13

Managing Performance and Usage

In this chapter, we cover the following recipes:

* Retrieving performance counters using Get-Counter
* Using WMI to retrieve performance counters
* Creating and using PLA data collector sets
* Reporting on performance data
* Generating performance-monitoring graphs
* Creating a system diagnostic report
* Reporting on printer usage
* Monitoring Hyper-V utilization and performance

# Introduction

Understanding how your infrastructure is being utilized is a key task for many IT pros today, and has been a challenge since the earliest days of computing. Windows NT 3.1 came with a marvelous tool, Performance Monitor (perfmon), that allowed you to see what the OS and applications are doing and what resources they are consuming. The Windows NT 3.1 Resource Kit even included an entire book on how to use perfmon and the various performance counters.

In addition to understanding how your individual servers are performing, you need to know what they are actually doing in terms of user work. Knowing, for example, the usage of printers is useful for capacity-planning purposes.

This chapter shows you how you can use PowerShell to obtain and display performance information and usage information using a variety of techniques, including cmdlets, WMI, and more.

Windows Server 2019 contains a subsystem known as Performance Logging and Alerting (PLA). PLA enables you to obtain performance details, logging information, and diagnostic data from local and remote computers (and more!).

PLA provides the underpinnings to familiar GUI tools such as Performance Monitor and Task Manager. These tools utilize the performance-monitoring framework built into Windows and help to visualize the outputs from PLA. You can also use PLA to generate data from which you can generate performance graphs.

In PLA, a counter set contains information about the performance of some aspect of your systems such as memory, disk devices, or network interfaces. A PLA counter set contains one or more individual counters. A counter provides a measurement of some aspect of the counter set. For example, the Memory counter set on Windows Server 2019 has 36 counters. Counters include Pages/Sec and Available Bytes.

A counter can be either single-instance or multi-instance. Counters such as \Memory\Page/Sec are single-instance counters, where each counter sample contains just one measurement (the current number of pages/second is being requested). Other counters, such as \Processor(\*)\% Processor Time, are multi-instance counters, returning counter samples for each processor in the server, plus one for the total CPU being consumed (helpfully named \_total). If your physical processor has hyper-threading enabled, you get two measurements for each physical core; thus, on a dual-processor hex-core system with hyper-threading, this counter would return 25 measurements (two measurements for each of the 12 cores plus one for total CPU utilization). A Hyper-V virtual machine with eight cores assigned would return nine counter samples (one for each virtual processor plus the total).

To get counter values from within a counter set, you use the Get-Counter cmdlet and specify the counter name (that is, the path) to the counter. The path is formatted as \\<ServerName\<CounterSetName>\<CounterName>. For example, the path to a counter on a remote server could look like this: \\DC1\Memory\Page Faults/sec. If you are getting counters on the local machine, you can omit the computer name prefix and just specify \Memory\Page Faults/sec. Note that the counter set and counter names can be long and can have spaces in the names. You often need to specify counter paths using PowerShell string quoting.

The Get-Counter cmdlet returns counter measurements as well as the available counter sets on a system. The PerformanceCounterSampleSet object, returned by Get-Counter, contains a CounterSamples property. This property contains one measurement of the counter (for single-instance counters) or an array of samples (for multi-instance counters).

In the Retrieving performance counters using Get-Counter recipe, you see how to use Get-Counter to return counter set details and counter measurements.

You can also use WMI to return performance counter details, as you can see in the Using WMI to retrieve performance counters recipe.

If you are using multiple counters and counter sets and plan to analyze performance data, using PLA data collector sets is the preferred approach. With the Creating and using PLA data collector sets recipe, you create and start a data collector set. The Reporting on performance data and Generating performance monitoring graphs recipes show how you can leverage the results from a PLA data collector.

PLA also has some built-in data collectors and predefined reports based on those collectors. You can easily get PLA to produce a Systems Diagnostic Report, as shown in the Creating a systems diagnostic report recipe. This report can provide insights into possible performance or other issues on your server.

When looking at the performance of a print server, it's useful to know how much usage a given printer is experiencing. The Reporting on printer usage recipe shows you how to retrieve printer usage information from your print server's event log and generate a report.

The final recipe in this chapter, Monitoring Hyper-V utilization and performance, demonstrates generating reports on your virtual machines (VMs) and VM servers. This recipe shows how to obtain basic performance information about a Hyper-V host and the VMs defined on the host.

# Retrieving performance counters using Get-Counter

Within Windows, a performance counter set is a set of performance counters. Each counter in a counter set measures an aspect of your system related to that counter set. In this recipe, we explore counter sets and counters on several servers.

## Getting ready

This recipe uses DC1, DC2, SRV1, SRV2, HV1, and HV2. These are servers that you have worked with in various recipes in this book. DC1 and DC2 are domain controllers, HV1 and HV2 host Hyper-V, and SRV1 and SRV2 are general-purpose Windows servers. All of these systems are running Windows 2019 Datacenter edition. Run this recipe from SRV1.

## How to do it...

1. Discover performance counter sets on SRV1:

$CounterSets = Get-Counter -ListSet \*

$CS1 = 'There are {0} counter sets on [{1}]'

$CS1 -f $CounterSets.Count,(hostname)

1. Discover performance counter sets on remote systems:

$Machines = 'DC1','DC2','HV1','HV2','SRV1','SRV2'

Foreach ($Machine in $Machines)

{

$RCounters = Get-Counter -ListSet \* -ComputerName $Machine

$CS2 = "There are {0} counters on [{1}]"

$CS2 -f $RCounters.Count, $Machine

}

1. List key performance counter sets:

Get-Counter -ListSet Processor, Memory, Network\*,\*Disk |

Sort-Object -Property CounterSetName |

Format-Table -Property CounterSetName

1. Get a description of the memory counter set:

Get-Counter -ListSet Memory |

Format-Table -Property Name, Description -Wrap

1. Get and display counters in the memory counter set on the localhost:

$CountersMem = (Get-Counter -ListSet Memory).Counter

'Memory counter set has [{0}] counters:' -f $CountersMem.Count

$CountersMem

1. Get and display a sample from each counter in the memory counter set:

$Counters = (Get-Counter -ListSet Memory).Counter

$FS = '{0,-19} {1,-60} {2,-10}'

$FS -f 'At', 'Counter', 'Value' # Display header row

foreach ($Counter in $Counters){

$C = Get-Counter -Counter $Counter

$T = $C.Timestamp # Time

$N = $C.CounterSamples.Path.Trim() # Counter Name

$V = $C.CounterSamples.CookedValue # Value

'{0,-15} {1,-59} {2,-14}' -f $T, $N, $V

}

1. Explore counter set types for key performance counters:

Get-Counter -ListSet Processor, Memory, Network\*, \*Disk\* |

Select-Object -Property CounterSetName, CounterSetType

1. Explore a local performance counter sample set:

$Counter1 = '\Memory\Page Faults/sec'

$PFS = Get-Counter -Counter $Counter1

$PFS

1. Look at a remote performance counter sample set on HV1:

$Counter2 = '\\HV1\Memory\Page Faults/sec'

$RPFS = Get-Counter -Counter $Counter1

$RPFS

1. Look inside a counter sample set:

$PFS | Get-Member -MemberType \*Property |

Format-Table -Wrap

1. What is inside a local multi-value counter sample:

$Counter3 = '\Processor(\*)\% Processor Time'

