Virtual memory constraints in 32-bit Windows: an Update

Mark B. Friedman
Demand Technology
1020 Eighth Avenue South, Suite 6
Naples, FL USA 34102
markf@demandtech.com

Abstract.

This paper discusses the signs that indicate a machine is suffering from a virtual memory constraint in 32-bit Windows. Machines configured with 2 GB or more of RAM installed are particularly vulnerable to this condition. It also discusses options to keep this from happening, including (1) changing the way 32-bit virtual address spaces are partitioned into private and shared ranges, (2) settings that govern the size of system memory pools, (3) hardware that supports 37-bit addressing, and (4) hardware that supports 64-bit addressing but can still run 32-bit applications in compatibility mode. Ultimately, the option of running Windows on 64-bit processors is the safest and surest way to relieve the virtual memory constraints associated with 32-bit Windows.

Introduction

The Microsoft Windows Server 2003 operating system creates a separate and independent virtual address space for each individual process that is launched. On 32-bit processors, each process virtual address space can be as large as 4 GB. (On 64-bit processors process virtual address spaces can be as large as 8 TB in current versions of Windows Server 2003.) There are many server workloads that can easily exhaust the 32-bit virtual address space available under Windows Server 2003. Machines configured with 2 GB or more of RAM installed appear to be particularly vulnerable to these virtual memory constraints for reasons that are discussed below. When Windows Server 2003 workloads exhaust their 32-bit virtual address space, the consequences are usually catastrophic. This paper discusses the signs and symptoms that indicate there is a serious virtual memory constraint.

This paper also discusses the features and options that system administrators can employ to forestall running short of virtual memory. The Windows Server 2003 operating system offers several forms of relief from the virtual memory constraints that arise on 32-bit machines. These include (1) options to change the manner in which 32-bit process virtual address spaces are partitioned into private addresses and shared system addresses, (2) settings that govern the size of key system memory pools, (3) hardware options that permit 37-bit addressing, and (4) hardware options that support 64-bit addressing. By selecting among these options, system administrators can avoid many situations where virtual memory constraints impact system availability and performance. Since these virtual memory addressing constraints arise inevitably as the size of RAM grows, the most effective way to deal with these constraints in the long run is to move to processors that can access 64-bit virtual addresses, running the 64-bit version Windows Server 2003.

Virtual addressing

Virtual memory is a feature supported by most advanced processors. Hardware support for virtual memory includes an address translation mechanism to map logical (i.e., virtual) memory addresses that application programs reference to physical (or real) memory hardware addresses. Virtual address translation is then performed transparently by the processor hardware during program execution. Only authorized operating system functions are capable of addressing physical memory locations directly. Figure 1 illustrates the hardware virtual address translation mechanism for Intel IA-32 processors. The hardware splits each 32-bot address reference into three segments. The high order ten bits of a virtual address are used to point to a specific Page Directory entry. The Page Directory entry points to a page of Page Table entries (PTEs), which is indexed using the middle ten bits of the virtual address to locate a specific four-byte PTE. The PTE entry that is cross-indexed in this fashion contains the high order 20 bits of the physical memory page that are then merged during translation with the low order 12-bits (enough to address 0-4095 bytes) of the virtual address to create the physical memory address. The operating system is responsible for building and maintaining the Page Directory and associated PTEs in the proper hardware-specified format on behalf of each process address space that is created. The OS also ensures that whenever a new program execution thread is dispatched that the processor's Control Register 3 is loaded with a pointer to the origin of the Page Directory for the process address space ***context*** that the running thread is associated with.

To implement virtual address translation, when an executable program's image file is first loaded into memory, the logical memory address range of the application is divided into fixed size chunks called ***pages.*** The operating system builds a PTE for each valid page of virtual memory that is loaded and

continues to build PTEs for additional pages that a process allocates (for data structures and other working storage, etc.) as it executes. The PTE supplies the mapping information for those virtual memory pages that reside in physical memory. This mapping is dynamic so that logical addresses that are frequently referenced tend to reside in physical memory, while infrequently referenced pages are relegated to paging files on secondary disk storage. The active subset of virtual memory pages associated with a single process address space that are currently resident in RAM is known as the process's ***working set.***

The low order bit of a PTE, known as the Present bit, is set to reflect the status of a virtual memory page in physical memory. If the Present bit is not set, the operating system stores information in the PTE that points to its location on the paging file. A reference to an invalid page by an executing thread causes the hardware to report an addressing exception or ***page fault***. The operating system then intercedes to allocate a free page in RAM, copy the contents of the page from disk to that physical memory location, update the PTE appropriately, and re-execute the instruction that originally failed. This page fault resolution process can have profound performance implications if performed frequently, due to the relatively slow speed of mechanical disks compared to the electronic speeds that prevail for accessing physical memory locations. Readers interested in a fuller understanding of this common hardware mechanism should consult [1].

Most performance-oriented treatises on virtual memory systems dissect the problems that can arise when physical memory is over-committed and excessive paging to disk occurs. Virtual memory systems usually work well because executing programs seldom require all their pages to be resident in physical memory concurrently in order to run. With virtual memory, only the ***active*** pages associated with a program's current working set remain resident in physical memory. On the other hand, virtual memory systems can run very poorly when the working sets of active processes greatly exceed the amount of RAM that the computer contains. A physical memory constraint is then transformed into an I/O bottleneck due to excessive amounts of activity to the paging disk (or disks).

Performance and capacity problems associated with virtual memory architectural constraints tend to receive far less scrutiny. These problems arise out of a hardware limitation, namely, the number of bits available to construct a memory address. The number of bits associated with a hardware memory address determines the memory addressing range. In the case of the 32-bit Intel-compatible processors that run Microsoft Windows, address registers are 32 bits wide, allowing for addressability of 0-4,294,967,295 bytes, which is conventionally denoted as a 4 GB range. This 4 GB range can be an architectural constraint, especially with workloads that need 2 GB or more of RAM to perform well. The suggestion that machines with 2 GB or more of RAM installed are most susceptible to addressability constraints is based on the observation that smaller machines are more prone to encounter paging performance problems before the architectural constraints discussed here are manifest.

Virtual memory constraints are manifest during periods of transition from one processor architecture to another. In the evolution of the Intel x86 processor, there was a period of transition from 16-bit addressing that was a feature of the original 8086 and 8088 processors that launched the PC revolution to the 24-bit segmented addressing mode of the 80286 to the current flat 32-bit addressing model that is implemented across all Intel IA-32 processors. Currently, the IA-32 architecture is in a state of transition with two flavors of machines capable of 64-bit addressing starting to become widely available. As 64-bit machines start to become more commonplace, they will definitively relieve the virtual memory constraints that are evident today on the 32-bit platform.

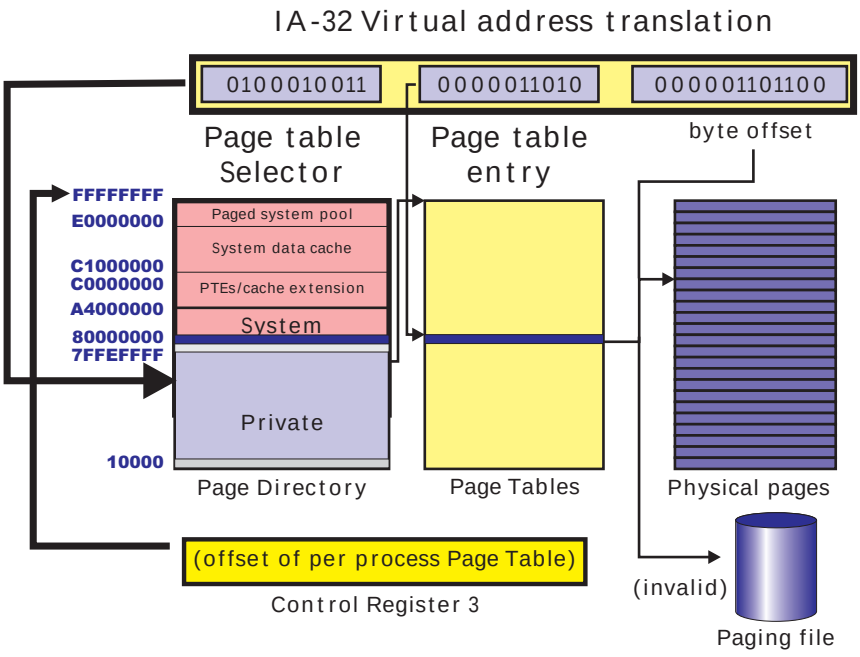

FIGURE 1. VIRTUAL MEMORY ADDRESS TRANSLATION ON 32-BIT INTEL-COMPATIBLE PROCESSORS.

Physical Address Extension. Functioning as a stop-gap solution on the road to full 64-bit memory addressing, Intel currently provides IA-32 machines with a Physical Address Extension (PAE) feature that supports a wider 37-bit address. PAE

provides the capability to address up to 128 GB of physical RAM. Virtual addresses remain 32-bits when PAE is active, but it requires 8-byte PTEs and a novel, 3-level address Page Directory structure, which is illustrated in Figure 2 for translating 32-bit virtual addresses into 37-bit physical addresses.

