating the Button Handler

```

C# Copy

private void ForecastButtonHandler(object sender, RoutedEventArgs e)
{
    // Change the status image and start the rotation animation.
    fetchButton.IsEnabled = false;
    fetchButton.Content = "Contacting Server";
    weatherText.Text = "";
    hideWeatherImageStoryboard.Begin(this);

    // Start fetching the weather forecast asynchronously.
    NoArgDelegate fetcher = new NoArgDelegate(
        this.FetchWeatherFromServer);

    fetcher.BeginInvoke(null, null);
}

```

When the button is clicked, we display the clock drawing and start animating it. We disable the button. We invoke the `FetchWeatherFromServer` method in a new thread, and then we return, allowing the [Dispatcher](#) to process events while we wait to collect the weather forecast.

- Fetching the Weather

```
C# Copy

private void FetchWeatherFromServer()
{
    // Simulate the delay from network access.
    Thread.Sleep(4000);

    // Tried and true method for weather forecasting - random number
    Random rand = new Random();
    String weather;

    if (rand.Next(2) == 0)
    {
        weather = "rainy";
    }
    else
    {
        weather = "sunny";
    }

    // Schedule the update function in the UI thread.
    tomorrowsWeather.Dispatcher.BeginInvoke(
        System.Windows.Threading.DispatcherPriority.Normal,
        new OneArgDelegate(UpdateUserInterface),
        weather);
}
```

To keep things simple, we don't actually have any networking code in this example. Instead, we simulate the delay of network access by putting our new thread to sleep for four seconds. In this time, the original UI thread is still running and responding to events. To show this, we've left an animation running, and the minimize and maximize buttons also continue to work.

When the delay is finished, and we've randomly selected our weather forecast, it's time to report back to the UI thread. We do this by scheduling a call to `UpdateUserInterface` in the UI thread using that thread's [Dispatcher](#). We pass a string describing the weather to this scheduled method call.

- Updating the UI

```
C# Copy
```

```

private void UpdateUserInterface(String weather)
{
    //Set the weather image
    if (weather == "sunny")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "SunnyImageSource"];
    }
    else if (weather == "rainy")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "RainingImageSource"];
    }

    //Stop clock animation
    showClockFaceStoryboard.Stop(this);
    hideClockFaceStoryboard.Begin(this);

    //Update UI text
    fetchButton.IsEnabled = true;
    fetchButton.Content = "Fetch Forecast";
    weatherText.Text = weather;
}

```

When the [Dispatcher](#) in the UI thread has time, it executes the scheduled call to `UpdateUserInterface`. This method stops the clock animation and chooses an image to describe the weather. It displays this image and restores the "fetch forecast" button.

Multiple Windows, Multiple Threads

Some WPF applications require multiple top-level windows. It is perfectly acceptable for one Thread/[Dispatcher](#) combination to manage multiple windows, but sometimes several threads do a better job. This is especially true if there is any chance that one of the windows will monopolize the thread.

Windows Explorer works in this fashion. Each new Explorer window belongs to the original process, but it is created under the control of an independent thread.

By using a WPF [Frame](#) control, we can display Web pages. We can easily create a simple Internet Explorer substitute. We start with an important feature: the ability to open a new explorer window. When the user clicks the "new window" button, we launch a copy of our window in a separate thread. This way, long-running or blocking operations in one of the windows won't lock all the other windows.

In reality, the Web browser model has its own complicated threading model. We've chosen it because it should be familiar to most readers.

The following example shows the code.

XAML

 Copy

```
<Window x:Class="SDKSamples.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MultiBrowse"
  Height="600"
  Width="800"
  Loaded="OnLoaded"
>
<StackPanel Name="Stack" Orientation="Vertical">
  <StackPanel Orientation="Horizontal">
    <Button Content="New Window"
      Click="NewWindowHandler" />
    <TextBox Name="newLocation"
      Width="500" />
    <Button Content="GO!"
      Click="Browse" />
  </StackPanel>

  <Frame Name="placeholder"
    Width="800"
    Height="550"></Frame>
</StackPanel>
</Window>
```

C#

 Copy

```
using System;
using System.Windows;
using System.Windows.Controls;
```



```

using System.Windows.Data;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        public Window1() : base()
        {
            InitializeComponent();
        }

        private void OnLoaded(object sender, RoutedEventArgs e)
        {
            placeholder.Source = new Uri("http://www.msn.com");
        }

        private void Browse(object sender, RoutedEventArgs e)
        {
            placeholder.Source = new Uri(newLocation.Text);
        }

        private void NewWindowHandler(object sender, RoutedEventArgs e)
        {
            Thread newWindowThread = new Thread(new ThreadStart(ThreadSt
            newWindowThread.SetApartmentState(ApartmentState.STA);
            newWindowThread.IsBackground = true;
            newWindowThread.Start();
        }

        private void ThreadStartingPoint()
        {
            Window1 tempWindow = new Window1();
            tempWindow.Show();
            System.Windows.Threading.Dispatcher.Run();
        }
    }
}

```

The following threading segments of this code are the most interesting to us in this context:

```
private void NewWindowHandler(object sender, RoutedEventArgs e)
{
    Thread newWindowThread = new Thread(new ThreadStart(ThreadStartingPo
newWindowThread.SetApartmentState(ApartmentState.STA);
newWindowThread.IsBackground = true;
newWindowThread.Start();
}
```

This method is called when the "new window" button is clicked. It creates a new thread and starts it asynchronously.

