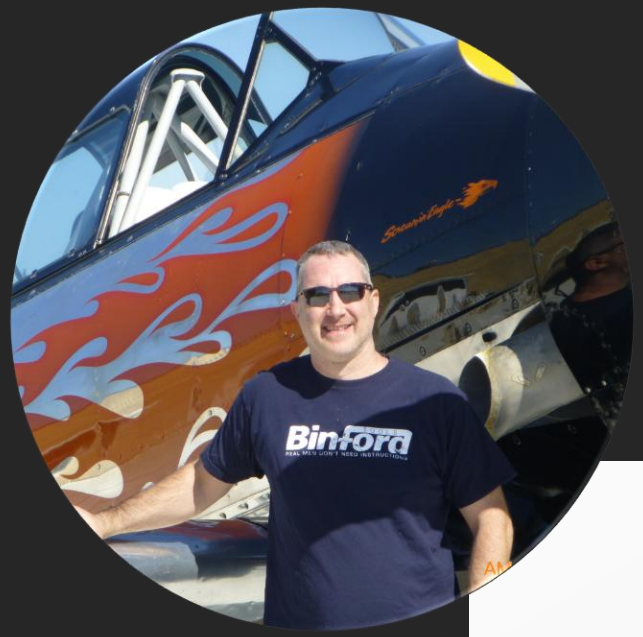
PowerShell Runspaces

More PowerShell, More Faster



Scott Corio

Sr. Enterprise Cloud Architect

Microsoft MVP
Azure

ScottCorio

DumpsterDave

blog.shitstormbrewing.beer

Why multithread?

- Process multiple items in parallel
  - Save on compute time in Azure
  - Get things done faster

- Keep User Interfaces Responsive
  - Prevents the UI from freezing

- Remote Systems
  - Managing multiple remote machines or Clusters

DEMO

Creating a Runspace

PowerShell Runspaces

- PowerShell runspaces form the very core of PowerShell
- When you open PowerShell, you are running an instance of the default runspace
- We can leverage runspaces individually, or as a collection of workers
- Multiple modules exist that make working with runspaces easier
  - But we’re going to focus on the OG so you can see how the sauce is made

Why use runspaces?

- Performance
  - Runspace Pools enable us to leverage many workers to break up larger tasks into smaller ones
  - Can split distinct tasks out such as doing work and writing output
- Security
  - Runspaces can run in different security contexts. Why run all your code as root if you don't need to
  - Something bad that happens in one runspace does not bleed into the others.
- Reduced Impact Surface
  - We can leverage custom runspaces to limit what users or processes have access to
- Why Else?

What about Real World Scenarios

- It's Unattended, why does speed matter?
  - If you run this on a platform that charges per minute (like Azure Automation) Saving a few minutes can save a few dollars

Advantages of Runspaces

- Can share information
  - Runspaces allow the two way sharing of **volatile** information between host and worker without exposing it to outside parties
  - Multiple workers can work on the same information at the same time*
- Lightweight
  - Each worker runs as a thread off of the main process instead of a separate process – less overhead
- Controllable
  - Can enter, exit, kill, debug… All without leaving home

Creating a simple runspace

```
$Runspace = [PowerShell]::Create()
$Runspace.AddScript({
  Params()
  DoStuff
})
$Params = @{KVP}
$Runspace.AddParameters($Params)
$Handle = $Runspace.BeginInvoke()
$output = $Runspace.PowerShell.EndInvoke($Handle)
$Runspace.PowerShell.Dispose()
```

Creating a (few) simple runspaces

```powershell
$RunspacePool = [runspacefactory]::CreateRunspacePool(1, 10)
$RunspacePool.Open()

$Spaces = [System.Collections.Generic.List[Object]]::new()

for ($i = 0; $i -lt 10; $i++) {
  $Runspace = [PowerShell]::Create()
  $Runspace.AddScript({
   Params()
   DoStuff
  })
  $Params = @{KVP}
  $Runspace.AddParameters($Params)
  $PowerShell.RunspacePool = $RunspacePool
  $Handle = $Runspace.BeginInvoke()

  $temp = '' | Select-Object PowerShell,Handle
 $temp.PowerShell = $PowerShell
 $temp.handle = $Handle
  [void]$Spaces.Add($Temp)
}

foreach ($rs in $Spaces) {
   $output = $rs.PowerShell.EndInvoke($rs.Handle)
   Write-Host $output -ForegroundColor Green
   $rs.PowerShell.Dispose()
}
```

DEMO

Creating Runspaces and Pools

Off to the races - Sharing information

- Runspaces are powerful but not safe by default
- Runspaces will let you do bad things
- Race Conditions can occur when information is not passed properly
- .NET has thread safe objects that should be used whenever information is to be passed back and forth
- Read-Only data can safely be passed as a simple parameter
- ArrayLists and Hashtables can be bi-directional. All other data types are read only (or are they?)

