**PowerShell multithreading simplified with concurrent objects - MNPSAG**

My name is Tim Curwick. I am an automation consultant at RBA, and a Microsoft MVP for PowerShell. My blog is MadWithPowerShell.com. You can reach me at [Tim.Curwick@RBAConsulting.com](mailto:Tim.Curwick@RBAConsulting.com) or at [MadWithPowerShell@gmail.com](mailto:MadWithPowerShell@gmail.com). You can follow me on Twitter @MadWPowerShell.

By the way, RBA is hiring. We have hired two people from this group in the last year. We have an immediate need for junior infrastructure engineers. We are also looking to start conversations with senior infrastructure architects, as those positions can open up at any time. If you are interested in hearing more, contact me. If you get the position through this group and through me, the group will get half of the referral bonus.

**Agenda**

Slide

Tonight we are talking about PowerShell multithreading.

Slide

We will briefly go over all of the existing methods we have used or not used.

Slide

We will talk about why to use multithreading.

We will talk about why not to use multithreading.

Slide

We will cover multithreading concepts and terminology, which will overlap with talking about multithreading related .Net objects.

Slide

Then we will walk through sample scripts, staring with a simple, single-threaded script, and adding more and more multithreading capabilities.

Slide

And I will share two useful functions that will make multithreading even easier.

**Background**

PowerShell 1.0 – Single threaded

PowerShell was designed as an easy to use scripting language for automating simple administrative tasks. As part of that vision, it was designed to be single threaded. We write a simple script, and PowerShell runs through it one line at a time. We can have complex flow control logic, but PowerShell is not going to do more than one thing at a time.

PowerShell 2.0 – Background jobs

But we quickly pushed the limits of PowerShell and wanted to do more. In PowerShell 2, they wanted to address this, but were not going to overhaul the architecture of PowerShell, so they gave us Background Jobs.

These jobs are not native to PowerShell. When PowerShell runs something as a job, it leverages a job scheduler internal to the OS, by putting a scriptblock in a queue for “immediate” execution. Then, whenever it gets around to it, the OS spins up a new instance of PowerShell to execute the scriptblock, and then holds the results until you ask for them.

Using PowerShell background jobs can be slow and cumbersome, and it can be complicated getting data into them at the start and out of them at the end, much less anytime in between.

PowerShell 3.0 – PowerShell Workflow

In PowerShell 3, they gave us PowerShell Workflow, with various parallel processing capabilities. But this, too, was not native to PowerShell. When you define a workflow, the PowerShell engine converts it from the PowerShell language to XAML. When you call a workflow, PowerShell puts it in the execution queue of the Windows Workflow Platform, which picks it up and executes it. Some PowerShell does not translate well, and behaves differently in a workflow than in a normal script. Other PowerShell is marked as untranslatable, and Windows Workflow spins up a new PowerShell session just to run that single line, leading to another variation on behavior. The steep learning curve and extra effort of working with workflows makes it almost never worth doing so.

Multiscripting

Using automation platforms such as System Center Orchestrator, or sometimes directly, we could parallelize work by kicking off multiple, independent single-threaded scripts simultaneously. Still limited, still not ideal.

Object oriented programming

When we started working with GUI forms for our scripts, we used events, such as clicking a button, to trigger different chunks of code, but these ran one at a time unless we layered on some type of multithreading.

[PowerShell]

Later we found out that we could directly use some of the .Net objects that PowerShell itself uses to run PowerShell. Doing multithreading that way was faster and provided better control than background jobs, but it was complicated, and cumbersome, and presented some of the same challenges as jobs for communicating with or amongst the threads.

[System.Collections.Concurrent]

But now, new techniques using some amazing new .Net objects make communication between threads easy and powerful, thereby making the direct method of multithreading much more useful and accessible. It’s still a little complicated and cumbersome, but manageable. That’s what we are going to talk about today.

**Why multithread**

Task complexity

As our automation becomes more sophisticated, and we automate more complex processes, it is often desirable to do more than one things at a time.

Improved user experience

Sometimes, that’s so your GUI form doesn’t freeze while waiting for a slow chunk of code to finish running.

Parallel processing

Sometimes, it’s because you have two long running, independent tasks to perform, and you can save time by running them both at the same time.