C#

 Copy

```
private void ThreadStartingPoint()
{
    Window1 tempWindow = new Window1();
    tempWindow.Show();
    System.Windows.Threading.Dispatcher.Run();
}
```

This method is the starting point for the new thread. We create a new window under the control of this thread. WPF automatically creates a new [Dispatcher](#) to manage the new thread. All we have to do to make the window functional is to start the [Dispatcher](#).

Technical Details and Stumbling Points

Writing Components Using Threading

The Microsoft .NET Framework Developer's Guide describes a pattern for how a component can expose asynchronous behavior to its clients (see [Event-based Asynchronous Pattern Overview](#)). For instance, suppose we wanted to package the `FetchWeatherFromServer` method into a reusable, nongraphical component. Following the standard Microsoft .NET Framework pattern, this would look something like the following.

```
public class WeatherComponent : Component
{
    //gets weather: Synchronous
    public string GetWeather()
    {
        string weather = "";

        //predict the weather

        return weather;
    }

    //get weather: Asynchronous
    public void GetWeatherAsync()
    {
        //get the weather
    }

    public event GetWeatherCompletedEventHandler GetWeatherCompleted;
}

public class GetWeatherCompletedEventArgs : AsyncCompletedEventArgs
{
    public GetWeatherCompletedEventArgs(Exception error, bool canceled,
        object userState, string weather)
        : base(error, canceled, userState)
    {
        _weather = weather;
    }

    public string Weather
    {
        get { return _weather; }
    }
    private string _weather;
}

public delegate void GetWeatherCompletedEventHandler(object sender,
    GetWeatherCompletedEventArgs e);
```

`GetWeatherAsync` would use one of the techniques described earlier, such as creating a background thread, to do the work asynchronously, not blocking the calling thread.

One of the most important parts of this pattern is calling the *MethodName* `Completed` method on the same thread that called the *MethodName* `Async` method to begin with. You could do this using WPF fairly easily, by storing [CurrentDispatcher](#)—but then the nongraphical component could only be used in WPF applications, not in Windows Forms or ASP.NET programs.

The [DispatcherSynchronizationContext](#) class addresses this need—think of it as a simplified version of [Dispatcher](#) that works with other UI frameworks as well.

C#

 Copy

```
public class WeatherComponent2 : Component
{
    public string GetWeather()
    {
        return fetchWeatherFromServer();
    }

    private DispatcherSynchronizationContext requestingContext = null;

    public void GetWeatherAsync()
    {
        if (requestingContext != null)
            throw new InvalidOperationException("This component can only

        requestingContext = (DispatcherSynchronizationContext)Dispatcher

        NoArgDelegate fetcher = new NoArgDelegate(this.fetchWeatherFromS

        // Launch thread
        fetcher.BeginInvoke(null, null);
    }

    private void RaiseEvent(GetWeatherCompletedEventArgs e)
    {
        if (GetWeatherCompleted != null)
            GetWeatherCompleted(this, e);
    }

    private string fetchWeatherFromServer()
    {
        // do stuff
        string weather = "";

        GetWeatherCompletedEventArgs e =
            new GetWeatherCompletedEventArgs(null, false, null, weather)
```

```

        SendOrPostCallback callback = new SendOrPostCallback(DoEvent);
        requestingContext.Post(callback, e);
        requestingContext = null;

        return e.Weather;
    }

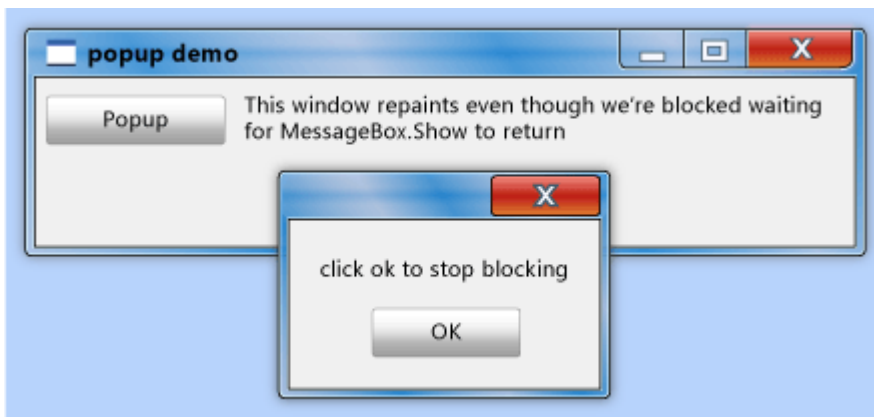
    private void DoEvent(object e)
    {
        //do stuff
    }

    public event GetWeatherCompletedEventHandler GetWeatherCompleted;
    public delegate string NoArgDelegate();
}

```

Nested Pumping

Sometimes it is not feasible to completely lock up the UI thread. Let's consider the [Show](#) method of the [MessageBox](#) class. [Show](#) doesn't return until the user clicks the OK button. It does, however, create a window that must have a message loop in order to be interactive. While we are waiting for the user to click OK, the original application window does not respond to user input. It does, however, continue to process paint messages. The original window redraws itself when covered and revealed.



