

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

This is the fourth post in my Pushing the Limits of Windows series that explores the boundaries of fundamental resources in Windows. This time, I'm going to discuss the limits on the maximum number of threads and processes supported on Windows. I'll briefly describe the difference between a thread and a process, survey thread limits and then investigate process limits. I cover thread limits first since every active process has at least one thread (a process that's terminated, but is kept referenced by a handle owned by another process won't have any), so the limit on processes is directly affected by the caps that limit threads.

Unlike some UNIX variants, most resources in Windows have no fixed upper bound compiled into the operating system, but rather derive their limits based on basic operating system resources that I've already covered. Process and threads, for example, require physical memory, virtual memory, and pool memory, so the number of processes or threads that can be created on a given Windows system is ultimately determined by one of these resources, depending on the way that the processes or threads are created and which constraint is hit first. I therefore recommend that you read the preceding posts if you haven't, because I'll be referring to reserved memory, committed memory, the system commit limit and other concepts I've covered:

- Pushing the Limits of Windows: Physical Memory
- Pushing the Limits of Windows: Virtual Memory
- Pushing the Limits of Windows: Paged and Nonpaged Pool

Processes and Threads

A Windows process is essentially container that hosts the execution of an executable image file. It is represented with a kernel process object and Windows uses the process object and its associated data structures to store and track information about the image's execution. For example, a process has a virtual address space that holds the process's private and shared data and into which the executable image and its associated DLLs are mapped. Windows records the process's use of resources for accounting and query by diagnostic tools and it registers the process's references to operating system objects in the process's handle table. Processes operate with a security context, called a token, that identifies the user account, account groups, and privileges assigned to the process.

Finally, a process includes one or more threads that actually execute the code in the process (technically, processes don't run, threads do) and that are represented with kernel thread objects. There are several reasons applications create threads in addition to their default initial thread: processes with a user interface typically create threads to execute work so that the main thread remains responsive to user input and windowing commands; applications that want to take advantage of multiple processors for scalability or that want to continue executing while threads are tied up waiting for synchronous I/O operations to complete also benefit from multiple threads.

