

Pushing the Limits of Windows: Physical Memory

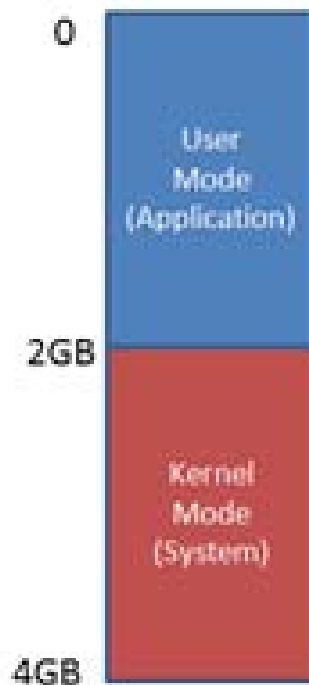
Mark Russinovich
(From Mark Russinovich Blog)

In my first Pushing the Limits of Windows post, I discussed physical memory limits, including the limits imposed by licensing, implementation, and driver compatibility. This time I'm turning my attention to another fundamental resource, virtual memory. Virtual memory separates a program's view of memory from the system's physical memory, so an operating system decides when and if to store the program's code and data in physical memory and when to store it in a file. The major advantage of virtual memory is that it allows more processes to execute concurrently than might otherwise fit in physical memory.

While virtual memory has limits that are related to physical memory limits, virtual memory has limits that derive from different sources and that are different depending on the consumer. For example, there are virtual memory limits that apply to individual processes that run applications, the operating system, and for the system as a whole. It's important to remember as you read this that virtual memory, as the name implies, has no direct connection with physical memory. Windows assigning the file cache a certain amount of virtual memory does not dictate how much file data it actually caches in physical memory; it can be any amount from none to more than the amount that's addressable via virtual memory.

Process Address Spaces

Each process has its own virtual memory, called an address space, into which it maps the code that it executes and the data that the code references and manipulates. A 32-bit process uses 32-bit virtual memory address pointers, which creates an absolute upper limit of 4GB (2^{32}) for the amount of virtual memory that a 32-bit process can address. However, so that the operating system can reference its own code and data and the code and data of the currently-executing process without changing address spaces, the operating system makes its virtual memory visible in the address space of every process. By default, 32-bit versions of Windows split the process address space evenly between the system and the active process, creating a limit of 2GB for each:



Applications might use Heap APIs, the .NET garbage collector, or the C runtime malloc library to allocate virtual memory, but under the hood all of these rely on the VirtualAlloc API. When an application runs out of address space then VirtualAlloc, and therefore the memory managers layered

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)


on top of it, return errors (represented by a NULL address). The Testlimit utility, which I wrote for the 4th Edition of Windows Internals to demonstrate various Windows limits, calls VirtualAlloc repeatedly until it gets an error when you specify the `-r` switch. Thus, when you run the 32-bit version of Testlimit on 32-bit Windows, it will consume the entire 2GB of its address space:

```
C:\Temp>testlimit -r

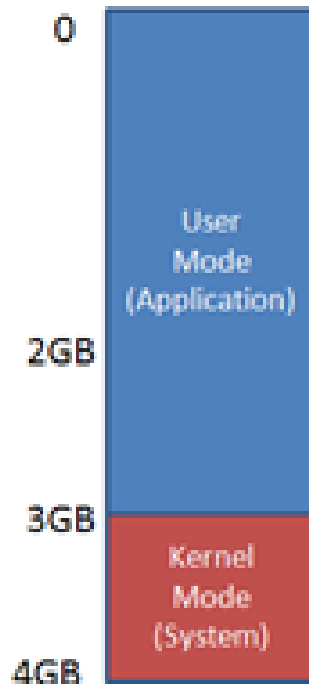
Testlimit v5.0 - tests Windows limits
By Mark Russinovich

Reserving private bytes (MB)...
Reserved 2010 MB of private memory. Lasterror: 8
Not enough storage is available to process this command.
```

2010 MB isn't quite 2GB, but Testlimit's other code and data, including its executable and system DLLs, account for the difference. You can see the total amount of address space it's consumed by looking at its Virtual Size in Process Explorer:

Process	Virtual Size
 Testlimit.exe	2,089,792 K

Some applications, like SQL Server and Active Directory, manage large data structures and perform better the more that they can load into their address space at the same time. Windows NT 4 SP3 therefore introduced a boot option, `/3GB`, that gives a process 3GB of its 4GB address space by reducing the size of the system address space to 1GB, and Windows XP and Windows Server 2003 introduced the `/userva` option that moves the split anywhere between 2GB and 3GB:



Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

To take advantage of the address space above the 2GB line, however, a process must have the 'large address space aware' flag set in its executable image. Access to the additional virtual memory is opt-in because some applications have assumed that they'd be given at most 2GB of the address space. Since the high bit of a pointer referencing an address below 2GB is always zero, they would use the high bit in their pointers as a flag for their own data, clearing it of course before referencing the data. If they ran with a 3GB address space they would inadvertently truncate pointers that have values greater than 2GB, causing program errors including possible data corruption.

All Microsoft server products and data intensive executables in Windows are marked with the large address space awareness flag, including Chkdsk.exe, Lsass.exe (which hosts Active Directory services on a domain controller), Smss.exe (the session manager), and Esentutl.exe (the Active Directory Jet database repair tool). You can see whether an image has the flag with the Dumpbin utility, which comes with Visual Studio:

```
C:\>dumpbin /headers windows\system32\chkdsk.exe
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file windows\system32\chkdsk.exe
PE signature found
File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
    8664 machine (x64)
      5 number of sections
4791914C time date stamp Fri Jan 18 21:57:32 2008
      0 file pointer to symbol table
      0 number of symbols
      F0 size of optional header
      22 characteristics
          Executable
          Application can handle large (>2GB) addresses
```

Testlimit is also marked large-address aware, so if you run it with the -r switch when booted with the 3GB of user address space, you'll see something like this:

```
C:\>testlimit -r
Testlimit v5.0 - tests Windows limits
By Mark Russinovich

Reserving private bytes (MB)...
Reserved 2962 MB of private memory. Lasterror: 8
Not enough storage is available to process this command.
```

Because the address space on 64-bit Windows is much larger than 4GB, something I'll describe shortly, Windows can give 32-bit processes the maximum 4GB that they can address and use the rest

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

for the operating system's virtual memory. If you run Testlimit on 64-bit Windows, you'll see it consume the entire 32-bit addressable address space:

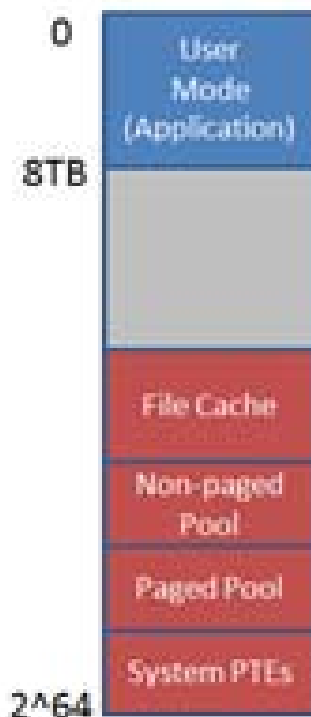
```
C:\>testlimit -r

Testlimit v5.0 - tests Windows limits
By Mark Russinovich

Reserving private bytes (MB)...
Reserved 4026 MB of private memory. Lasterror: 8
Not enough storage is available to process this command.
```

64-bit processes use 64-bit pointers, so their theoretical maximum address space is 16 exabytes (2^{64}). However, Windows doesn't divide the address space evenly between the active process and the system, but instead defines a region in the address space for the process and others for various system memory resources, like system page table entries (PTEs), the file cache, and paged and non-paged pools.

The size of the process address space is different on IA64 and x64 versions of Windows where the sizes were chosen by balancing what applications need against the memory costs of the overhead (page table pages and translation lookaside buffer - TLB - entries) needed to support the address space. On x64, that's 8192GB (8TB) and on IA64 it's 7168GB (7TB - the 1TB difference from x64 comes from the fact that the top level page directory on IA64 reserves slots for Wow64 mappings). On both IA64 and x64 versions of Windows, the size of the various resource address space regions is 128GB (e.g. non-paged pool is assigned 128GB of the address space), with the exception of the file cache, which is assigned 1TB. The address space of a 64-bit process therefore looks something like this:



Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

The figure isn't drawn to scale, because even 8TB, much less 128GB, would be a small sliver. Suffice it to say that like our universe, there's a lot of emptiness in the address space of a 64-bit process.

