All Topics, C#, .NET >> Files and Folders >> File System
http://www.codeproject.com/cs/files/directorywatcherservice.asp

C# (C# 2.0)
Windows, .NET (.NET 2.0)
Win32, VS (VS2005)
Dev
Posted: **1 Jan 2007**
Updated: **8 Feb 2007**
Views: **16,325**

Generic Directory Watcher Service

By Luke Stratman.

This service watches for filesystem events in directories and runs specified programs in response to those events.

18 votes for this article.
Popularity: 5.9. Rating: **4.7** out of 5.

- Download Directory Watcher service - 11.3 Kb
- Download source files - 26.2 Kb

Introduction

As I was experimenting with the media center functionality of Windows Vista RC2 one afternoon, I realized that it would be great to have a service running that would automatically transcode recordings from Microsoft's heavyweight DVR-MS format to a more svelte WMV file. Well, a utility (DVRMSToolbox) already exists to handle this scenario. But what if you wanted to add more functionality, like reorganizing the recording by automatically renaming it and copying it to another directory? Thus was born the idea for the Directory Watcher, a generic service that watches directories contained in the configuration file for fileysystem events and then runs specified applications in response to those events.

Background

The Directory Watcher service makes use of several .NET features and programming concepts to do its job, the first and foremost of which is the `FileSystemWatcher` class. This is a built-in class in the .NET framework that resides in the `System.IO` namespace and handles all of the heavy lifting involved in watching a directory for filesystem events. The real work that this service does is to spawn and regulate handler threads in response to these events which segues nicely into the next concept, the `CountingSempahore`.

The `CountingSemaphore` class is a custom class that is implemented by this service and extends the standard, binary semaphore behavior provided by the built-in .NET `Monitor` class. Semaphores are used to provide thread synchronization by controlling access to a critical section by multiple executing threads. .NET provides binary semaphore, or mutex, functionality through use of the `lock` keyword or through direct use of the Monitor class: these two constructs ensure that only one thread at a time can access a section of code.

However, what we need in this application is a resource-based semaphore: instead of restricting access to a section of code to a single thread, we want n number of threads to be able to access the section at a given time. Through this, we can control the number of processes being executed at a given time in response to filesystem events. The CountingSemaphore class provides this: the first n number of threads entering a critical section will claim one resource and be allowed to execute, but subsequent threads will block trying to claim a resource and will be forced to wait until one of the initial threads finishes before it can execute.

Finally, we make use of framework's ability to runtime-compile code in order to provide a measure of scripting support. When trying to do something extremely simple, like send an email to someone, to handle a filesystem event it's often a PITA to compile and maintain an entirely separate executable for the task. So, the service allows small snippets of .NET code to be specified that will handle events (they must meet certain criteria that will be covered later). We make use of the classes in the CodeDom namespace to perform runtime-compilation and generate assemblies in memory for these snippets. The service's filesystem event handler code then invokes the handler classes defined therein in addition to giving the user the option of using traditional, pre-compiled applications to handle events.

Implementation details

Configuration

One of my favorite improvements in .NET 2.0 are the dramatically improved configuration classes: ConfigurationElement, ConfigurationSection, ConfigurationElementCollection, etc. They make it relatively painless to define configuration data classes, declaratively populate them through the *App.config* file, and access them programatically (as opposed to being forced to load the *config* file contents into an XmlDocument object and perform XPath queries against it). The Directory Watcher service makes full use of this functionality; below is a sample *App.config* file that mimics the IIS SMTP pickup directory behavior, but uses Cygwin's exim MTA instead:

Collapse

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="watchInformation"
       type="DirectoryWatcher.WatchInformation, DirectoryWatcher"/>
  </configSections>
  <appSettings>
    <add key="maxConcurrentProcesses" value="20"/>
  </appSettings>
  <watchInformation>
```

```
    <directoriesToWatch>
      <directoryToWatch path="c:\Cygwin\var\spool\exim\pickup">
        <fileSetsToWatch>
          <fileSetToWatch>
            <eventsToWatch>
              <eventToWatch type="Created"/>
            </eventsToWatch>
            <programsToExecute>
              <programToExecute path="c:\Cygwin\bin\exim-4.52-2.exe"
                 arguments="-odf -t" redirectFileToStdin="true"/>
              <programToExecute path="c:\Cygwin\bin\rm.exe"
                 arguments="&quot;{P}&quot;"/>
            </programsToExecute>
          </fileSetToWatch>
        </fileSetsToWatch>
      </directoryToWatch>
    </directoriesToWatch>
  </watchInformation>
</configuration>
```

All of the data contained therein is referenced through classes that inherit from `ConfigurationElement`, `ConfigurationSection`, or `ConfigurationElementCollection`. The `<configSections/>` node contains the class and assembly reference that tells .NET's `ConfigurationManager` class that the `<watchInformation/>` node and all of its children are represented by the `WatchInformation` class. If we look at the code for this class, we see that it is very simple:

```
public class WatchInformation : ConfigurationSection
{
  /// <summary>
  /// Collection of directories that we are to watch for filesystem changes.
  /// </summary>
  [ConfigurationProperty("directoriesToWatch", IsRequired = true)]
  public DirectoryToWatchCollection DirectoriesToWatch
  {
    get
    {
      return (DirectoryToWatchCollection)base["directoriesToWatch"];
    }
  }
}
```

By flagging the `DirectoriesToWatch` property with the `ConfigurationProperty` attribute, it tells us that the `<directoriesToWatch/>` node is represented by an object of type `DirectoryToWatchCollection`. To retrieve the information from the configuration file for a property, all you have to do is reference `base ["nodeOrAttributeName"]` and cast it as the type the node represents. .NET's configuration classes will take care of all of the rest. This behavior follows recursively through the rest of the child nodes; an exhaustive analysis of all of the configuration classes used in this application isn't necessary since the classes basically just map nodes and attributes in the *config* file to properties in the classes. The MSDN documentation on the `System.Configuration` namespace provides a thorough coverage of the topic here.

However, there is definitely a gotcha with regards to the configuration classes: they don't provide a direct way for you to store configuration information for a property in a text node instead of an attribute. For instance, when you look at a typical line in the *config* file,

```
<programToExecute path="c:\Cygwin\bin\exim-4.52-2.exe" arguments="-odf -t"
  redirectFileToStdin="true"/>
```

you see that values for each property (of the `ProgramToExecute` class in this case) are contained in an attribute. This works fine for simple values, but what about for free-form text like snippets of .NET code that we need to be able to specify in order to provide runtime-compilation support? Storing that sort of data in an attribute just isn't practical, but the configuration classes just don't provide default functionality for storing data in text nodes: they see all nodes as complex objects and all attributes as simple values. This is where the old way of doing XML serialization provided a lot more flexibility: you would simply flag the property that you wanted stored in a node with an `[XmlElement()]` attribute instead of with an `[XmlAttribute()]` attribute and you were off to the races. While there is a way to get this functionality in the configuration classes, it's not really documented by Microsoft (or anyone else, as far as I could tell) and involved me hacking around in the `System.Configuration` assembly (thank God for Lutz Roeder's .NET Reflector) to figure out what was going on. It's actually pretty simple and here's the code for `ProgramCode`, the configuration class for runtime-compiled code where we had to implement this functionality:

Collapse

```
public class ProgramCode : ConfigurationElement
{
  /// <summary>
  /// The text of the actual code that we are to compile.
  /// </summary>
  private string text = null;
  /// <summary>
  /// Line number in the configuration file for this element
  /// (used for possible exception messages).
  /// </summary>
  private int lineNumber = 0;
  /// <summary>
  /// Path to the configuration file in which this element resides
  /// (used for possible exception messages).
  /// </summary>
  private string fileName = "";
  /// <summary>
  /// Assembly that results when we compile the code.
```

```
    /// </summary>
    private Assembly assembly = null;

    /// <summary>
    /// The language for this code snippet.
    /// </summary>
    [ConfigurationProperty("language", IsRequired = true)]
    public ProgramLanguage Language
    {
      get
      {
        return (ProgramLanguage)base["language"];
      }
    }

    /// <summary>
    /// Text representing the actual code.
    /// </summary>
    public string Text
    {
      get
      {
        return text;
      }
    }

    /// <summary>
    /// Collection of referenced assemblies for this snippet of code.
    /// </summary>
    [ConfigurationProperty("referencedAssemblies")]
    public ReferencedAssemblyCollection ReferencedAssemblies
    {
      get
      {
        return (ReferencedAssemblyCollection)base["referencedAssemblies"];
      }
    }

    /// <summary>
    /// Assembly representing the compiled results of the code.
    /// </summary>
    public Assembly Assembly
    {
      get
      {
        // Omitted for brevity's sake, we'll cover this later
      }
    }

    /// <summary>
    /// Handler for the case where we encounter an unrecognized element
    /// while attempting to deserialize the class from XML; deals with
    /// "custom" properties, specifically the Text property, whose value is set
    /// in an element node instead of an attribute.
```