Queue-based tasks

Sometimes, you have two long-running dependent tasks to perform, and you wish you didn’t have to wait for task A to complete (and fill up all of your memory) before task B starts working the results, but it’s a circumstance where a pipeline is not practical.

Performance enhancement

Sometimes, Microsoft or other vendors have poor API’s that force you to query every VM or every Azure service or every user one by one, and you have dozens or thousands or hundreds of thousands of them to query, and you need to query more than one at a time if you want the script to finish before you retire.

**Limits of multithreading**

Next bottleneck

Multithreading removes a performance bottleneck. But just one. And there is usually another one really close behind that one, limiting the benefits of multithreading.

If your task was already using up all of your CPU or memory or IOPS or bandwidth or whatever, multitasking isn’t going to help at all, and could make it worse.

When it does help, it’s not a magic infinite multiplier. It can only help as much as the available space in your next smallest bottleneck.

When I tried multithreading website testing, performance increased about threefold, and then I maxed out my wifi connection.

You have to manage your threads carefully if memory is your next bottleneck, because at a certain point paging becomes the dominant activity of your memory, CPU, and disk, and then almost nothing gets done.

When I tried massively multithreading active directory calls, something on the network decided I was a denial of service attack, and throttled me down to zero improvement overall.

But sometimes the results are more dramatic. When the bottleneck is Exchange Online, we can use 3 service accounts and 9 threads to improve performance to about 9 times normal.

Complexity

Multithreading adds complexity to your script. It is also difficult to debug code running in another thread. It’s no more difficult that it is for C# developers whose code always runs in another thread, but as PowerShell scripters we are spoiled by the ease of developing in an environment where we get to interact with the main thread of the script in real time.

When the extra development time of a multithreaded script is balanced against the potentially small benefits, sticking with a simple script is often the right decision.

Terms and .Net objects

Thread

A thread is a single process running a scriptblock one line at a time. While our main script running in PowerShell is itself a thread, when we talk about threads in this context, we are usually talking about any additional threads that we are creating and managing.

Each thread is completely independent, and knows nothing about the parent or other threads other than what we explicitly send to it. It shares none of the variables, modules, connections, etc. of the other threads. Any needed variables need to be passed in. Any needed modules need to be loaded in each thread. This adds overhead and limits the advantages of adding additional threads for quick tasks.

Any needed connections need to be reestablished in each thread. There are some exceptions, such as Azure PowerShell connections. Azure PowerShell stores connection tokens in the user profile. As child threads will run as the same user as the parent, they can find and use the tokens for an existing connection.

Run space

A run space is a reserved virtual space within the OS where a thread runs. Unlike most things we create in PowerShell, we want to dispose of any run spaces and threads we create, especially during development and testing. Even after the script we ran in a thread in a run space is complete, as long as the main PowerShell session is still running, the run space and thread persist, along with the reservations within the OS, taking up memory and other resources.

Often not a huge problem, but if you are testing a script over and over, you accumulate more and more threads.

If you accidentally spin up too many threads, performance of the computer will suffer. If you accidentally spin up way too many threads, you will crash Windows. Doing it intentionally for testing purposes, I spun up about 6,000 threads, each doing nothing, before it crashed my work laptop.

Run space pool

A run space pool is a managed collective run space wherein multiple threads can play.

You set a maximum number of threads that can play in the pool at one time. If you are going to be spinning up large numbers of short-lived threads, one for each of many tasks to be performed, you can use this maximum to throttle how many threads run at a time. Above the max, the remaining threads wait patiently for one of the threads in the pool to get out before the lifeguard lets the next one in.

However, in all of the examples today, we are using a different architecture to handle task queueing, and we will set the max to match the number of threads that will run.

[PowerShell]

PowerShell is the somewhat confusing name for the .Net object that is essentially the thread we are creating. Another way to think of it is as a separate instance of the PowerShell engine within which we will define and run the thread. It’s fully qualified name is [System.Management.Automation.PowerShell], but it has a task accelerator, which is like an alias for objects, which is just [PowerShell].

This is the object to which we will assign a scriptblock and parameters, and then tell it to invoke the scriptblock, thereby starting the new thread running.

When the thread is complete, the [PowerShell] object has the contents of all of the streams from the thread; the output stream, error stream, etc. But today we are going to focus on different techniques for getting data from the thread.