Some thread must be in charge of the message box window. WPF could create a new thread just for the message box window, but this thread would be unable

to paint the disabled elements in the original window (remember the earlier discussion of mutual exclusion). Instead, WPF uses a nested message processing system. The [Dispatcher](#) class includes a special method called [PushFrame](#), which stores an application's current execution point then begins a new message loop. When the nested message loop finishes, execution resumes after the original [PushFrame](#) call.

In this case, [PushFrame](#) maintains the program context at the call to [MessageBox.Show](#), and it starts a new message loop to repaint the background window and handle input to the message box window. When the user clicks OK and clears the pop-up window, the nested loop exits and control resumes after the call to [Show](#).

Stale Routed Events

The routed event system in WPF notifies entire trees when events are raised.

XAML	 Copy
<pre><Canvas MouseLeftButtonDown="handler1" Width="100" Height="100" > <Ellipse Width="50" Height="50" Fill="Blue" Canvas.Left="30" Canvas.Top="50" MouseLeftButtonDown="handler2" /> </Canvas></pre>	

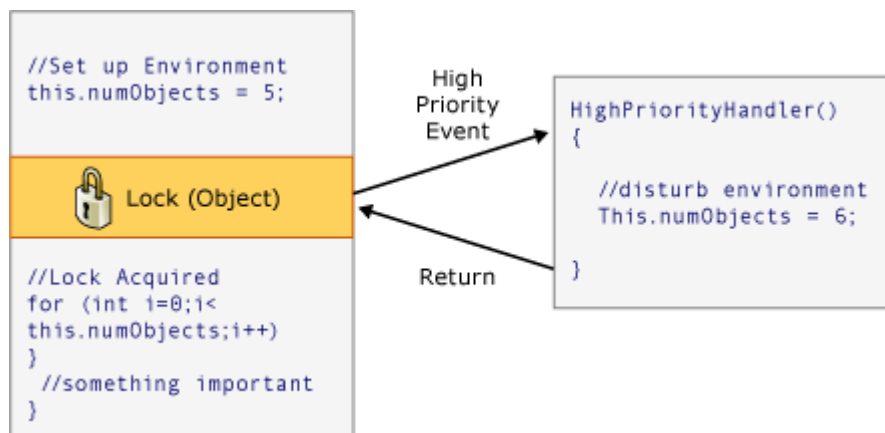
When the left mouse button is pressed over the ellipse, `handler2` is executed. After `handler2` finishes, the event is passed along to the [Canvas](#) object, which uses `handler1` to process it. This happens only if `handler2` does not explicitly mark the event object as handled.

It's possible that `handler2` will take a great deal of time processing this event. `handler2` might use [PushFrame](#) to begin a nested message loop that doesn't return for hours. If `handler2` does not mark the event as handled when this message loop is complete, the event is passed up the tree even though it is very old.

Reentrancy and Locking

The locking mechanism of the common language runtime (CLR) doesn't behave exactly as one might imagine; one might expect a thread to cease operation completely when requesting a lock. In actuality, the thread continues to receive and process high-priority messages. This helps prevent deadlocks and make interfaces minimally responsive, but it introduces the possibility for subtle bugs. The vast majority of the time you don't need to know anything about this, but under rare circumstances (usually involving Win32 window messages or COM STA components) this can be worth knowing.

Most interfaces are not built with thread safety in mind because developers work under the assumption that a UI is never accessed by more than one thread. In this case, that single thread may make environmental changes at unexpected times, causing those ill effects that the [DispatcherObject](#) mutual exclusion mechanism is supposed to solve. Consider the following pseudocode:



Most of the time that's the right thing, but there are times in WPF where such unexpected reentrancy can really cause problems. So, at certain key times, WPF calls [DisableProcessing](#), which changes the lock instruction for that thread to use the WPF reentrancy-free lock, instead of the usual CLR lock.

So why did the CLR team choose this behavior? It had to do with COM STA objects and the finalization thread. When an object is garbage collected, its `Finalize` method is run on the dedicated finalizer thread, not the UI thread. And therein lies the problem, because a COM STA object that was created on the UI thread can only be disposed on the UI thread. The CLR does the equivalent of a [BeginInvoke](#) (in this case using Win32's `SendMessage`). But if the UI thread is busy, the finalizer thread is stalled and the COM STA object can't be disposed, which creates a serious memory leak. So the CLR team made the tough call to make locks work the way they do.

The task for WPF is to avoid unexpected reentrancy without reintroducing the memory leak, which is why we don't block reentrancy everywhere.

See Also

[Single-Threaded Application with Long-Running Calculation Sample](#)