When you run the 64-bit version of Testlimit (Testlimit64) on 64-bit Windows with the `-r` switch, you'll see it consume 8TB, which is the size of the part of the address space it can manage:

```
C:\>testlimit64 -r

Testlimit v5.0 - tests Windows limits
By Mark Russinovich

Reserving private bytes (MB)...
Reserved 8388544 MB of private memory. Lasterror: 8
Not enough storage is available to process this command.
```

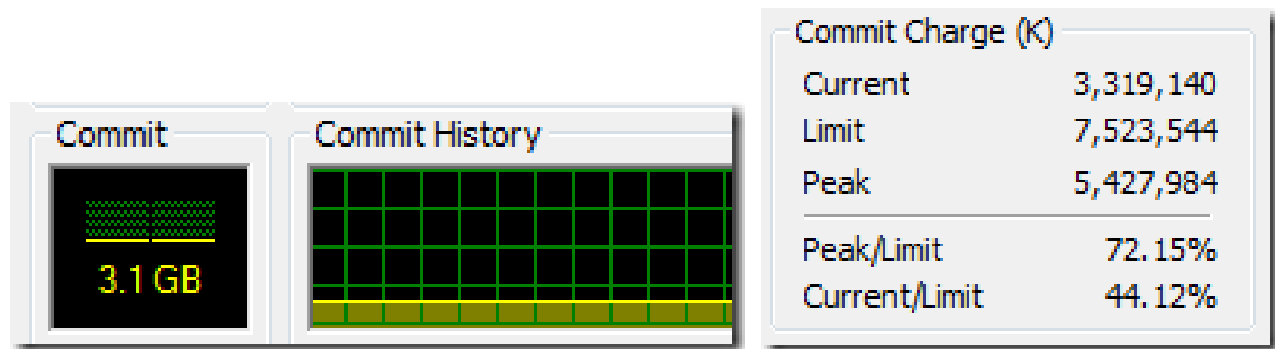
Process	Virtual Size
testlimit64.exe	8,589,913,132 K

Committed Memory

Testlimit's `-r` switch has it reserve virtual memory, but not actually commit it. Reserved virtual memory can't actually store data or code, but applications sometimes use a reservation to create a large block of virtual memory and then commit it as needed to ensure that the committed memory is contiguous in the address space. When a process commits a region of virtual memory, the operating system guarantees that it can maintain all the data the process stores in the memory either in physical memory or on disk. That means that a process can run up against another limit: the commit limit.

As you'd expect from the description of the commit guarantee, the commit limit is the sum of physical memory and the sizes of the paging files. In reality, not quite all of physical memory counts toward the commit limit since the operating system reserves part of physical memory for its own use. The amount of committed virtual memory for all the active processes, called the current commit charge, cannot exceed the system commit limit. When the commit limit is reached, virtual allocations that commit memory fail. That means that even a standard 32-bit process may get virtual memory allocation failures before it hits the 2GB address space limit.