$CPU = Get-Counter -Counter $Counter3

$CPU

1. View a multi-value counter sample on HV2:

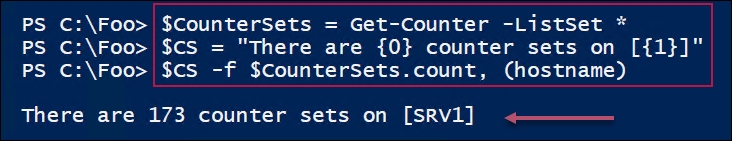
$Counter4 = '\\HV2\Processor(\*)\% Processor Time'

$CPU = Get-Counter -Counter $Counter4

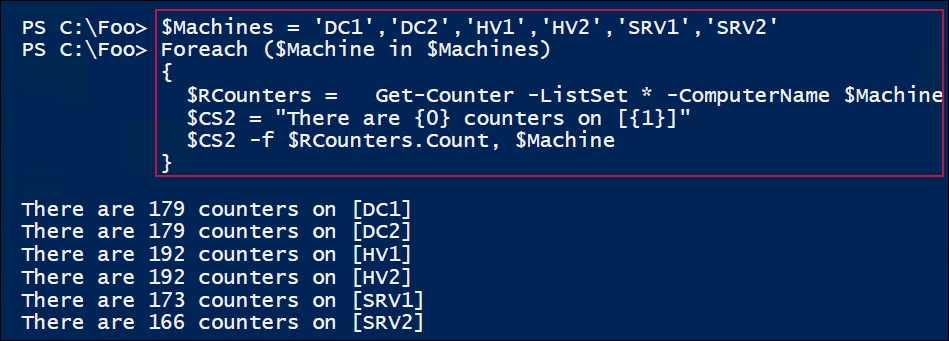
$CPU

## How it works...

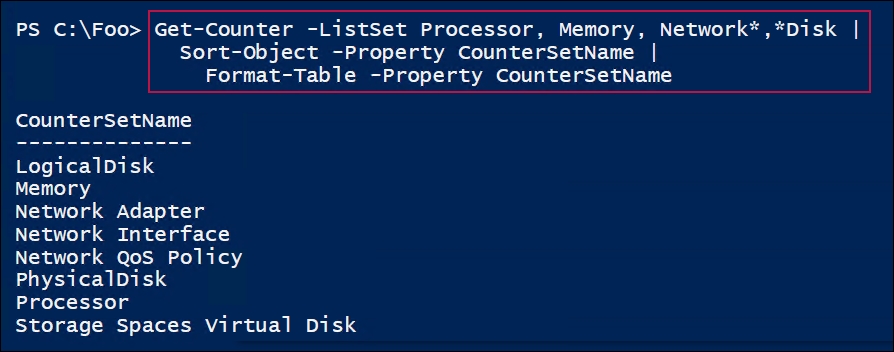
In step 1, you use the Get-Counter cmdlet to determine how many counter sets exist on SRV1, which looks like this:



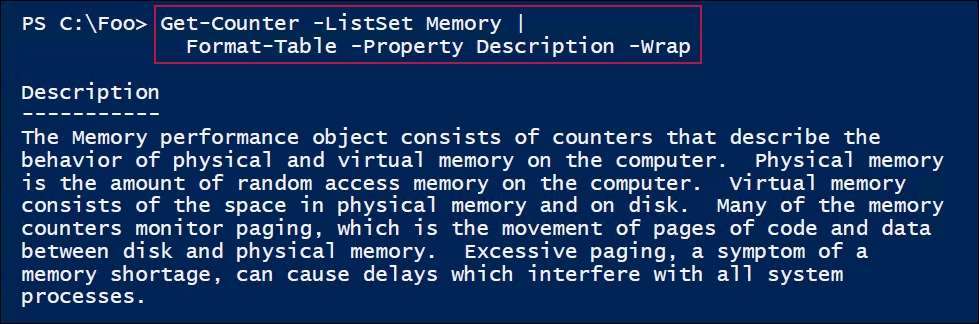
With step 2, you determine how many performance counter sets exist on remote servers, which looks like this:



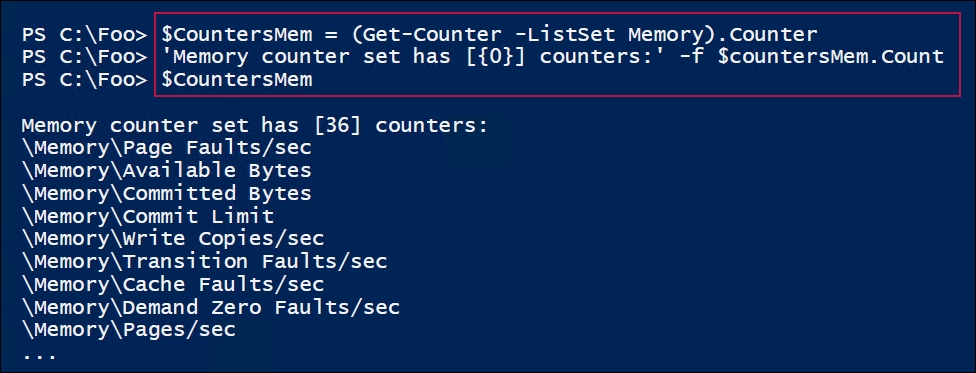
In step 3, you retrieve and view key counter sets, which looks like this:



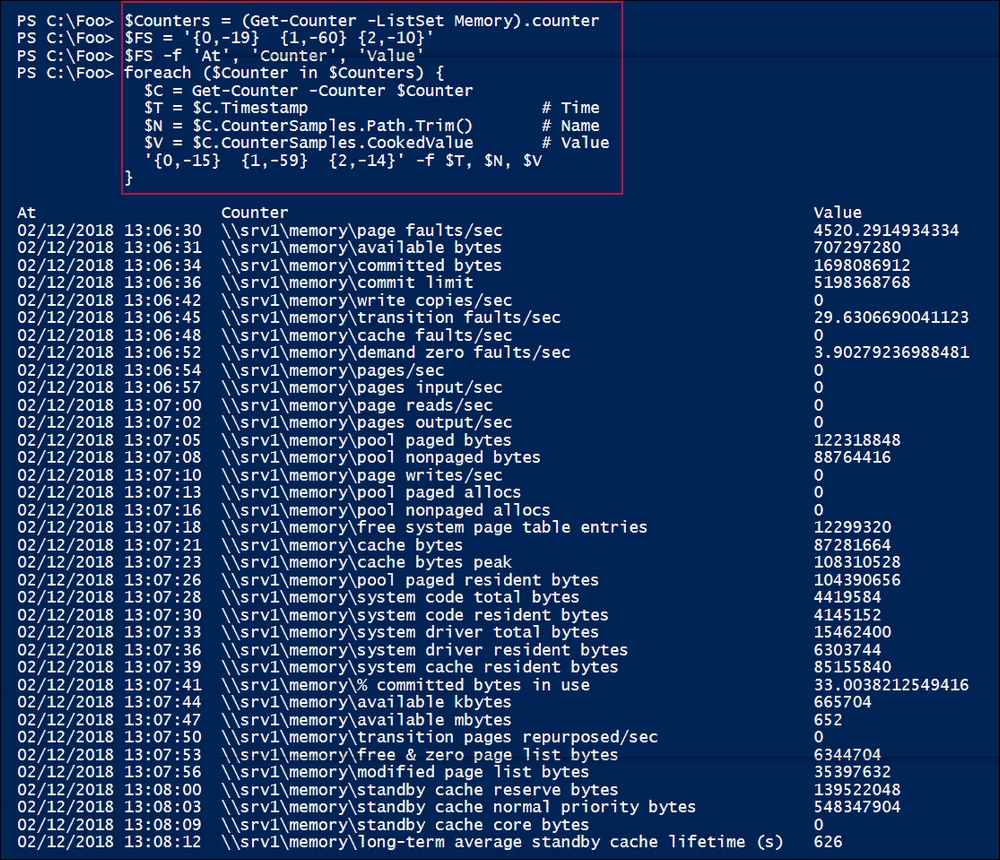
With step 4, you get a description of the Memory counter set:



