

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

In previous Pushing the Limits posts, I described the two most basic system resources, physical memory and virtual memory. This time I'm going to describe two fundamental kernel resources, paged pool and nonpaged pool, that are based on those, and that are directly responsible for many other system resource limits including the maximum number of processes, synchronization objects, and handles.

Paged and nonpaged pools serve as the memory resources that the operating system and device drivers use to store their data structures. The pool manager operates in kernel mode, using regions of the system's virtual address space (described in the Pushing the Limits post on virtual memory) for the memory it sub-allocates. The kernel's pool manager operates similarly to the C-runtime and Windows heap managers that execute within user-mode processes. Because the minimum virtual memory allocation size is a multiple of the system page size (4KB on x86 and x64), these subsidiary memory managers carve up larger allocations into smaller ones so that memory isn't wasted.

For example, if an application wants a 512-byte buffer to store some data, a heap manager takes one of the regions it has allocated and notes that the first 512-bytes are in use, returning a pointer to that memory and putting the remaining memory on a list it uses to track free heap regions. The heap manager satisfies subsequent allocations using memory from the free region, which begins just past the 512-byte region that is allocated.

Nonpaged Pool

The kernel and device drivers use nonpaged pool to store data that might be accessed when the system can't handle page faults. The kernel enters such a state when it executes interrupt service routines (ISRs) and deferred procedure calls (DPCs), which are functions related to hardware interrupts. Page faults are also illegal when the kernel or a device driver acquires a spin lock, which, because they are the only type of lock that can be used within ISRs and DPCs, must be used to protect data structures that are accessed from within ISRs or DPCs and either other ISRs or DPCs or code executing on kernel threads. Failure by a driver to honor these rules results in the most common crash code, IRQL_NOT_LESS_OR_EQUAL.

Nonpaged pool is therefore always kept present in physical memory and nonpaged pool virtual memory is assigned physical memory. Common system data structures stored in nonpaged pool include the kernel and objects that represent processes and threads, synchronization objects like mutexes, semaphores and events, references to files, which are represented as file objects, and I/O request packets (IRPs), which represent I/O operations.

Paged Pool

Paged pool, on the other hand, gets its name from the fact that Windows can write the data it stores to the paging file, allowing the physical memory it occupies to be repurposed. Just as for user-mode virtual memory, when a driver or the system references paged pool memory that's in the paging file, an operation called a page fault occurs, and the memory manager reads the data back into physical memory. The largest consumer of paged pool, at least on Windows Vista and later, is typically the Registry, since references to registry keys and other registry data structures are stored in paged pool. The data structures that represent memory mapped files, called sections internally, are also stored in paged pool.

Device drivers use the ExAllocatePoolWithTag API to allocate nonpaged and paged pool, specifying the type of pool desired as one of the parameters. Another parameter is a 4-byte Tag, which drivers

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

are supposed to use to uniquely identify the memory they allocate, and that can be a useful key for tracking down drivers that leak pool, as I'll show later.