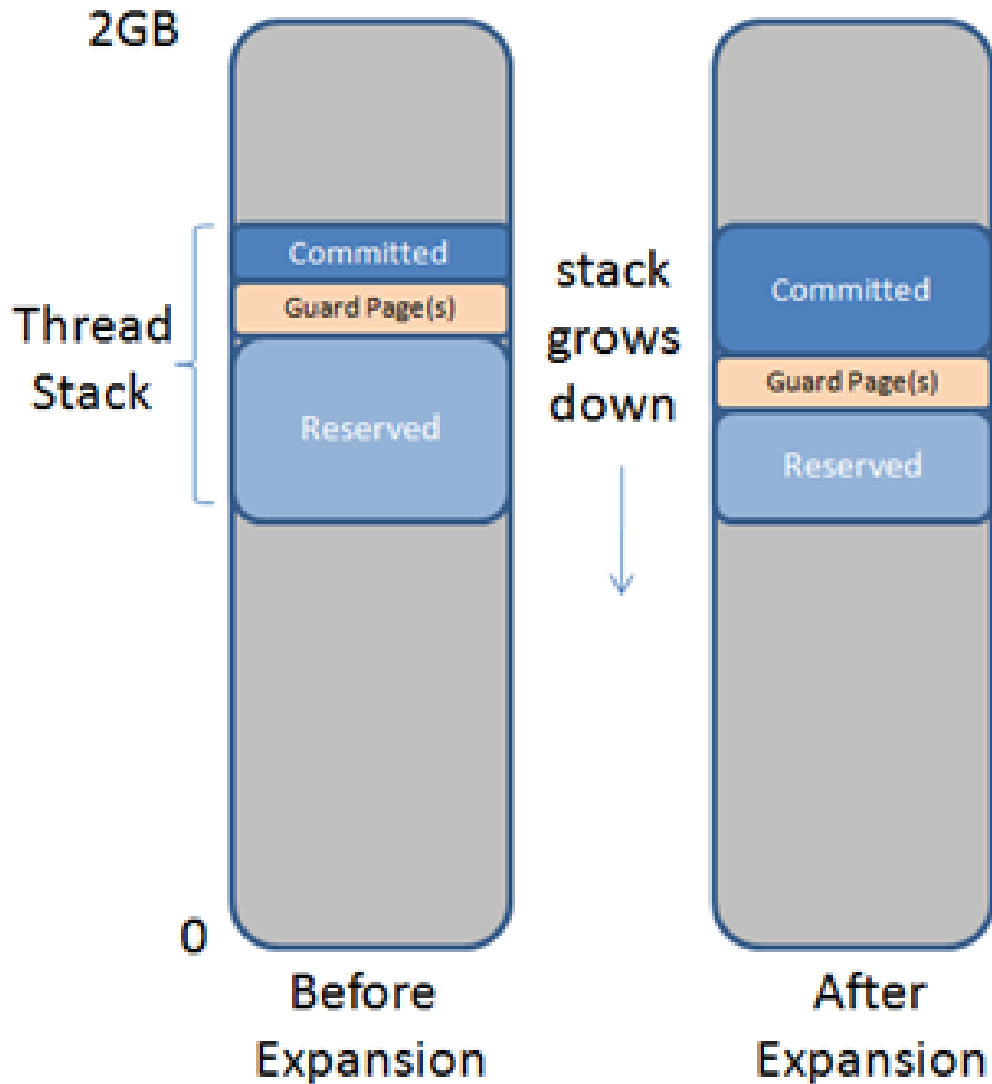
Thread Limits

Besides basic information about a thread, including its CPU register state, scheduling priority, and resource usage accounting, every thread has a portion of the process address space assigned to it, called a stack, which the thread can use as scratch storage as it executes program code to pass function parameters, maintain local variables, and save function return addresses. So that the system's virtual memory isn't unnecessarily wasted, only part of the stack is initially allocated, or committed and the rest is simply reserved. Because stacks grow downward in memory, the system places guard pages beyond the committed part of the stack that trigger an automatic commitment of additional memory (called a stack expansion) when accessed. This figure shows how a stack's

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

committed region grows down and the guard page moves when the stack expands, with a 32-bit address space as an example (not drawn to scale):



The Portable Executable (PE) structures of the executable image specify the amount of address space reserved and initially committed for a thread's stack. The linker defaults to a reserve of 1MB and commit of one page (4K), but developers can override these values either by changing the PE values when they link their program or for an individual thread in a call to `CreateThread`. You can use a tool like `Dumpbin` that comes with Visual Studio to look at the settings for an executable. Here's the `Dumpbin` output with the `/headers` option for the executable generated by a new Visual Studio project:

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

```
0 Win32 version
2DE000 size of image
  400 size of headers
2DFE62 checksum
  2 subsystem (Windows GUI)
  8140 DLL characteristics
      Dynamic base
      NX compatible
      Terminal Server Aware
100000 size of stack reserve
  1000 size of stack commit
100000 size of heap reserve
  1000 size of heap commit
  0 loader flags
  10 number of directories
  0 [ 0] RVA [size] of Export Directory
```

Converting the numbers from hexadecimal, you can see the stack reserve size is 1MB and the initial commit is 4K and using the new Sysinternals VMMap tool to attach to this process and view its address space, you can clearly see a thread stack's initial committed page, a guard page, and the rest of the reserved stack memory:

Address	Type	Size	Committed	Total WS	Private...	Blocks	Protection	Details
00AF0000	Thread Stack	1,024 K	8 K	4 K	4 K	3	Read/Write	Thread ID: 1276
00AF0000	Reserved	1,016 K						
00BEE000	Private	4 K	4 K				Read/Write/Guard	
00BEF000	Private	4 K	4 K	4 K	4 K		Read/Write	

Because each thread consumes part of a process's address space, processes have a basic limit on the number of threads they can create that's imposed by the size of their address space divided by the thread stack size.

