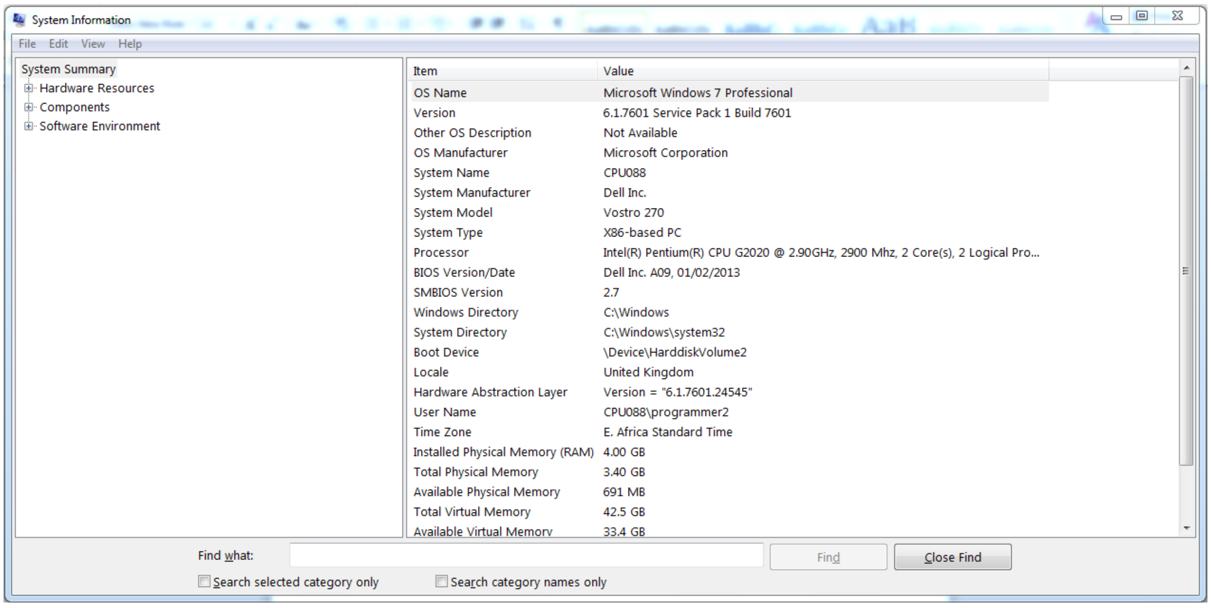
POWERSHELL SYSTEM INVENTORY

Contents

Introduction

Current operating systems have graphical interfaces that can allow us to easily see any information about a computer system. For instance, one can run the command msinfo32 in windows run (windows key + R) and get a system summary as follows:



Item	Value
OS Name	Microsoft Windows 7 Professional
Version	6.1.7601 Service Pack 1 Build 7601
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	CPU088
System Manufacturer	Dell Inc.
System Model	Vostro 270
System Type	X86-based PC
Processor	Intel(R) Pentium(R) CPU G2020 @ 2.90GHz, 2900 Mhz, 2 Core(s), 2 Logical Pro...
BIOS Version/Date	Dell Inc. A09, 01/02/2013
SMBIOS Version	2.7
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume2
Locale	United Kingdom
Hardware Abstraction Layer	Version = "6.1.7601.24545"
User Name	CPU088\programmer2
Time Zone	E. Africa Standard Time
Installed Physical Memory (RAM)	4.00 GB
Total Physical Memory	3.40 GB
Available Physical Memory	691 MB
Total Virtual Memory	42.5 GB
Available Virtual Memory	33.4 GB

There are more of such tools in the control panel, system and security, administrative tools. These tools are very useful when troubleshooting or when creating a system inventory. There are however circumstances where such tools cannot work effectively. For example, if one wanted to automate the process of gathering such information on a regular basis and store an inventory record. Another case is where one wants to gather such information from many machines connected to a network. Luckily, we can create PowerShell scripts that can execute automatically to perform such kind of work.