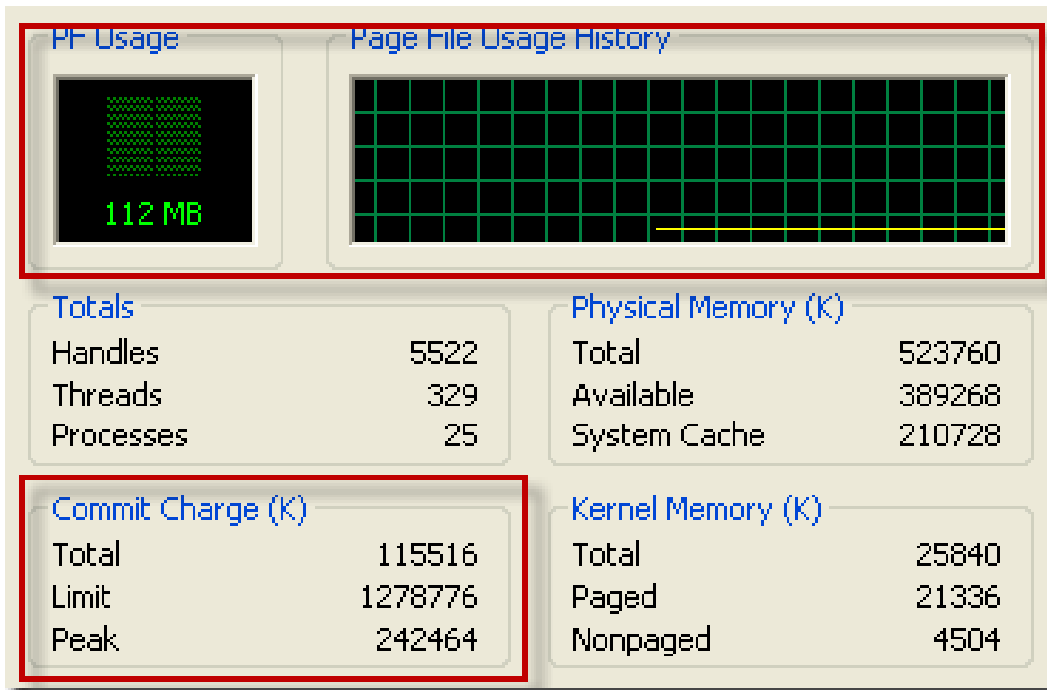
The current commit charge and commit limit is tracked by Process Explorer in its System Information window in the Commit Charge section and in the Commit History bar chart and graph:



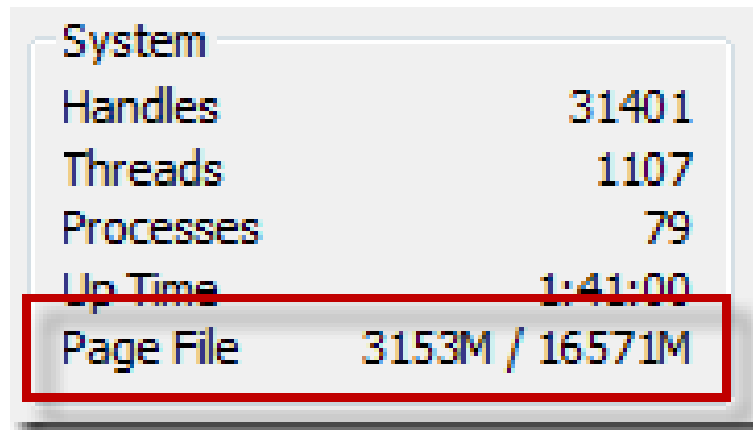
Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

Task Manager prior to Vista and Windows Server 2008 shows the current commit charge and limit similarly, but calls the current commit charge "PF Usage" in its graph:



On Vista and Server 2008, Task Manager doesn't show the commit charge graph and labels the current commit charge and limit values with "Page File" (despite the fact that they will be non-zero values even if you have no paging file):



You can stress the commit limit by running Testlimit with the -m switch, which directs it to allocate committed memory. The 32-bit version of Testlimit may or may not hit its address space limit before hitting the commit limit, depending on the size of physical memory, the size of the paging files and the current commit charge when you run it. If you're running 32-bit Windows and want to see how the system behaves when you hit the commit limit, simply run multiple instances of Testlimit until one hits the commit limit before exhausting its address space.

Note that, by default, the paging file is configured to grow, which means that the commit limit will grow when the commit charge nears it. And even when when the paging file hits its maximum size,

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

Windows is holding back some memory and its internal tuning, as well as that of applications that cache data, might free up more. Testlimit anticipates this and when it reaches the commit limit, it sleeps for a few seconds and then tries to allocate more memory, repeating this indefinitely until you terminate it.