Handler

Upon invoking the script in a PowerShell object, we get another object, which we can use to check if the thread is finished, and in conjunction with the PowerShell object, stop and/or get results from the thread.

This object is actually of type [System.Management.Automation.PowerShellAsyncResult], but we are going to call it a handler, because that’s shorter, and consistent with various community discussions about multithreading.

Dictionary

If you are unfamiliar with it, a dictionary is a .Net object class that is very similar to the hashtables we are used to using in PowerShell. It’s a collection of key/value pairs that we use the same way we use hashtables.

$D[‘Status’] = ‘Active’

$D.Add( ‘Status’, ‘Active’ )

$D[‘Status’]

$D.Status

$D.Remove( ‘Status’ )

$D.Keys

$D.Values

You cannot add or addplus a hashtable to a dictionary. If you do, it kind of works, but PowerShell converts the dictionary into a hashtable.

The easiest way to create a dictionary is to cast an empty hashtable as a dictionary. When we do so, we need to specify the types we want to use for the keys and the values.

$Processed = [System.Collections.Generic.Dictionary[String,System.Collections.ArrayList]]@{}

$Status = [System.Collections.Generic.Dictionary[String,String]]@{}

$ThreadStatus = [System.Collections.Generic.Dictionary[Int,String]]@{}

We can make the key and value types very broad, allowing us to put anything we want in them.

$Status = [System.Collections.Generic.Dictionary[Object,Object]]@{}

But the Dictionary works more efficiently if we can be more specific.

[System.Collections.Concurrent.ConcurrentDictionary]

In namespace System.Collections.Concurrent, there are several classes specifically designed for objects to be safely shared between threads. We are going to use two in particular. The first is the ConcurrentDictionary.

The ConcurrentDictionary is simply a dictionary that multiple threads can share. .Net allows only one thread at a time read from or write to the ConcurrentDictionary, preventing any conflicts. When necessary, threads patiently wait their turn.

This makes it extremely easy to pass data in and out of and between threads.

We define a ConcurrentDictionary the same way we do a Dictionary, by casting an empty hashtable.

$Processed = [System.Collections.Concurrent.ConcurrentDictionary[String,System.Collections.ArrayList]]@{}

$Status = [System.Collections.Concurrent.ConcurrentDictionary[String,String]]@{}

$ThreadStatus = [System.Collections.Concurrent.ConcurrentDictionary[Int,String]]@{}

With ConcurrentDictionaries, it is important to be specific about the key and value types. Unlike regular Dictionaries, ConcurrentDictionaries won’t let you use generic types. Dictionaries understand that a [String] is a type of [Object], and let’s you use a string where you defined an [Object], but ConcurrentDictionaries do not.

[System.Collections.Queue]

If you are unfamiliar with it, a Queue is a specialized collection, designed for adding things to it, and taking things from it in the same order they went in.

$Q = [System.Collections.Queue]@()

$Q.Enque( ‘First’ )

$Q.Enque ( ‘Second’ )

$Q.Enque ( ‘Third’ )

$Q.Deque() # Give us ‘First’ and removes it from $Q

[System.Collections.Concurrent.BlockingCollection]

The BlockingCollection is the magical object that makes multithreading really easy.

It is similar in concept to, but used differently from, a Queue. Plus it has special magic.

Despite being more like a specialized array than a hashtable, we create a BlockingCollection by casting an empty hashtable. Like the ConcurrentDictionary, we need to define the type of objects that will be put in the BlockingCollection.

$UserQueue = [System.Collections.Concurrent.BlockingCollection[String]]@{}

$OutputQueue = [System.Collections.Concurrent.BlockingCollection[PSObject]]@{}

We add objects with the .Add() method.

$UserQueue.Add( ‘Tim.Curwick@RBAConsulting.com’ )

$UserQueue.Add( ‘Joe.Artz@Target.com’ )

We can get objects out with the .Take() method.

$UserQueue.Take() # Gives us ‘Tim.Curwick@RBAConsulting.com’ and removes it from the collection

But we generally won’t do it that way. We are simply going to do this.