Task list

The task at hand requires us to write a PowerShell script that can allow us to:

1) Get a list of computers to scan from a text file.
2) Get inventory data from the machines which will include:
    - Name of the computer
    - CPU: Number of cores, clock speed, manufacturer
    - RAM: Total amount, number of slots
    - All local disk drives
    - IP address of each
    - Username and description
    - Operating system version
    - Uptime
3) Output a text file containing the inventory data suitably formatted.

The solution

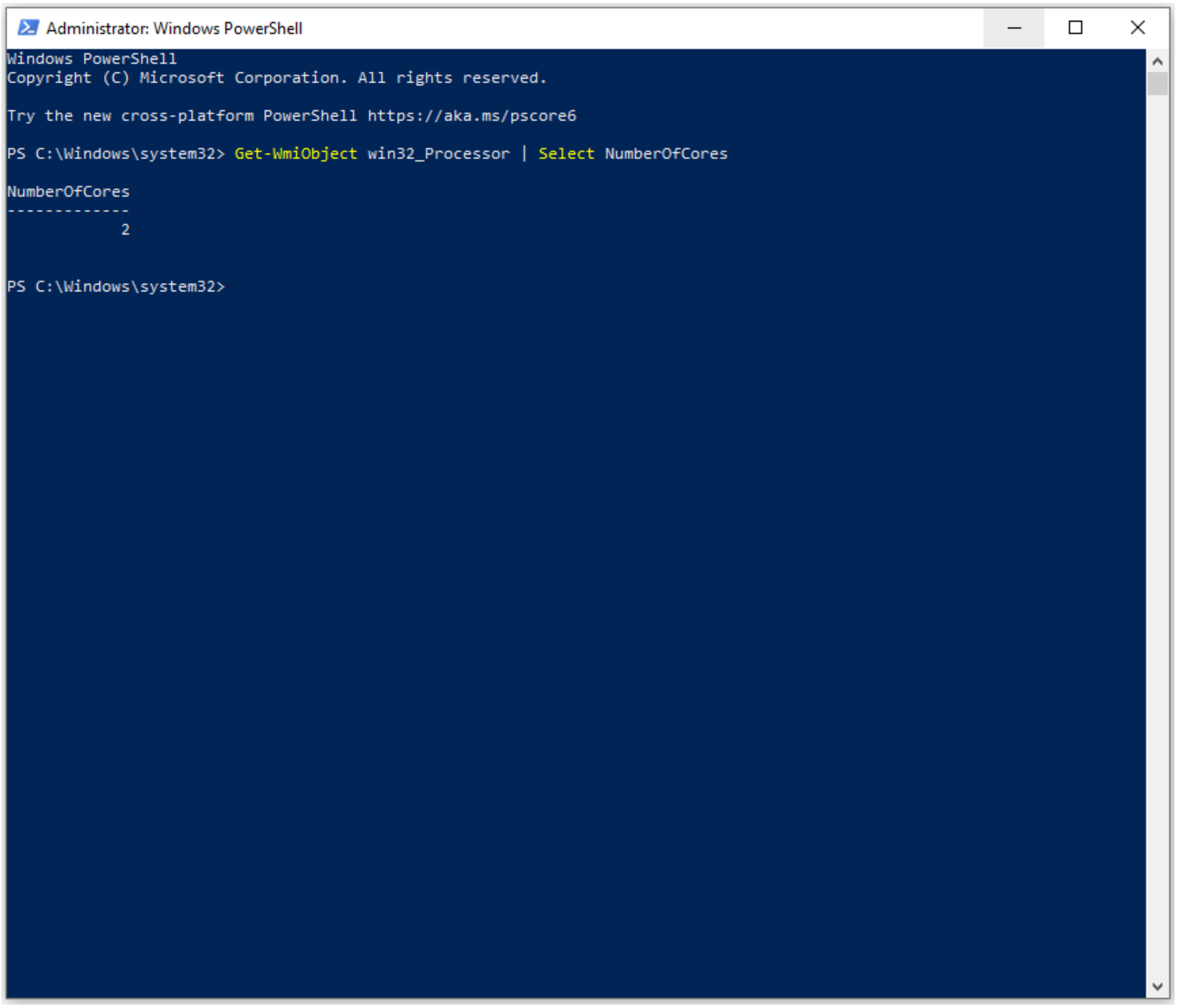
To be able to complete these tasks successfully, we will utilize either Get-CimInstance or Get-WmiObject. CIM stands for Common Information Model. It is an object-oriented, open-source standard for retrieving, displaying, and managing computer, network, application, and services information.