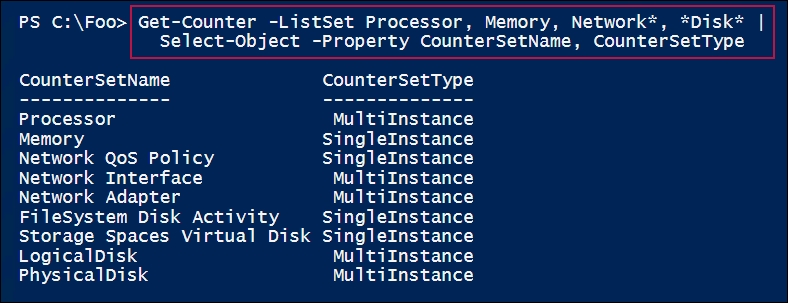
In step 5, you retrieve the counters in the Memory counter set (showing just the first few counters), which looks like this:



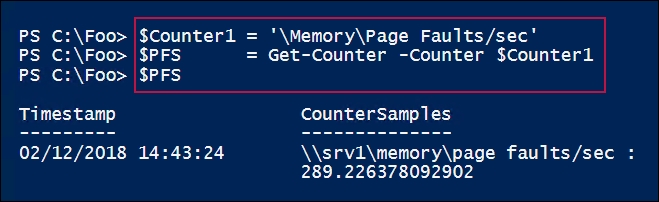
In step 6, you use Get-Counter to retrieve counter data from the Memory counter set, which looks like this:



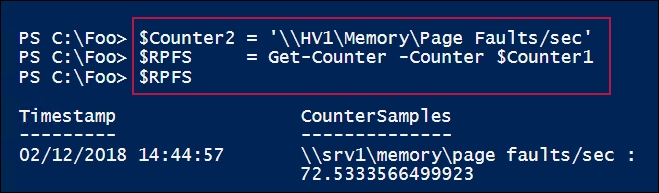
In step 7, you determine which counter set names produce single- or multi-instance counters, which looks like this:



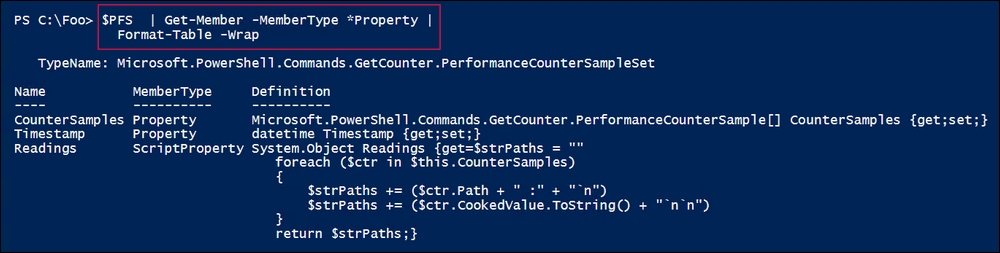
In step 8, you retrieve a specific memory counter (page faults per second) from SRV1, which looks like this:



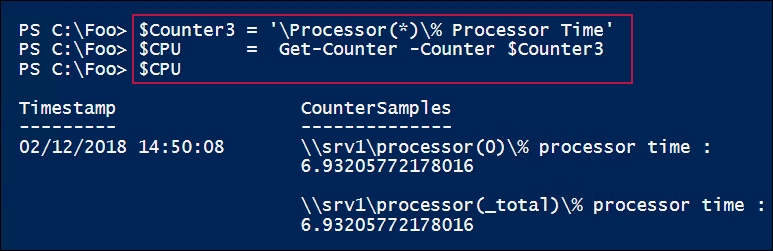
In step 9, you retrieve a counter value from a remote system, HV1, which looks like this:



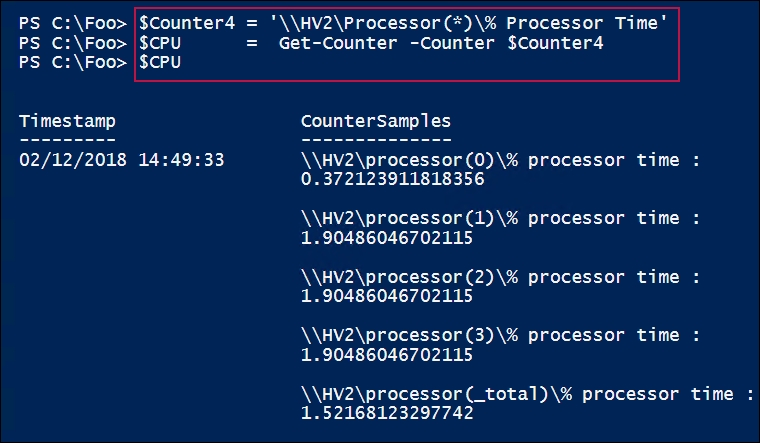
In step 10, you view the properties of one counter, which looks like this:



In step 11, you view a multi-value counter sample on the local machine, which looks like this:



In step 12, you retrieve a multi-value counter from HV2, contains counter values for each processor in the HV1 computer, which looks like this:



## There's more...

In step 5, you use Get-Counter to get the values of the counter on the Memory counter set (on SRV1). As you can see, this is not particularly fast. Using Get-Counter to retrieve larger numbers of counter values is not a great approach—using PLA data collector sets (as discussed in the Creating and using PLA data collector sets recipe) is much more efficient.

In step 8, you can see that the PerformanceCounterSampleSet object has both a CounterSamples property and a Readings script property, which returns the same basic information. The PerformanceCounterSampleSet object is a structured object with properties for counter path and counter values. The Readings script property is a small PowerShell script that returns a string made up of the path and the cooked data value. For viewing from the PowerShell console, use the Readings property, whereas for analysis, having a structured object can be useful.

In step 11 and step 12, you look at two multi-valued processor counters. Each step returns one counter measurement for each CPU, plus a total for all CPUs in the system. In step 11, you view the processor usage on SRV1, which has just one CPU (and produces two counter values), while in step 12, HV2 has four processors, so you see five counter samples (one for each processor plus a total).

# Using WMI to retrieve performance counters

Another way to access performance information is via WMI. You can use either the WMI or the CIM cmdlets to access a large number of performance counters, as an alternative to using Get-Counter.