```
  /// </summary>
  /// <param name="elementName">
  /// Name of the unrecognized element.
  /// </param>
  /// <param name="reader">
  /// Reader object that is involved in the deserialization.
  /// </param>
  /// <returns>
  /// True if we actually recognize the element, false otherwise.
  /// </returns>
  protected override bool OnDeserializeUnrecognizedElement(string elementName,
                                                           XmlReader reader)
  {
    if (elementName == "text")
    {
      text = reader.ReadString();
      reader.Read();

      return true;
    }

    return base.OnDeserializeUnrecognizedElement(elementName, reader);
  }

  /// <summary>
  /// Instantiates the object using data stored in XML;
  /// records the filename and line number
  /// (for use later in possible exceptions) and then calls the base method.
  /// </summary>
  /// <param name="reader">
  /// Reader object that is involved in the deserialization.
  /// </param>
  /// <param name="serializeCollectionKey">
  /// True to serialize only the collection key properties, false otherwise.
  /// </param>
  protected override void DeserializeElement(XmlReader reader,
                                             bool serializeCollectionKey)
  {
    lineNumber = ((IConfigErrorInfo)reader).LineNumber;
    fileName = ((IConfigErrorInfo)reader).Filename;

    base.DeserializeElement(reader, serializeCollectionKey);
  }

  /// <summary>
  /// Called after the deserialization process is complete;
  /// validates the object's data.
  /// </summary>
  protected override void PostDeserialize()
  {
    if (text == null)
      throw new ConfigurationErrorsException
        ("\"text\" is a required element.", fileName, lineNumber);
        base.PostDeserialize();
    }
}
```

First, you see that the Text property (which holds the actual text of the code that we're going to compile) is not decorated with a ConfigurationProperty attribute. This is because we don't want the configuration deserialization logic to try to automatically deserialize the node. Instead, we're going to rely on OnDeserializeUnrecognizedElement: this method is called during deserialization when a node that is not mapped to a property is encountered. It passes in the name of the element and the XmlReader object involved in the deserialization and allows for custom deserialization within the method. So we override this method: since the <text/> node won't be mapped to a configuration property, this method will be invoked when the deserializer encounters it. We check to see if the element name is "text" and, if so, all we have to do is set an internal member, text, to the string value of the node, advance the reader to the next node in the DOM, and return true. The return value seems to contradict what's documented on MSDN for this method: I don't know if I'm misreading their documentation but, to set the record straight, you want to return true when the node was actually valid and was processed, and false otherwise. A false return value will cause the configuration deserializer to throw an exception.

Service

Startup logic

The code in the service's OnStart event handler is fairly straightforward:

Collapse

```
protected override void OnStart(string[] args)
{
  WriteToEventLog(EventLogEntryType.Information,
                  "Starting up the Directory Watcher service.");

  try
  {
    // Get the section from the configuration file that contains the
    // directories/file sets that we are to watch
    watchInformation =
       (WatchInformation)ConfigurationManager.GetSection("watchInformation");

    // Get the maximum number of concurrent processes that can be active
    // (if specified)
    if (ConfigurationManager.AppSettings["maxConcurrentProcesses"] != null)
      maxConcurrentProcesses =
         Convert.ToUInt32(ConfigurationManager.AppSettings
                       ["maxConcurrentProcesses"]);

    WriteToEventLog(EventLogEntryType.Information,
                "Using a concurrent process count of {0}.",
                maxConcurrentProcesses);

    // Instantiate the regulation semaphore to enforce the maximum process
    // count
    executionRegulator = new CountingSemaphore(maxConcurrentProcesses);

    foreach (DirectoryToWatch directoryToWatch in
             watchInformation.DirectoriesToWatch)
    {
      foreach (FileSetToWatch fileSetToWatch in
               directoryToWatch.FileSetsToWatch)
      {
        // If we're using any runtime-compiled code, validate each assembly
        // that was generated to make sure that only one class exists in it
        // that implements the IFileSystemEventHandler interface
        foreach (ProgramToExecute programToExecute in
                 fileSetToWatch.ProgramsToExecute)
        {
          if (programToExecute.Code.Text != null)
            ValidateAssembly(programToExecute.Code.Assembly);
        }

        // Create and instantiate an individual FileSystemWatcher for
        // each wildcard file set and a single FileSystemWatcher object
        // to handle all regular expression file sets
        if (fileSetToWatch.MatchExpressionType ==
            MatchExpressionType.Wildcard ||
            watchers[directoryToWatch.Path][""] == null)
        {
          FileSystemWatcher watcher =
             new FileSystemWatcher(directoryToWatch.Path);

          // Set the filter to the match expression for wildcard file
          // sets, and blank (to capture changes for all files and do
          // the actual matching in the event handler) for regular
          // expression file sets
          watcher.Filter =
            (fileSetToWatch.MatchExpressionType ==
             MatchExpressionType.Wildcard ?
             fileSetToWatch.MatchExpression : "");