When a machine is booted in PAE mode, the Windows Server 2003 operating system builds PTEs and Page Directories in the proper format to support extended addressing. Individual process address spaces are still limited to 4 GB in size when Windows Server 2003 runs in PAE mode, so utilizing physical memory above 4 GB can be challenging, as discussed in more detail below. Because 64-bit PAE PTEs consume twice the amount of space as conventional 32-bit PTEs, enabling PAE mode should be restricted to those configurations where more than 4 GB of RAM is installed and PAE-mode is required to address the physical RAM installed above the 4 GB line.

PAE is an interim fix, a pit stop on the road towards full 64-bit addressing in Intel hardware. As a stop-gap solution, it has an element of ***deja vu*** about it. It bears an uncanny resemblance to interim actions taken by other processor manufacturers for families of machines that similarly evolved from 16 or 24-bit machines to 32-bit addresses, and ultimately to today's 64-bit machines. In the IBM mainframe world, there was a prolonged focus on Virtual Storage Constraint Relief (VSCR) for its popular 24-bit OS/360 hardware and software in almost every subsequent release of new hardware and software that IBM produced from 1980 to the present day. The popular book ***The Soul of a New Machine*** [2], chronicled the development of Data General's 32-bit address machines to keep pace with the Digital Equipment Corporation's 32-bit Virtual Address Extension (VAX) of its original 16-bit minicomputer architecture. Even though today's hardware and software engineers are hardly ignorant of the relevant past history, they are still condemned to repeat the cycle of delivering stopgap solutions that provide a modicum of virtual memory constraint relief until the next major architectural step forward is taken.

Process virtual address spaces.

The Windows operating system constructs a separate virtual memory address space on behalf of each running process, potentially addressing up to 4 GB of virtual memory on 32-bit machines. Each 32-bit process virtual address space is divided into two equal parts, as depicted in Figure 3. The lower 2 GB of each process address space consists of private addresses associated with that specific process only. This 2 GB range of addresses refers to pages that can only be accessed by threads running in that process address space context. Each per process virtual address space can range from 0x0001 0000 to address 0x7fff ffff, spanning 2 GBs, potentially. (The first 64K addresses are protected from being accessed - it is possible to trap many common programming errors that way.) Each process gets its own unique set of user addresses in this range. Furthermore, no thread running in one process can access virtual memory addresses in the private range that is associated with another process. As it ex-

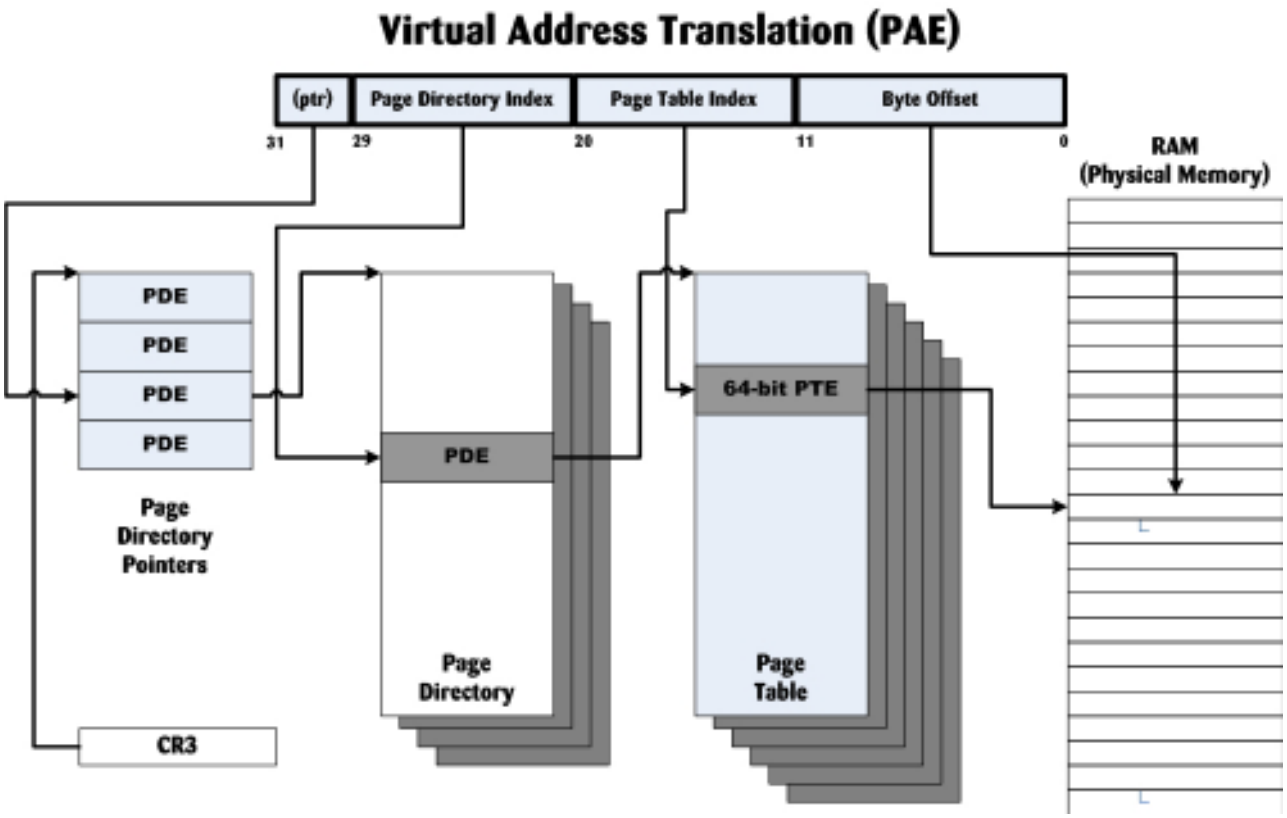

FIGURE 2. VIRTUAL ADDRESS TRANSLATION IN PAE-MODE.

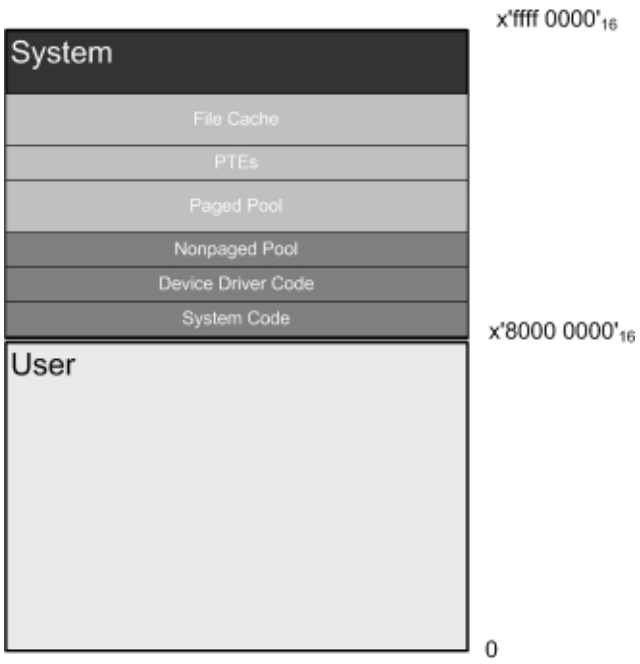

FIGURE 3. THE 4 GB ADDRESS SPACE LAYOUT USED IN 32-BIT INTEL-COMPATIBLE MACHINE.

ecutes, a process allocates virtual memory from its user address space for code and data structures of all types.

The 2 GB upper limit on the size of a user-mode process virtual address space can be a constraint for a variety of application programs. How to recognize those processes that are affected and what actions can be taken to relieve their capacity constraints is the focus of this discussion. One initial complication is that it is seldom possible for a process to allocate its complete virtual memory range. This is due to fragmentation that occurs because virtual memory is allocated and de-allocated in non-uniform chunks. Because of this fragmentation, a virtual memory allocation request by a process can fail before the 2 GB limit on addressability is completely exhausted.

A process that allocates virtual memory, but neglects to free it, suffers from a memory leak, a program bug that will eventually exhaust the supply of virtual memory available to a process. Application programs with virtual memory leaks will not be discussed any further here, but please refer to an earlier version of this paper for a full account of their detection and diagnosis [4].

Since the operating system builds a unique address space for every process, Figure 4 is perhaps a better picture of what User virtual address spaces look like. Notice that the System portion of each process address space is identical. One set of System Page Table Entries (PTEs) maps the System portion of the virtual address space for every process. Because System addresses are common to all processes, they offer a convenient way for processes to communicate with each other, when necessary.