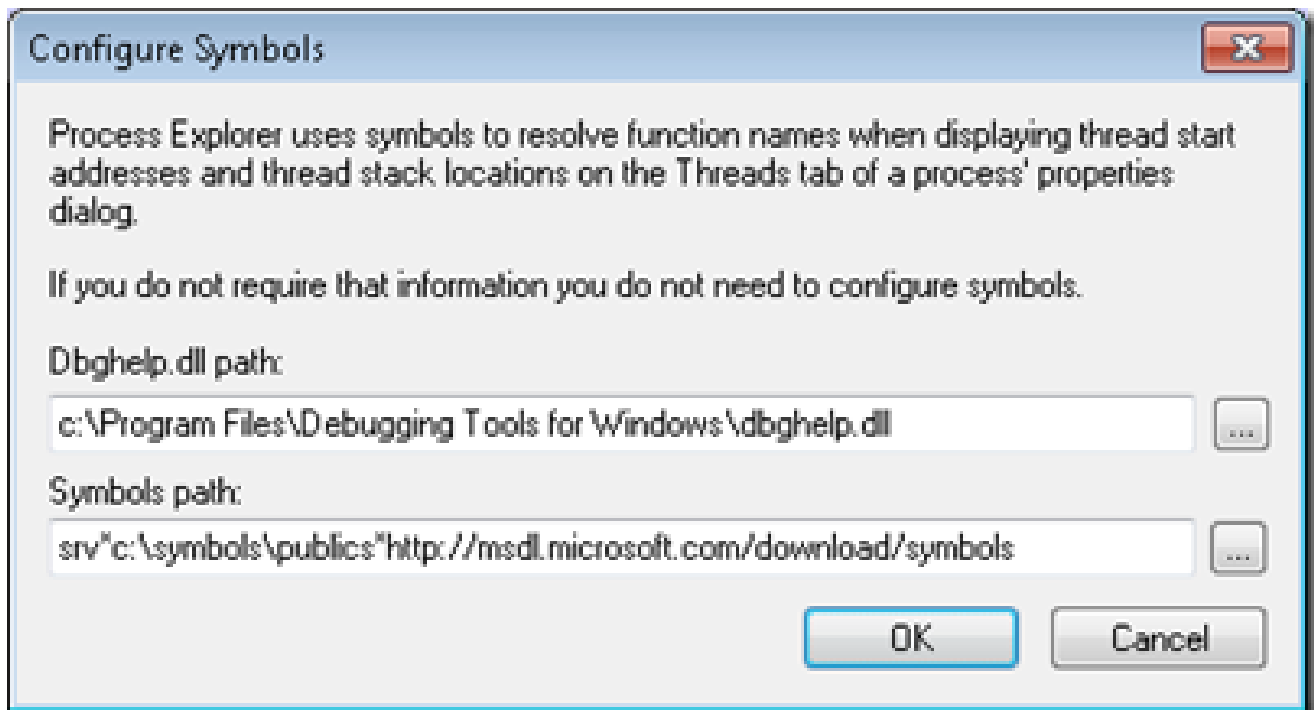
Viewing Paged and Nonpaged Pool Usage

There are three performance counters that indicate pool usage:

- Pool nonpaged bytes
- Pool paged bytes (virtual size of paged pool – some may be paged out)
- Pool paged resident bytes (physical size of paged pool)

However, there are no performance counters for the maximum size of these pools. They can be viewed with the kernel debugger !vm command, but with Windows Vista and later to use the kernel debugger in local kernel debugging mode you must boot the system in debugging mode, which disables MPEG2 playback.

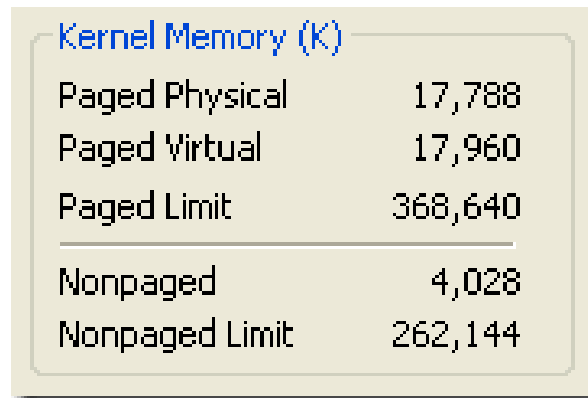
So instead, use Process Explorer to view both the currently allocated pool sizes, as well as the maximum. To see the maximum, you'll need to configure Process Explorer to use symbol files for the operating system. First, install the latest Debugging Tools for Windows package. Then run Process Explorer and open the Symbol Configuration dialog in the Options menu and point it at the dbghelp.dll in the Debugging Tools for Windows installation directory and set the symbol path to point at Microsoft's symbol server:



After you've configured symbols, open the System Information dialog (click System Information in the View menu or press Ctrl+I) to see the pool information in the Kernel Memory section. Here's what that looks like on a 2GB Windows XP system:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)



Kernel Memory (K)	
Paged Physical	17,788
Paged Virtual	17,960
Paged Limit	368,640
<hr/>	
Nonpaged	4,028
Nonpaged Limit	262,144

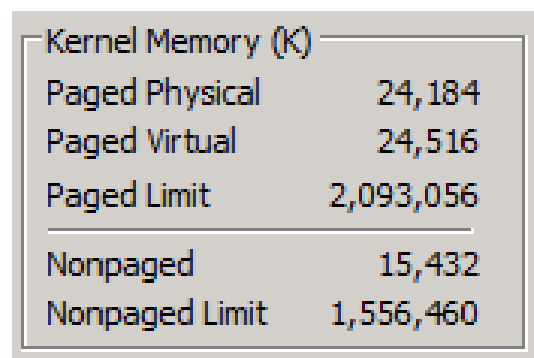
2GB 32-bit Windows XP

Nonpaged Pool Limits

As I mentioned in a previous post, on 32-bit Windows, the system address space is 2GB by default. That inherently caps the upper bound for nonpaged pool (or any type of system virtual memory) at 2GB, but it has to share that space with other types of resources such as the kernel itself, device drivers, system Page Table Entries (PTEs), and cached file views.

Prior to Vista, the memory manager on 32-bit Windows calculates how much address space to assign each type at boot time. Its formulas takes into account various factors, the main one being the amount of physical memory on the system. The amount it assigns to nonpaged pool starts at 128MB on a system with 512MB and goes up to 256MB for a system with a little over 1GB or more. On a system booted with the /3GB option, which expands the user-mode address space to 3GB at the expense of the kernel address space, the maximum nonpaged pool is 128MB. The Process Explorer screenshot shown earlier reports the 256MB maximum on a 2GB Windows XP system booted without the /3GB switch.