          // Attach handlers to the various filesystem events that we're
          // supposed to watch for
          if (fileSetToWatch.EventsToWatch["All"] != null)
          {
            watcher.Changed += new FileSystemEventHandler(watcher_OnChanged);
            watcher.Created += new FileSystemEventHandler(watcher_OnChanged);
            watcher.Deleted += new FileSystemEventHandler(watcher_OnChanged);
            watcher.Renamed += new RenamedEventHandler(watcher_OnChanged);
          }

          else
          {
            foreach (EventToWatch eventToWatch in
                     fileSetToWatch.EventsToWatch)
            {
              if (eventToWatch.Type == WatcherChangeTypes.Changed)
                watcher.Changed +=
                   new FileSystemEventHandler(watcher_OnChanged);

              else if (eventToWatch.Type == WatcherChangeTypes.Created)
                watcher.Created +=
                   new FileSystemEventHandler(watcher_OnChanged);

              else if (eventToWatch.Type == WatcherChangeTypes.Deleted)
                watcher.Deleted +=
                   new FileSystemEventHandler(watcher_OnChanged);

              else if (eventToWatch.Type == WatcherChangeTypes.Renamed)
                watcher.Renamed +=
                   new RenamedEventHandler(watcher_OnChanged);
            }
          }

          // Create a new dictionary entry for this directory path if
          // it doesn't exist already
          if (!watchers.ContainsKey(directoryToWatch.Path))
            watchers[directoryToWatch.Path] =
               new Dictionary<string, FileSystemWatcher>();

          // Add the watcher to the list for this directory and
          // enable it
          watchers[directoryToWatch.Path][watcher.Filter] = watcher;
          watcher.EnableRaisingEvents = true;
        }

        WriteToEventLog(EventLogEntryType.Information,
                        "Added watcher for the path \"{0}\"" +
                        "and the {1} \"{2}\".",
                        directoryToWatch.Path,
                        (fileSetToWatch.MatchExpressionType ==
                         MatchExpressionType.Wildcard ?
                         "wildcard expression" : "regular expression"),
                        fileSetToWatch.MatchExpression);
      }
    }
  }

  // Log any exceptions that occur during startup
  catch (Exception exception)
  {
    WriteToEventLog(EventLogEntryType.Error,
                    "Exception occurred while starting the service." +
                    "\n\nType: {0}\nMessage:{1}",
                    exception.GetType().FullName, exception.Message);
    throw;
  }

  base.OnStart(args);
}
```

We start by getting access to the configuration data in the *App.config* file by using .NET's built-in `ConfigurationManager` class. That one line of code is all that's necessary; the `ConfigurationManager` takes care of reading the data from the *config* file and instantiating all of the child properties:

```
watchInformation = (WatchInformation)ConfigurationManager.GetSection
                            ("watchInformation");
```

For each file set, we then check its handler programs and if any are represented by runtime compiled code, then we do a validation of the assembly that was generated to make sure that it meets our requirements. The validation functions that are involved are as follows:

Collapse

```
protected static void ValidateAssembly(Assembly assembly)
{
  FindEventHandlerType(assembly);
}

protected static Type FindEventHandlerType(Assembly assembly)
{
  Type eventHandlerType = null;

  foreach (Type type in assembly.GetTypes())
  {
    if (type.GetInterface("IFileSystemEventHandler") != null)
    {
      // If we've already found a qualifying type, then throw an exception
      if (eventHandlerType != null)
        throw new ArgumentException(
           String.Format("Multiple classes implementing " +
                         "IFileSystemEventHandler were found in {0}.",
                         assembly.FullName));

      eventHandlerType = type;
    }
  }

  // If no qualifying types were found, then throw an exception
  if (eventHandlerType == null)
    throw new ArgumentException(
       String.Format("No classes implementing IFileSystemEventHandler " +
                     "were found in {0}.", assembly.FullName));

  return eventHandlerType;
}
```

It just does a simple check of the assembly's types and makes sure that one, and only one, class (the main handler class) implements the `IFileSystemEventHandler` interface which is defined as follows:

```
public interface IFileSystemEventHandler
{
  /// <summary>
  /// Handler function that is called whenever a filesystem event occurs.
  /// </summary>
  /// <param name="e">
  /// Arguments (file name, directory, event type, etc.) associated with
  /// the event.
  /// </param>
  void OnFileSystemEvent(FileSystemEventArgs e);
}
```

It's then a simple matter of iterating over each directory, iterating over each file set contained within the directory, and creating `FileSystemWatcher` objects appropriately. The only wrinkle occurs when handling wildcard (`*.txt`) vs. regular expression (`^log_(october|november)`) matched file sets. The `FileSystemWatcher` class contains a `Filter` property that, when set, instructs the object to watch for events only for files matching the specified wildcard. So, when we preparing to watch a wildcard matched file set, we just set the `Filter` property and move on. However, for regular expression matched file sets, we have to do an evaluation of whether or not a file is within a particular set in the event handling code, so, for each directory path containing at least one regular expression matched file set, we declare a single, catch-all `FileSystemWatcher` object by setting its `Filter` property to a blank string (`""`). The event handler function (covered later) that is invoked whenever a change is detected will then take care of the actual regular expression evaluation to decide which file set (if any) a file belongs to and will invoke its handler applications accordingly.

Runtime compilation

Support for runtime-compiled code in the service is accomplished through the use of the `CodeDom` namespace and is implemented in the `ProgramCode` configuration class in the form of a property called `Assembly`, which is shown below:

Collapse

```
public Assembly Assembly
{
  get
  {
    // If the code has not already been compiled, do so now
    if (assembly == null)
    {
      CodeDomProvider codeProvider = null;

      // Get the proper code provider based on the code's language
      if (Language == ProgramLanguage.CSharp)
        codeProvider = new CSharpCodeProvider();
      else if (Language == ProgramLanguage.VisualBasic)
        codeProvider = new VBCodeProvider();

      CompilerParameters compilerParameters = new CompilerParameters();

      // Set the compiler options so that we don't generate an assembly
      // on disk and we create a library assembly that does not contain
      // debug information
      compilerParameters.GenerateExecutable = false;
      compilerParameters.GenerateInMemory = true;
      compilerParameters.IncludeDebugInformation = false;
      compilerParameters.CompilerOptions = "/target:library /optimize";
      compilerParameters.ReferencedAssemblies.Add("System.dll");
      compilerParameters.ReferencedAssemblies.Add(
         AppDomain.CurrentDomain.BaseDirectory +
         "\\DirectoryWatcher.exe");

      // Add any assembly references (besides System.dll and
      // DirectoryWatcher.exe, which everyone gets) specified for this
      // code
      foreach (ReferencedAssembly referencedAssembly in
               ReferencedAssemblies)
        compilerParameters.ReferencedAssemblies.Add(
          referencedAssembly.Name);

      // Generate the assembly
      CompilerResults results =
         codeProvider.CompileAssemblyFromSource(
             compilerParameters, text);

      // Check the return code and throw an exception if the compilation failed
      if (results.NativeCompilerReturnValue != 0)
        throw new CompilationException(results.Errors);

      assembly = results.CompiledAssembly;
    }

    return assembly;
  }
}
```

The `CodeDom` classes make it very easy to do this compilation. To start, we have to get the proper code provider type based on the language of the code snippet, then we have to set the compiler parameters appropriately. We want the resulting assembly to be as compact as possible so we explicitly exclude debug information and specify a compiler option for optimization. We also don't want to create any assembly files on disk, so we set the option telling the compiler to generate it in memory only. We then add the necessary assembly references (everyone gets `System` and *`DirectoryWatcher.exe`* since they're the bare minimum required for a handler class to operate) and invoke the compilation function. If we get a return code of 0, we know we succeeded and now have an `Assembly` object from which we can create types and handle filesystem events. Here is an example runtime-compiled program that sends an email notification by creating a file in the SMTP pickup directory used as an example earlier ("Use your EasyButton to find my EasyButton? Won't that, like . . . tear a hole in the universe or something?"):

Collapse

```
<programToExecute>
  <code language="CSharp">
    <text>
      <![CDATA[
        using System;
        using System.IO;
        using DirectoryWatcher;

        public class NotifyClass : IFileSystemEventHandler
        {
          public NotifyClass()
          {
          }

          public void OnFileSystemEvent(FileSystemEventArgs e)
          {
            string tempFileName = Path.GetTempFileName();
            StreamWriter writer = new StreamWriter(tempFileName);

            writer.WriteLine("To: lstratman@gmail.com");
            writer.WriteLine("From: lstratman@gmail.com");
            writer.WriteLine("Subject: File change notification");
            writer.WriteLine("");
            writer.WriteLine(e.FullPath + " has changed.");
            writer.Close();

            File.Move(tempFileName,
                      "c:\\Cygwin\\var\\spool\\exim\\" +
                      pickup\\email.txt");
          }
        }
      ]]>
    </text>
  </code>
</programToExecute>
```

Event handling logic

When a filesystem event is detected, a handler function is invoked that is responsible for running the necessary applications to respond to the event:

Collapse

```
protected void watcher_OnChanged(object source, FileSystemEventArgs e)
{
  FileSystemWatcher watcher = (FileSystemWatcher)source;
  DirectoryToWatch directoryToWatch =
     watchInformation.DirectoriesToWatch[watcher.Path];
```