32-bit Thread Limits

Even if the process had no code or data and the entire address space could be used for stacks, a 32-bit process with the default 2GB address space could create at most 2,048 threads. Here's the output of the Testlimit tool running on 32-bit Windows with the `-t` switch (create threads) confirming that limit:

```
C:\Temp>testlimit -t

Testlimit v5.01 - tests Windows limits
By Mark Russinovich

Creating threads...
Created 2025 threads. Lasterror: 8
Not enough storage is available to process this command.
```

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

Again, since part of the address space was already used by the code and initial heap, not all of the 2GB was available for thread stacks, thus the total threads created could not quite reach the theoretical limit of 2,048.

I linked the Testlimit executable with the large address space-aware option, meaning that if it's presented with more than 2GB of address space (for example on 32-bit systems booted with the /3GB or /USERVA Boot.ini option or its equivalent BCD option on Vista and later increaseuserva), it will use it. 32-bit processes are given 4GB of address space when they run on 64-bit Windows, so how many threads can the 32-bit Testlimit create when run on 64-bit Windows? Based on what we've covered so far, the answer should be roughly 4096 (4GB divided by 1MB), but the number is actually significantly smaller. Here's 32-bit Testlimit running on 64-bit Windows XP:

```
C:\temp>testlimit -t
Testlimit v5.01 - tests Windows limits
By Mark Russinovich
Creating threads...
Created 3204 threads. Lasterror: 8
Not enough storage is available to process this command.
```

The reason for the discrepancy comes from the fact that when you run a 32-bit application on 64-bit Windows, it is actually a 64-bit process that executes 64-bit code on behalf of the 32-bit threads, and therefore there is a 64-bit thread stack and a 32-bit thread stack area reserved for each thread. The 64-bit stack has a reserve of 256K (except that on systems prior to Vista, the initial thread's 64-bit stack is 1MB). Because every 32-bit thread begins its life in 64-bit mode and the stack space it uses when starting exceeds a page, you'll typically see at least 16KB of the 64-bit stack committed. Here's an example of a 32-bit thread's 64-bit and 32-bit stacks (the one labeled "Wow64" is the 32-bit stack):

Address	Type	Size	Committed	Total WS	Private WS	Blocks	Protection	Details
00250000	Thread Stack	256 K	28 K	16 K	16 K	3	Read/Write	Thread ID: 3268
00250000	Reserved	228 K						
00289000	Private	12 K	12 K				Read/Write/Guard	
0028C000	Private	16 K	16 K	16 K	16 K		Read/Write	
02160000	Thread Stack (Wow64)	1,024 K	12 K	4 K	4 K	3	Read/Write	Thread ID: 3268
02160000	Reserved	1,012 K						
0225D000	Private	8 K	8 K				Read/Write/Guard	
0225F000	Private	4 K	4 K	4 K	4 K		Read/Write	

32-bit Testlimit was able to create 3,204 threads on 64-bit Windows, which given that each thread uses 1MB+256K of address space for stack (again, except the first on versions of Windows prior to Vista, which uses 1MB+1MB), is exactly what you'd expect. I got different results when I ran 32-bit Testlimit on 64-bit Windows 7, however:

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

```
C:\Talks\Tools\Testlimit\Release>testlimit -t
Testlimit v5.01 - tests windows limits
By Mark Russinovich
Creating threads...
Created 2925 threads. Lasterror: 8
Not enough storage is available to process this command.
```

The difference between the Windows XP result and the Windows 7 result is caused by the more random nature of address space layout introduced in Windows Vista, Address Space Layout Randomization (ASLR), that leads to some fragmentation. Randomization of DLL loading, thread stack and heap placement, helps defend against malware code injection. As you can see from this VMMMap output, there's 357MB of address space still available, but the largest free block is only 128K in size, which is smaller than the 1MB required for a 32-bit stack:

Stack	3,744,512 K	
System	41,028 K	
Free	357,628 K	
Address	Type	Size
F8310000	Free	128 K
F8780000	Free	128 K
F8E40000	Free	128 K
F8F60000	Free	128 K