The memory manager in 32-bit Windows Vista and later, including Server 2008 and Windows 7 (there is no 32-bit version of Windows Server 2008 R2) doesn't carve up the system address statically; instead, it dynamically assigns ranges to different types of memory according to changing demands. However, it still sets a maximum for nonpaged pool that's based on the amount of physical memory, either slightly more than 75% of physical memory or 2GB, whichever is smaller. Here's the maximum on a 2GB Windows Server 2008 system:



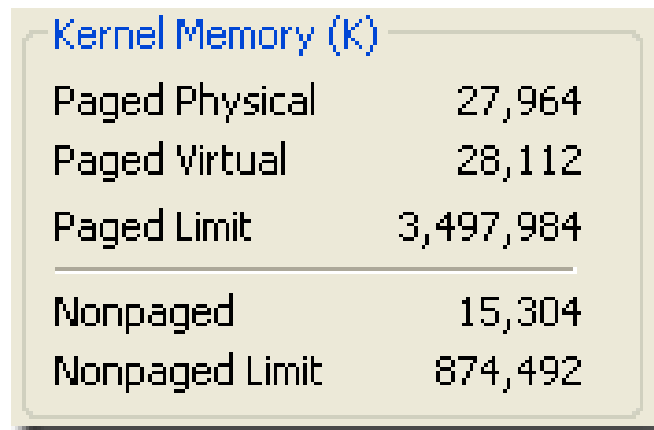
Kernel Memory (K)	
Paged Physical	24,184
Paged Virtual	24,516
Paged Limit	2,093,056
<hr/>	
Nonpaged	15,432
Nonpaged Limit	1,556,460

2GB 32-bit Windows Server 2008

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

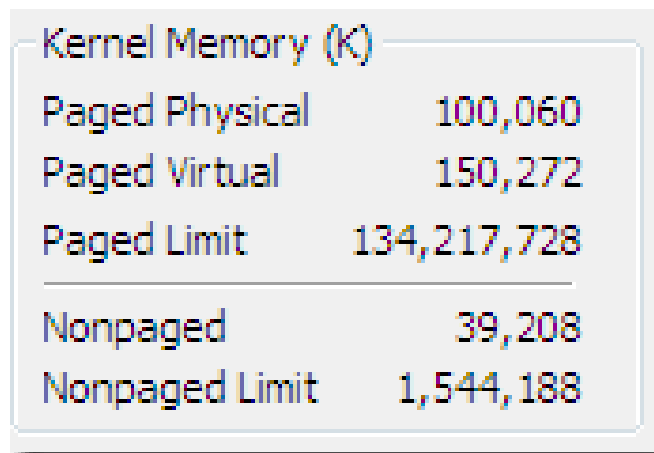
64-bit Windows systems have a much larger address space, so the memory manager can carve it up statically without worrying that different types might not have enough space. 64-bit Windows XP and Windows Server 2003 set the maximum nonpaged pool to a little over 400K per MB of RAM or 128GB, whichever is smaller. Here's a screenshot from a 2GB 64-bit Windows XP system:



Kernel Memory (K)	
Paged Physical	27,964
Paged Virtual	28,112
Paged Limit	3,497,984
<hr/>	
Nonpaged	15,304
Nonpaged Limit	874,492

2GB 64-bit Windows XP

64-bit Windows Vista, Windows Server 2008, Windows 7 and Windows Server 2008 R2 memory managers match their 32-bit counterparts (where applicable – as mentioned earlier, there is no 32-bit version of Windows Server 2008 R2) by setting the maximum to approximately 75% of RAM, but they cap the maximum at 128GB instead of 2GB. Here's the screenshot from a 2GB 64-bit Windows Vista system, which has a nonpaged pool limit similar to that of the 32-bit Windows Server 2008 system shown earlier.



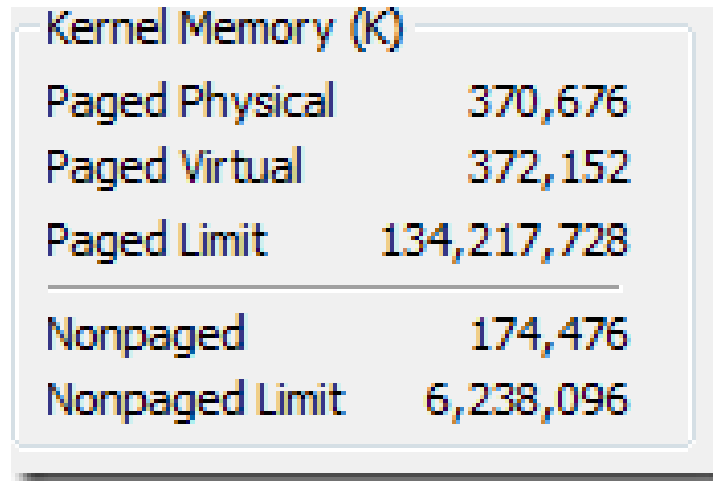
Kernel Memory (K)	
Paged Physical	100,060
Paged Virtual	150,272
Paged Limit	134,217,728
<hr/>	
Nonpaged	39,208
Nonpaged Limit	1,544,188