```
    // If this watcher is a wildcard watcher, grab its programs from the
    // configuration section
    if (watcher.Filter != "")
    {
      ProgramToExecuteCollection programsToExecute =
         directoryToWatch.FileSetsToWatch[watcher.Filter].ProgramsToExecute;
      AddProgramsToQueue(programsToExecute, e.FullPath, e.ChangeType);
    }

    // Otherwise, go through the list of regular expression file sets for this
    // directory, see if any of them match the file that was modified, and, if
    // they do, get their programs from the configuration section
    else
    {
      foreach (FileSetToWatch fileSetToWatch in
               directoryToWatch.FileSetsToWatch)
      {
        if (fileSetToWatch.MatchExpressionType ==
            MatchExpressionType.RegularExpression &&
            fileSetToWatch.MatchRegex.IsMatch(e.Name))
          AddProgramsToQueue(fileSetToWatch.ProgramsToExecute, e.FullPath,
                             e.ChangeType);
      }
    }
  }
```

Again, it's pretty straightforward: if the source `FileSystemWatcher` object wasn't a catch-all for regular expression matched file sets (i.e. didn't have its `Filter` property set to `""`), then we reference the config data to get the list of programs that we're supposed to execute for this file set and pass them to another function responsible for starting up the programs. Otherwise, we iterate over each file set for the directory and, if it's a regular expression file set, we apply it against the file name and, if it matches, we pass its program list to the aforementioned program function. That program function is as follows:

Collapse

```
protected void AddProgramsToQueue(ProgramToExecuteCollection programsToExecute,
                                  string filePath,
                                  WatcherChangeTypes eventType)
{
  List<ExecutionInstance> executionInstances =
     new List<ExecutionInstance>();
  Thread executionThread =
     new Thread(new ParameterizedThreadStart(RunPrograms));

  // Loop through each program and create an ExecutionInstance object for it
  foreach (ProgramToExecute programToExecute in programsToExecute)
  {
    ExecutionInstance executionInstance;

    // If we're running a pre-compiled application, create the necessary
    // ProcessStartInfo object
    if (programToExecute.Code.Text == null)
    {
      ProcessStartInfo startInfo =
         new ProcessStartInfo(programToExecute.Path);
      FileInfo fileInfo = new FileInfo(filePath);

      startInfo.Arguments =
         programToExecute.Arguments.Replace("{P}", filePath);
      startInfo.Arguments =
         startInfo.Arguments.Replace("{F}", fileInfo.Name);
      startInfo.Arguments =
         startInfo.Arguments.Replace("{E}", fileInfo.Extension);
      startInfo.Arguments =
         startInfo.Arguments.Replace("{D}", fileInfo.DirectoryName);

      if (fileInfo.Extension != "")
        startInfo.Arguments =
           startInfo.Arguments.Replace("{f}", fileInfo.Name.Substring(0,
                                        fileInfo.Name.Length -
                                        fileInfo.Extension.Length - 1));

      else
        startInfo.Arguments =
           startInfo.Arguments.Replace("{f}", fileInfo.Name);

      startInfo.UseShellExecute = false;
      startInfo.RedirectStandardInput =
         programToExecute.RedirectFileToStdin;

      executionInstance =
         new ExecutionInstance(startInfo, eventType, filePath,
                               programToExecute.RedirectFileToStdin);
    }

    // Otherwise, we're using runtime-compiled code and we need to create
    // an instance of the class that implements IFileSystemEventHandler
    else
    {
      IFileSystemEventHandler eventHandler =
         CreateEventHandlerInstance(programToExecute.Code.Assembly);
      executionInstance = new ExecutionInstance(eventHandler, eventType,
                                                 filePath);
    }

    executionInstances.Add(executionInstance);
  }

  // Start the thread that will execute the programs
  executionThread.Start(executionInstances);
}
```

This function is responsible for taking a list of programs to execute and then spinning off a worker thread to execute those programs in sequence. The fact that an event can invoke more than one program is one reason why this must be handled in a separate thread: using the example of the SMTP pickup directory from the configuration section, we run one command to send the email and another to clean the message file up from the pickup directory. We obviously don't want the cleanup command run until the message sending command finishes, so we use a thread to start a program, wait until it completes, start the next program, and repeat until we reach the end of the list.

Another reason for spinning off a thread is that we want the calling event handler function to return as quickly as possible: if the event handler actually blocked waiting for the handling programs to run, then it's possible for the `FileSystemWatcher`'s internal buffers to fill up and for events to be dropped. So, this function creates a list of `ProcessStartInfo` (if the program is a traditional, pre-compiled application) and `IFileSystemEventHandler` (if the program is represented by runtime-compiled code) objects representing each program that should be run and then starts up another thread to run them in sequence. Finally, that thread's startup function looks like this:

Collapse