When using WMI, the naming structure for counter information is different from using Get-Counter. With WMI, counters are exposed via separate WMI classes whose names are slightly different from those you use with Get-Counter. Effectively, with WMI, each performance counter set is a WMI class.

You find the WMI performance counters in the ROOT\CimV2 namespace; they have names that begin with Win32\_Perf. For example, the Memory performance counter set contains 36 separate counters. The Win32\_PerfFormattedData\_PerfOS\_Memory WMI class contains 46 properties, including the numerous individual performance counters.

With WMI, you get all the measurements back in one call to Get-CimInstance, whereas you would need to call Get-Counter for each counter sample. This provides better performance than you see with Get-Counter, but WMI is still fairly slow when compared to the PLA data collector sets you use in later recipes in this chapter.

This recipe gets performance counters from local and remote machines using the CIM cmdlet set. The CIM cmdlet set is preferable to the older WMI commands as it is a little faster, and it can make use of WinRM for remote sessions.

## Getting ready

You run this recipe on SRV1. This recipe uses the CIM cmdlets. You could revise this recipe to make use of the WMI cmdlets, which might be useful in cases where you are communicating with an older system that does not have PowerShell remoting up and running.

## How to do it...

1. Find performance-related counters in the Root\CimV2 namespace:

Get-CimClass -ClassName Win32\*perf\* | Measure-Object |

Select-Object -Property Count

Get-CimClass -ClassName Win32\*perfFormatted\* | Measure-Object |

Select-Object -Property Count

Get-CimClass -ClassName Win32\*perfraw\* | Measure-Object |

Select-Object -Property Count

1. Find key performance classes for the OS:

Get-CimClass "win32\_PerfFormatted\*PerfOS\*" |

Select-Object -Property CimClassName

1. Find key performance classes for the disk:

Get-CimClass "Win32\_PerfFormatted\*Disk\*" |

Select-Object -Property CimClassName

1. Find key performance classes for the disk:

Get-CimInstance -ClassName Win32\_PerfFormattedData\_PerfOS\_Memory |

Select-Object -Property PagesPerSec, AvailableMBytes

1. Get CPU counter samples:

Get-CimInstance -ClassName Win32\_PerfFormattedData\_PerfOS\_Processor |

Where-Object Name -eq '\_Total' |

Select-Object -Property Name, PercentProcessortime

1. Get CPU counter samples from a remote system:

$CHT = @{

ClassName = 'Win32\_PerfFormattedData\_PerfOS\_Memory'

ComputerName = 'DC1'

}

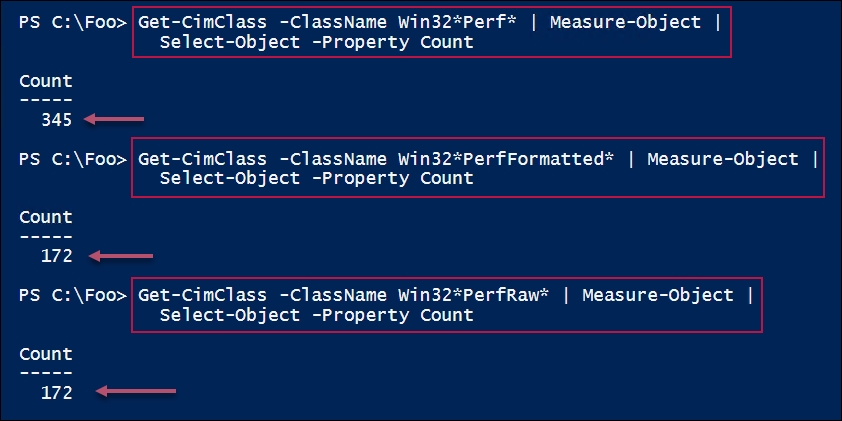
Get-CimInstance @CHT |

Select-Object -Property PSComputerName, PagesPerSec,

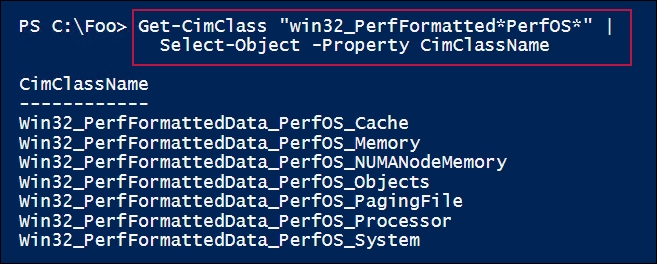
AvailableMBytes

## How it works...

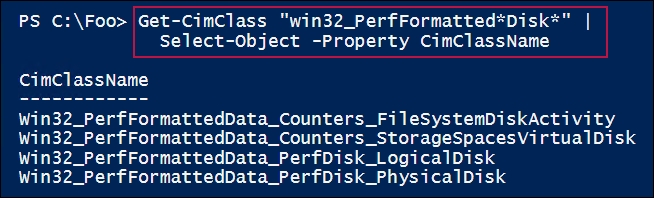
In step 1, you search for the performance-related counters in the Root\CIMV2 namespace, which looks like this:



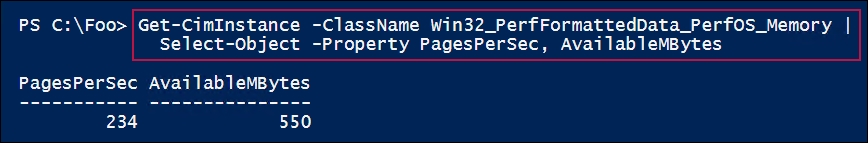
In step 2, you search for WMI classes relating to the OS performance, which looks like this:



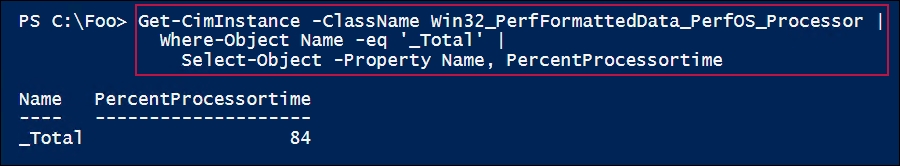
In step 3, you search for WMI classes relating to disk performance, which looks like this:



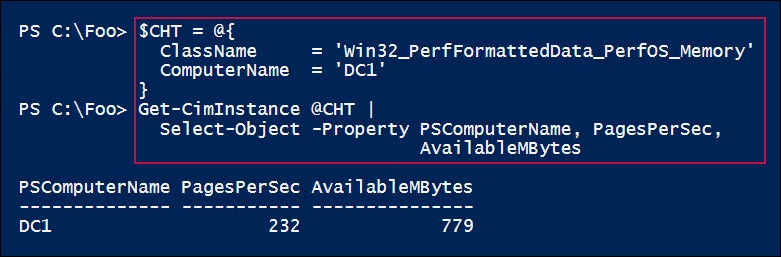
In step 4, you get two memory counters from SRV1, which looks like this:



In step 5, you retrieve the PerfOS\_Processor class and display just the name and the percentage of CPU time being used on SRV1, which looks like this:



With step 6, you retrieve the memory performance information, but remotely from DC1, which looks like this:



## There's more...

In step 1, you saw there were 172 WMI classes containing formatted data and the same number containing uncooked values, out of a total of 345 performance classes in the Root\CIMV2 namespace on SRV1. The one additional class is Win32\_Perf, which returns all the other performance class instances (on SRV1, using Get-CimInstance to return the class results in 2,891 entries).