2GB 32-bit Windows Server 2008

Finally, here's the limit on an 8GB 64-bit Windows 7 system:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)



Kernel Memory (K)	
Paged Physical	370,676
Paged Virtual	372,152
Paged Limit	134,217,728
<hr/>	
Nonpaged	174,476
Nonpaged Limit	6,238,096

8GB 64-bit Windows 7

Here's a table summarizing the nonpaged pool limits across different version of Windows:

	32-bit	64-bit
XP, Server 2003	up to 1.2GB RAM: 32-256 MB > 1.2GB RAM: 256MB	min(~400K/MB of RAM, 128GB)
Vista, Server 2008, Windows 7, Server 2008 R2	min(~75% of RAM, 2GB)	min(~75% of RAM, 128GB)

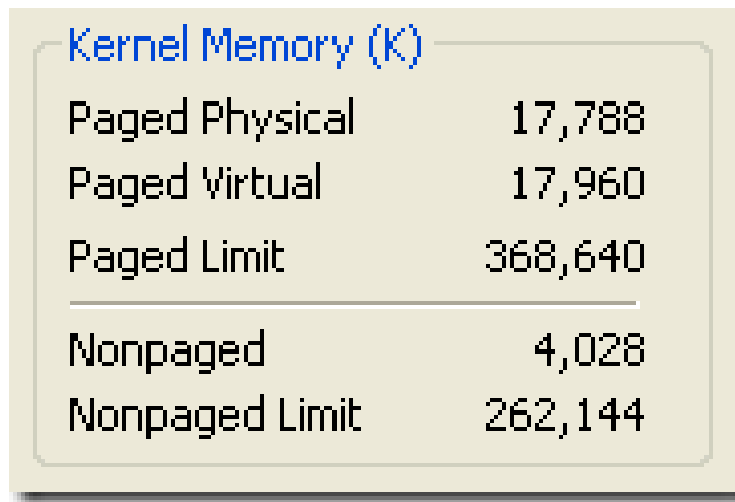
Paged Pool Limits

The kernel and device drivers use paged pool to store any data structures that won't ever be accessed from inside a DPC or ISR or when a spinlock is held. That's because the contents of paged pool can either be present in physical memory or, if the memory manager's working set algorithms decide to repurpose the physical memory, be sent to the paging file and demand-faulted back into physical memory when referenced again. Paged pool limits are therefore primarily dictated by the amount of system address space the memory manager assigns to paged pool, as well as the system commit limit.

On 32-bit Windows XP, the limit is calculated based on how much address space is assigned other resources, most notably system PTEs, with an upper limit of 491MB. The 2GB Windows XP System shown earlier has a limit of 360MB, for example:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

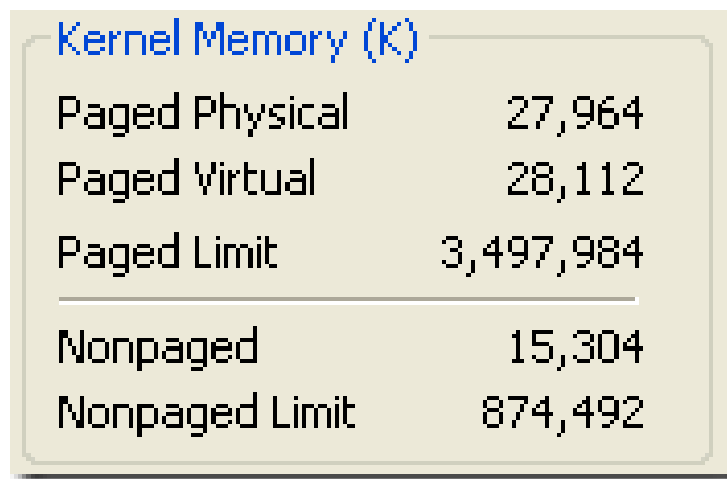


Kernel Memory (K)	
Paged Physical	17,788
Paged Virtual	17,960
Paged Limit	368,640
<hr/>	
Nonpaged	4,028
Nonpaged Limit	262,144

2GB 32-bit Windows XP

32-bit Windows Server 2003 reserves more space for paged pool, so its upper limit is 650MB. Since 32-bit Windows Vista and later have dynamic kernel address space, they simply set the limit to 2GB. Paged pool will therefore run out either when the system address space is full or the system commit limit is reached.