WMI stands for Windows Management Instrumentation. It is Microsoft's own implementation of CIM and has been around since Windows NT 4.0. Originally, Microsoft used DCOM (Distributed COM) and RPC (Remote Procedure Call) to allow remote connections, but this had shortcomings when traversing networks. Starting with Windows 8 and Server 2012, Microsoft moved to WSMan (Web Services Management) for remote management, though older protocols are still supported for backward compatibility.

In our case we will use Get-WmiObject which contains classes with methods which will be able to get the necessary information for us. For example, win32_Processor is a class in the WmiObject who's superclass is CMI_Processor and can be used to fetch information about the subject computer's processor. There are many methods within the class. For example, to get the number of cores of a host computer processor, one would use the command

Get-WmiObject win32_Processor | Select NumberOfCores

The method being called here is 'NumberOfCores' and it gives us the number of cores as an integer.



Administrator: Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Windows\system32> Get-WmiObject win32_Processor | Select NumberOfCores

NumberOfCores
-------------
            2


PS C:\Windows\system32>
```

The Code

The code is as follows:

```
$fpath = "C:\Inventory"

gc $fpath\computers.txt | % {
    $computer = $_

    $computerOS = Get-WmiObject Win32_OperatingSystem -ComputerName
$computer
    $computerSystem = Get-WmiObject Win32_ComputerSystem -ComputerName
$computer
    $computerCPU = Get-WmiObject Win32_Processor -ComputerName $computer
    $computerRAM = Get-WmiObject Win32_PhysicalMemory -ComputerName
$computer
    $computerHDD = Get-WMIObject Win32_LogicalDisk -filter "deviceid='C:'" -
ComputerName $computer

    function NetAdapter {
            return Get-WmiObject win32_networkadapterconfiguration -ComputerName
$computer | where { $_.ipaddress -like "1*" } | select MACAddress, IPAddress | select -
First 1
    }
    function UpTime {
            $LastBoot = (Get-WmiObject -ComputerName $computer -Query "SELECT
LastBootUpTime FROM Win32_OperatingSystem")
            $LastBootTime =
$Lastboot.ConvertToDateTime($LastBoot.LastBootUpTime)
            return $Span = New-TimeSpan -Start $LastBootTime -End (Get-Date)
    }

    $computerNAC = NetAdapter
    $computerUT = UpTime

    $Object = New-Object PSObject -Property @{
            "Machine: " = $computerSystem.Name
            "CPU: " = "Cores: " + [String] $computerCPU.NumberOfCores + ", Clock
Speed: " + [String] $computerCPU.MaxClockSpeed + ", Manufacturer: " +
$computerCPU.Manufacturer
            "RAM: " = "Total Amount: {0:N2}" -f
($computerSystem.TotalPhysicalMemory/1GB) + "GB" + ", Slots:" +
$computerRAM.count
            "HDD: " = "ID: " + $computerHDD.deviceid + ", HDD_Capacity: {0:N2}" -f
($computerHDD.Size/1GB) + "GB" + ", HDD_Space: {0:P2}" -f
($computerHDD.FreeSpace/$computerHDD.Size) + ", Free (" + "{0:N2}" -f
($computerHDD.FreeSpace/1GB) + "GB)"
            "Network Interfaces: " = "IP Address: " + [String] $computerNAC.IPAddress
+ " MAC Address: " + [String] $computerNAC.MACAddress
            "Local Users: " = "Username: " + $computerSystem.UserName + ",
Workgroup: " + $computerSystem.Workgroup
            "OperatingSystem: " = "OS: " + $computerOS.caption + ", Version: " +
$computerOS.Version
            "Uptime: " = $computerUT
    }
    Write-Output $Object

} | export-csv $fpath\result.csv -notype
```

The line-by-line explanation is as follows:

```
$fpath = "C:\Inventory"
```

This line stores the path to our working directory. It is where the PowerShell script resides and will be run from there. It is also where a text file containing the computers to be scanned is kept. The path is also used to store the exported results.

