CONTENTS

OVERVIEW ......................................................................................................................................................... 3

EXERCISE 1: INSTRUMENTATION USING PERFORMANCE COUNTERS .................................................. 5
    Task 1 – Adding a Class to Expose Performance Counters ........................................................................ 5
    Task 2 – Exposing Diagnostic Information .................................................................................................. 12

EXERCISE 2: PERFORMANCE ANALYSIS .................................................................................................. 18
    Task 1 – Instrumenting the Application ........................................................................................................ 18

SUMMARY ............................................................................................................................................................ 21
Overview

Windows 7 builds on top of the instrumentation platform provided in Windows Vista and previous versions of Windows to expose diagnostic information from applications. There is a variety of mechanisms for exposing instrumentation and control information, including:

- Event Tracing for Windows (ETW)
- Windows Performance Counters
- Windows Management Instrumentation (WMI)
- Event Log

Analyzing application performance and consuming instrumentation information are critical tasks for troubleshooting applications in real-world scenarios. The Windows instrumentation mechanisms and the Windows Performance Toolkit provide the necessary tools to streamline the diagnostics and analysis experience.

Objectives

In this Hands-On Lab, you will learn how to:

- Expose instrumentation information from an application using Windows performance counters
- Analyze application performance using the Windows Performance Toolkit

System Requirements

You must have the following items to complete this lab:

- Microsoft Visual Studio 2008
- Windows 7
- Windows 7 SDK
- Microsoft Windows Performance Toolkit

Note: The techniques used in this lab do not represent recommended development practices. In a practical scenario, you would have a separate elevated process (or installer) to register performance counters for your application. Additionally, you might consider using multi-instance performance counters to allow multiple instances of your applications to coexist on a single machine (i.e. terminal
services, multiple sessions). Breaking this recommendation hinders the user experience in a multi-user scenario such as Fast User Switching or Terminal Services.
Exercise 1: Instrumentation Using Performance Counters

In this exercise, you will add instrumentation to an existing application using Windows performance counters. The application is designed to copy files from a source directory to a destination directory, and it reports progress visually in its UI; however, to consume progress information programmatically or from a remote machine, instrumentation information must be exported to the operating system.

To complete this exercise, you must launch Visual Studio as an administrator or ensure that when launching the application it runs elevated (e.g. using right-click, Run as Administrator on the application’s main executable). This is required for performance counters registration.

Task 1 – Adding a Class to Expose Performance Counters

To expose instrumentation information from the application, you will create a class that exposes Windows performance counters indicating the progress of the file copy operation.

1. Open the starting solution Begin.sln located under the %TrainingKitInstallfolder%\InstrumentationAndPerformance\Ex1-PerfCounters\Begin folder, choosing the language of your preference (C# or VB).

2. Right-click the FileCopier project in Solution Explorer and choose Add New Item. Then choose the Class template (C#) or the Module template (Visual Basic) and name it FileCopyPerformanceCounters.

3. (For C# users only) Make the class static as there will be no need for more than one instance of it.

   C#
   
   public static class FileCopyPerformanceCounters
   {
   }

4. Add a namespace directive at the top of the file for the System.Diagnostics namespace.

   C#
   
   using System.Diagnostics;

   Visual Basic
   
   Imports System.Diagnostics
5. Add a new public method called **Initialize**. In this method, you will initialize the `PerformanceCounterCategory` and `PerformanceCounter` objects required to expose performance counters to the operating system. To do this, add the following code (shown in **bold**).

```csharp
public static void Initialize()
{
}
```

```visualbasic
Public Sub Initialize()
End Sub
```

6. Create a new instance of the `CounterCreationDataCollection` class.

```csharp
public static void Initialize()
{
    CounterCreationDataCollection counters = new CounterCreationDataCollection();
}
```

```visualbasic
Public Sub Initialize()
    Dim counters As New CounterCreationDataCollection()
End Sub
```

7. Create a new instance of the `CounterCreationData` class and initialize it with:

   a. "**Total Files Copied**" as the counter name.

   b. A description string of your choosing as the counter help string.

   c. A `PerformanceCounterType.NumberOfItems32` as the counter type.

To do this, add the following code (shown in **bold**).

```csharp
public static void Initialize()
{
    CounterCreationDataCollection counters = new CounterCreationDataCollection();
    CounterCreationData counter = new CounterCreationData("Total Files Copied",
        "Total number of files copied by the application.",
```
8. Use the `Add` method of the `CounterCreationDataCollection` object to add the counter data created in the previous step to the collection created previously.