64-bit Windows XP and Windows Server 2003 set their maximums to four times the nonpaged pool limit or 128GB, whichever is smaller. Here again is the screenshot from the 64-bit Windows XP system, which shows that the paged pool limit is exactly four times that of nonpaged pool:



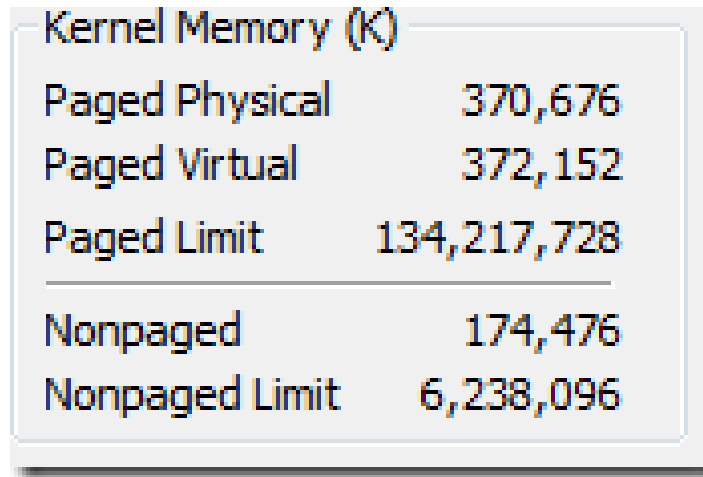
Kernel Memory (K)	
Paged Physical	27,964
Paged Virtual	28,112
Paged Limit	3,497,984
<hr/>	
Nonpaged	15,304
Nonpaged Limit	874,492

2GB 64-bit Windows XP

Finally, 64-bit versions of Windows Vista, Windows Server 2008, Windows 7 and Windows Server 2008 R2 simply set the maximum to 128GB, allowing paged pool's limit to track the system commit limit. Here's the screenshot of the 64-bit Windows 7 system again:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)



Kernel Memory (K)	
Paged Physical	370,676
Paged Virtual	372,152
Paged Limit	134,217,728
<hr/>	
Nonpaged	174,476
Nonpaged Limit	6,238,096

8GB 64-bit Windows 7

Here's a summary of paged pool limits across operating systems:

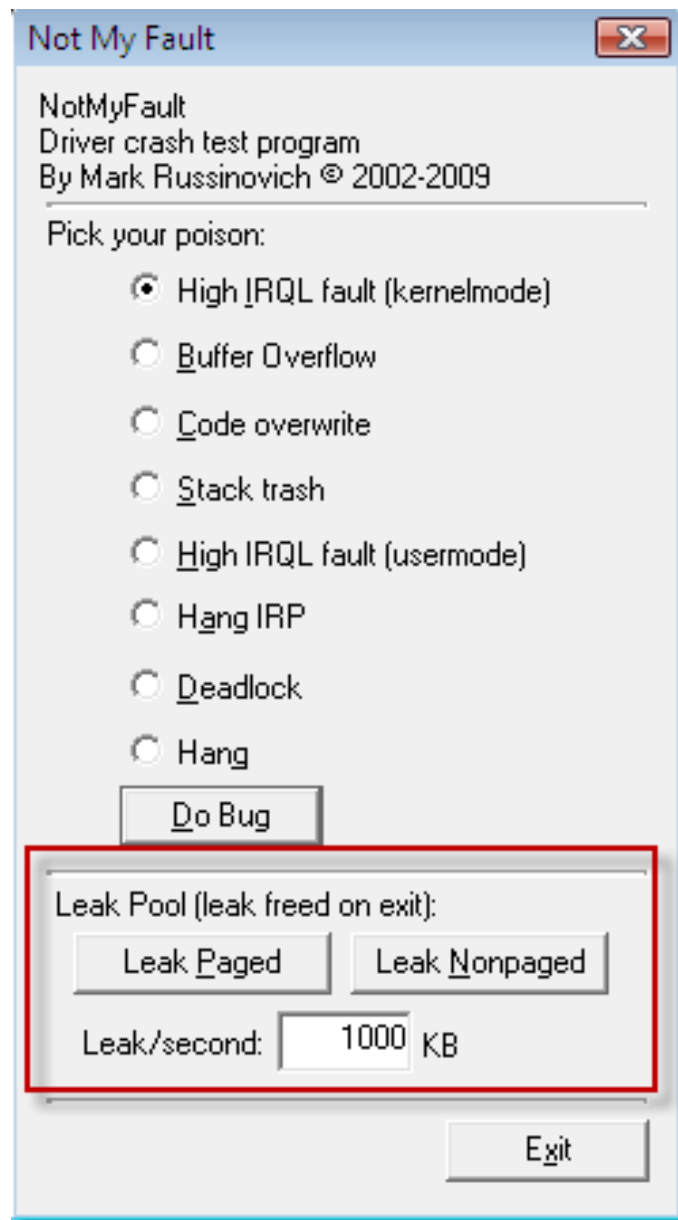
	32-bit	64-bit
XP, Server 2003	XP: up to 491MB Server 2003: up to 650MB	min(4 * nonpaged pool limit, 128GB)
Vista, Server 2008, Windows 7, Server 2008 R2	min(system commit limit, 2GB)	min(system commit limit, 128GB)