# Creating and using PLA data collector sets

In the previous two recipes, you retrieved individual counter objects either by using Get-Counter or via WMI. That works, but retrieving performance data is slow. It took over a minute and 40 seconds to retrieve the performance counters in a local machine's Memory counter set. Using these methods for large-scale performance data collection does not scale well.

The PLA subsystem provides an efficient mechanism to perform the data collection. PLA allows you to create a data collector set. This is an object representing the counters whose values you wish to collect. Once you create the data collector set, you can direct Windows to start collecting the data and to output it for later analysis. You have options as to how to output the data—you can use a binary log file, a comma-delimited file, and more. Once you have the data collected and output, you can analyze it, as you can see in the Reporting on performance data recipe.

There is no direct cmdlet support for setting up and using performance data collection. Instead, you use the PLA COM objects that are built into Windows.

## Getting ready

You run this on SRV1, a domain-joined server.

## How to do it...

1. Create and populate a new performance data collector set:

$Name = 'SRV1 Collector Set'

$SRV1CS = New-Object -COM Pla.DataCollectorSet

$SRV1CS.DisplayName = $Name

$SRV1CS.Duration = 12\*3600 # 12 hours - 19:00

$SRV1CS.SubdirectoryFormat = 1

$SRV1CS.SubdirectoryFormatPattern = 'yyyy\-MM'

$JPHT = @{

Path = "$Env:SystemDrive"

ChildPath = "\PerfLogs\Admin\$Name"

}

$SRV1CS.RootPath = Join-Path @JPHT

$SRV1Collector = $SRV1CS.DataCollectors.CreateDataCollector(0)

$SRV1Collector.FileName = "$Name\_"

$SRV1Collector.FileNameFormat = 1

$SRV1Collector.FileNameFormatPattern = "\-MM\-dd"

$SRV1Collector.SampleInterval = 15

$SRV1Collector.LogFileFormat = 3 # BLG format

$SRV1Collector.LogAppend = $True

1. Define counters of interest:

$Counters = @(

'\Memory\Pages/sec',

'\Memory\Available MBytes',

'\Processor(\_Total)\% Processor Time',

'\PhysicalDisk(\_Total)\% Disk Time',

'\PhysicalDisk(\_Total)\Disk Transfers/sec' ,

'\PhysicalDisk(\_Total)\Avg. Disk Sec/Read',

'\PhysicalDisk(\_Total)\Avg. Disk Sec/Write',

'\PhysicalDisk(\_Total)\Avg. Disk Queue Length'

)

1. Add the counters to the collector:

$SRV1Collector.PerformanceCounters = $Counters

1. Create a schedule—start tomorrow morning at 07:00:

$StartDate =

Get-Date -Day $((Get-Date).Day+1) -Hour 7 -Minute 0 -Second 0

$Schedule = $SRV1CS.Schedules.CreateSchedule()

$Schedule.Days = 127

$Schedule.StartDate = $StartDate

$Schedule.StartTime = $StartDate

1. Create, add, and start the collector set:

try

{

$SRV1CS.Schedules.Add($Schedule)

$SRV1CS.DataCollectors.Add($SRV1Collector)

$SRV1CS.Commit("$Name" , $null , 0x0003) | Out-Null

$SRV1CS.Start($false);

}

catch

{

Write-Host "Exception Caught starting PLA DC: " $\_.Exception

Return

}

## How it works...

In step 1, you create and configure a new data collector set object (on SRV1). In step 2, you specify the performance counters you wish the data collector to collect. With step 3, you add those counters to the data collector. In step 4, you create a schedule telling the data collector when it should collect data. Finally, in step 5, you save the data collection details and start the data collection.

None of these steps produce output.

## There's more...

In step 1, you assign the data collector's LogFileFormat to be 1. This tells PLA to create a log in binary format. When you create a data collector set, you could alternatively output using comma-separated values or tab-separated values, or as SQL records.

Setting the data collector's LogFileFormat property to 1 creates data in a comma-separated value format, while setting the property to 2 results in a tab-separated value format. Depending on the tools you use to analyze the output, different formats may be more appropriate.

In the GitHub repository supporting this book, you can find alternatives to this recipe that store the logging data in comma-separated and tab-separated formats. Note that other recipes in this chapter use different log file formats.

Adding a counter set and activating it generates data, and that consumes disk space. After using a counter set to analyze an issue, you might wish to stop data collection and possibly remove the counter set; you could do that as follows:

# Stop data collection

$DCStRemote = New-Object -COM Pla.DataCollectorSet

$Name = 'SRV1 Collector Set'

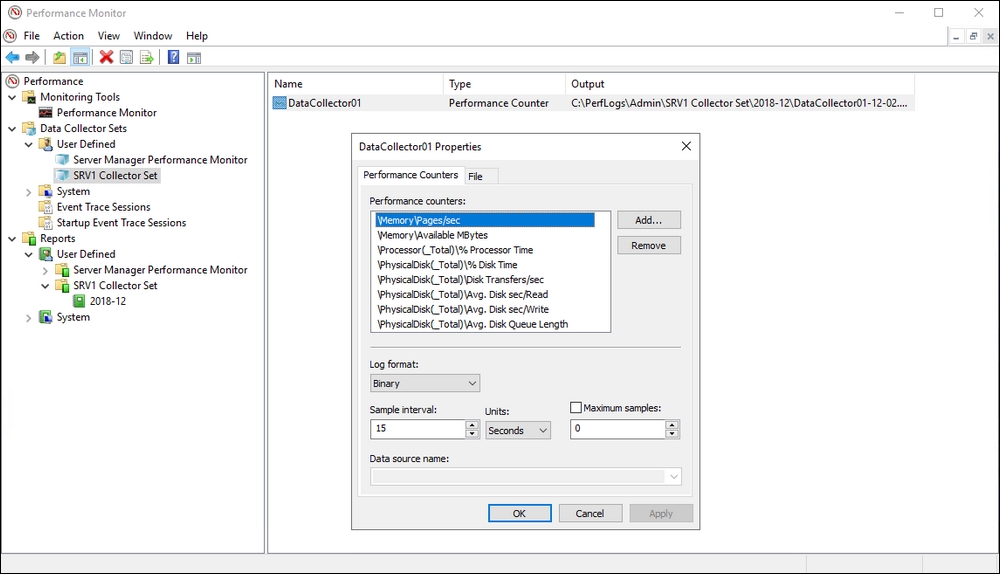
$DCstRemote.Query($Name,'LocalHost')

$DCstRemote.Stop($true)

# Remove the counter set

$DCstRemote.Delete()

As noted, the steps in this recipe produce no output. Once you have completed the steps in this recipe, you can view the data collector inside perfmon, which looks like this:



# Reporting on performance data

Once you have created performance information using a PLA data collector set, you can use PowerShell to analyze the data.

In this recipe, you create a very simple report on the CPU usage of SRV1. The source of the report is the information logged as a result of the Creating and using PLA data collection sets recipe. As noted earlier, PLA can output the performance data in a variety of formats. In the Creating and using PLA data collector sets recipe, you used a binary log file format. This recipe, on the other hand, makes uses of a CSV format.

## Getting ready

This recipe uses PLA data collection output logged in a CSV format from SRV1. To create CSV output, use the Creating and using PLA data collector sets recipe and change the value of the log file format to 1.