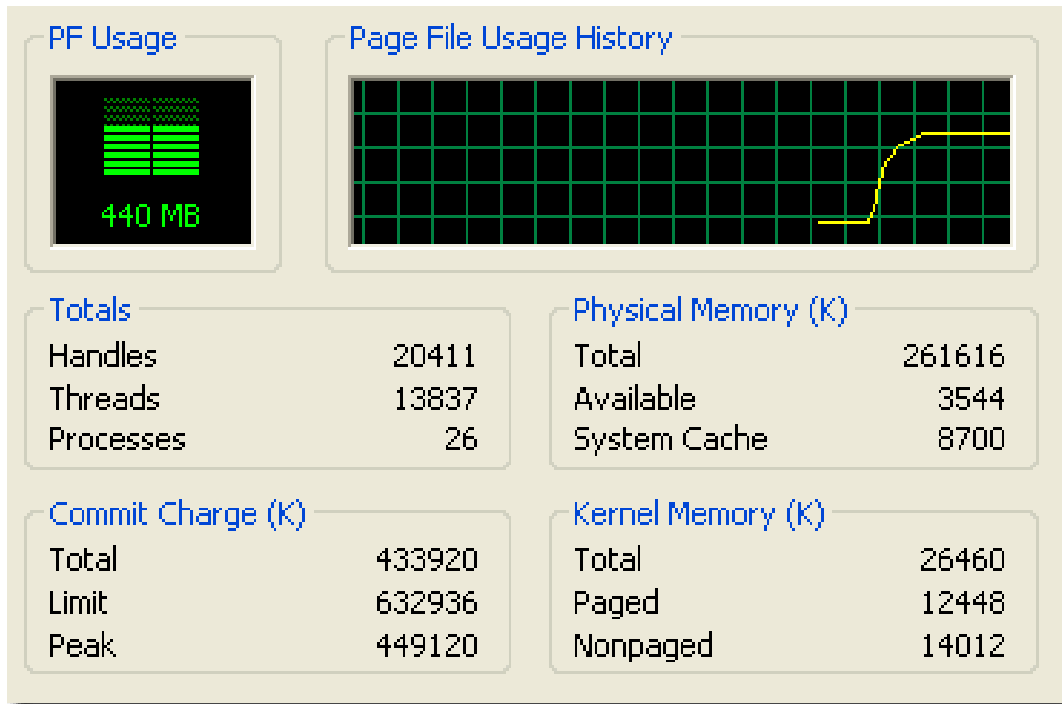
As I mentioned, a developer can override the default stack reserve. One reason to do so is to avoid wasting address space when a thread's stack usage will always be significantly less than the default 1MB. Testlimit sets the default stack reservation in its PE image to 64K and when you include the `-n` switch along with the `-t` switch, Testlimit creates threads with 64K stacks. Here's the output on a 32-bit Windows XP system with 256MB RAM (I did this experiment on a small system to highlight this particular limit):

```
C:\Temp>Testlimit.exe -t -n
Testlimit v5.01 - tests windows limits
By Mark Russinovich
Creating threads with min stack reserve...
Created 14225 threads. Lasterror: 1450
Insufficient system resources exist to complete the requested service.
```

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

Note the different error, which implies that address space isn't the issue here. In fact, 64K stacks should allow for around 32,000 threads (2GB/64K = 32,768). What's the limit that's being hit in this case? A look at the likely candidates, including commit and pool, don't give any clues, as they're all below their limits:



It's only a look at additional memory information in the kernel debugger that reveals the threshold that's being hit, resident available memory, which has been exhausted:

```
kd> !vm

*** Virtual Memory Usage ***
Physical Memory:          65404 (    261616 Kb)
Page File: \??\C:\pagefile.sys
  Current:    393216 Kb  Free Space:    300672 Kb
  Minimum:    393216 Kb  Maximum:    786432 Kb
Available Pages:          3742 (    14968 Kb)
ResAvail Pages:           71 (     284 Kb)
***** Running out of physical memory *****

Locked IO Pages:         59 (     236 Kb)
Free System PTEs:       213328 (   853312 Kb)
```

Resident available memory is the physical memory that can be assigned to data or code that must be kept in RAM. Nonpaged pool and nonpaged drivers count against it, for example, as does memory