   **C#**
   ```csharp
   public static void Initialize()
   {
       ...
       CounterCreationData counter = new CounterCreationData(
           "Total Files Copied",
           "Total number of files copied by the application.",
           PerformanceCounterType.NumberOfItems32);
       counters.Add(counter);
   }
   ```

   **Visual Basic**
   ```vbnet
   Public Sub Initialize()
   ...
   Dim counter As New CounterCreationData("Total Files Copied", "Total number of files copied by the application.", PerformanceCounterType.NumberOfItems32)
   counters.Add(counter)
   End Sub
   ```

9. Create another instance of the `CounterCreationData` class and initialize it with:
   a. "Files Copied" as the counter name.
   b. The rest of the parameters should be identical to the parameters used to initialize the previous `CounterCreationData` object.
Additionally, add it to the `CounterCreationDataCollection` object.

### C#

```csharp
public static void Initialize()
{
    ...
    counters.Add(counter);
    counter = new CounterCreationData("% Files Copied",
                                      "Percent of files copied in the current operation.",
                                      PerformanceCounterType.NumberOfItems32);
    counters.Add(counter);
}
```

### Visual Basic

```vbnet
Public Sub Initialize()
    ...
    counters.Add(counter)
    counter = New CounterCreationData("% Files Copied", _
                                       "Percent of files copied in the current operation.", _
                                       PerformanceCounterType.NumberOfItems32)
    counters.Add(counter)
End Sub
```

10. Use the `Exists` and `Delete` static methods of the `PerformanceCounterCategory` class to check whether the `FileCopier` performance counter category exists. If it does, delete it so that in the next step you can recreate it.

### C#

```csharp
public static void Initialize()
{
    ...
    counter = new CounterCreationData("% Files Copied",
                                       "Percent of files copied in the current operation.",
                                       PerformanceCounterType.NumberOfItems32);
    counters.Add(counter);

    if (PerformanceCounterCategory.Exists("FileCopier"))
        PerformanceCounterCategory.Delete("FileCopier");
}
```

### Visual Basic

```vbnet
Public Sub Initialize()
    ....
```
counter = New CounterCreationData("% Files Copied", _
"Percent of files copied in the current operation.", _
PerformanceCounterType.NumberOfItems32)
counters.Add(counter)

If PerformanceCounterCategory.Exists("FileCopier") Then
PerformanceCounterCategory.Delete("FileCopier")
End Sub

11. Call the Create static method of the PerformanceCounterCategory class with:

   a. "FileCopier" as the category name.

   b. A description string of your choosing as the category help string,
      PerformanceCounterCategoryType.SingleInstance as the category type.

   c. The CounterCreationDataCollection instance as the final parameter.

   **C#**

```
public static void Initialize()
{
    ...
    if (PerformanceCounterCategory.Exists("FileCopier"))
        PerformanceCounterCategory.Delete("FileCopier");

    PerformanceCounterCategory.Create(
        "FileCopier",
        "Instrumentation of the FileCopier application.",
        PerformanceCounterCategoryType.SingleInstance,
        counters);
}
```

**Visual Basic**

```
Public Sub Initialize()
    ...

    If PerformanceCounterCategory.Exists("FileCopier") Then
        PerformanceCounterCategory.Delete("FileCopier")

    PerformanceCounterCategory.Create("FileCopier", _
        "Instrumentation of the FileCopier application.", _
        PerformanceCounterCategoryType.SingleInstance,
        counters)

End Sub
```
12. Add two private static data members of the **PerformanceCounter** type to the class (C#) or module (Visual Basic), called **totalFilesCounter** and **percentDoneCounter**.

**C#**

```csharp
public static class FileCopyPerformanceCounters
{
    private static PerformanceCounter totalFilesCounter;

    private static PerformanceCounter percentDoneCounter;

    ...
}
```

**Visual Basic**

```vbnet
Module FileCopyPerformanceCounters
    Private totalFilesCounter As PerformanceCounter

    Private percentDoneCounter As PerformanceCounter

    ...
End Module
```

13. In the **Initialize** method, initialize the counters with a new instance of the **PerformanceCounter** class, passing "**FileCopier**" as the category name, the relevant counter name ("Total Files Copied" or "% Files Copied") as the counter name and **false** as the last parameter.