## How to do it...

1. Import the CSV file of counter samples:

$Folder = 'C:\PerfLogs\Admin'

$File = Get-ChildItem -Path $Folder\\*.csv -Recurse

1. Import the performance counters:

$Counters = Import-Csv $File.FullName

"$($Counters.Count) counters in $($File.FullName)"

1. Fix the issue with the first row in the counters:

$Counters[0] = $Counters[1]

1. Obtain basic CPU stats:

$CN = '\\SRV1\Processor(\_Total)\% Processor Time'

$HT = @{

Name = 'CPU'

Expression = {[System.Double] $\_.$CN}

}

$Stats = $counters |

Select-Object -Property \*,$HT |

Measure-Object -Property CPU -Average -Minimum -Maximum

1. Add the 95th percentile value of the CPU:

$CN = '\\SRV1\Processor(\_Total)\% Processor Time'

$Row = [int]($Counters.Count \* .95 )

$CPU = ($Counters.$CN | Sort-Object)

$CPU95 = [double] $CPU[$Row]

$AMHT = @{

InputObject = $Stats

Name = 'CPU95'

MemberType = 'NoteProperty'

Value = $CPU95

}

Add-Member @AMHT

1. Combine the results into a single report:

$Stats.CPU95 = $Stats.CPU95.ToString('n2')

$Stats.Average = $Stats.Average.ToString('n2')

$Stats.Maximum = $Stats.Maximum.ToString('n2')

$Stats.Minimum = $Stats.Minimum.ToString('n2')

1. Display the CPU performance summary:

$Stats |

Format-Table -Property Property,Count,Maximum,CPU95,Minimum

## How it works...

This recipe uses a CSV file of performance data created by using PLA. In step 1, you import the data in the CSV file. This step produces no output.

In step 2, you display the number of counter samples imported, which looks like this:



In step 3, you fix a known error with PLA affecting the first row returned (PLA returns an invalid first row). The fix is to assume the first two rows have identical values. That has the potential to introduce some degree of error into detailed calculations. The impact of such an error is small and mitigated by having a significant number of samples.

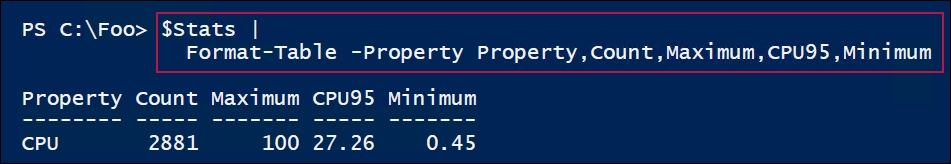
In step 4, you analyze the data sample and derive basic CPU usage statistics. In step 5, you create a 95th percentile value for CPU usage on SRV1.

With step 6, you work out an approximation of the 95th percentile CPU time. This is a good number to track, as it is a measure of how high, in general, the CPU load is on the computers that you are monitoring (95 percent of the time). It eliminates infrequent but high CPU measurements which might be misleading. You calculate this by first counting the total number of rows returned (sorted by CPU utilization), and then calculating an index that contains a value of 0.95 times the number of rows.

You then use this index to get that row from a list of sorted CPU values. With 100 rows of data returned, this calculation would return row 95 (that is, the 95th-highest CPU reading in the sample set).

Assuming you have a significant number of samples, this approach gets you the row that is a good approximation of the 95th-percentile CPU time measurement. At the end of this step, you add the value as a note property (CPU95).

In step 7, you display a summary of the CPU usage on SRV1, including a maximum, minimum, and 95th-percentile average CPU utilization, which looks like this:



## There's more...

In this recipe, we reported on just one counter, the total CPU time on just one computer (SRV1). Although there was a maximum CPU utilization during the sampling of 100%, most of the time, CPU utilization was at or below 27.26%. That suggests SRV1 is not CPU-bound.

This recipe just reported on a single performance counter. It would be straightforward to update your data-collection process to include more counters, which enables you to report on more performance information. You could include counters for networking, storage counters, and more. These other counters may help you to understand just why CPU utilization is high.

You could also make use of data collector output from the different hosts in your infrastructure. You could then adjust this recipe to report on these additional counter values from all different hosts. Knowing that overall performance load is high on Hyper-V Host HV1 but low on the host HV2 might suggest moving a VM or two between the hosts.

# Generating a performance-monitoring graph

In the Reporting on performance data recipe, you saw how you could take the data logged by a PLA data collector set and create a performance report. The report in that recipe showed CPU utilization of SRV1. That output is in the form of a table and is a summary of the performance of the server.

Another way to view the performance data is in the form of a graph. PowerShell does not have direct cmdlet support for displaying rich graphs, but the .NET Framework's System.Windows.Forms.DataVisualization namespace does.

This recipe uses the data visualization's Chart object to create a chart and save it as a Portable Network Graphic (PNG) file. You then display the graphic on your workstation.

## Getting ready

You run this recipe on SRV1. This recipe uses the output of the PLA data collector set similar to the one you created and started in the Creating and using PLA data collector sets recipe. Note that the input to this recipe is a CSV file produced by PLA.

## How to do it...

1. Load the assembly containing the DataVisualization classes:

Add-Type -AssemblyName System.Windows.Forms.DataVisualization

1. Import the CSV data from earlier, and fix the row 0 issue:

$CSVFile = Get-ChildItem -Path C:\PerfLogs\Admin\\*.csv -rec

$Counters = Import-Csv $CSVFile

$Counters[0] = $Counters[1] # fix row 0 issue

1. Create a chart object:

$Type = 'System.Windows.Forms.DataVisualization.Charting.Chart'

$CPUChart = New-Object -TypeName $Type

1. Define the chart dimensions:

$CPUChart.Width = 1000

$CPUChart.Height = 600

$CPUChart.Titles.Add("SRV1 CPU Utilisation") | Out-Null

1. Create and define the chart area:

$Type = 'System.Windows.Forms.DataVisualization.' +

'Charting.ChartArea'

$ChartArea = New-Object -TypeName $Type

$ChartArea.Name = "SRV1 CPU Usage"

$ChartArea.AxisY.Title = "% CPU Usage"

$CPUChart.ChartAreas.Add($ChartArea)

1. Get the date/time column:

$Name = ($counters[0] |

Get-Member |

Where-Object MemberType -EQ "NoteProperty")[0].Name

1. Add the counter sample values to the chart:

$CPUChart.Series.Add("CPUPerc") | Out-Null

$CPUChart.Series["CPUPerc"].ChartType = "Line"

$CPUCounter = '\\SRV1\Processor(\_Total)\% Processor Time'

$Counters |

ForEach-Object {$CPUChart.Series["CPUPerc"].Points.AddXY($\_.$Name,$\_.$CPUCounter)|

Out-Null

}

1. Ensure the output folder exists, then save the chart image as a PNG file in the folder:

$NIHT = @{

Path = 'C:\Perflogs\Reports'

ItemType = 'Directory'

ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT # create the folder if it does not exist

$CPUChart.SaveImage("C:\PerfLogs\Reports\Srv1CPU.Png", 'PNG')

1. Use the mspaint.exe application to view the chart image:

mspaint.exe C:\PerfLogs\Reports\SRV1cpu.Png

## How it works...

Like the Reporting on performance data recipe, the steps in this recipe produce no output (except step 9 where you view the chart). That is usual when you use many objects in the .NET Framework or when you use COM objects.

