WPF Threads

Build More Responsive Apps With The Dispatcher

Shawn Wildermuth

This article discusses:

- Threading in WPF
- Using the Dispatcher
- Non-UI thread handling
- Using timers

This article uses the following technologies:
.NET Framework 3.0, WIndows Presentation Foundation

Contents

It would be a shame if you put months of your life into creating an intuitive, natural, and even beautiful interface only to have your users tapping their fingers on their collective desks waiting for it to respond. Watching your application screech to a halt because of a long-running process is just painful. However, creating applications that are responsive takes some careful planning, which usually entails having long-running processes work in other threads so that the UI thread is free to keep up with the user.

My first real experience with responsiveness goes way back to Visual C++® and MFC and the first grid I ever wrote. I was helping write a pharmacy application that had to be able to show every drug in a complex formulary. The problem was that there were 30,000 drugs, so we decided to give the appearance of responsiveness by filling the first screenful of drugs in the UI thread (in about 50 milliseconds), then using a background thread to finish filling the non-visible drugs (in about 10 seconds). The project went well and I learned the valuable lesson that user perception can be more important than reality.

Windows® Presentation Foundation (WPF) is a great technology for creating compelling user interfaces, but that doesn't mean you don't have to take the responsiveness of your application into account. The simple fact is, no matter what type of long-running processes are involved—whether getting large results from a database, making asynchronous Web service calls, or any number of other potentially intensive operations—making your application more responsive is guaranteed to make your users much happier in the long run. But before you can start using an asynchronous programming model in your WPF application, it is important that you understand the WPF threading model. In this article I will not only introduce you to that threading model, but I'll also show you how the Dispatcher-based objects work and explain how to use the BackgroundWorker so that you can create compelling and responsive user interfaces.

The Threading Model

All WPF applications start out with two important threads, one for rendering and one for managing the user interface. The rendering thread is a hidden thread that runs in the background, so the only thread that you ordinarily deal with is the UI thread. WPF requires that most of its objects be tied to the UI thread. This is known as thread affinity, meaning you can only use a WPF object on the thread on which it was created. Using it on other threads will cause a runtime exception to be thrown. Note that the WPF threading model interoperates well with Win32®-based APIs. This means that WPF can host or be hosted by any HWND-based API (Windows Forms, Visual Basic®, MFC, or even Win32).

The thread affinity is handled by the Dispatcher class, a prioritized message loop for WPF applications. Typically your WPF projects have a single Dispatcher object (and therefore a single UI thread) that all user interface work is channeled through.

Unlike typical message loops, each work item that is sent to WPF is sent through the Dispatcher with a specific priority. This allows for both ordering of items by priority and deferring certain types of work until the system has time to handle them. (For example, some work items can be deferred until the system or application is idle.) Supporting item prioritization allows WPF to allow certain types of work to have more access, and therefore more time on a thread, than other work.

Later in this article I will demonstrate that the rendering engine is allowed to update the user interface at a higher priority than the input system. This means that animations will continue to update the user interface no matter what the user is doing with the mouse, keyboard, or ink system. This can make the user interface appear more responsive. For example, let's assume you were writing a music-playing application (like Windows Media® Player). You would most likely want the information about the music playing (including the progress bar and other information) to show up whether the user was using the interface or not. To the user, this can make the interface appear more responsive to what they are most interested in (listening to music, in this case).

In addition to using the Dispatcher's message loop to channel items of work through the user interface thread, every WPF object is aware of the Dispatcher that is responsible for it (and therefore, the UI thread that it lives on). This means that any attempts to update WPF objects from secondary threads will fail. This is the responsibility of the DispatcherObject class.

MSDN Magazine Blog

5 Questions: Imagine That
In the October issue of MSDN Magazine, Don't Get Me Started columnist David Platt wrote about his experience acting as a judge at the Imagine Cup fina... More...
Thursday, Nov 3

MSDN Magazine November Issue Preview
The November issue of MSDN Magazine will be hitting the street next week, so it's a good time to offer a preview of what to expect. Leading things of... More...
Wednesday, Oct 26

More MSDN Magazine Blog entries >

Current Issue

Browse All MSDN Magazines

Subscribe to MSDN Flash newsletter

Receive the MSDN Flash e-mail newsletter every other week, with news and information personalized to your interests and areas of focus.