**C#**

```csharp
public static void Initialize()
{
    ...
    PerformanceCounterCategory.Create(
        "FileCopier",
        "Instrumentation of the FileCopier application.",
        PerformanceCounterCategoryType.SingleInstance, counters);

    totalFilesCounter = new PerformanceCounter(
        "FileCopier", "Total Files Copied", false);

    percentDoneCounter = new PerformanceCounter(
        "FileCopier", "% Files Copied", false);
}
```

**Visual Basic**

```vbnet
Public Sub Initialize()
```
... PerformanceCounterCategory.Create("FileCopier", _
"Instrumentation of the FileCopier application.", _
PerformanceCounterCategoryType.SingleInstance, _
counters)

totalFilesCounter = New PerformanceCounter( _
"FileCopier", "Total Files Copied", False)
percentDoneCounter = New PerformanceCounter( _
"FileCopier", "% Files Copied", False)
End Sub

14. In MainForm.cs (C#) or MainForm.vb (Visual Basic), locate the constructor and add a call to FileCopyPerformanceCounters.Initialize:

C#
public MainForm()
{
    InitializeComponent();

    FileCopyPerformanceCounters.Initialize();
}

Visual Basic
Public Sub New()
    Me.InitializeComponent()

    FileCopyPerformanceCounters.Initialize()
End Sub

15. Press F5 to launch the application. Ensure that no exceptions are thrown. This likely means that the counters were successfully registered. If there is an exception, prior to debugging make sure that your Visual Studio instance is launched with administrator permissions (or that you are launching the application itself with administrator permissions).
Task 2 – Exposing Diagnostic Information

In this task, you will use the framework created in the previous step to expose diagnostic information about the application’s progress as it copies files from the source to the destination.

1. In the `FileCopyPerformanceCounters` class (C#) or module (Visual Basic), add two methods called `UpdateTotalFiles` and `UpdatePercentDone` which receive a single integer parameter.

   In the body of each method, set the `RawValue` property of the relevant `PerformanceCounter` instance (`totalFilesCounter` or `percentDoneCounter`) to the value specified by the method’s parameter. To do this, add the following code (shown in bold):

   **C#**
   ```
   public static class FileCopyPerformanceCounters
   {
   ...
   
   public static void UpdateTotalFiles(int totalFiles)
   {
   totalFilesCounter.RawValue = totalFiles;
   }
   
   public static void UpdatePercentDone(int percentDone)
   {
   percentDoneCounter.RawValue = percentDone;
   }
   }
   ```

   **Visual Basic**
   ```
   Module FileCopyPerformanceCounters
   ```
2. Update the number of files copied. To do this, open the `MainForm.cs` file (C#) or `MainForm.vb` (Visual Basic). Then follow these steps:

   a. (For C# users) Locate the `BtnCopy_Click` method and navigate to the code that registers for the `DoWork` method of the `worker` instance.

   b. (For Visual Basic users) Locate the `worker_DoWork` Sub.

   c. Inside the loop that copies files, insert a call to the `FileCopyPerformanceCounters.UpdateTotalFiles` method with the current number of files that have been copied.

**C#**

```csharp
private void BtnCopy_Click(object sender, EventArgs args)
{
    ...
    this.worker.DoWork += (o, e) =>
    {
        string[] files = Directory.GetFiles(source);
        for (int i = 0; i < files.Length; ++i)
        {
            Thread.Sleep(1000);
            File.Copy(files[i], Path.Combine(dest, Path.GetFileName(files[i])));
            this.worker.ReportProgress((int)((100.0f * i) / files.Length));
            FileCopyPerformanceCounters.UpdateTotalFiles(i);
        }
    };
    ...
}
```

**Visual Basic**

```vbnet
Private Sub worker_DoWork() Handles worker.DoWork
    Dim files = Directory.GetFiles(Me.txtSourceDirectory.Text)
    For i = 0 To files.Length - 1
```
### Instrumentation and Performance Hands-On Lab

```csharp
Thread.Sleep(1000)
File.Copy(files(i), Path.Combine(Me.txtDestinationDirectory.Text, Path.GetFileName(files(i))))
Me.worker.ReportProgress(CInt(Fix((100.0F * i) / files.Length)))

FileCopyPerformanceCounters.UpdateTotalFiles(i)
Next i
End Sub
```

3. Update the percentage of files copied. To do this follow these steps:

   a. (For **C# users**) In the **BtnCopy_Click** method navigate to the code that registers for the **ProgressChanged** method of the **worker** instance.

   b. (For **Visual Basic users**) Locate the **worker_ProgressChanged** Sub.

   c. Insert a call to the **FileCopyPerformanceCounters.UpdatePercentDone** method and pass to it the **ProgressPercentage** property of the **ProgressChangedEventArgs** parameter.