Two available approaches

- System.Collections
  - System.Collections provides thread safety via the Synchronized ArrayList and HashTable objects
  - Very rudimentary. Locks the entire collection for every read/write operation (but not modify)
  - No TypeSafety with System.Collections.
- System.Collections.Generic has TypeSafety, but no thread Safety
  - Does not provide thread synchronization
- System.Collections.Concurrent
  - Built-in thread safety implementation of ICollection class
  - Presents the IsSynchronized and SynchRoot properties (but doesn't use them)
  - Objects are both Thread-safe and Type-safe

DEMO

Sharing Basic Information

System.Collections.Concurrent

- BlockingCollection<T>
  - Provides Blocking and Bounding (Know when it's in use and completed)
- ConcurrentBag<T>
  - Unordered List
- ConcurrentDictionary<TKey,TValue>
  - Key/Value pair
- ConcurrentQueue<T>
  - First-in/First-Out
- ConcurrentStack<T>
  - First-in/Last-Out

When to use each

- BlockingCollection<T>
  - When you have one thread "Building" the collection and another "Consuming" it and the consumer needs to wait for the builder to complete.
- ConcurrentBag<T>
  - When order doesn't matter and you just need to add stuff to a list
- ConcurrentDictionary<TKey,TValue>
  - When you need a named index or key to lookup a specific value or pass a collection of non-thread safe values in a thread safe manner
- ConcurrentQueue<T>
  - When you need to process items in the order they're received, like displaying log entries
- ConcurrentStack<T>
  - When you need to process in reverse order like finding the root of a certificate chain and then re-assembling it so that the subject certificate is first out

DEMO

Really Sharing Information

Blocking Collection

- Blocking collections should only be used when you need to block other threads from doing stuff until you are done adding things to the collection
- Example: You want to check if there's enough space available to copy files. You have to wait until you know the total space of the files that are going to be moved before you can check free space. If you have 10 threads that are going to do the copy, you don't want them to start until you've finished gathering files and checking for free space.
- If a `TryTake()` is called, that call will fail/return null even if there are things in the collection if `IsAddingCompleted` is not `true`

Concurrent Bag

- Concurrent Bags are lists.
- They are un-ordered, but can be sorted.

Concurrent Dictionary

- Concurrent Dictionaries allow one to add or update items in a safe manner
- When one thread is updating a specific key value pair, it will block other threads from updating that pair until it is done, but not block access to the other pairs in the collection
- You must write an Update Factory Function to call these methods
  - ***{ param($key, $oldValue) <Your Formula> }***
  - params should accept 2 values (and only 2) which are the key and old value
  - Your formula should do what it needs to do and then output the new value
  - Be careful with increment/decrement operators (++/--). They don't work like you think. OldValue is **read only** and cannot be updated by the function.

Concurrent Queue/Stack

- Are like Ordered bags. The only difference is which end new things go into and which end they come out of.
- You can "Peek" at them to check for items before doing work
- All actions will enforce a lock, but it's very fast
- Push/Add/Enqueue do not have built in failure mechanisms because they should never fail (unless you destroy the object and then try to work with it again)
- Pop/Dequeue have failure mechanisms built in to ensure success. Because of this, the return value is true/false. You must use a `[ref]` value to receive the data
  - Be sure to declare you variable **BEFORE** you call pop/dequeue.
  - ```
    $value = $null
    $Success = $Collection.TryDequeue([ref]$Value)
    ```


    ```

Handling Handcuffs

- By default, each runsapce we create has access to all of the modules and data structure types that PowerShell has access to
- There can be cases where we don't need or want the runspace to have access to these resources
- By eliminating things we don't need, we can also reduce the memory footprint of each runspace
- Less things to load means faster startup times
  - If your default profile takes 10 seconds to load and you have to load 100 runspaces, you're gonna have a bad time.
- With Constrained runspaces, if you don't add something to the runspace, you **DON'T** have access to it. This includes namespaces.

An Example Handcuffed Runspace

```
$InitialSessionState = [System.Management.Automation.Runspaces.InitialSessionState]::Create()
$InitialSessionState.LanguageMode = 'Full'
$InitialSessionState.ExecutionPolicy = 'Bypass'
$InitialSessionState.ImportPSModule('Microsoft.PowerShell.Utility')

#Create a constrained pool
$RunspacePool = [runspacefactory]::CreateRunspacePool(1, 10, $InitialSessionState, $Host)

#Create a constrained worker
$PowerShell = [powershell]::Create($InitialSessionState)
```

DEMO

Constrained Runspaces

Debugging Runspaces

- Runspaces are Isolated threads
- Get-Runspace - Shows the active Runspaces for your process
- Debug-Runspace - Allows for integrations with the runspace
  - Debugger is part of the runspace and will give you TONS of data.
  - Some extensions/environments don't play well with the debugger
- If preserving the PowerShell object, retrieve streams like verbose and errors
- If reusing Runspaces, be aware of shared state conflicts

Debugger Commands

- l, l <s>, l <s> <n>
  - List current line along with a couple before and after for context
  - list starting with line <s>
  - list <n> lines starting with line <s>
- s
  - Step into the next statement to allow debugging line by line
- v
  - Step over the current statement, skipping function calls, but executing them
- o
  - Step out of the current function, returning to the calling function
- c
  - continue until the next breakpoint is hit
- q
  - Quit debugger and stop script execution
- k
  - Get call stack (Get-PSCallStack
- ?
  - Display Help
- d
  - detach from debugger and continue exection

DEMO

Debugging

Q&A