Shared system addresses

The upper half of each per process address space in the range of '0x8000 0000' to '0xffff ffff' consists of system addresses common to all virtual address spaces. All running processes have access to the same set of addresses in the system range. This feat is accomplished by combining the system's page tables with each unique per process set of page tables. Commonly addressable system virtual memory locations play an important role in various forms of interprocess communication, or IPC. Win32 API functions can be used to allocate portions of commonly addressable system areas to share data between two or more distinct processes. For example, the mechanism Windows Server 2003 uses that allows multiple process address spaces to access common runtime modules known as Dynamically Linked Libraries (DLLs) utilizes this form of shared memory addressing. (DLLs are library modules that contain subroutines and functions which can be called dynamically at run-time, instead of being linked statically to the programs that utilize them.)

User mode threads running inside a process cannot directly address memory locations in the system range because system virtual addresses are allocated using Privileged mode. This restricts memory access to kernel threads that run in Privileged mode. This is a form of security that restricts access to system memory to authorized kernel threads. When an application execution thread calls a system function, it transfers control to an associated kernel mode thread, and, in the process, routinely changes the execution state from User mode to Privileged. It is in this fashion that an application thread gains access to system virtual memory addresses.

System virtual memory constraints

It is entirely possible for virtual memory in the system range to be exhausted. It is also possible for one of the four major system memory pools to reach

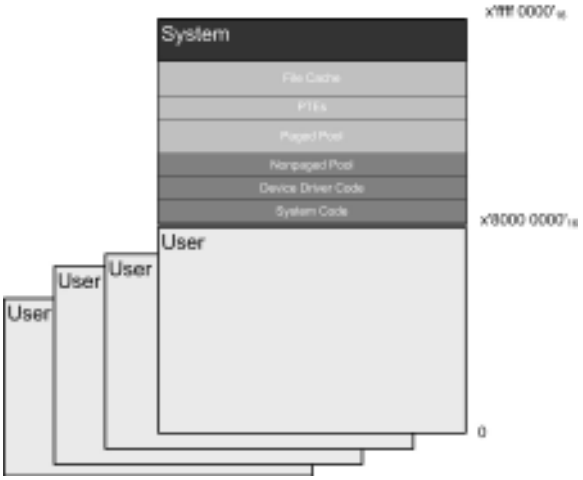

FIGURE 4. USER PROCESSES SHARE THE SYSTEM PORTION OF THE 4 GB VIRTUAL ADDRESS SPACE.

its maximum allocation level before the full range of system virtual memory addresses is exhausted. There are two sets of circumstances in 32-bit Windows where concerns about running out of system virtual memory are heightened. These are (1) Terminal Server machines intended to support large numbers of User mode processes and threads and (2) machines booted with the ***/3GB*** option that extends the User private area to 3 GB, but shrinks the system area down to 1 GB.

The shared system address range is used mainly to store data areas for the following four components of the operating system:

- The system file cache that holds segments of recently accessed file system objects. The initial size of the file cache in 32-bit Windows is 512 MB in Windows Server 2003, but can extend to about 960 MB, all of which is allocated on demand.
- System PTEs, which are necessary to map system pages into physical memory. System PTEs are used to map any virtual memory pages allocated in the system area. The main consumers of PTEs are I/O buffers, including those used by high resolution video cards. The stacks for kernel-mode threads are also allocated from the System PTE pool. The OS earmarks about 660 MB for the pool of System PTEs, but if virtual memory is available the pool maximum can be extended to 900 MB, or more.
- The Nonpaged Pool where data structures accessed by kernel and driver functions that must always be resident in physical memory are stored. Nonpaged pool structures include TCP/IP session-oriented connection data, I/O buffers of all kinds, and kernel stacks (i.e., working storage) for kernel-mode threads. By default, the initial maximum size of the Nonpaged pool that the OS calculates is 256 MB.
- The Paged Pool where data structures that can be paged out are stored. The initial maximum size of the Paged pool that the OS calculates is normally 512 MB, but if virtual memory is available the Paged Pool maximum can be extended to 650 MB, or more.

Operating system and driver resident code and the system's Hyperspace, a work area used by the operating system's Memory Manager, also occupy some amount of system area virtual memory, but these are usually more limited in size.

It is a mistake to think of these four major system pools as being essentially static in size. Adding up the potential size of these four major system memory pools, it is readily apparent that the system's virtual memory requirements would exceed 2 GB, assuming all the virtual memory earmarked for these pools could actually be allocated. At initialization, the operating system calculates preset allocation targets for the size of these shared data areas in the system range, with room left over for several designated overflow areas. These initial allocations are preliminary. Over time, as system virtual memory allocations occur, the OS manages its remaining free memory locations dynamically, servicing memory allocation requests that point to the Nonpaged or Paged pool on a first-come, first-served basis until all free memory in the system range is ultimately allocated. Thus, by design, running out of virtual memory in the system range is a distinct possibility.

When operating system functions exhaust the amount of virtual memory available for System PTEs or data areas in either the Nonpaged or Paged pools, the results are usually catastrophic. When there are no free system PTEs available, the OS will refuse to allocate additional virtual memory within the system range. If either the Nonpaged or Paged pools is exhausted, system functions that need working storage cannot allocate virtual memory. System functions that fail due to a shortage of virtual memory usually result in a system crash - the dreaded blue screen of death in Windows. In contrast, note that a shortage of virtual memory for the file cache is seldom catastrophic, although such a shortage could manifest itself as a severe performance problem.

In the case of extreme virtual memory shortages in the system range, the fatal memory allocation failure is apt to have the appearance of being randomly distributed across any of the stressed data areas, something that also complicates identification and diagnosis of the problem. Troubleshooting the problem will normally require viewing a crash dump, where the evidence of a virtual memory constraint is usually clear and unambiguous.

There are several tuning parameters that are also available to offer "hints" to the operating system in order to handle situations where one specific pool tends to run out of space before the others do. The operating system also provides performance counters that can assist you in setting these configuration parameters. The associated performance counters that track virtual memory usage in the system range are also discussed below.

Terminal Server. Because they require system area storage to support large numbers of User mode processes and threads, large Terminal Server machines are among the most likely workloads to encounter a shortage of system area virtual memory. Each desktop process that Terminal Server executes requires a corresponding kernel mode thread, so the virtual memory area devoted to the pool of System PTEs can become depleted. Another shared system memory construct is the session space where data structures that are shared by desktop processes are stored. Session space is usually of modest size on a typical server. However, Terminal Server machines, where multiple sessions are supported, are a notable exception. In Terminal Server, it is necessary to store global data structures shared by all the processes in

the session in the system area. Session space global data structures include an individual copy of the Win32k.sys kernel mode Windows driver, the session desktop, windows, security tokens, and other objects unique to each User session. Terminal Server machines may also rely on the effectiveness of the system file cache to meet performance requirements.

/3GB boot option. When you use the ***/3GB*** boot option to extend the size of the User private address range to 3 GB, the system area is shrunk in half to 1 GB. This is the other scenario that most frequently leads to virtual memory shortages in the system area.

Virtual memory Commit Limit

The Commit Limit is an upper limit on the total number of virtual memory pages the operating system will allocate. When the system is up against its Commit Limit, no more requests for virtual memory can be honored. This is usually catastrophic for the process that encounters this limitation. Fortunately, this is a straightforward problem that is easy to monitor and easy to anticipate - up to a point. Servers with only 1-2 GB of RAM installed are liable to run up against the virtual memory Commit Limit before the virtual memory constraints imposed by the 32-bit architecture are manifest. Programs with memory leaks, a bug that refers to programs that allocate virtual memory continuously, but fail to release it subsequently, are also apt to be caught by the Commit Limit.

The operating system builds page tables on behalf of each process that is created. A process's page tables get built on demand as virtual memory locations are allocated, potentially mapping the entire virtual process address space range. The Win32 VirtualAlloc API call provides both for reserving contiguous virtual address ranges and committing specific virtual memory addresses. Reserving virtual memory does not trigger building page table entries because you are not yet using the virtual memory address range to store data. Reserving a range of virtual memory addresses is something your application might want to do in advance for a data file intended to be mapped into virtual storage. Only later when the file is being accessed are those virtual memory pages actually allocated (or committed).

In contrast, committing virtual memory addresses causes the Virtual Memory Manager to fill out a page table entry (PTE) to map the address into RAM. Alternatively, a PTE contains the address of the page on one or more paging files that are defined that allow virtual memory pages to spill over onto disk. Any unreserved and unallocated process virtual memory addresses are considered free.

Commit Limit

The Commit Limit is the upper limit on the total number of page table entries (PTEs) the operating system will build on behalf of all running processes. The virtual memory Commit Limit prevents the system from building a page table entry (PTE) for a virtual memory page that cannot fit somewhere in either RAM or the paging files.

The Commit Limit is the sum of the amount of physical memory plus the allotted space on the paging files. When the Commit Limit is reached, it is no longer possible for a process to allocate virtual memory. Programs making routine calls to VirtualAlloc to allocate memory will fail.