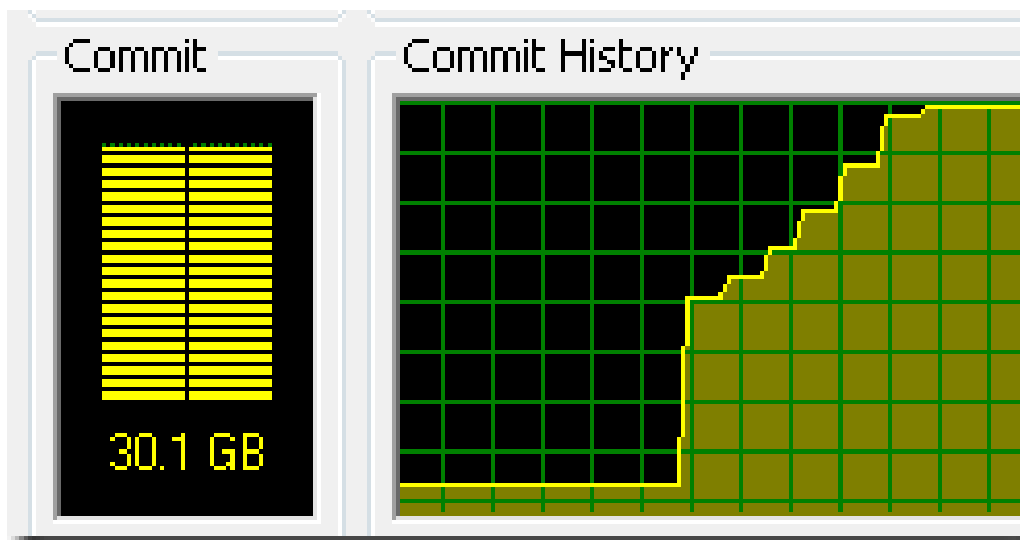
If you run the 64-bit version of Testlimit, it will almost certainly will hit the commit limit before exhausting its address space, unless physical memory and the paging files sum to more than 8TB, which as described previously is the size of the 64-bit application-accessible address space. Here's the partial output of the 64-bit Testlimit running on my 8GB system (I specified an allocation size of 100MB to make it leak more quickly):

```
C:\>testlimit64 -m 100

Testlimit v5.0 - tests Windows limits
By Mark Russinovich

Leaking private bytes 100 MB at a time...
Leaked 13900 MB of private memory (13900 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 2200 MB of private memory (16100 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 2600 MB of private memory (18700 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 2600 MB of private memory (21300 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 3400 MB of private memory (24700 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 3900 MB of private memory (28600 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 200 MB of private memory (28800 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Leaked 0 MB of private memory (28800 MB total leaked). Lasterror: 1455
The paging file is too small for this operation to complete.
Sleeping for 5 seconds to allow for paging file expansion...
```

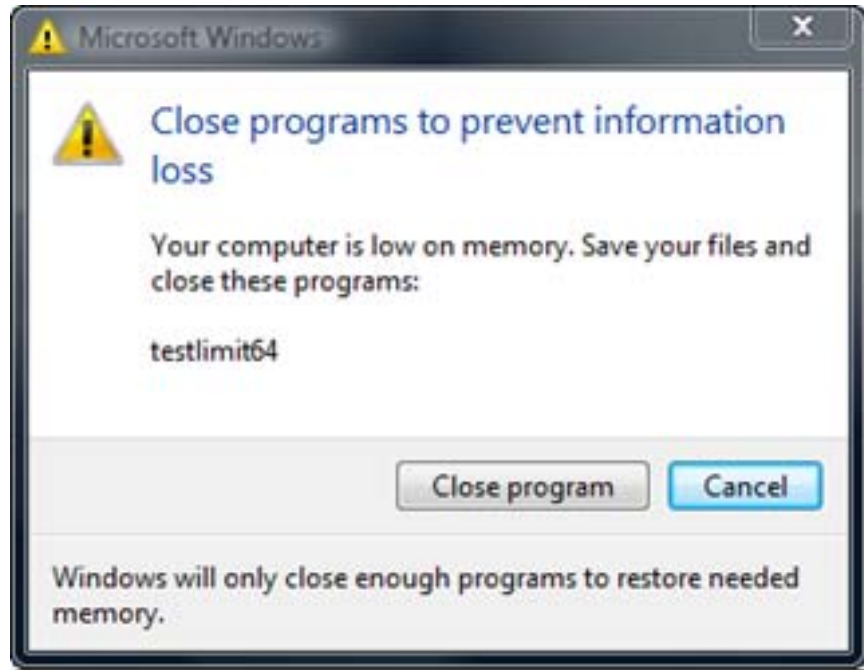
And here's the commit history graph with steps when Testlimit paused to allow the paging file to grow:



Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

When system virtual memory runs low, applications may fail and you might get strange error messages when attempting routine operations. In most cases, though, Windows will be able present you the low-memory resolution dialog, like it did for me when I ran this test:



After you exit Testlimit, the commit limit will likely drop again when the memory manager truncates the tail of the paging file that it created to accommodate Testlimit's extreme commit requests. Here, Process Explorer shows that the current limit is well below the peak that was achieved when Testlimit was running:

Commit Charge (K)	
Current	3,215,576
Limit	16,969,052
Peak	33,431,160
Peak/Limit	197.01%
Current/Limit	18.95%

Process Committed Memory

Because the commit limit is a global resource whose consumption can lead to poor performance, application failures and even system failure, a natural question is 'how much are processes contributing the commit charge'? To answer that question accurately, you need to understand the different types of virtual memory that an application can allocate.