ForEach ( $User in $UserQueue.GetConsumingEnumerable() )

{

$ADUser = Get-ADUser -Filter { UserPrincipalName -eq $User }

If ( $ADUser.c = ‘US’ )

{

$Result = $ADUser | Select-Object -Property DistinguishedName, SID

$ResultQueue.Add( $Result )

}

}

Behind the scenes, ForEach uses an enumerator of a collection to loop through it. This magic method creates a cool consuming enumerator for ForEach to use.

When this loop runs, if there is anything in BlockingCollection $UserQueue, the first user is assigned to variable $User and is removed from the Queue. If there are multiple threads watching the same $UserQueue BlockingCollection, each user only goes to a single thread.

If the user queue is empty, the for ForEach sits and waits. If a different thread adds a new user to the $UserQueue, the ForEach takes it out of the BlockingCollection and processes it.

When the process that is adding users to the queue is finished, it closes the queue by running:

$UserQueue.CompleteAdding()

This sets property $UserQueue.IsAddingComplete to $True.

If adding is complete, once all users have been processed and the BlockingCollection is empty, ForEach recognizes that it is done, and the script continues after the loop.

**Example**

Slides

**Starting a thread**

How do we run a new thread?

DEMO

Any objects that are going to be passed to the thread need to be defined before we can pass them. This includes any as yet empty shared objects.

$UserQueue = [System.Collections.Concurrent.BlockingCollection[String]]@{}

$OutputQueue = [System.Collections.Concurrent.BlockingCollection[PSObject]]@{}

Any parameters are passed as a hashtable. We can create that now. The variable names used inside and outside the thread don’t have to match, but it’s much easier to work with when they do.

$ThreadParameters = @{

UserQueue = $UserQueue

OutputQueue = $OutputQueue

SomeOtherVariable = $SomeOtherVariable }

Then we define the script that will run in the thread. In the parameter block, the parameter types must match the types of the object we are sending in. The parameter names must match the keys defined in the parameter hashtable we are sending in.

$ThreadScript = {

Param (

[System.Collections.Concurrent.BlockingCollection[String]]$UserQueue

[System.Collections.Concurrent.BlockingCollection[PSObject]]$OutputQueue

[String]$SomeOtherVariable

<# Thread code here #>

}

We need a run space or a run space pool for the thread to play in. We can’t just let them run wild.

Because the .Net developers didn’t think to create a constructor for run space pools, we use a static method on a different object, the [RunspacePoolFactory] to create a run space pool. We specify the minimum and maximum number of threads the pool can allow to run at one time. And we open the pool for playtime.

# Create runspace pool

$RunspacePool = [runspacefactory]::CreateRunspacePool( 1, 7 )

$RunspacePool.Open()

Then we create the thread, and tell it which pool to play in.

# Create thread

$PowerShell = [PowerShell]::Create()

$PowerShell.RunspacePool = $RunspacePool

We add the script and the parameters we defined to the thread.

# Add script

[void]$PowerShell.AddScript( $ThreadScript )

# Add parameters

[void]$PowerShell.AddParameters( $ThreadParameters )

And finally, we start the thread. This returns the handler.

# Start thread

$Handler = $PowerShell.BeginInvoke()

We will need the PowerShell thread object and the handler object later, so we save those in an object.

$RunningThread = [PSCustomObject]@{

PowerShell = $PowerShell

Handler = $Handler }

**Invoke-Thread**

It’s pretty straightforward, but we will want to do that multiple times in most multithreading scripts, so I created a function to help with that.

The function takes a run space pool, a scriptblock and a parameter hashtable, starts the thread, and returns the custom object with [PowerShell] thread object and the handler object.

I also added functionality so that if the names of the parameters match the names of the variables to be passed to those parameters, you don’t have to build and include a parameter hashtable. Simply use the -UseEmbeddedParameters switch, and it parses the script to get the names of the parameters, and uses the values of any matching variables found to build the parameter hashtable for you.

This would fail if any of the parameter names matched the parameter names of this function, so I appended Unique to each of the functions parameters to mitigate the risk of such a conflict.

**Example architecture**

So let’s do some multithreading.

When getting information on Azure VMs using Azure PowerShell, you can’t get all of the information on all of the VMs with a single query, due to poor API design. You have to make a query to get a list of all of the VMs. Then, you have to make a separate query for each of the VMs. Then you have to make another query for each of the VMs with a different switch to get additional information. And the API is slow, so this takes a while.

Here is what that looks like without multithreading. Greatly simplified