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

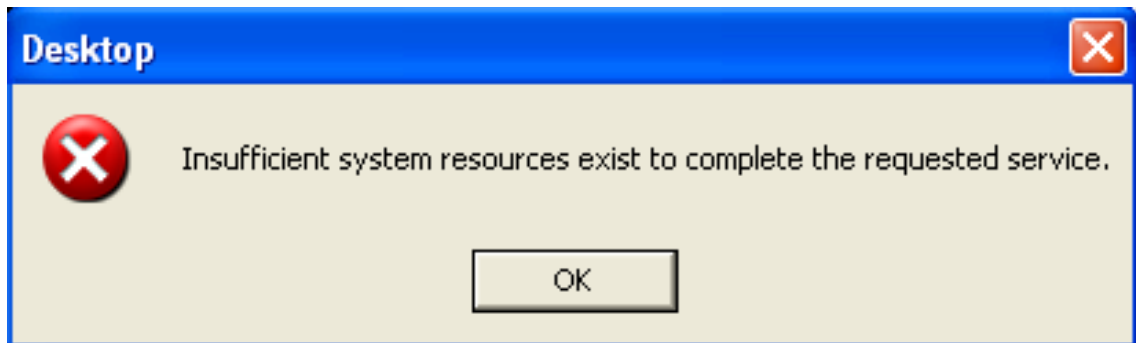
that's locked in RAM for device I/O operations. Every thread has both a user-mode stack, which is what I've been talking about, but they also have a kernel-mode stack that's used when they run in kernel mode, for example while executing system calls. When a thread is active its kernel stack is locked in memory so that the thread can execute code in the kernel that can't page fault.

A basic kernel stack is 12K on 32-bit Windows and 24K on 64-bit Windows. 14,225 threads require about 170MB of resident available memory, which corresponds to exactly how much is free on this system when Testlimit isn't running:

```
kd> !vm

*** Virtual Memory Usage ***
Physical Memory:          65404 (    261616 Kb)
Page File: \??\C:\pagefile.sys
  Current:      393216 Kb  Free Space:    343788 Kb
  Minimum:     393216 Kb  Maximum:    786432 Kb
Available Pages:         51489 (   205956 Kb)
ResAvail Pages:         42748 (   170992 Kb)
Locked IO Pages:         59 (     236 Kb)
Free System PTEs:       269800 (  1079200 Kb)
```

Once the resident available memory limit is hit, many basic operations begin failing. For example, here's the error I got when I double-clicked on the desktop's Internet Explorer shortcut:



As expected, when run on 64-bit Windows with 256MB of RAM, Testlimit is only able to create 6,600 threads – roughly half what it created on 32-bit Windows with 256MB RAM - before running out of resident available memory:

```
C:\>testlimit -t -n

Testlimit v5.01 - tests windows limits
By Mark Russinovich

Creating threads with min stack reserve...
Created 6662 threads. Lasterror: 1450
Insufficient system resources exist to complete the requested service.
```

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

The reason I said “basic” kernel stack earlier is that a thread that executes graphics or windowing functions gets a “large” stack when it executes the first call that’s 20K on 32-bit Windows and 48K on 64-bit Windows. Testlimit’s threads don’t call any such APIs, so they have basic kernel stacks.

64-bit Thread Limits

Like 32-bit threads, 64-bit threads also have a default of 1MB reserved for stack, but 64-bit processes have a much larger user-mode address space (8TB), so address space shouldn’t be an issue when it comes to creating large numbers of threads. Resident available memory is obviously still a potential limiter, though. The 64-bit version of Testlimit (Testlimit64.exe) was able to create around 6,600 threads with and without the `-n` switch on the 256MB 64-bit Windows XP system, the same number that the 32-bit version created, because it also hit the resident available memory limit. However, on a system with 2GB of RAM, Testlimit64 was able to create only 55,000 threads, far below the number it should have been able to if resident available memory was the limiter ($2\text{GB}/24\text{K} = 89,000$):

```
C:\temp>testlimit64.exe -t
Testlimit v5.01 - tests Windows limits
By Mark Russinovich
Creating threads...
Created 54965 threads. Lasterror: 1455
The paging file is too small for this operation to complete.
```

In this case, it’s the initial thread stack commit that causes the system to run out of virtual memory and the “paging file is too small” error. Once the commit level reached the size of RAM, the rate of thread creation slowed to a crawl because the system started thrashing, paging out stacks of threads created earlier to make room for the stacks of new threads, and the paging file had to expand. The results are the same when the `-n` switch is specified, because the threads have the same initial stack commitment.