**C#**

```csharp
private void BtnCopy_Click(object sender, EventArgs args)
{
    ...
    this.worker.WorkerReportsProgress = true;
    this.worker.ProgressChanged += (o, e) =>
    {
        this.BeginInvoke((MethodInvoker)delegate
        {
            progressBar.Value = e.ProgressPercentage;
            FileCopyPerformanceCounters.UpdatePercentDone(e.ProgressPercentage);
        });
    };
    ...
}
```

**Visual Basic**

```vbnet
Private Sub worker_ProgressChanged(ByVal sender As Object, ByVal e As ProgressChangedEventArgs) Handles worker.ProgressChanged
    progressBar.Value = CInt(e.ProgressPercentage)
    FileCopyPerformanceCounters.UpdatePercentDone(e.ProgressPercentage)
End Sub
```
4. Press **F5** to launch the application and choose a source and destination directory by using the **Browse** buttons next to the source and destination text boxes.

   **Note:** Make sure to have some files at the source directory, and to use an empty destination directory.

5. Launch Performance Monitor (). You can do this by typing `perfmon` into the Start Menu search box.

6. In the left navigation panel, select **Performance Monitor** under the **Monitoring Tools** node.

   ![Performance Monitor](image)

   **Figure 2**
   
   *Opening the Performance Monitor*

7. Click the green + button to open the **Add Counters** dialog box.

8. Find the **% FilesCopied** and **Total Files Copied** counters under the **FileCopier** category in the **Available counters** section. Select them and click **Add**.
9. Right-click the % Processor Time counter and select Hide Selected Counters.

10. Back in the File Copier application, click the Copy button and inspect the performance counters values going up as the copy operation is in progress.
In this exercise, you have added instrumentation capabilities to an existing application by exposing Windows performance counters and consuming them from the Windows Performance Monitor. The full exercise solution can be found in the %TrainingKitInstallfolder%\InstrumentationAndPerformance\Ex1-PerfCounters\End folder, depending on the language of your preference.
Exercise 2: Performance Analysis

In this exercise, you will analyze the performance of the application developed in the previous exercise to discover its CPU and I/O utilization. You will use the Windows Performance Toolkit to perform the analysis without modifying the application code or using any third-party profilers.

To complete this exercise, you must have the Microsoft Windows Performance Toolkit installed on your system. Note that on a 32-bit system, the 32-bit version of Windows Performance Toolkit should be installed; on a 64-bit system, the 64-bit version of the Windows Performance Toolkit should be installed.

Task 1 – Instrumenting the Application

To instrument the application, you will use the Windows Performance Toolkit command-line tool (xperf.exe) with the “BASE” collection profile.

1. Launch the complete application from Exercise 1.

   **Note:** If you did not complete the exercise, you can use the application from the complete solution provided in the %TrainingKitInstallfolder%\InstrumentationAndPerformance\Ex1-PerfCounters\End folder, choosing the language of your preference (C# or VB).

2. Select a source and destination directory for the file copy operation.

3. Open a command line with administrator privileges (type “Command Prompt” into the Start Menu search box and then right-click the first result and select “Run as administrator”).

4. Navigate to the installation directory of the Windows Performance Toolkit (if you installed the toolkit to the default installation directory, you can use the command cd C:\Program Files\Microsoft Windows Performance Toolkit).

5. Run the following command to turn on the base instrumentation profile.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>xperf -on BASE</td>
</tr>
</tbody>
</table>

6. Click the Copy button in the application to start the copy operation.

7. When the copy operation completes, go back to the command prompt and run the following command to generate the collected instrumentation data into the result.etl file.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>xperf -d result.etl</td>
</tr>
</tbody>
</table>
8. Run the following command to open the Windows Performance Toolkit UI.

**Command**

xperf result.etl

9. From the list on the left, choose “Disk Utilization by Process” and “CPU Sampling by Process”.

![Figure 6](image)

*Analyzing the instrumentation data*

10. From the combo boxes that appear in the top right corner of each of the two graphs, select the **FileCopier.exe** process only.

**Note:** You will need to get the FileCopier process id from the Task Manager’s Processes section. To open it, right-click the taskbar, and select Start Task Manager.
11. Inspect the CPU utilization in correlation to the disk utilization, and determine whether the disk or CPU is a bottleneck in the application.

12. Select a region of the disk utilization graph and right-click it, then select **Summary Table** and inspect the various disk accesses performed by the process.

13. Select a region of the disk utilization graph and right-click it, then select **Detail Graph** and inspect the disk head activity when the files were being copied by the application.

14. When done, go back to the command prompt and run the following command to delete the instrumentation file.

```
Command

del result.etl
```

In this exercise, you have analyzed the performance characteristics of an application without modifying its source code by using the Windows Performance Toolkit's command line and graphical user interface.
Summary

In this lab you have exposed instrumentation information from an application using Windows performance counters, so that it can be inspected programatically and remotely. Additionally, you have analyzed the performance of an application externally, without modifying its source code, by using the Windows Performance Toolkit.