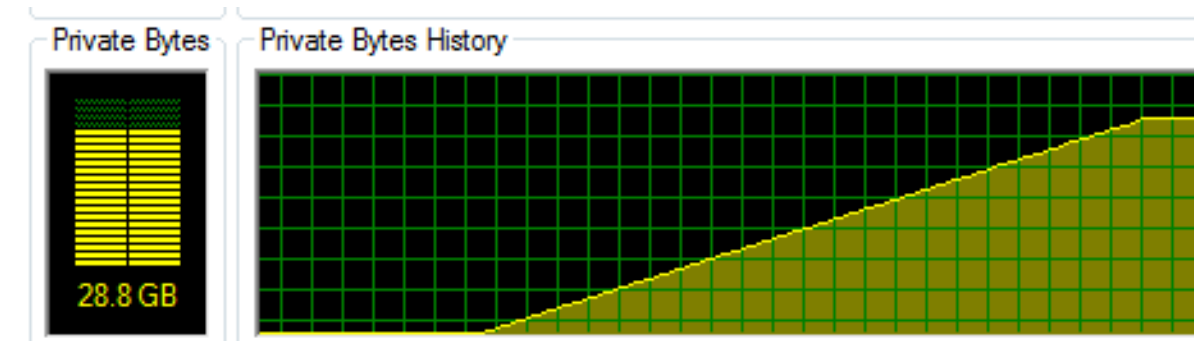
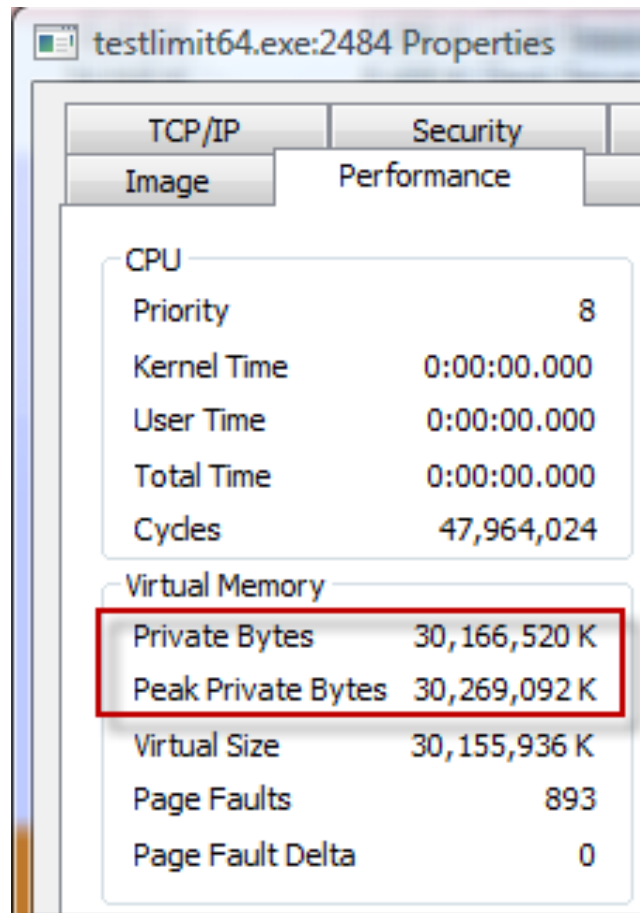
Not all the virtual memory that a process allocates counts toward the commit limit. As you've seen, reserved virtual memory doesn't. Virtual memory that represents a file on disk, called a file mapping view, also doesn't count toward the limit unless the application asks for copy-on-write semantics,

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

because Windows can discard any data associated with the view from physical memory and then retrieve it from the file. The virtual memory in Testlimit's address space where its executable and system DLL images are mapped therefore don't count toward the commit limit. There are two types of process virtual memory that do count toward the commit limit: private and pagefile-backed.