DispatcherObject

In the hierarchy of classes in WPF, most derive centrally from the DispatcherObject class (through other classes). As shown in **Figure 1**, you can see that the DispatcherObject virtual class is sandwiched just below Object in the hierarchy of most WPF classes.

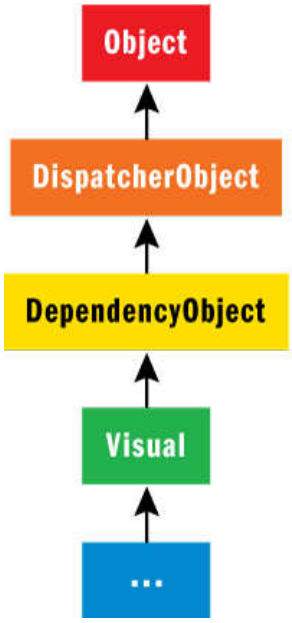

Figure 1 **DispatcherObject Derivation**

The DispatcherObject class has two chief duties: to provide access to the current Dispatcher that an object is tied to and provide methods to check (CheckAccess) and verify (VerifyAccess) that a thread has access to the object (derived from DispatcherObject). The difference between CheckAccess and VerifyAccess is that CheckAccess returns a Boolean value that represents whether the current thread can use the object and VerifyAccess throws an exception if the thread does not have access to the object. By providing this basic functionality, all the WPF objects support being able to determine whether they can be used on a particular thread—specifically, the UI thread. If you are writing your own WPF objects, such as controls, all methods you use should call VerifyAccess before they perform any work. This guarantees that your objects are only used on the UI thread, as shown in **Figure 2**.

Figure 2 Using VerifyAccess and CheckAccess

```
public class MyWpfObject : DispatcherObject
{
    public void DoSomething()
    {
        VerifyAccess();

        // Do some work
    }

    public void DoSomethingElse()
    {
        if (CheckAccess())
        {
            // Something, only if called
            // on the right thread
        }
    }
}
```

With this in mind, be careful to be on the UI thread when calling any DispatcherObject-derived object such as Control, Window, Panel, and so on. If you make a call to a DispatcherObject from a non-UI thread, it will throw an exception. Instead, if you are working on a non-UI thread, you'll need to use the Dispatcher to update DispatcherObjects.

Using the Dispatcher

The Dispatcher class provides a gateway to the message pump in WPF and provides a mechanism to route work for processing by the UI thread. This is necessary to meet the thread affinity demands, but since the UI thread is blocked for each piece of work routed through the Dispatcher, it is important to keep the work that the Dispatcher does small and quick. It is better to break apart larger pieces of work for the user interface into small discrete blocks for the Dispatcher to execute. Any work that doesn't need to be done on the UI thread should instead be moved off onto other threads for processing in the background.

Typically you will use the Dispatcher class to send worker items to the UI thread for processing. For example, if you want to do some work on a separate thread using the Thread class, you could create a ThreadStart delegate with some work to do on a new thread as shown in **Figure 3**.

Figure 3 Updating UI with Non-UI Thread—The Wrong Way

```
// The Work to perform on another thread
ThreadStart start = delegate()
{
    // ...

    // This will throw an exception
    // (it's on the wrong thread)
    statusText.Text = "From Other Thread";
};

// Create the thread and kick it started!
new Thread(start).Start();
```

This code fails because setting the Text property of the statusText control (a TextBlock) is not being called on the UI thread. When the code attempts to set the Text on the TextBlock, the TextBlock class internally calls its VerifyAccess method to ensure that the call is coming from the UI thread. When it determines the call is from a different thread, it throws an exception. So how can you use the Dispatcher to make the call on the UI thread?

The Dispatcher class provides access to invoke code on the UI thread directly. **Figure 4** shows the use of the Dispatcher's Invoke method to call a method called SetStatus to change the TextBlock's Text property for you.