```
gc $fpath\computers.txt | % { }
```

This command reads the text file cumputers.txt and loops through the contents.

```
$computer = $_
```

This command assigns the current line read from the file to a variable $computer which will hold the name of the current computer to which we will perform the operations that follow.

```
$computerOS = Get-WmiObject Win32_OperatingSystem -ComputerName $computer
$computerSystem = Get-WmiObject Win32_ComputerSystem -ComputerName $computer
$computerCPU = Get-WmiObject Win32_Processor -ComputerName $computer
$computerRAM = Get-WmiObject Win32_PhysicalMemory -ComputerName $computer
$computerHDD = Get-WMIObject Win32_LogicalDisk -filter "deviceid='C:'" -
ComputerName $computer
$computerNAC = Get-WmiObject win32_networkadapterconfiguration -ComputerName
$computer | where { $_.ipaddress -like "1*" } | select MACAddress, IPAddress | select -First
1
```

These set of commands utilize the WmiObject to fetch information about the various components of the computer that we want to know about. For example, Get-WmiObject Win32_OperatingSystem gets the operating system information of the computer in the $computer variable. Notice the $computer variable at the end of each command to specify the machine which we are fetching data for. Win32_Processor gets the processor information. Win32_PhysicalMemory gets the RAM information. Win32_LogicalDisk gets the disk drive information. Win32_ComputerSystem gets the general system information and win32_NetworkAdapterConfiguration gets the network adapter information but with some conditions attached to the command like the ‘where { $_.ipaddress -like "1*" }’ which specifies the kind of format we expect the ipaddress to contain. We are also specifying that we only need the macaddress and the ipaddress methods from this class.

```
function UpTime {
	$LastBoot = (Get-WmiObject -ComputerName $computer -Query "SELECT
LastBootUpTime FROM Win32_OperatingSystem")
	$LastBootTime = $Lastboot.ConvertToDateTime($LastBoot.LastBootUpTime)
	return $Span = New-TimeSpan -Start $LastBootTime -End (Get-Date)
}
```

This is a function definition. The function UpTime is being used to calculate the uptime which was one of the required tasks. The system is able to get the last boot-up time through the method ‘LastBootUpTime’ which is in the Win32_OperatingSystem class.

```
$LastBoot = (Get-WmiObject -ComputerName $computer -Query "SELECT
LastBootUpTime FROM Win32_OperatingSystem")
```

The last boot-up time is in seconds so we need to convert it to a date-time value (in purple).

```
$LastBootTime = $Lastboot.ConvertToDateTime($LastBoot.LastBootUpTime)
```

The uptime thus can be calculated by subtracting the last boot-up time from the current date-time.

```
return $Span = New-TimeSpan -Start $LastBootTime -End (Get-Date)
```

Return the result from the function call which will contain the uptime in days, hours, minutes and seconds.

```
$computerUT = UpTime
```

Call the function and assign the value to a variable $computerUT.

```
$Object = New-Object PSObject -Property @{ }
```

Create a new data object to store the information that we will get from the various methods that we will run. This will enable easy and quick access to the information not to mention the good order the information will be stored in. The information can be accessed through indices or by referencing. For example, to access the first computer's RAM information, one can use indices like $Object[0][2] which specifies the first row [0] and the 3rd column [2]. We can also just reference the column as $Object[0][“RAM”]

```
"Machine: " = $computerSystem.Name
```

This line fetches the computer name and assigns it to a column titled ‘Machine:’ It call the method ‘Name’ from the class ‘computerSystem’.

```
"CPU: " = "Cores: " + [String] $computerCPU.NumberOfCores + ", Clock Speed: " +
[String] $computerCPU.MaxClockSpeed + ", Manufacturer: " +
$computerCPU.Manufacturer
```

This line fetches the CPU information from the class computerCPU. It calls the methods ‘NumberOfCores’, ‘MaxClockSpeed’, and ‘Manufacturer’. The information is stored in a column named ‘CPU’.