Process Limits

The number of processes that Windows supports obviously must be less than the number of threads, since each process has one thread and a process itself causes additional resource usage. 32-bit Testlimit running on a 2GB 64-bit Windows XP system created about 8,400 processes:

```
C:\temp>testlimit -p
Testlimit v5.0 - tests Windows limits
By Mark Russinovich
Creating processes...
Created 8396 processes. Lasterror: 8
Not enough storage is available to process this command.
```

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

A look in the kernel debugger shows that it hit the resident available memory limit:

```
Available Pages:      101067 (    404268 Kb)
ResAvail Pages:      106 (        424 Kb)
***** Running out of physical memory *****
Locked IO Pages:      0 (          0 Kb)
Free System PTEs:    33478182 ( 133912728 Kb)
```

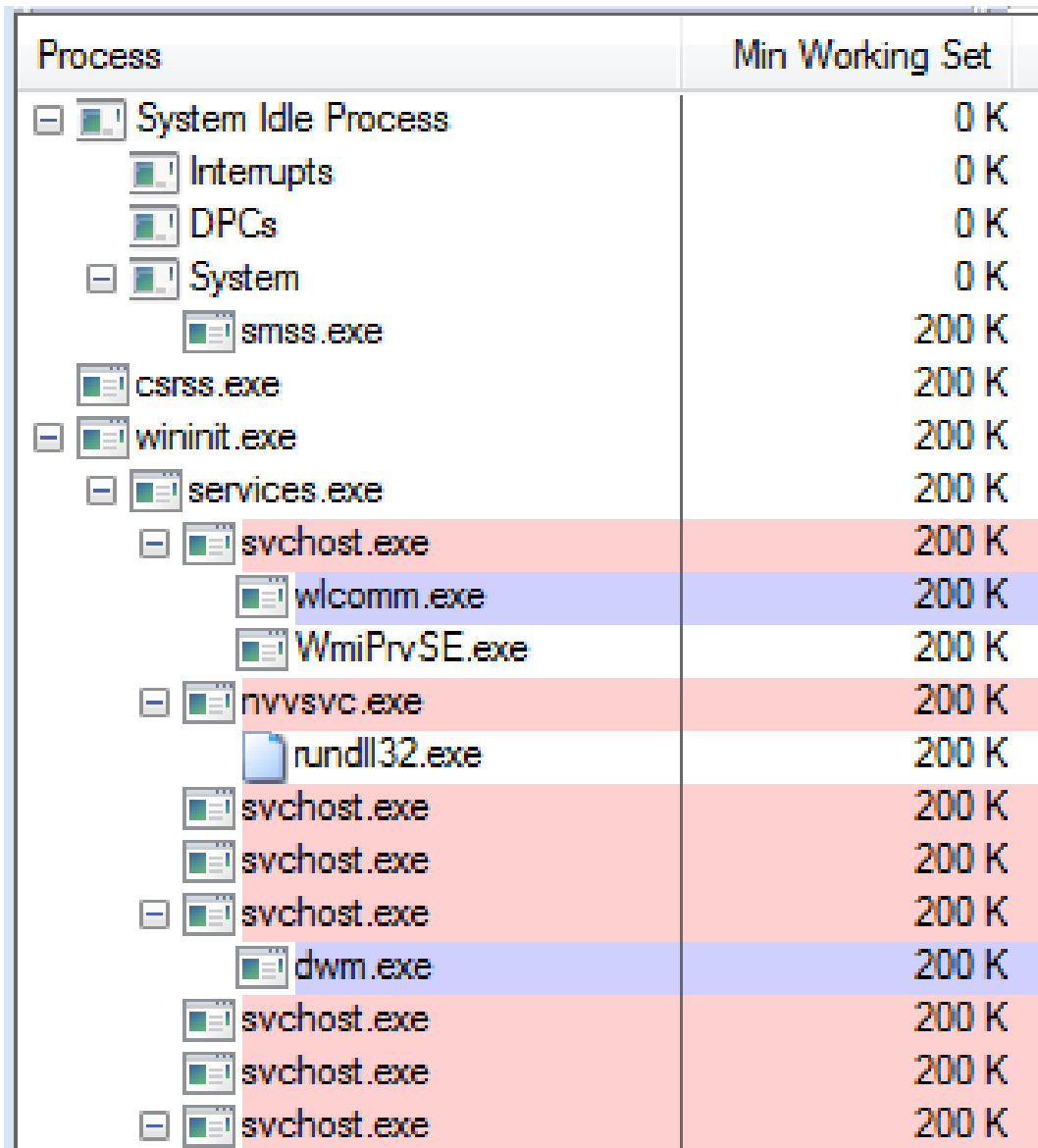
If the only cost of a process with respect to resident available memory was the kernel-mode thread stack, Testlimit would have been able to create far more than 8,400 threads on a 2GB system. The amount of resident available memory on this system when Testlimit isn't running is 1.9GB:

```
Available Pages:      412035 (   1648140 Kb)
ResAvail Pages:      483382 (   1933528 Kb)
Locked IO Pages:      0 (          0 Kb)
```