Testing Pool Limits

Because the kernel pools are used by almost every kernel operation, exhausting them can lead to unpredictable results. If you want to witness first hand how a system behaves when pool runs low, use the Notmyfault tool. It has options that cause it to leak either nonpaged or paged pool in the increment that you specify. You can change the leak size while it's leaking if you want to change the rate of the leak and Notmyfault frees all the leaked memory when you exit it:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

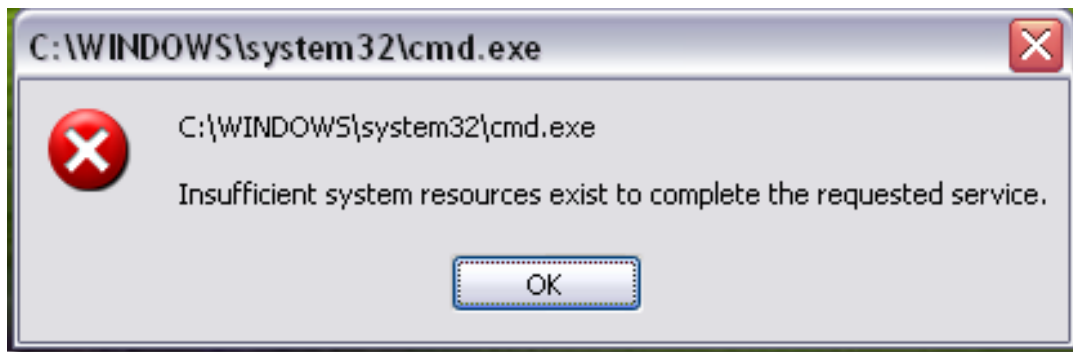


Don't run this on a system unless you're prepared for possible data loss, as applications and I/O operations will start failing when pool runs out. You might even get a blue screen if the driver doesn't handle the out-of-memory condition correctly (which is considered a bug in the driver). The Windows Hardware Quality Laboratory (WHQL) stresses drivers using the Driver Verifier, a tool built into Windows, to make sure that they can tolerate out-of-pool conditions without crashing, but you might have third-party drivers that haven't gone through such testing or that have bugs that weren't caught during WHQL testing.

I ran Notmyfault on a variety of test systems in virtual machines to see how they behaved and didn't encounter any system crashes, but did see erratic behavior. After nonpaged pool ran out on a 64-bit Windows XP system, for example, trying to launch a command prompt resulted in this dialog:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)



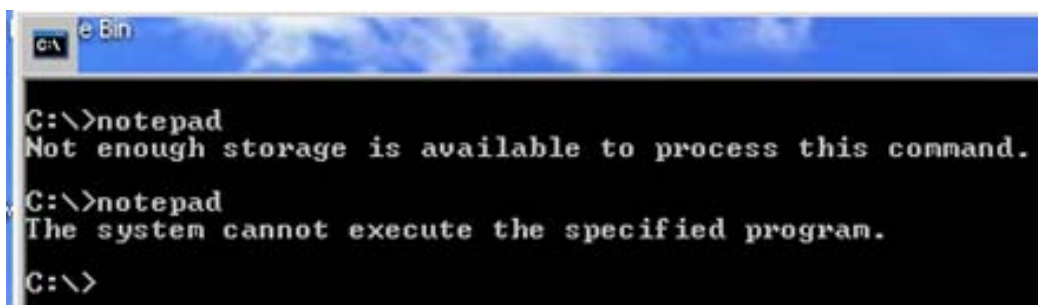
On a 32-bit Windows Server 2008 system where I already had a command prompt running, even simple operations like changing the current directory and directory listings started to fail after nonpaged pool was exhausted:

```
C:\Temp>cd \  
Insufficient system resources exist to complete the requested service.  
C:\Temp>dir  
Insufficient system resources exist to complete the requested service.  
C:\Temp>_
```

On one test system, I eventually saw this error message indicating that data had potentially been lost. I hope you never see this dialog on a real system!



Running out of paged pool causes similar errors. Here's the result of trying to launch Notepad from a command prompt on a 32-bit Windows XP system after paged pool had run out. Note how Windows failed to redraw the window's title bar and the different errors encountered for each attempt:

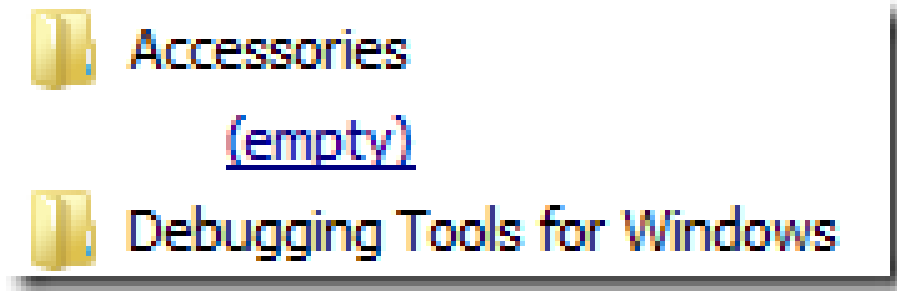


```
e Bin  
C:\>notepad  
Not enough storage is available to process this command.  
C:\>notepad  
The system cannot execute the specified program.  
C:\>
```