Private virtual memory is the kind that underlies the garbage collector heap, native heap and language allocators. It's called private because by definition it can't be shared between processes. For that reason, it's easy to attribute to a process and Windows tracks its usage with the Private Bytes performance counter. Process Explorer displays a process private bytes usage in the Private Bytes column, in the Virtual Memory section of the Performance page of the process properties dialog, and displays it in graphical form on the Performance Graph page of the process properties dialog. Here's what Testlimit64 looked like when it hit the commit limit:



Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

Pagefile-backed virtual memory is harder to attribute, because it can be shared between processes. In fact, there's no process-specific counter you can look at to see how much a process has allocated or is referencing. When you run Testlimit with the -s switch, it allocates pagefile-backed virtual memory until it hits the commit limit, but even after consuming over 29GB of commit, the virtual memory statistics for the process don't provide any indication that it's the one responsible:

Virtual Memory	
Private Bytes	2,024 K
Peak Private Bytes	2,024 K
Virtual Size	49,588 K
Page Faults	883
Page Fault Delta	0

For that reason, I added the -l switch to Handle a while ago. A process must open a pagefile-backed virtual memory object, called a section, for it to create a mapping of pagefile-backed virtual memory in its address space. While Windows preserves existing virtual memory even if an application closes the handle to the section that it was made from, most applications keep the handle open. The -l switch prints the size of the allocation for pagefile-backed sections that processes have open. Here's partial output for the handles open by Testlimit after it has run with the -s switch:

```
C:\Windows\system32>handle -a -l -p testlimit64
Handle v3.42
Copyright (C) 1997-2008 Mark Russinovich
Sysinternals - www.sysinternals.com

-----
testlimit64.exe pid: 2676 NTDEV\markruss
 38: Section
    Pagefile      1048576 bytes
 3C: Section
    Pagefile      1048576 bytes
 40: Section
    Pagefile      1048576 bytes
 44: Section
    Pagefile      1048576 bytes
...
```

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)

You can see that Testlimit is allocating pagefile-backed memory in 1MB blocks and if you summed the size of all the sections it had opened, you'd see that it was at least one of the processes contributing large amounts to the commit charge.

How Big Should I Make the Paging File?

Perhaps one of the most commonly asked questions related to virtual memory is, how big should I make the paging file? There's no end of ridiculous advice out on the web and in the newsstand magazines that cover Windows, and even Microsoft has published misleading recommendations. Almost all the suggestions are based on multiplying RAM size by some factor, with common values being 1.2, 1.5 and 2. Now that you understand the role that the paging file plays in defining a system's commit limit and how processes contribute to the commit charge, you're well positioned to see how useless such formulas truly are.

Since the commit limit sets an upper bound on how much private and pagefile-backed virtual memory can be allocated concurrently by running processes, the only way to reasonably size the paging file is to know the maximum total commit charge for the programs you like to have running at the same time. If the commit limit is smaller than that number, your programs won't be able to allocate the virtual memory they want and will fail to run properly.