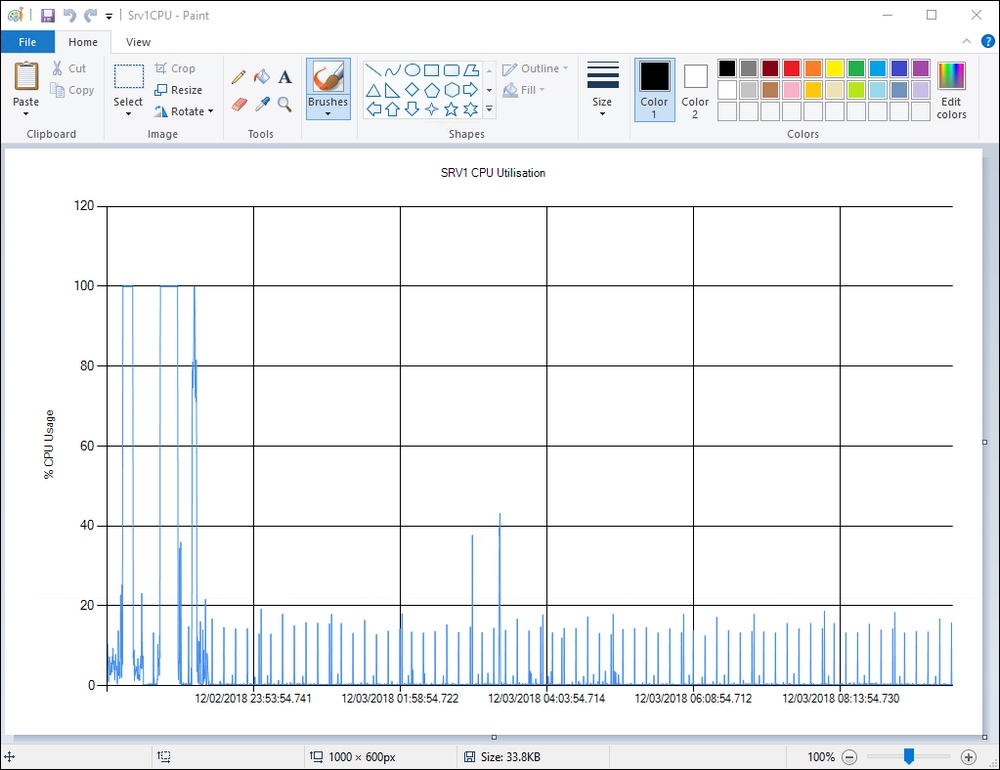
In step 1, you load the assembly containing the .NET classes you are using in this recipe. By default, this is an assembly that is not loaded by PowerShell.

With step 2, you import the PLA-created CPU data for SRV1. This file consists of a number of counter samples. You configured these details in the Creating and using PLA data collector sets recipe.

In step 3, you create a chart object, and in step 4, you define the chart's dimensions. In step 5, you create and configure a chart area object then add it to the chart.

In step 6, you get the name of the time and date column within the performance counters. Step 7 adds the data to the chart. Step 8 saves the chart as a PNG file.

Executing step 9 invokes the mspaint.exe application, which displays the PNG file, as follows:



## There's more...

This recipe showed you how to create a simple report graphing one counter, CPUutilization, across several hours of monitoring one server. You could add a second series, such as memory pages per second, to the chart. The result could be a customized graph that is similar to what you see in Performance Monitor. You could also incorporate data from more servers to the chart.

To support ongoing server monitoring, consider creating scheduled tasks to create the performance graphs and email the resultant output to those who need to know. Or have the scheduled task create a new web page on your intranet and drop the graphs into the page.

# Creating a system diagnostic report

The PLA subsystem that you have been working with in this chapter has an additional system-defined report known as the System Diagnostic Report. This report monitors a system for a period then provides a detailed report on the server.

## Getting ready

You use the SRV1 server that you have used in other recipes in this chapter.

## How to do it...

1. Start the built-in data collector on the local system, which generates the report:

$PerfReportName="System\System Diagnostics"

$DataSet = New-Object -ComObject Pla.DataCollectorSet

$DataSet.Query($PerfReportName,$null)

$DataSet.Start($true)

1. Output a message, then wait for the data collector to finish:

"Sleeping for [$($Dataset.Duration)] seconds"

Start-Sleep -Seconds $Dataset.Duration

1. Get the report and save it as HTML:

$Dataset.Query($PerfReportName,$null)

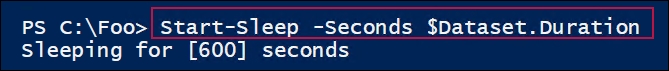
$PerfReport = $Dataset.LatestOutputLocation + "\Report.html"

1. View the report:

& $PerfReport

## How it works...

In step 1, you create a DataCollectorSet object, which starts the System Diagnostic Reporting process. With step 2, you wait for PLA to complete building the report, which looks like this:



In step 3, after the report has completed, you save it as HTML. Then, in step 4, you display it, which looks like this:



## There's more...

In step 4, you view the report in your browser. You could adapt this recipe to run as a scheduled task on every server in your network once daily. Store the output in a central place and you can easily access it to begin troubleshooting a server.

# Reporting on printer usage

Knowing who is using your printing devices and how much can be important in terms of capacity planning.

By default, Windows does not log printer usage information. But it is simple to turn on this logging and use the results.

## Getting ready

You use this recipe on the PSRV host.

## How to do it...

1. Run wevtutil.exe to turn on printer monitoring on the PSRV host:

$LogName = 'Microsoft-Windows-PrintService/Operational'

wevtutil.exe sl $LogName /enabled:true

1. Define a function that returns objects for each printer job completed on the server:

Function Get-PrinterUsage {

# 2.1 Get events from the print server event log

$LogName = 'Microsoft-Windows-PrintService/Operational'

$Dps = Get-WinEvent -LogName $LogName |

Where-Object ID -eq 307

Foreach ($Dp in $Dps) {

# 2.2 Create an ordered hash table

$Document = [ordered] @{}

# 2.3 Populate the hash table with properties from the

# Event Log entry

$Document.Id = $Dp.Properties[0].value

$Document.Type = $Dp.Properties[1].value

$Document.User = $Dp.Properties[2].value

$Document.Computer = $Dp.Properties[3].value

$Document.Printer = $Dp.Properties[4].value

$Document.Port = $Dp.Properties[5].value

$Document.Bytes = $Dp.Properties[6].value

$Document.Pages = $Dp.Properties[7].value

# 2.4 Create an object for this printer usage entry

$UEntry = New-Object -TypeName PSObject -Property $Document

# 2.5 And give it a more relecant tyhpe name

$UEntry.PsTypeNames.Clear()

$UEntry.PsTypeNames.Add('Reskit.PrintUsage')

# 2.6 Output the entry

$UEntry

} # End of foreach

} # End of function

1. Set and use an alias to get printer usage:

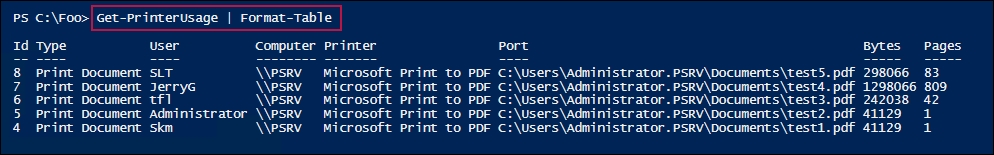
Get-PrinterUsage | Format-Table

## How it works...

In step 1, which produces no output, you turn on event logging of completed printer jobs. By default, this logging is not enabled.