Paging file extension

Before the Commit Limit is reached, Windows Server 2003 will alert you to the possibility that virtual memory may soon be exhausted. Whenever a paging file becomes 90% full, a distinctive warning message is issued to the Console. A System Event log message with an ID of 26, illustrated in Figure 5, is also generated that documents the condition.

Following the instructions in the message directs you to the Virtual Memory control (see Figure 6) from the Advanced tab of the System applet in the Control

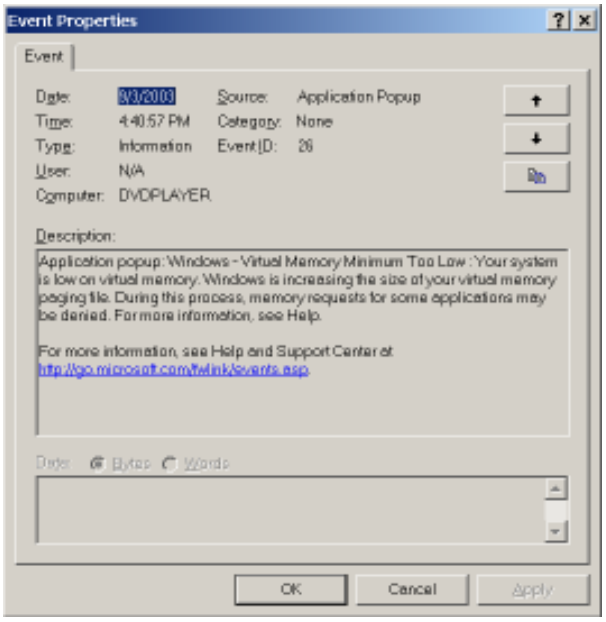

FIGURE 5. OUT OF VIRTUAL MEMORY EVENT LOG ERROR MESSAGE.

Panel where additional paging files can be defined or the existing paging files can be extended (assuming disk space is available and the page file does not already exceed 4 GB).

Windows Server 2003 creates an initial paging file automatically when the operating system is first installed. The default paging file is built on the same logical drive where the OS is installed. The initial paging file is built with a minimum allocation equal to 1.5 times the amount of physical memory. It is defined by default so that it can extend to approximately two times the initial allocation.

The Virtual Memory applet illustrated in Figure 6 allows you to set initial and maximum values that define a range of allocated paging file space on disk for each paging file created. When the system

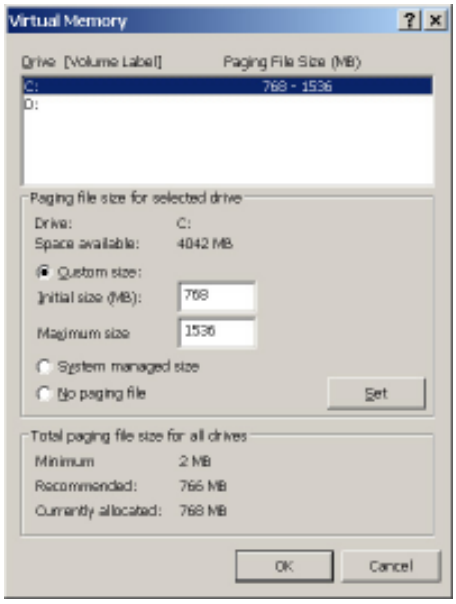

FIGURE 6. THE APPLET FOR CONFIGURING THE LOCATION AND SIZE OF THE PAGING FILES.

appears to be running out of virtual memory, the Memory Manager will automatically extend a paging file that is running out of space, has a range defined, and is currently not at its maximum allocation value. This extension, of course, is also subject to space being available on the specified logical disk. The automatic extension of the paging file increases the amount of virtual memory available for allocation requests. This extension of the Commit Limit may be necessary to keep the system from crashing.

It is possible, but by no means certain, that extending the paging file automatically may have some performance impact. When the paging file allocation extends, it no longer occupies a contiguous set of disk sectors. Because the extension fragments the paging file, I/O operations to disk may suffer from longer seek times. On balance, this potential performance degradation is far outweighed by availability considerations. Without the paging file extension, the system is vulnerable to running out of virtual memory entirely and crashing.

Note that a fragmented paging file is not necessarily always a serious performance liability. Because your paging files coexist on physical disks with other application data files, some disk seek arm movement back and forth between the paging file and application data files is unavoidable. Having a big chunk of the paging file surrounded by application data files may actually reduce overall average seek distances on your paging file disk. [3]

Monitoring virtual memory usage

Performance counters that track overall virtual memory usage are available, as well as ones that allow you to drill down to see the amount of virtual memory allocated by process. These counters make it possible to identify a process that is leaking virtual memory or identify those machines that need more RAM to perform well. Unfortunately, a number of usage issues arise when using these counters to assist in identifying 32-bit server machines encountering virtual memory constraints. Critical gaps in the range of performance monitoring measurement data available make it difficult to anticipate all the varieties of virtual memory shortages that can occur.

One serious problem is that while current allocation levels to the major system pools can be monitored using performance monitoring tools, the maximum sizes of these are only available using debugging tools. Another potential problem is that system level measurements of virtual memory allocations cannot always be reconciled with the process level measurements. A final concern is that the performance counter that tracks the amount of space available for system PTEs provides a measurement that is inconsistent with the other system virtual memory usage counters that are all reported in bytes.

Monitoring Committed Bytes

The Memory\Committed Bytes and Memory\Commit Limit provide a convenient way to determine if your system is running up against the Commit Limit, as illustrated in Figure 7. It shows a 32-bit Windows Server 2003 machine with 4 GB of RAM predominantly configured to run Terminal Server User sessions. As Users activate their sessions in the morning, the number of Committed Bytes allocated increases steadily. Committed Bytes starts to level off after 10:00 AM, at which time there is still plenty of room to accommodate additional virtual memory allocations. The number of Committed Bytes never comes close to exhausting the Commit Limit, which for

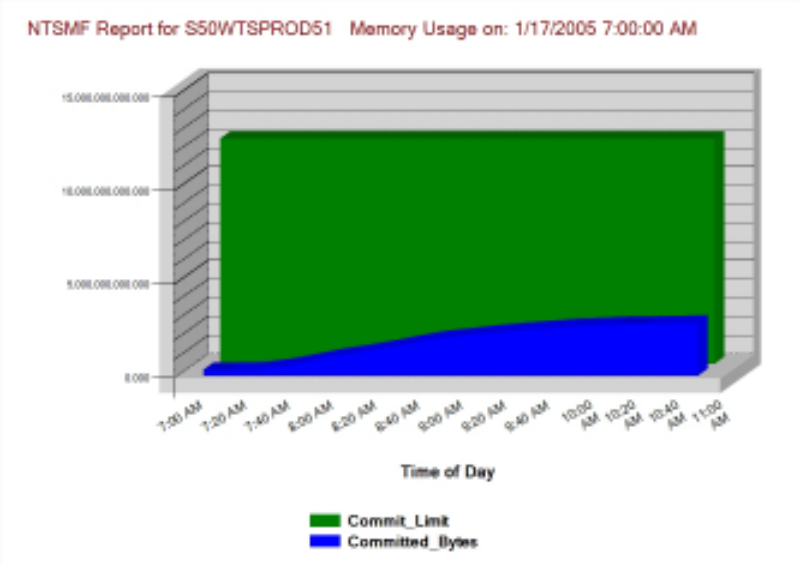

FIGURE 7. MONITORING COMMITTED BYTES.

this system is about 12 GB. The large gap between the upward trending Committed Bytes shown in the foreground and the static Commit Limit behind it is conveniently viewed as capacity headroom for virtual memory growth that should permit this workload to expand.

The situation illustrated in Figure 7 is typical of a server configured with a combination of ample RAM and paging file space where virtual memory growth is not likely to be constrained by the virtual memory Commit Limit. Ironically, because their overall virtual memory growth is unconstrained by the Commit Limit, these are precisely the same machines that can be subject to fatal virtual memory shortages when one of the system memory areas within the 2 GB range allotted to system virtual addresses is exhausted due to the 32-bit addressing constraint.

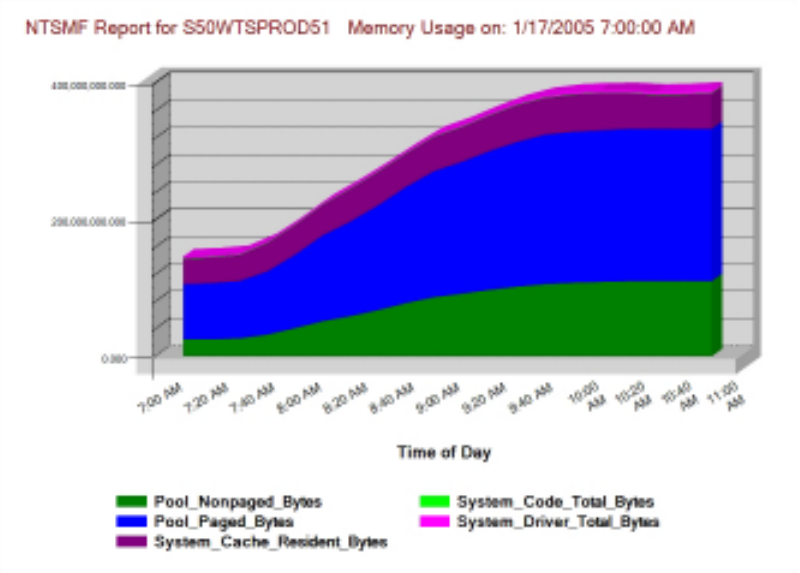

FIGURE 8. MONITORING SYSTEM VIRTUAL MEMORY ALLOCATIONS BY POOL.