```
public static void RunPrograms(object source)
{
  List<ExecutionInstance> executionInstances =
              (List<ExecutionInstance>)source;

  // If we're watching for a create event, we first try to open the file in
  // exclusive mode; this is to account for the "long copy" scenario where
  // the create event is fired when the copy first starts, but we need to
  // wait until the copy completes before we begin our processing
  if (executionInstances[0].EventType == WatcherChangeTypes.Created)
  {
    FileStream fileStream = null;

    while (fileStream == null)
    {
      try
      {
        fileStream =
           File.Open(executionInstances[0].FilePath, FileMode.Open,
                     FileAccess.Read, FileShare.None);
      }

      // Catch the IOException that will be thrown when we fail to open
      // the file in exclusive mode
      catch (IOException exception)
      {
        string warningTrap = exception.Message;
        Thread.Sleep(1000);
      }

      // Log any other unhandled exceptions that are thrown
      catch (Exception exception)
      {
        WriteToEventLog(EventLogEntryType.Error,
                        "Unhandled exception occurred while waiting for " +
                        "\"{0}\" to become available." +
                        "\n\nType: {1}\nMessage:{2}",
                        executionInstances[0].FilePath,
                        exception.GetType().FullName,
                        exception.Message);
      }
    }

    fileStream.Close();
  }

  // Claim a resource from the counting semaphore and enter the critical
  // section
  executionRegulator.P();

  foreach (ExecutionInstance executionInstance in executionInstances)
  {
    try
    {
      // If we're running a pre-compiled application, start the
      // process
      if (executionInstance.EventHandler == null)
      {
        WriteToEventLog(EventLogEntryType.Information,
                        "Running program in response to event for " +
                        "{3} being {4}:\n\"{0}\"{1}{2}.",
                        executionInstance.StartInfo.FileName,
                        (executionInstance.StartInfo.Arguments != "" ?
                         " " + executionInstance.StartInfo.Arguments :
                         ""),
                        (executionInstance.RedirectFileToStdin ?
                         " < \"" + executionInstance.FilePath +
                         "\"" : ""),
                        executionInstance.FilePath,
                        executionInstance.EventType.ToString().ToLower());

        Process executionProcess =
           Process.Start(executionInstance.StartInfo);

        // If we're redirecting the file to the standard input stream, open
        // it up and read its contents into the stream in 1 KB chunks
        if (executionInstance.RedirectFileToStdin)
        {
          BinaryReader binaryReader =
             new BinaryReader(File.Open(executionInstance.FilePath,
                                        FileMode.Open));
          BinaryWriter binaryWriter =
             new BinaryWriter(executionProcess.StandardInput.BaseStream);
          byte[] buffer = new byte[1024];
          int readSize = binaryReader.Read(buffer, 0, buffer.Length);

          while (readSize != 0)
          {
            binaryWriter.Write(buffer, 0, readSize);
            readSize = binaryReader.Read(buffer, 0, buffer.Length);
          }

          binaryReader.Close();
          binaryWriter.Close();
        }
        // Wait for the process to exit and then clean it up
        executionProcess.WaitForExit();
        executionProcess.Close();
      }

      // Otherwise, invoke the OnFileSystemEvent() method for the
      // handler class defined in the runtime-compiled code
      else
      {
        WriteToEventLog(EventLogEntryType.Information,
                        "Invoking {0}.OnFileSystemEvent() in " +
                        "response to event for {1} being {2}.",
                        executionInstance.EventHandler.GetType().Name,
                        executionInstance.FilePath,
                        executionInstance.EventType.ToString().ToLower());

        FileInfo fileInfo = new FileInfo(executionInstance.FilePath);
        FileSystemEventArgs eventArguments =
           new FileSystemEventArgs(executionInstance.EventType,
                                   fileInfo.DirectoryName,
                                   fileInfo.Name);

        executionInstance.EventHandler.OnFileSystemEvent(
           eventArguments);
      }
    }

    // Log any exceptions that occur while running the program
    catch (Exception exception)
    {
      WriteToEventLog(EventLogEntryType.Error,
                      "Exception occurred while running \"{0}\"." +
                      "\n\nType: {1}\nMessage:{2}",
                      executionInstance.StartInfo.FileName,
                      exception.GetType().FullName, exception.Message);
    }
  }