```
"RAM: " = "Total Amount: {0:N2}" -f ($computerSystem.TotalPhysicalMemory/1GB) +
"GB" + ", Slots:" + $computerRAM.count
```

This line fetches the RAM information from the class computerSystem and computerRAM. It calls the method ‘TotalPhysicalMemory’ in the computerSystem class which returns the RAM capacity in bytes. We convert that value into gigabytes by dividing by the size of 1GB. It also calls the method count in the computerRAM class to give us the number RAM sticks available in the computer.

```
"HDD: " = "ID: " + $computerHDD.deviceid + ", HDD_Capacity: {0:N2}" -f
($computerHDD.Size/1GB) + "GB" + ", HDD_Space: {0:P2}" -f
($computerHDD.FreeSpace/$computerHDD.Size) + ", Free (" + "{0:N2}" -f
($computerHDD.FreeSpace/1GB) + "GB)"
```

This line fetches the disk drive information from the computerHDD class. It calls the mothed 'deviceid' to get the disk drive id. It calls the methods 'Size' to get the disk capacity in bytes. It is converted to gigabytes by dividing by 1GB. It also calls the 'FreeSpace' to get the free space in bytes then converted to gigabytes.

```
"Network Interfaces: " = "IP Address: " + [String] $computerNAC.IPAddress + ", MAC Address: " + [String] $computerNAC.MACAddress
```

This line fetches the network information from the computerNAC class. It calls the 'IPAddress' method to get the ipaddress of the computer and the 'MACAddress' method to get the macaddress of the computer.

```
"Local Users: " = "Username: " + $computerSystem.UserName + ", Workgroup: " + $computerSystem.Workgroup
```

This line fetches the user information from the class computerSystem. It calls the method 'UserName' to get the computer username and the 'Workgroup' method to get the workgroup they belong to.

```
"OperatingSystem: " = "OS: " + $computerOS.Caption + ", Version: " + $computerOS.Version
```

This line fetches the operating system information from the computerOS class. It calls the method 'Caption' to get the operating system and 'Version' method to get the version of the operating system.

```
"Uptime: " = $computerUT
```

This line adds the uptime calculated by the UpTime function to the data object.

```
Write-Output $Object
```

This line outputs the data object to the PowerShell terminal. An example can be seen below at the results of running the script on a local machine.

```
| export-csv $fpath\result.csv -notype
```

This line exports the data into a csv file called results. The path to the file was specified at the beginning of the script and stored in the variable $fpath. Each of the columns and rows are entered correctly in the file. We can also output to a text file with the command:

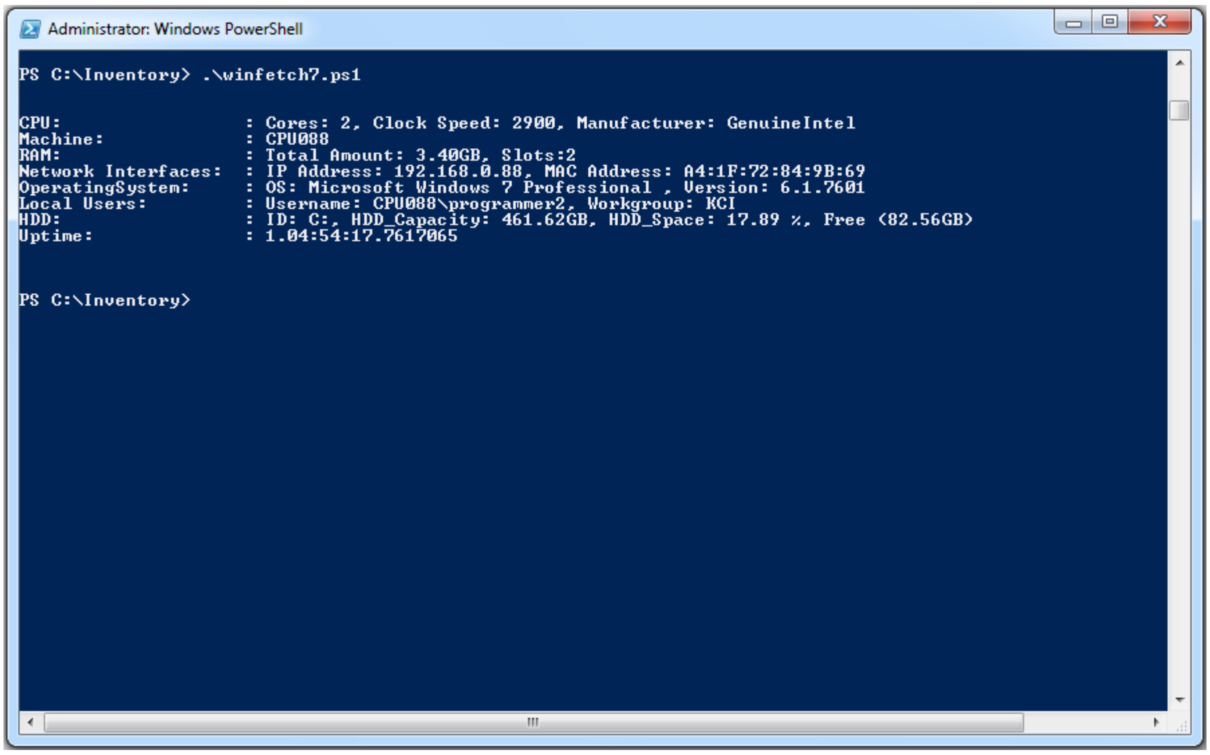
```
| Out-File result.txt
```

Running the script

The script can be run on a windows PowerShell interface by typing:

```
.\ winfetch.ps1
```

The resulting output is as follows:



Administrator: Windows PowerShell

```
PS C:\Inventory> .\winfetch7.ps1

CPU:                 : Cores: 2, Clock Speed: 2900, Manufacturer: GenuineIntel
Machine:             : CPU088
RAM:                 : Total Amount: 3.40GB, Slots:2
Network Interfaces:  : IP Address: 192.168.0.88, MAC Address: A4:1F:72:84:9B:69
OperatingSystem:     : OS: Microsoft Windows 7 Professional , Version: 6.1.7601
Local Users:         : Username: CPU088\programmer2, Workgroup: KCI
HDD:                 : ID: C:, HDD_Capacity: 461.62GB, HDD_Space: 17.89 %, Free (82.56GB)
Uptime:              : 1.04:54:17.7617065


PS C:\Inventory>
```

Problems and solutions

There were some challenges encountered. For one, some computers do not have the Get-CimInstance or Common Information Model. This was overcome by using Get-WmiObject.

The Get-CimInstance does not have a NetworkAdapterConfiguration method that is used to fetch the network information. To overcome this, I had to use CmdLet to write a script for Get-NetworkInfo which contained parameters that specify the parameters to enable in order to get information about the network.

Calculating the uptime was also challenging because of the confusion of converting from seconds to days, hours and minutes. Before the calculation, both the start and end parameters had to be in the same format with the result being cast into a new timespan.

There was also a challenge in getting the documentation for the various methods in the various classes.

Conclusion

The task was challenging but resulted in learning about PowerShell classes and their methods. This was a good experience that opens the door to exploring the windows PowerShell more.

References

1. PowerShell Documentation. Retrieved from https://docs.microsoft.com/en-us/powershell/
2. PowerShell Fundamentals. Retrieved from https://powershell.one/fundamentals
3. The Most USEFUL PowerShell CmdLets and more. Retrieved from https://www.improvescripting.com/
4. PowerShell Guide: Get-CimInstance And Get-WmiObject. Retrieved from https://www.pdq.com/blog/powershell-guide-get-ciminstance-and-get-wmiobject/
5. Programming Language Guide. Retrieved from https://www.autoscripts.net/categories/shell-bash/
6. Get-CIMInstance (PowerShell 3.0+). Retrieved from https://ss64.com/ps/get-ciminstance.html
7. PowerShell Howtos. Retrieved from https://www.delftstack.com/howto/powershell/
8. POWERSHELL. Retrieved from https://www.itprotoday.com/windows-and-user-productivity/powershell