Monitoring system memory pool usage

Figure 8 reports the values for five system memory pool allocation counters for the machine illustrated in Figure 7. The following five counters report on system virtual memory allocations:

- Pool Nonpaged Bytes
- Pool Paged Bytes
- System Code Total Bytes
- System Driver Total Bytes
- System Cache Resident Bytes

These five performance counters report the portion of the overall number of Committed Bytes that are associated with virtual memory allocations in the system range.

As illustrated in Figure 8, the number of system code and driver virtual bytes is usually minimal, and can usually be safely dispensed with. The current size of both the Nonpaged and Paged Pools should both be monitored carefully. The Cache Resident Bytes counters tracks the current usage of physical RAM by the file cache function, so it is a physical memory allocation counter that is here grouped with counters that measure virtual memory usage. There is a corresponding physical memory allocation counter called Pool Paged Resident Bytes that reports the number of allocated paged pool bytes that are currently resident in physical memory. Since the pages in Nonpaged Pool are always resident in physical memory, one counter that tracks the size of the Nonpaged pool is sufficient.

In this example machine illustrated in Figure 7 and 8, neither the Nonpaged or Paged pool is approaching its maximum size. Because the Paged Pool and Nonpaged Pools are dynamically re-sized as virtual memory in the system range is allocated, it is not always easy to pinpoint exactly when you have exhausted one of these pools. Fortunately, there is at least one server application, the file Server service, that reports on Paged Pool memory allocation failures when they occur. Non-zero values of the Server\Pool Paged Failures counter indicate virtual memory problems even for machines not primarily intended to serve as network file servers.

Nonpaged and paged pool maximums. Unfortunately, there are no counters available that report the maximum size of the Nonpaged and Paged pools, so even the most careful monitoring procedures leave you exposed to fatal virtual memory shortages striking by surprise. To determine the Nonpaged and paged pool maximum sizes, you need to run the system debugger.

The kernel debugger ***!vm*** command extension provides more detail than the performance counters do in tracking how virtual memory is allocated. You can access the maximum allocation limits of these two pools and monitor current allocation levels, as illustrated in Listing 1.

The NonPagedPool Max and PagedPool Maximum rows show the values for these two virtual memory allocation limits. This information on virtual memory usage is also useful in a post-mortem review of a crash dump to determine if a critical shortage of virtual memory was the source of the problem.

System PTEs Another counter worth tracking in this context is Free System Page Table Entries, which reports the number of available system PTEs to service new allocations. It is necessary to chart the Free System Page Table Entries counter separately from the five system virtual memory usage counters shown in Figure 8 because it measures free space available, rather than usage, as in Figure 10, which is adapted from measurement data supplied by Microsoft from a series of Terminal Server benchmark stress tests.

System PTEs are built and used by system functions to address system virtual memory areas. They

```
lkd> !vm
*** Virtual Memory Usage ***
Physical Memory:    130927   (  523708 Kb)
Page File: \??\C:\pagefile.sys
Current:    786432Kb Free Space:    773844Kb
Minimum:    786432Kb Maximum:      1572864Kb
        Available Pages:     73305   (  293220 Kb)
        ResAvail Pages:      93804   (  375216 Kb)
        Locked IO Pages:       248   (     992 Kb)
        Free System PTEs:   205776   (  823104 Kb)
        Free NP PTEs:        28645   (  114580 Kb)
        Free Special NP:         0   (       0 Kb)
        Modified Pages:        462   (    1848 Kb)
        Modified PF Pages:     460   (    1840 Kb)
        NonPagedPool Usage:   2600   (   10400 Kb)
        NonPagedPool Max:    33768   (  135072 Kb)
        PagedPool 0 Usage:    2716   (   10864 Kb)
        PagedPool 1 Usage:     940   (    3760 Kb)
        PagedPool 2 Usage:     882   (    3528 Kb)
        PagedPool Usage:      4538   (   18152 Kb)
        PagedPool Maximum:  138240   (  552960 Kb)
        Shared Commit:        4392   (   17568 Kb)
        Special Pool:            0   (       0 Kb)
        Shared Process:       2834   (   11336 Kb)
        PagedPool Commit:     4540   (   18160 Kb)
        Driver Commit:        1647   (    6588 Kb)
        Committed pages:     48784   (  195136 Kb)
        Commit limit:       320257   ( 1281028 Kb)
```

LISTING 1. OUTPUT FROM THE !VM DEBUGGER COMMAND

are allocated from a pool that is also used to service allocations for the stacks of any kernel mode threads that exist. When the system virtual memory range is exhausted, the number of Free System PTEs drops to zero and no more system virtual memory of any type can be allocated. On 32-bit systems with large amounts of RAM (1-2 GB, or more), it is also important to track the number of Free System PTEs.

In Figure 9, Free System PTEs is tracked against the current number of active Terminal Server User sessions. Free System PTEs are plotted on a logarithmic scale as the number of active Terminal Server User sessions is increased during the course of this benchmark. When the number of free System PTEs drops below 100, it is no longer possible to add more Users to the system. In fact, it is apparent that some processes and threads associated with currently active Sessions start to fail when the pool where System PTEs are allocated becomes depleted. Processor utilization vs. the number of Users is plotted on the right y-axis to illustrate the progress of the benchmark. [5]

After subtracting the five system virtual memory allocation counters discussed above from Committed Bytes, the remainder consists roughly of virtual memory allocations made by private area process address spaces. In the machine illustrated in Figures 7 and 8, the five system virtual memory allocation counters account for about 400 MB of the 2.84 GB Committed Bytes total reported at the end of the interval. This indicates that various running process address spaces are responsible for allocating the remaining 2.44 GB of virtual memory. Unfortunately, a measurement anomaly associated with accounting for virtual memory allocations that are performed on behalf of shared program components makes it difficult to account for virtual memory allocations on a process by process basis in a straightforward manner. This measurement anomaly at the process address space level is discussed in the next section.

Accounting for process memory usage

When a memory leak occurs, or the system is otherwise running out of virtual memory, it is often useful to drill down to the individual process. There are three counters at the process level that describe how each process is allocating virtual memory. These are Process(n)\Virtual Bytes, Process(n)\Private Bytes, and Process(n)\Pool Paged Bytes.

Process(n)\Virtual Bytes shows the full extent of each process's virtual address space, including shared memory segments that are used to map data files and shareable image file DLLs in memory. The Process(n)\Working Set bytes counter, in contrast, tracks how many pages in RAM that virtual memory associated with the process currently has allocated. An interesting measurement anomaly is that Process(n)\Working Set bytes is often greater than Process(n)\Virtual Bytes. This is due to the way

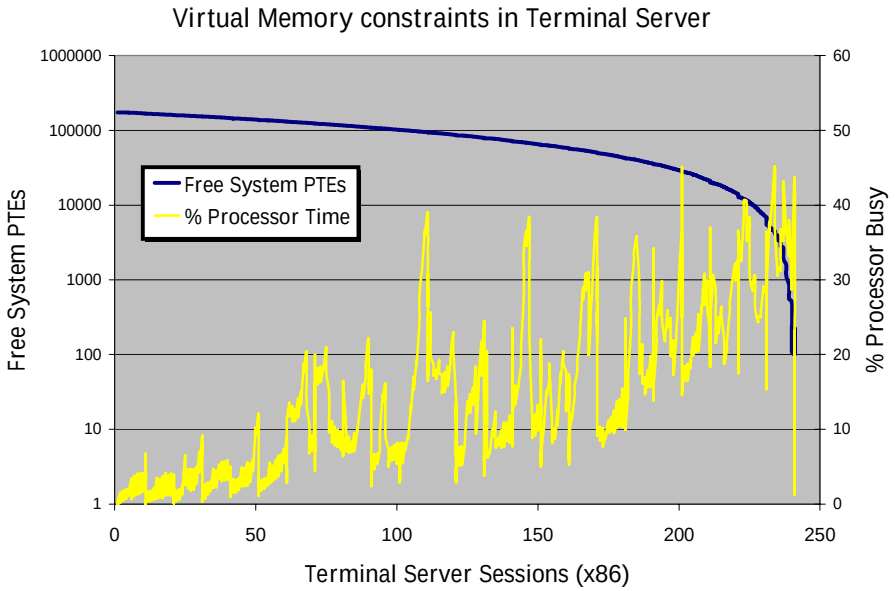

FIGURE 9. FREE SYSTEM PTES IN A VIRTUAL MEMORY-CONSTRAINED TERMINAL SERVER BENCHMARK WORKLOAD.

