The on-going Perf Improvement Work in PowerShell

Dongbo Wang

Performance and Profiling

- Steady-state perf benefits greatly from .NET Core improvement ☺
  - Example: Base64.ps1 runs about 30% faster in PowerShell 6.2 than in Windows PowerShell 5.1

```
## Windows PowerShell 5.1 (.NET Framework 4.8)
PS> Measure-Command { F:\demos\base64.ps1 } |
      Foreach-Object TotalMilliseconds
2214.4445
```

```
## PowerShell 6.2 (.NET Core 2.1)
PS> Measure-Command { F:\demos\base64.ps1 } |
      Foreach-Object TotalMilliseconds
1557.9588
```

* Base64.ps1 implements the Base64 algorithm and does a round-trip conversion from a long string (18000 characters) to Base64

- PowerShell startup time ☹

PowerShell Version	.NET Version	Configuration	pwsh -noprofile -c echo 1
Windows PowerShell 5.1	.NET Framework 4.8	NGEN + Background JIT	249.8 ms (baseline)
PowerShell 6.1	.NET Core 2.0	CrossGen + Background JIT	494.5 ms (~2x slower)
PowerShell 6.2	.NET Core 2.1	CrossGen + Background JIT + Tiered Compilation	367.6 ms (47% slower) (25.6% better than 6.1)

* data collected on machine with: i7-6700 CPU 3.40 GHz, 8 Logical processors, 64 GB memory, Windows 10

Performance and Profiling

-- NGEN vs. CrossGen

- NGEN can reduce the JIT compilation almost to 0 at startup of Windows PowerShell 5.1 (`> powershell -noprofile -c echo 1`)

Jitting Trigger	Num Compilations	% of total jitted compilations	Jit Time msec	Jit Time (% of total process CPU)
TOTAL	13	100.0	9.4	1.7
Foreground	13	100.0	9.4	1.7
Multicore JIT Background	0	0.0	0.0	0.0
Tiered Compilation Background	0	0.0	0.0	0.0

- CrossGen still requires a lot JITTING at startup of PowerShell 6.2 (`> pwsh -noprofile -c echo 1`)

Jitting Trigger	Num Compilations	% of total jitted compilations	Jit Time msec	Jit Time (% of total process CPU)
TOTAL	903	100.0	472.7	28.9
Foreground	398	44.1	120.0	7.3
Multicore JIT Background	13	1.4	26.6	1.6
Tiered Compilation Background	492	54.5	326.1	19.9

Performance and Profiling

-- CrossGen: now-built-in in .NET Core 3.0 SDK

- Announced for .NET Core 3.0 Preview 6
  - R2R Compilation: the “PublishReadyToRun” property
  - Assembly Linking: the “PublishTrimmed” property
  - Publish using an explicit RuntimeIdentifier

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <PublishTrimmed>true</PublishTrimmed>
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
</Project>

## Publish using an explicit runtime identifier
> dotnet publish -r win-x64 -c Release
```

Performance and Profiling
-- Tooling

- WinDbg works well with .NET Core when investigating dumps
  - Crossgen'ed assembly? Force loading the non-crossgen'ed symbols
    `.reload /i <System.Management.Automation.dll>`
- Windows Performance Recorder/Windows Performance Analyzer
  - Reference Set
- PerfView
  - Memory allocation / GC Statistics / JIT Statistics / CPU stacks
  - Crossgen'ed assembly? PerfView can generate PDBs for crossgen'ed assemblies
    `copy crossgen.exe to the application base folder`
- Profiling on non-Windows?
  - Today we only do perf profiling on Windows and just hope the changes could be similarly effective on Linux and macOS.

Current perf improvement work

- Guidelines
  - Should be well guided by the profiling traces
  - Should be low hanging fruits
    - Simple changes/refactoring trade for good results
    - Very low regression risk
- What's next?
  - Add performance tests to track the perf improvement and regression in the long term (#9996)
  - Go through all perf related issues in the PowerShell repo
    - Get better understanding with profiling and fix low hanging fruits
    - Track the findings that require more fundamental changes
- Examples

Examples

- An example of low-hanging fruits (simple change for good results)
  - One-line change to avoid the allocation of `CultureInfo` in the `ClrFacade.GetAssemblies()`
  - https://github.com/PowerShell/PowerShell/pull/10024
- Walk through the analysis/code change of `1..1e5 | foreach { 1+1 }`
  - Focused on avoid unnecessary allocations to reduce GC pressure
    - “GC Heap Alloc Ignore Free Stacks” from PerfView profiling
    - Analyze code to understand why huge boxing operations were involved
    - https://github.com/PowerShell/PowerShell/pull/10047