In step 2, you create a function that returns objects representing completed printer jobs. The function first combs the Microsoft-Windows-PrintService/Operational event log for printer job details. Turning on event logging results in an event log entry for each completed job. For each event log entry found, the function extracts the details of the print job, creates a custom object documenting the job, and returns that object. Before returning the object, the function also changes the object returned to be of the Reskit.PrintUsage type.

In step 3, you call the function to return output that looks like this:



## There's more...

In step 2, you created and returned a custom object for each event log entry found. In doing so, you change the type name used by PowerShell for this object. This enables you to create custom-display XML to display the object nicely. By default, the object would be displayed in a list. Your custom-display XML could force the output to be a table (much in the way the display XML for Get-Process is structured).

# Monitoring Hyper-V utilization and performance

This report gathers information about a Hyper-V server and the VMs running on that server.

## Getting ready

You run this recipe on the HV1 Hyper-V host.

## How to do it...

1. Create a basic report hash table:

$ReportHT = [Ordered] @{}

1. Get the host details and add them to the report hash table:

$HostDetails = Get-CimInstance -ClassName Win32\_ComputerSystem

$ReportHT.HostName = $HostDetails.Name

$ReportHT.Maker = $HostDetails.Manufacturer

$ReportHT.Model = $HostDetails.Model

1. Add the PowerShell version information to the report hash table:

$ReportHT.PSVersion = $PSVersionTable.PSVersion.ToString()

1. Add OS information to the report hash table:

$OS = Get-CimInstance -Class Win32\_OperatingSystem

$ReportHT.OSEdition = $OS.Caption

$ReportHT.OSArch = $OS.OSArchitecture

$ReportHT.OSLang = $OS.OSLanguage

$ReportHT.LastBootTime = $os.LastBootUpTime

$Now = Get-Date

$UTD = [float] ("{0:n3}" -f (($Now –

$OS.LastBootUpTime).Totaldays))

$ReportHT.UpTimeDays = $UTD

1. Add a count of processors in the host to the report hash table:

$PHT = @{

ClassName = 'MSvm\_Processor'

Namespace = 'Root/Virtualization/v2'

}

$Proc = Get-CimInstance @PHT

$ReportHT.CPUCount = ($Proc |

Where-Object ElementName -Match 'Logical Processor').Count

1. Add the current host CPU usage to the report hash table:

$Cname = '\\.\processor(\_total)\% processor time'

$CPU = Get-Counter -Counter $Cname

$ReportHT.HostCPUUsage = $CPU.CounterSamples.CookedValue

1. Add the total host physical memory to the report hash table:

$Memory = Get-Ciminstance -Class Win32\_ComputerSystem

$HostMemory = [float]("{0:n2}" -f

($Memory.TotalPhysicalMemory/1GB))

$ReportHT.HostMemoryGB = $HostMemory

1. Add the memory allocated to VMs to the report hash table:

$Sum = 0

Get-VM | Foreach-Object {$sum += $\_.MemoryAssigned + $Total}

$Sum = [float] ( "{0:N2}" -f ($Sum/1gb) )

$ReportHT.AllocatedMemoryGB = $Sum

1. Create the host report object from the hash table:

$Reportobj = New-Object -TypeName PSObject -Property $ReportHT

1. Create the report header:

$Report = "Hyper-V Report for: $(hostname)`n"

$Report += "At: [$(Get-Date)]"

1. Add the report object to the report:

$Report += $Reportobj | Out-String

1. Get the VM details on the local VM host and create a container array for individual VM-related objects:

$VMs = Get-VM -Name \*

$VMHT = @() # to be an array of hash tables

1. Get VM details for each VM into an object added to the hash table container:

Foreach ($VM in $VMs) {

# Create VM Report hash table for this VM

$VMReport = [ordered] @{}

# Add VM's Name

$VMReport.VMName = $VM.VMName

# Add Status

$VMReport.Status = $VM.Status

# Add current VM uptime

$VMReport.Uptime = $VM.Uptime

# Add VM CPU

$VMReport.VMCPU = $VM.CPUUsage

# Replication mode and status

$VMReport.ReplMode = $VM.ReplicationMode

$VMReport.ReplState = $Vm.ReplicationState

# Create an object from Hash table and add to array

$VMR = New-Object -TypeName PSObject -Property $VMReport

$VMHT += $VMR

}

1. Convert the array of hash tables to a nice string, finishing the report creation:

$Report += $VMHT | Format-Table | Out-String

1. Display the report:

$Report

## How it works...

This recipe creates a report that describes both the Hyper-V host and the VM on the host. In the first part, you build a hash table containing information about the host itself which you add to the report. Then, you report on the VMs running on this server (HV1).

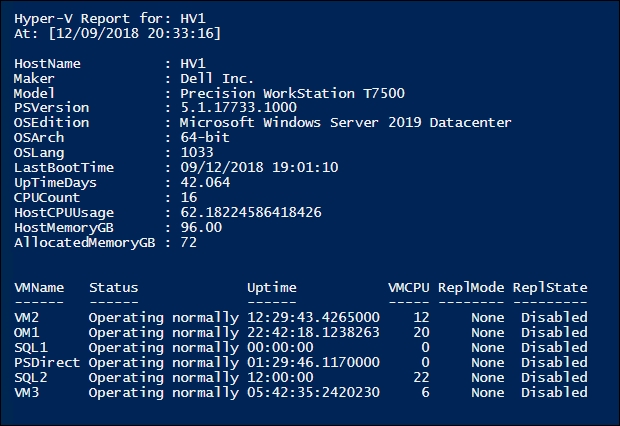
In step 1, you create an ordered hash table ($ReportHT). You create an ordered hash table so that the order of rows is maintained. With step 2, you add basic host details to the hash table.

In step 3, you add details of the PowerShell version running on the host. With step 4, you add operating system details to the hash table along with (in step 5) a count of processors available in the host. Then you add current host CPU usage in step 6, and total host physical memory in step 7. In step 8, you calculate the total amount of memory assigned to VMs and add it to the hash table.

In step 9, you create a summary report object with properties coming from the hash table. In step 10, you create a basic report header to which, in step 11, you convert the report object to a string and add it the report.

With step 12 and step 13, you retrieve details of each of the VMs assigned to HV1, and with step 14, you add those details to the report.

Finally, in step 15, you view the report, which looks like this:



## There's more...

In this recipe, you create and use two hash tables. The first hash table holds details about the VM host overall. If you were reporting on multiple Hyper-V servers, you could create hash tables relating to each server and then convert those two hash tables into objects to report on. If you are going to expand this recipe to cover multiple servers, using objects to report from is easier.

This recipe chose some basic performance and usage information to report. As ever with performance analysis, you could add more. For example, you could add network throughput, storage usage information to the report header. And you could expand the information reported on each VM to include more details. An important thing to consider is when to stop adding details to your reports.