Figure 4 Updating the UI

```
// The Work to perform on another thread
ThreadStart start = delegate()
{
  // ...

  // Sets the Text on a TextBlock Control.
  // This will work as its using the dispatcher
  Dispatcher.Invoke(DispatcherPriority.Normal,
                    new Action<string>(SetStatus),
                    "From Other Thread");
};
// Create the thread and kick it started!
new Thread(start).Start();
```

The Invoke call takes three pieces of information: the priority of the item to be executed, a delegate that describes what work to perform, and any parameters to pass into the delegate described in the second parameter. By calling Invoke, it queues up the delegate to be called on the UI thread. Using the Invoke method ensures that you are going to block until the work is performed on the UI thread.

As an alternative to using the Dispatcher synchronously, you can use the BeginInvoke method of the Dispatcher to asynchronously queue up a work item for the UI thread. Calling the BeginInvoke method returns an instance of the DispatcherOperation class that contains information about the execution of the work item, including the current status of the work item and the result of the execution (once the work item has completed). The use of the BeginInvoke method and the DispatcherOperation class is shown in **Figure 5**.

Figure 5 Updating the UI Asynchronously

```
// The Work to perform on another thread
ThreadStart start = delegate()
{
    // ...

    // This will work as its using the dispatcher
    DispatcherOperation op = Dispatcher.BeginInvoke(
        DispatcherPriority.Normal,
        new Action<string>(SetStatus),
        "From Other Thread (Async)");

    DispatcherOperationStatus status = op.Status;
    while (status != DispatcherOperationStatus.Completed)
    {
        status = op.Wait(TimeSpan.FromMilliseconds(1000));
        if (status == DispatcherOperationStatus.Aborted)
        {
            // Alert Someone
        }
    }
};

// Create the thread and kick it started!
new Thread(start).Start();
```

Unlike a typical message pump implementation, the Dispatcher is a priority-based queue of work items. This allows for better responsiveness because the more important pieces of work are executed before less important pieces of work. The nature of the prioritization is exemplified by the priorities that are specified in the DispatchPriority enumeration (as shown in **Figure 6**).

Figure 6 DispatchPriority Prioritization Levels (in Priority Order)

Priority	Description
Inactive	Work items are queued but not processed.
SystemIdle	Work items are only dispatched to the UI thread when the system is idle. This is the lowest priority of items that are actually processed.
ApplicationIdle	Work items are only dispatched to the UI thread when the application itself is idle.
ContextIdle	Work items are only dispatched to the UI thread after higher-priority work items are processed.
Background	Work items are dispatched after all layout, rendering, and input items are processed.
Input	Work items are dispatched to the UI thread at the same priority as user input.
Loaded	Work items are dispatched to the UI thread after all layout and rendering are complete.
Render	Work items are dispatched to the UI thread at the same priority as the rendering engine.
DataBind	Work items are dispatched to the UI thread at the same priority as data binding.
Normal	Work items are dispatched to the UI thread with normal priority. This is the priority at which most application work items should be dispatched.
Send	Work items are dispatched to the UI thread with the highest priority.

Generally you should always use DispatcherPriority.Normal for the priority of work items that update the appearance of the UI (like the example I used earlier). But there are times when you should use different priorities. Of particular interest are the three idle priorities (ContextIdle, ApplicationIdle, and SystemIdle). These priorities allow you to specify work items that are only executed under very low workloads.

BackgroundWorker

Now that you have a sense of how the Dispatcher works, you might be surprised to know that you will not find use for it in most cases. In Windows Forms 2.0, Microsoft introduced a class for non-UI thread handling to simplify the development model for user interface developers. This class is called the BackgroundWorker. **Figure 7** shows typical usage of the BackgroundWorker class.