Dividing the amount of resident memory Testlimit used (1.9GB) by the number of processes it created (8,400) yields 230K of resident memory per process. Since a 64-bit kernel stack is 24K, that leaves about 206K unaccounted for. Where's the rest of the cost coming from? When a process is created, Windows reserves enough physical memory to accommodate the process's minimum working set size. This acts as a guarantee to the process that no matter what, there will be enough physical memory available to hold enough data to satisfy its minimum working set. The default working set size happens to be 200KB, a fact that's evident when you add the Minimum Working Set column to Process Explorer's display:

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)



Process	Min Working Set
System Idle Process	0 K
Interrupts	0 K
DPCs	0 K
System	0 K
smss.exe	200 K
csrss.exe	200 K
wininit.exe	200 K
services.exe	200 K
svchost.exe	200 K
wlcomm.exe	200 K
WmiPrvSE.exe	200 K
nvvsvc.exe	200 K
rundll32.exe	200 K
svchost.exe	200 K
svchost.exe	200 K
svchost.exe	200 K
dwm.exe	200 K
svchost.exe	200 K
svchost.exe	200 K
svchost.exe	200 K

The remaining roughly 6K is resident available memory charged for additional non-pageable memory allocated to represent a process. A process on 32-bit Windows will use slightly less resident memory because its kernel-mode thread stack is smaller.

As they can for user-mode thread stacks, processes can override their default working set size with the `SetProcessWorkingSetSize` function. `Testlimit` supports a `-n` switch, that when combined with `-p`, causes child processes of the main `Testlimit` process to set their working set to the minimum possible, which is 80K. Because the child processes must run to shrink their working sets, `Testlimit` sleeps after it can't create any more processes and then tries again to give its children a chance to execute. `Testlimit` executed with the `-n` switch on a Windows 7 system with 4GB of RAM hit a limit other than resident available memory: the system commit limit:

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

```
C:\Talks\Tools\Testlimit\Release>testlimit -p -n
Testlimit v5.01 - tests Windows limits
By Mark Russinovich

Creating processes with min working set...
Created 4047 processes. Lasterror: 1450
Insufficient system resources exist to complete the requested service.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 5701 processes. Lasterror: 1450

runtime error R6028
- unable to initialize heap
Insufficient system resources exist to complete the requested service.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 6231 processes. Lasterror: 1455
The paging file is too small for this operation to complete.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 6379 processes. Lasterror: 1455
The paging file is too small for this operation to complete.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 6497 processes. Lasterror: 1455
The paging file is too small for this operation to complete.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 6560 processes. Lasterror: 1450
Insufficient system resources exist to complete the requested service.
Sleeping for 5 seconds to allow for processes to shrink working sets...
Created 6560 processes. Lasterror: 1455
The paging file is too small for this operation to complete.
```