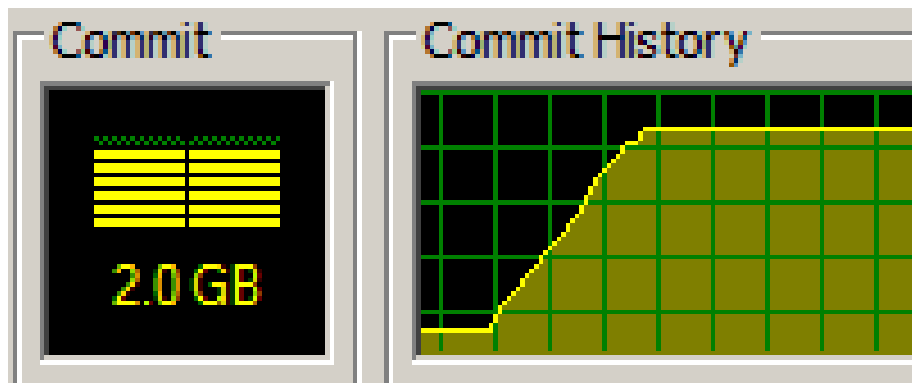
Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

And here's the start menu's Accessories folder failing to populate on a 64-bit Windows Server 2008 system that's out of paged pool:



Here you can see the system commit level, also displayed on Process Explorer's System Information dialog, quickly rise as Notmyfault leaks large chunks of paged pool and hits the 2GB maximum on a 2GB 32-bit Windows Server 2008 system:



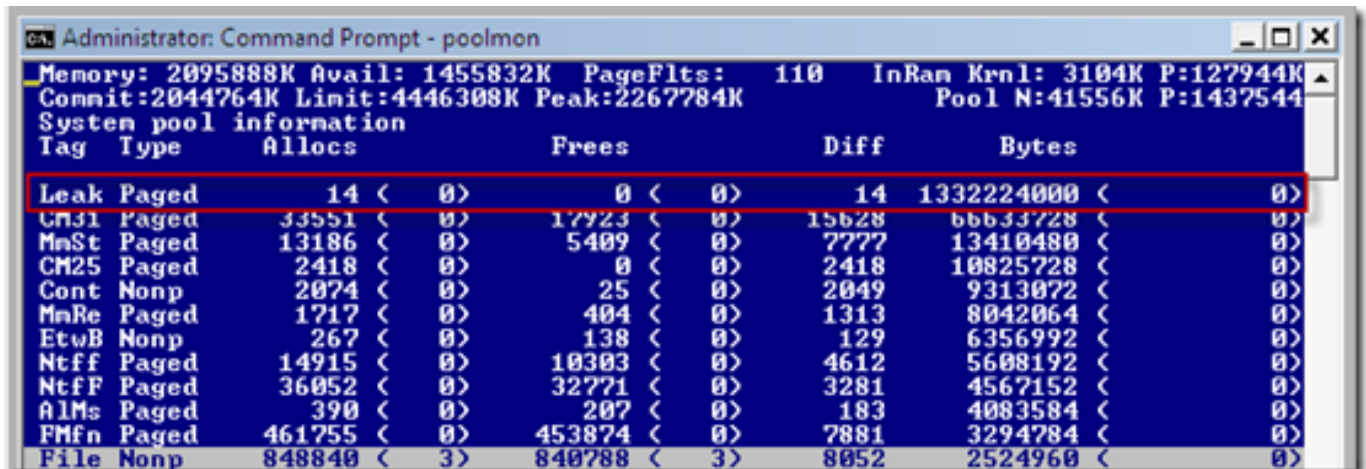
The reason that Windows doesn't simply crash when pool is exhausted, even though the system is unusable, is that pool exhaustion can be a temporary condition caused by an extreme workload peak, after which pool is freed and the system returns to normal operation. When a driver (or the kernel) leaks pool, however, the condition is permanent and identifying the cause of the leak becomes important. That's where the pool tags described at the beginning of the post come into play.