  // Release the resource and exit the critical section
  executionRegulator.V();
}
```

The first thing it does is a trick to account for the "long copy" scenario. If a file is being created, but is being copied from somewhere, it could take a while if the file is large or if the transport protocol is slow. In either case, we don't want to start our handler applications until the file finishes copying, but the create event is fired when the copy operation first starts. So, we try to open the file with an exclusive lock, which will throw an `IOException` while the copy is still in progress. So, we catch that exception, sleep for one second, try again, and repeat until the open operation is successful. We then close the handle and continue executing. Before actually starting the programs, we claim a resource from the `CountingSemaphore`:

```
// Claim a resource from the counting semaphore and enter the critical section
```

```
executionRegulator.P();
```

If there are resources available, then the call will return immediately and we will continue executing. Otherwise, the call will block until a resource becomes available. This gives us a way to control the number of running processes in an environment where the churn on a directory may be very high. Once a resource is claimed, we branch depending on what type of program (pre- or runtime-compiled) this is. If it's a pre-compiled program, we start the process and, if we're supposed to redirect the file to the process' standard input, we open a handle to the file and read it into the standard input stream. If it's a runtime compiled program, we create a `FileSystemEventArgs` object and invoke the `OnFileSystemEvent()` method of the `IFileSystemEventHandler` interface. After we've finished executing the program, we release our resource back to the `CountingSemaphore`:

```
// Release the resource and exit the critical section
executionRegulator.V();
```

CountingSemaphore

The implementation of the `CountingSemaphore` class used for this service is basically a simple wrapper around the built-in `Monitor` class. It's all that was needed for this project, but you can check out a more full-featured implementation here.

Collapse

```
public class CountingSemaphore
{
  /// <summary>
  /// Resource limit for the semaphore
  /// </summary>
  private uint count;

  /// <summary>
  /// Default constructor; resource count is 1, meaning the class will act
  /// like a standard, binary semaphore.
  /// </summary>
  public CountingSemaphore() : this(1)
  {
  }

  /// <summary>
  /// Constructor that allows you to set the number of resources that should
  /// be available for consumption.
  /// </summary>
  /// <param name="count">Number of resources that should be available
  /// for consumption</param>
  public CountingSemaphore(uint count)
  {
    this.count = count;
  }

  /// <summary>
  /// Function that should be called when leaving the critical section; frees
  /// up one resource.
  /// </summary>
  public void AddOne()
  {
    V();
  }

  /// <summary>
  /// Function that should be called when entering a critical section; claims
  /// one resource or waits if no resources are available.
  /// </summary>
  public void WaitOne()
  {
    P();
  }

  /// <summary>
  /// Function that should be called when entering a critical section; claims
  /// one resource or waits if no resources are available.
  /// </summary>
  public void P()
  {
    lock(this)
    {
      while (count <= 0)
        Monitor.Wait(this, Timeout.Infinite);

      count--;
    }
  }

  /// <summary>
  /// Function that should be called when leaving the critical section;
  /// frees up one resource.
  /// </summary>
  public void V()
  {
    lock(this)
    {
      count++;
      Monitor.Pulse(this);
    }
  }
}
```

Errata

Be sure to alter the account that the service runs under according to your needs. When watching folders only on the local machine, you can typically leave it as the Local

System account, unless the ACLs on a directory explicitly exclude the SYSTEM user in which case you'll have to switch to a user with rights to that directory. When watching folders on a network share, however, you need to switch the account to one with local network access: on Windows XP and 2003 you can use the Network Service account (provided that the share is open to everyone), and on Windows 2000 you can use a domain account. However please note that, per the comments by Jeffrey Walton, this service (specifically the `FileSystemWatcher` class that it depends on) will not work when watching network shares on a non-Windows OS box, such as a share on an IBM eServer or a Samba share on a Linux box.

History

- 2007-01-01 - Initial publication.
- 2007-01-02 - Added the errata section.
- 2007-01-06 - Added support for runtime-compiled event handling code.
- 2007-02-08 - Fixed a bug where specifying `<eventToWatch type="All"/>` in the *config* file wasn't attaching event handlers properly.

About Luke Stratman

I'm a senior software engineer at a small financial services firm in Herndon, VA, just west of Washington, DC. I specialize in web-based application development and have extensive experience in .NET 1.1/2.0, ASP.NET, web service implementations, and SQL Server 2000/2005 database design and administration. I've also worked with wide variety of other languages, APIs, and technologies including C/C++ (in both Windows and Linux), DirectX, Java, Perl, and PHP.

Outside of work, I'm a rabid DC sports fan and I love the outdoors, especially when I get a chance to hike or kayak.

Click here to view Luke Stratman's online profile.

THE CODE PROJECT
The Visual Studio .NET Developer's Homepage

Discussions and Feedback

29 comments have been posted for this article. Visit **http://www.codeproject.com/cs/files/directorywatcherservice.asp** to post and view comments on this article.

Updated: 8 Feb 2007