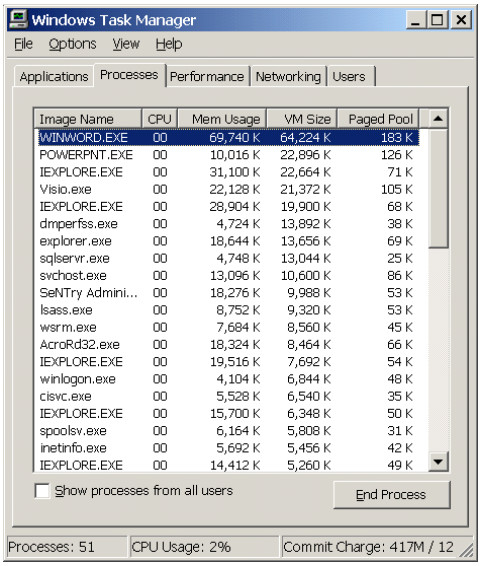

FIGURE 10. WORKING SET BYTES COMPARED TO VIRTUAL BYTES.

memory accounting is performed for shared DLLs, as discussed further below.

If a process is leaking memory, you should be able to tell by monitoring Process(n)\Private Bytes or Process(n)\ Pool Paged Bytes, depending on the type of memory leak. If the memory leak is allocating, but not freeing, virtual memory in the process's private region, this will be reflected in monotonically increasing values of the Process(n)\Private Bytes counter. If the memory leak is allocating, but not freeing, virtual memory in the system range, this will be reflected in monotonically increasing values of the Process(n)\Pool Paged Bytes counter.

Shared DLLs

Modular programming techniques encourage building libraries containing common routines that can be shared easily among running programs. In the Microsoft Windows programming environment, these shared libraries are known as Dynamic Link Libraries, or DLLs, and they are used extensively by Microsoft and other developers. The widespread use of shared DLLs complicates the bookkeeping that is done to figure out both the number of virtual pages and physical memory-resident pages associated with each process working set.

The OS counts all the resident pages associated with shared DLLs as part of every process working set that has the DLL loaded. All the resident pages of the DLL, whether the process has recently accessed them or not, are counted in the process working set. Processes can be charged for resident DLL pages they may never have touched, but at least this double counting is performed consistently across all processes that have the DLL loaded.

Unfortunately, this working set accounting procedure does make it difficult to account precisely for how physical memory is being used. It also leads to a measurement anomaly that is illustrated in Figure 10. For example, because the resident pages associated with shared DLLs are included in the process working set, it is not unusual for a process to acquire a working set larger than the number of Committed virtual memory bytes that it has requested. Notice the number of processes in Figure 10 with more working set bytes (Mem Usage) than Committed Virtual bytes (VM Size). None of the virtual memory Committed bytes associated with shared DLLs are included in the Process Virtual Bytes counter even though all the resident bytes associated with them are included in the Process Working set counter.

Trying to account for virtual memory usage at the process level leads to another measurement anomaly. For the machine shown in Figures 7 and 8, the number of Committed Bytes not associated with system virtual memory was calculated as approximately 2.4 GB. When you add up the number of virtual bytes consumed by all processes - this is readily accomplished by accessing the Process(_Total)\Virtual bytes counter - it will likely exceed the value of system-wide Committed Bytes that is reported. Just as with the Process(*)\Working set counter, it is not possible to break out which processes account for virtual memory allocation consistently due to double counting of shared memory segments.

Figure 11 charts the Process(*)\Virtual Bytes counter for only the 150 largest process virtual addresses for the Terminal Server machine shown previously in Figure 7 and 8 over a one hour period beginning at 10:00 AM. Each bar in the chart represents the virtual byte count of an individual process address space. The number of virtual bytes that are shown allocated at the process level exceeds 10 GB for these 150 active processes, which far exceeds the Committed Bytes measurements for the same interval, which were shown back in Figure 7 to be only about 4 GB. The whopping difference between the expected measurement and the actual ones is disconcerting, to say the least.

Extended virtual addressing

The Windows 32-bit server operating systems support several extended virtual addressing options suitable for large Intel machines configured with 4 GB or more of RAM. These include:

- ***/3GB*** Boot switch, which allows the definition of process address spaces larger than 2 GB, up to a maximum of 3 GB.
- Physical Address Extension (PAE), which provides support for 37-bit physical addresses on Intel Xeon 32-bit processors. With PAE support enabled, 32-bit Intel processors can be configured to address as much as 128 GB of RAM.
- Address Windowing Extensions (AWE), which permits 32-bit process address spaces access to

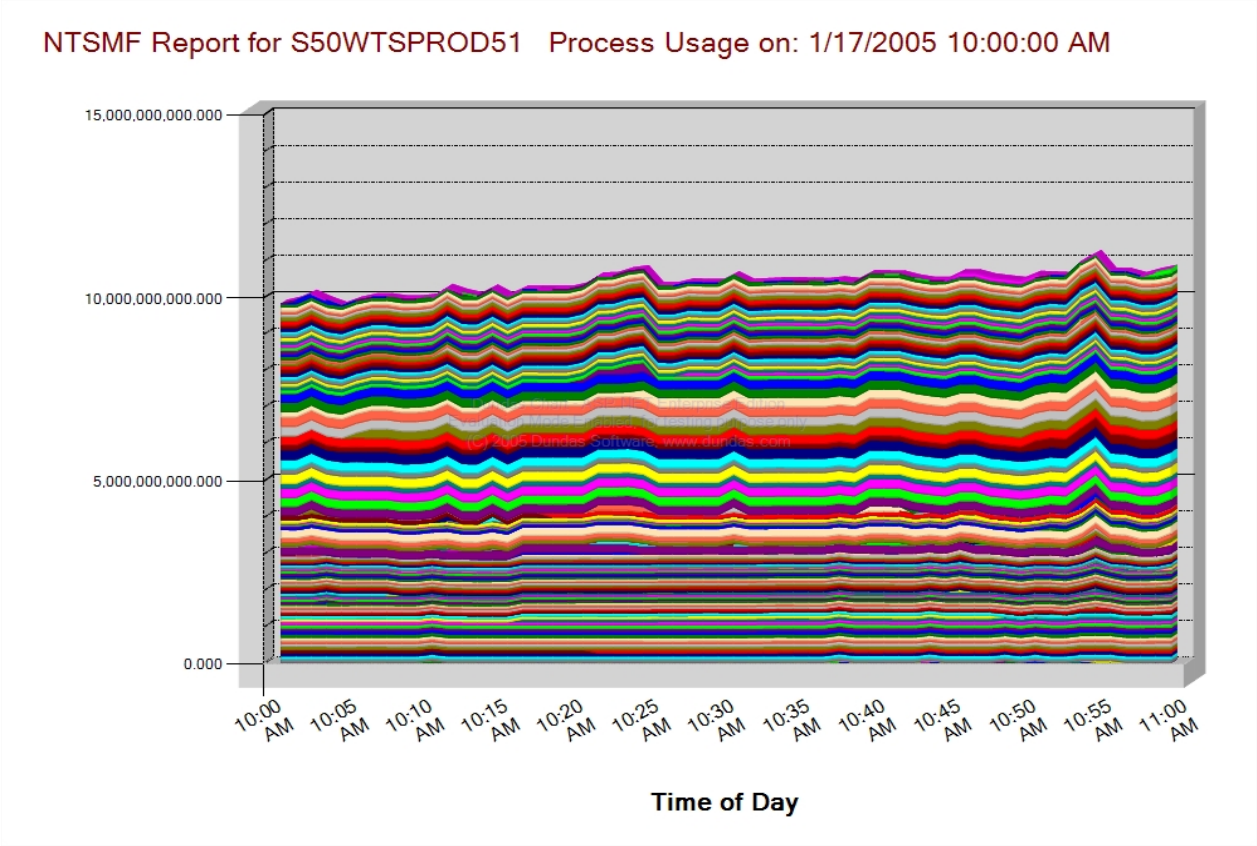

FIGURE 11. THE SUM OF PROCESS(*)\VIRTUAL BYTES COUNTERS CAN FAR EXCEED THE VALUE OF THE MEMORY\COMMITTED BYTES COUNTER SHOWN IN FIGURE 7 FOR THE SAME MACHINE.

physical addresses outside their 4 GB virtual address limitations. AWE is used most frequently in conjunction with PAE

Any combination of these extended addressing options can be deployed, depending on the circumstances. Under the right set of circumstances, one or more of these extended addressing functions can significantly enhance performance of 32-bit applications, which, however, are still limited to using 32-bit virtual addresses. These extended addressing options relieve the pressure of the 32-bit limit on addressability in different ways that are each discussed in more detail below. But, they are also subject to the addressing limitations posed by 32-bit virtual addresses. Ultimately, the 32-bit virtual addressing limit remains a barrier to performance and scalability.