Here you can see the kernel debugger reporting not only that the system commit limit had been hit, but that there have been thousands of memory allocation failures, both virtual and paged pool allocations, following the exhaustion of the commit limit (the system commit limit was actually hit several times as the paging file was filled and then grown to raise the limit):

```
PagedPool Usage:          61328 (    245312 Kb)
PagedPool Maximum:      33554432 ( 134217728 Kb)

***** 8135 pool allocations have failed *****

Session Commit:          25294 (    101176 Kb)
Shared Commit:           221965 (    887860 Kb)
Special Pool:             0 (          0 Kb)
Shared Process:          194324 (    777296 Kb)
PagedPool Commit:        61392 (    245568 Kb)
Driver Commit:           9347 (     37388 Kb)

Committed pages:        2153741 (   8614964 Kb)
Commit limit:           2394660 (   9578640 Kb)

***** 26710 commit requests have failed *****
```

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

The baseline commitment before Testlimit ran was about 1.5GB, so the threads had consumed about 8GB of committed memory. Each process therefore consumed roughly 8GB/6,600, or 1.2MB. The output of the kernel debugger's !vm command, which shows the private memory allocated by each active process, confirms that calculation:

```
Total Private:          2194912 (      8779648 Kb)
 0328 svchost.exe        35205 (      140820 Kb)
 13e00 WindowsLiveWri   18088 (       72352 Kb)
 0b4c svchost.exe        17800 (       71200 Kb)
 0608 SearchIndexer.    12766 (       51064 Kb)
 0fec iexplore.exe      10472 (       41888 Kb)
 1c55c taskmgr.exe        9495 (       37980 Kb)
 0620 dwm.exe            7851 (       31404 Kb)
 4b98 WINWORD.EXE       7245 (       28980 Kb)
 0530 svchost.exe        7154 (       28616 Kb)
 . . .
 1c368 testlimit64.exe   278 (        1112 Kb)
 1c328 testlimit64.exe   278 (        1112 Kb)
 1c320 testlimit64.exe   278 (        1112 Kb)
 1c31c testlimit64.exe   278 (        1112 Kb)
 1c314 testlimit64.exe   278 (        1112 Kb)
 1c2ec testlimit64.exe   278 (        1112 Kb)
 1c294 testlimit64.exe   278 (        1112 Kb)
 1c250 testlimit64.exe   278 (        1112 Kb)
 1c24c testlimit64.exe   278 (        1112 Kb)
 1c224 testlimit64.exe   278 (        1112 Kb)
 1c220 testlimit64.exe   278 (        1112 Kb)
 1c208 testlimit64.exe   278 (        1112 Kb)
 1c160 testlimit64.exe   278 (        1112 Kb)
 1bf70 testlimit64.exe   278 (        1112 Kb)
 11100 testlimit64.exe   278 (        1112 Kb)
```

The initial thread stack commitment, described earlier, has a negligible impact with the rest coming from the memory required for the process address space data structures, page table entries, the handle table, process and thread objects, and private data the process creates when it initializes.

How Many Threads and Process is Enough?

So the answer to the questions, “how many threads does Windows support?” and “how many processes can you run concurrently on Windows?” depends. In addition to the nuances of the way that the threads specify their stack sizes and processes specify their minimum working sets, the two major factors that determine the answer on any particular system include the amount of physical memory and the system commit limit. In any case, applications that create enough threads or processes to get anywhere near these limits should rethink their design, as there are almost always alternate ways to accomplish the same goals with a reasonable number. For instance, the general

Pushing the Limits of Windows: Processes and Threads

Mark Russinovich
(From Mark Russinovich Blog)

goal for a scalable application is to keep the number of threads running equal to the number of CPUs (with NUMA changing this to consider CPUs per node) and one way to achieve that is to switch from using synchronous I/O to using asynchronous I/O and rely on I/O completion ports to help match the number of running threads to the number of CPUs.