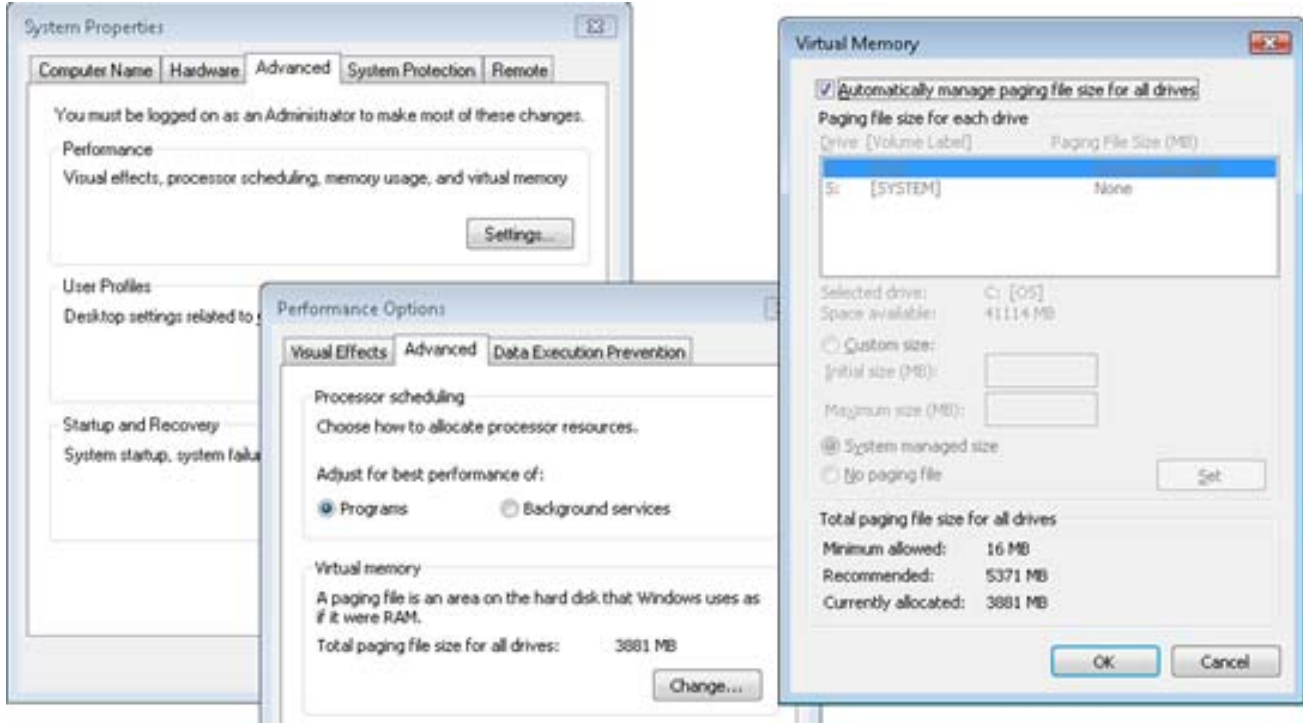
So how do you know how much commit charge your workloads require? You might have noticed in the screenshots that Windows tracks that number and Process Explorer shows it: Peak Commit Charge. To optimally size your paging file you should start all the applications you run at the same time, load typical data sets, and then note the commit charge peak (or look at this value after a period of time where you know maximum load was attained). Set the paging file minimum to be that value minus the amount of RAM in your system (if the value is negative, pick a minimum size to permit the kind of crash dump you are configured for). If you want to have some breathing room for potentially large commit demands, set the maximum to double that number.

Some feel having no paging file results in better performance, but in general, having a paging file means Windows can write pages on the modified list (which represent pages that aren't being accessed actively but have not been saved to disk) out to the paging file, thus making that memory available for more useful purposes (processes or file cache). So while there may be some workloads that perform better with no paging file, in general having one will mean more usable memory being available to the system (never mind that Windows won't be able to write kernel crash dumps without a paging file sized large enough to hold them).

Paging file configuration is in the System properties, which you can get to by typing "sysdm.cpl" into the Run dialog, clicking on the Advanced tab, clicking on the Performance Options button, clicking on the Advanced tab (this is really advanced), and then clicking on the Change button:

Pushing the Limits of Windows: Physical Memory

Mark Russinovich
(From Mark Russinovich Blog)



You'll notice that the default configuration is for Windows to automatically manage the page file size. When that option is set on Windows XP and Server 2003, Windows creates a single paging file that's minimum size is 1.5 times RAM if RAM is less than 1GB, and RAM if it's greater than 1GB, and that has a maximum size that's three times RAM. On Windows Vista and Server 2008, the minimum is intended to be large enough to hold a kernel-memory crash dump and is RAM plus 300MB or 1GB, whichever is larger. The maximum is either three times the size of RAM or 4GB, whichever is larger. That explains why the peak commit on my 8GB 64-bit system that's visible in one of the screenshots is 32GB. I guess whoever wrote that code got their guidance from one of those magazines I mentioned!

A couple of final limits related to virtual memory are the maximum size and number of paging files supported by Windows. 32-bit Windows has a maximum paging file size of 16TB (4GB if you for some reason run in non-PAE mode) and 64-bit Windows can have paging files that are up to 16TB in size on x64 and 32TB on IA64. For all versions, Windows supports up to 16 paging files, where each must be on a separate volume.