Extended process private virtual addresses

The ***/3GB*** boot switch extends the per process private address range from 2 to 3 GB. Windows Server 2003 permits a different partitioning of user and system addressable storage locations using the ***/3GB*** boot switch. This extends the private User address range to 3 GB and shrinks the system area to 1 GB, as illustrated in Figure 12. The ***/3GB*** switch supports an additional subparameter ***/userva=<SizeInMB>***, where ***SizeinMB*** can be any value for the size of the largest User virtual address between 2048 and 3072.

Only applications compiled and linked with the IMAGE_FILE_LARGE_ADDRESS_AWARE switch enabled can allocate a private address space larger than 2 GB. Applications that are Large Address Aware include MS SQL Server, MS Exchange, MS Internet Information System (IIS), Oracle, and SAS.

For example, there is a database-oriented process called store.exe that is a central component of the MS Exchange Server application. The store.exe process in Exchange corresponds to the MS Exchange Information Store, the component that manages a repository of mail folders, and other messaging content. Like other data-intensive server applications, the store process relies on memory-resident caching of frequently accessed objects to avoid disk I/O and improve performance. The store process will frequently benefit from running with the ***/3GB*** switch on because it can then acquire approximately 1 GB more private virtual memory for its internal data cache. In fact, Microsoft recommends using the ***/3GB*** switch with ***userva=3030*** for any Exchange Server machine with at least 1 GB of RAM. See, for example, KB article 810371 entitled "Using the /Userva switch on Windows Server 2003-based computers that are running Exchange Server" posted on the Microsoft TechNet web site.

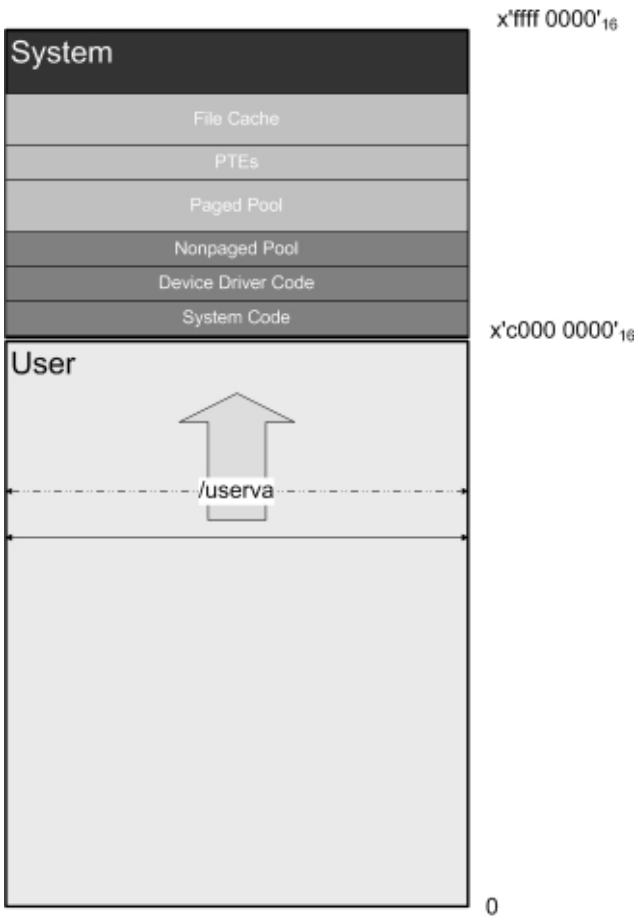

FIGURE 12. THE /USERVA BOOT SWITCH TO INCREASE THE SIZE OF THE USER VIRTUAL ADDRESS RANGE.

Extending the user private area using the ***/3GB*** switch also shrinks the size of the system virtual address range. This can have serious drawbacks. While the ***/3GB*** option allows an application to grow its private virtual addressing range, it forces the operating system to squeeze into a narrower range of addresses. When the ***/3GB*** boot option is employed, the initial default allocations of the four main system memory pools are halved. In certain circumstances, this narrower range of system virtual addresses may not be adequate, and critical system functions can easily be constrained by virtual addressing limits. These concerns are discussed in more detail in the section entitled "Fine-tuning system virtual memory."

PAE

The Physical Address Extension (PAE) enables applications to address more than 4 GB of physical memory. It is supported by most current Intel processors running Windows Server 2003 Enterprise Edition or Windows 2000 Advanced Server. Instead of 32-bit addressing, PAE extends physical addresses to use 37 bits, allowing machines to be configured with as much as 128 GB of RAM.

When PAE is enabled, the operating system builds an additional virtual memory translation layer that is used to map 32-bit virtual addresses into this 128 GB physical address range, as illustrated in Figure 2. PAE utilizes 8-byte PTEs that are wide enough to contain the necessary extra address bits. A ***/PAE*** boot.ini switch informs the operating system to build PTEs in the format required for extended addressing. On newer server machines that support hot-add memory, Windows Server 2003 will boot in PAE mode automatically to ensure continuous operation if physical memory ever increases above the 4 GB upper limit for standard addressing.

Server application processes running on machines with PAE enabled are still limited to using 32-bit virtual addresses. However, 32-bit server applications facing virtual memory addressing constraints can exploit PAE in two basic ways:

1. They can expand sideways by deploying multiple application server processes. Both MS SQL Server 2000 and IIS 6.0 support sideways scalability with the ability to define and run multiple application processes. Similarly, a Terminal Server machine with PAE enabled can potentially support the creation of more process address spaces than a machine limited to 4 GB physical addresses.
2. They can indirectly access physical memory addresses outside their 4 GB limit using the Address Windowing Extensions (AWE) API calls. Using AWE calls, server applications like SQL Server and Oracle can allocate database buffers in physical memory locations outside their 4 GB process virtual memory limit and manipulate them.

The PAE support the OS provides maps 32-bit process virtual addresses into the 37-bit physical addresses that the processor hardware supports. An application process, still limited to 32-bit virtual addresses, need not be aware that PAE is active. When PAE is enabled, operating system functions can only use addresses up to 16 GB. Only applications using AWE can access RAM addresses above 16 GB to the 128 GB maximum. Large Memory Enabled (LME) device drivers can also directly address buffers above 4 GB using 64-bit pointers. Direct I/O for the full physical address space is supported if both the devices and drivers support 64-bit addressing. For devices and drivers limited to 32-bit addresses, the operating system is responsible for copying buffers located in physical addresses greater than 4 GB to buffers in RAM below the 4 GB line than can be directly addressed using 32-bits.

Figure 13 illustrates an expands sideways scenario for SQL Server 2000 where three named instances of the sqlservr.exe process are running concurrently on a machine configured with 12 GB of RAM. When the machine is booted with the ***/3GB*** option, each individual SQL Server instance can expand to use 3 GB of RAM. PAE-mode addressing allows multiple 32-bit 4 GB virtual memory process address spaces to be loaded anywhere in physical

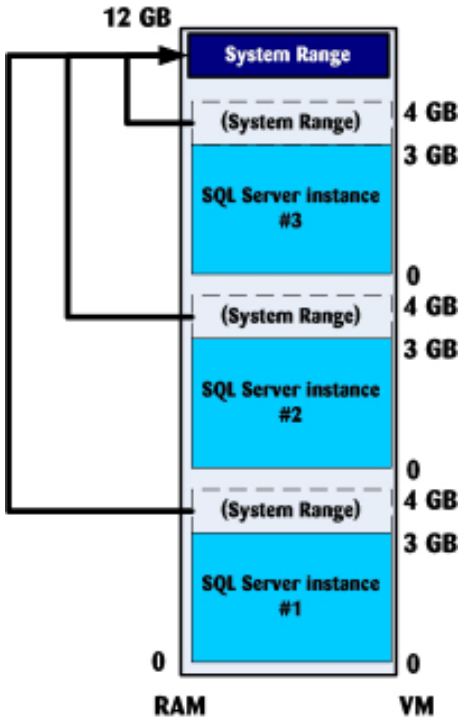

FIGURE 13. EXPANDING SIDEWAYS BY RUNNING MULTIPLE INSTANCES OF 32-BIT SQL SERVER 2000.

memory and execute. Each 4 GB private address space allows for a 1 GB range of virtual memory addresses where operating system code and data structures reside. Each SQL Server process address space is shown pointing to the same common OS pages resident in physical memory.

While expanding sideways by defining more process address spaces is a straightforward solution to virtual memory addressing constraints in selected circumstances, it is not a general purpose solution to the problem. When a single SQL Server database process is virtual memory-constrained running with no more than a 3 GB User-mode private area, extensive application changes may be required to partition the database across multiple instances of SQL Server. Moreover, not every processing task can be readily partitioned into subtasks that can be parceled out to multiple processes. While PAE brings extended physical addressing, there are no additional hardware-supported functions that extend interprocess communication (IPC). IPC functions in Windows rely on operating system shared memory, which is still constrained by 32-bit addressing. Machines with PAE enabled often run better without the ***/3GB*** option because the operating system range is subject to becoming rapidly depleted otherwise.