Tracking Pool Leaks

When you suspect a pool leak and the system is still able to launch additional applications, Poolmon, a tool in the Windows Driver Kit, shows you the number of allocations and outstanding bytes of allocation by type of pool and the tag passed into calls of ExAllocatePoolWithTag. Various hotkeys cause Poolmon to sort by different columns; to find the leaking allocation type, use either 'b' to sort by bytes or 'd' to sort by the difference between the number of allocations and frees. Here's Poolmon running on a system where Notmyfault has leaked 14 allocations of about 100MB each:

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)



Administrator: Command Prompt - poolmon

Memory: 2095888K Avail: 1455832K PageFlts: 110 InRam Krn1: 3104K P:127944K
Commit:2044764K Limit:4446308K Peak:2267784K Pool N:41556K P:1437544

System pool information

Tag	Type	Allocs	Frees	Diff	Bytes
Leak	Paged	14 < 0>	0 < 0>	14	1332224000 < 0>
Cm31	Paged	33551 < 0>	17923 < 0>	15628	66633728 < 0>
MmSt	Paged	13186 < 0>	5409 < 0>	7777	13410480 < 0>
Cm25	Paged	2418 < 0>	0 < 0>	2418	10825728 < 0>
Cont	Nonp	2074 < 0>	25 < 0>	2049	9313072 < 0>
MmRe	Paged	1717 < 0>	404 < 0>	1313	8042064 < 0>
Etwb	Nonp	267 < 0>	138 < 0>	129	6356992 < 0>
Ntff	Paged	14915 < 0>	10303 < 0>	4612	5608192 < 0>
Ntff	Paged	36052 < 0>	32771 < 0>	3281	4567152 < 0>
AlMs	Paged	390 < 0>	207 < 0>	183	4083584 < 0>
PMfn	Paged	461755 < 0>	453874 < 0>	7881	3294784 < 0>
File	Nonp	848840 < 3>	840788 < 3>	8052	2524960 < 0>

After identifying the guilty tag in the left column, in this case 'Leak', the next step is finding the driver that's using it. Since the tags are stored in the driver image, you can do that by scanning driver images for the tag in question. The Strings utility from Sysinternals dumps printable strings in the files you specify (that are by default a minimum of three characters in length), and since most device driver images are in the %Systemroot%\System32\Drivers directory, you can open a command prompt, change to that directory and execute "strings * | findstr <tag>". After you've found a match, you can dump the driver's version information with the Sysinternals Sigcheck utility. Here's what that process looks like when looking for the driver using "Leak":

```
C:\Windows\System32\drivers>strings * | findstr Leak
C:\Windows\System32\drivers\myfault.sys: Leak

C:\Windows\System32\drivers>sigcheck myfault.sys

sigcheck v1.60 - sigcheck
Copyright (C) 2004-2008 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\Windows\System32\drivers\myfault.sys:
    Verified:      Signed
    Signing date:  3:43 PM 3/25/2009
    Strong Name:   Unsigned
    Publisher:     Sysinternals
    Description:   Crash Test Driver
    Product:       Sysinternals Myfault
    Version:       2.0
    File version:  2.0 built by: WinDDK

C:\Windows\System32\drivers>
```

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

If a system has crashed and you suspect that it's due to pool exhaustion, load the crash dump file into the Windbg debugger, which is included in the Debugging Tools for Windows package, and use the !vm command to confirm it. Here's the output of !vm on a system where Notmyfault has exhausted nonpaged pool:

```
lkd> !vm
*** Virtual Memory Usage ***
Physical Memory:          262010 (   1048040 Kb)
Page File: \??\C:\pagefile.sys
  Current:    1355240 Kb  Free Space:    1341636 Kb
  Minimum:   1355240 Kb  Maximum:      4193280 Kb
Available Pages:          18144 (    72576 Kb)
ResAvail Pages:           56535 (   226140 Kb)
Locked IO Pages:           0 (         0 Kb)
Free System PTEs:        273266 (  1093064 Kb)
Modified Pages:           1056 (    4224 Kb)
Modified PF Pages:        1029 (    4116 Kb)
NonPagedPool Usage:      191968 (   767872 Kb)
NonPagedPool Max:        193023 (   772092 Kb)
***** Excessive NonPaged Pool Usage *****
PagedPool 0 Usage:        3620 (    14480 Kb)
```

Once you've confirmed a leak, use the !poolused command to get a view of pool usage by tag that's similar to Poolmon's. !poolused by default shows unsorted summary information, so specify 1 as the option to sort by paged pool usage and 2 to sort by nonpaged pool usage:

```
lkd> !poolused 2
Sorting by NonPaged Pool Consumed

Pool Used:
Tag           NonPaged          Paged
Allocs      Used      Allocs      Used
-----
Leak         1334 770807912          0          0
VtPI          1 4194304            0            0
LSwi          1 2623568            0            0
EtwB          86 1506568           10    323584
RaRS         1000 688000            0            0
```

Use Strings on the system where the dump came from to search for the driver using the tag that you find causing the problem.

Pushing the Limits of Windows: Paged and Nonpaged Pool

Mark Russinovich
(From Mark Russinovich Blog)

So far in this blog series I've covered the most fundamental limits in Windows, including physical memory, virtual memory, paged and nonpaged pool. Next time I'll talk about the limits for the number of processes and threads that Windows supports, which are limits that derive from these.