Figure 7 Using a BackgroundWorker in WPF

```
BackgroundWorker _backgroundWorker = new BackgroundWorker();

...

// Set up the Background Worker Events
_backgroundWorker.DoWork += _backgroundWorker_DoWork;
backgroundWorker.RunWorkerCompleted +=
    _backgroundWorker_RunWorkerCompleted;

// Run the Background Worker
_backgroundWorker.RunWorkerAsync(5000);

...

// Worker Method
void _backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // Do something
}

// Completed Method
void _backgroundWorker_RunWorkerCompleted(
    object sender,
    RunWorkerCompletedEventArgs e)
{
    if (e.Cancelled)
    {
        statusText.Text = "Cancelled";
    }
    else if (e.Error != null)
    {
        statusText.Text = "Exception Thrown";
    }
    else
    {
        statusText.Text = "Completed";
    }
}
```

The BackgroundWorker component works well with WPF because underneath the covers it uses the AsyncOperationManager class, which in turn uses the SynchronizationContext class to deal with synchronization. In Windows Forms, the AsyncOperationManager hands off a WindowsFormsSynchronizationContext class that derives from the SynchronizationContext class. Likewise, in ASP.NET it works with a different derivation of SynchronizationContext called AspNetSynchronizationContext. These SynchronizationContext-derived classes know how to handle the cross-thread synchronization of method invocation.

In WPF, this model is extended with a DispatcherSynchronizationContext class. By using BackgroundWorker, the Dispatcher is being employed automatically to invoke cross-thread method calls. The good news is that since you are probably already familiar with this common pattern, you can continue using BackgroundWorker in your new WPF projects.

DispatcherTimer

WPF Threading Resources

- Windows Presentation Foundation Virtual Labs
- WPF in the .NET Framework Developer Center
- WPF Fundamentals: Threading Model, MSDN Library
- *Programming the Windows Presentation Foundation* by Chris Sells and Ian Griffiths (O'Reilly, 2005)

Periodic execution of code is a common task in development within the Microsoft® .NET Framework, but using timers in .NET has been, at best, confusing. If you look for a Timer class in the .NET Framework Base Class Library (BCL) you will find no fewer than three Timer classes: System.Threading.Timer, System.Timers.Timer, and System.Windows.Forms.Timer. Each of these timers is different. Alex Calvo's article in *MSDN Magazine* explains when to use each of these Timer classes (see msdn.microsoft.com/msdnmag/issues/04/02/TimersinNET).

For WPF applications, there is a new type of timer that utilizes the Dispatcher—the DispatcherTimer class. Like other timers, the DispatcherTimer class supports specifying an interval between ticks as well as the code to run when the timer's event fires. You can see a fairly pedestrian use of the DispatcherTimer in **Figure 8**.

Figure 8 DispatcherTimer Class in Action

```
// Create a Timer with a Normal Priority
_timer = new DispatcherTimer();

// Set the Interval to 2 seconds
_timer.Interval = TimeSpan.FromMilliseconds(2000);

// Set the callback to just show the time ticking away
// NOTE: We are using a control so this has to run on
// the UI thread
_timer.Tick += new EventHandler(delegate(object s, EventArgs a)
{
    statusText.Text = string.Format(
        "Timer Ticked:  {0}ms", Environment.TickCount);
});

// Start the timer
_timer.Start();
```

Because the DispatcherTimer class is tied to Dispatcher you can also specify the DispatcherPriority as well as which Dispatcher to use. The DispatcherTimer class uses the Normal priority as the current Dispatcher by default, but you can override these values:

```
_timer = new DispatcherTimer(
    DispatcherPriority.SystemIdle, form1.Dispatcher);
```

It is well worth all the effort involved in planning your worker processes to enable more responsive applications. Doing some initial research can make that planning more succesful. I suggest exploring some of the sites mentioned in the "WPF Threading References" sidebar before you begin—along with this article, they should give you a good foundation for developing responsive apps.

Shawn Wildermuth is a Microsoft MVP, MCSD.NET, MCT, and the founder of Wildermuth Consulting Services. Shawn is also the author of Pragmatic ADO.NET (Addison-Wesley, 2002) and the coauthor of four Microsoft Certification Training Kits as well as the upcoming Prescriptive Data Architectures. He can be contacted through his Web site at wildermuthconsulting.com.