PAE is required if want to enable the OS support for Cache Coherent Non-Uniform Memory Architecture (known as ccNUMA or sometimes NUMA, for short) machines, but it is not enabled automatically. On both AMD64 and x64-based systems running in long mode, PAE is required, is enabled automatically, and cannot be disabled.

AWE

The Address Windowing Extension (AWE) is an API that allows programs to address physical memory locations outside of their 4 GB virtual addressing range. AWE is used by applications in conjunction with PAE to extend their addressing range beyond 32-bits. Since process virtual addresses remain limited to 32-bits, AWE is a marginal solution that should be deployed cautiously. It has many pitfalls, limitations, and potential drawbacks.

The AWE API calls maneuver around the 32-bit address restriction by placing responsibility for virtual address translation into the hands of an application process. AWE works by defining an in-process buffering area called the AWE region that is used to map allocated physical pages dynamically to 32-bit virtual addresses. See Figure 14 for an illustration of the three step procedure necessary to use AWE, which is similar to managing overlay structures.

1. The first step is to allocate an AWE region using nonpaged physical memory within the process address space by making a call to AllocateUserPhysicalPages in the AWE API. AllocateUserPhysicalPages locks down the pages in the AWE region and returns a Page Frame array structure that is the normal mechanism the operating system uses to keep track of which physical memory pages are mapped to which process virtual address pages. (An application must have the Lock Pages in Memory User Right to use this function.) Initially, of course, there are no virtual addresses mapped to the AWE region.
2. Then, the AWE application reserves physical memory (which may or not be in the range above 4 GB) using a call to VirtualAlloc, specifying the MEM_PHYSICAL and MEM_RESERVE flags. Because physical memory is being reserved, the operating system does not build page tables entries (PTEs) to address these data areas. (Because physical memory addresses are requested, User mode threads in the process cannot directly access these physical memory addresses. (Only authorized kernel threads can.)
3. Next, the process requests that the physical memory that was acquired be mapped to the AWE region using the MapUserPhysicalPages function. Once physical pages are mapped to the AWE region, the region is now addressable by User mode threads running inside the process.

Using the AWE API calls, multiple sets of physical memory blocks, extending to 128 GB, can be mapped dynamically, one at a time, into the AWE region. The application, of course, must keep track of which set of physical memory buffers is currently mapped to the AWE region, what set is currently required to handle the current request, and perform virtual

address unmapping and mapping as necessary to ensure addressability to the right physical memory locations.

In this example of an AWE implementation, the User process allocates four large blocks of physical memory that are literally outside the address space and one AWE region of the same size ***within*** the process virtual address space. Because the physical blocks of RAM are the same size as the AWE region of virtual memory, the AWE call to MapUserPhysicalPages is used one physical address memory block at a time. In the example, the AWE region and the reserved physical memory blocks that are mapped to the AWE region are all the same size, but this is not required. Any sort of overlay structure can be implemented. For instance, there could be more than one AWE region, and each AWE region does not have to be the same size. Nor do physical memory blocks have to match the size of the AWE region. Applications can map multiple reserved physical memory blocks to the same AWE region, provided the AWE region address ranges they are mapped to are distinct and do not overlap.

In this example the User process private address space extends to 3 GB. It is desirable for processes using AWE to acquire an extended private area so that they can create a large enough AWE mapping region to manage physical memory overlays effectively. Obviously, frequent unmapping and remapping of physical blocks would slow down memory access considerably. The AWE mapping and unmapping functions, which involve binding physical addresses to a process address space's PTEs, must be synchronized across multiple threads executing on multiprocessors. Compared to performing physical I/O operations to AWE-enabled access to memory-resident buffers, of course, the speed-up in access times using AWE is still considerable.

AWE limitations. Besides forcing User processes to develop their own dynamic memory management routines, AWE has other limitations. For example, AWE regions and their associated reserved physical memory blocks must be allocated in pages. An AWE application can determine the page size using a call to GetSystemInfo. Physical memory can only be mapped into one process at a time. (Processes can still share data in non-AWE region virtual addresses.) Nor can a physical page be mapped into more than one AWE region at a time inside the same process address space. These limitations are apparently due to system virtual addressing constraints, which are significantly more serious when the ***/3GB*** switch is in effect. Executable code (.exe, .dll, files, etc.) can be stored in an AWE region, but not executed from there. Similarly, AWE regions cannot be used as data buffers for graphics device output. Each AWE memory allocation must also be freed as an atomic

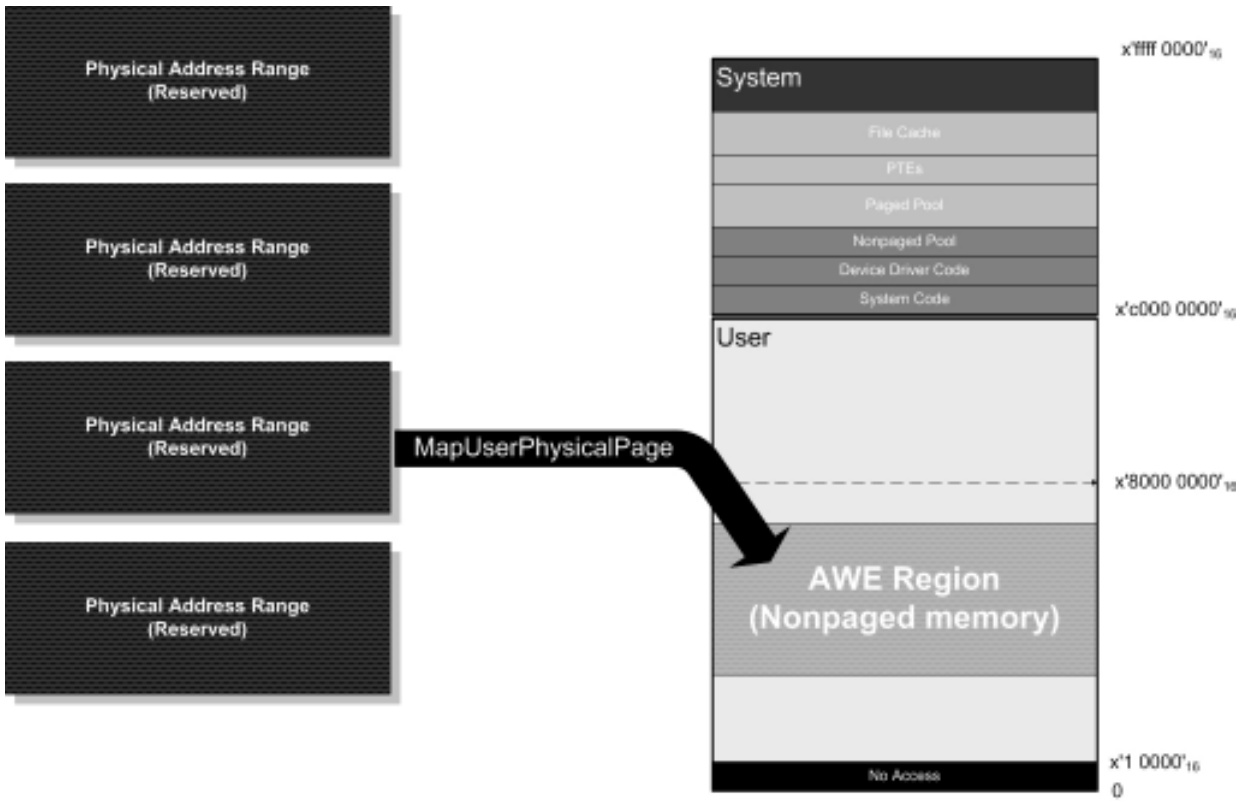

FIGURE 14. AN AWE IMPLEMENTATION ALLOWING THE PROCESS TO ADDRESS LARGE AMOUNTS OF PHYSICAL MEMORY.

unit. It is not possible to free only part of an AWE region.

The physical pages allocated for an AWE region and associated reserved physical memory blocks are never paged out - they are locked in RAM until the application explicitly frees the entire AWE region (or exits, in which case the OS will clean up automatically). Applications that use AWE must be careful not to acquire so much physical memory that other applications can run out of memory to allocate. If too many pages in memory are locked down by AWE regions and the blocks of physical memory reserved for the AWE region overlays, contention for the RAM that remains can lead to excessive paging or prevent creation of new processes or threads due to lack of resources in the system area's Nonpaged Pool.

Application support

Database applications like MS SQL Server, Oracle and MS Exchange that rely on memory-resident caches to reduce the amount of I/O operations they perform are susceptible to running out of addressable private area in 32-bit Windows. These server applications all support the ***/3GB*** boot switch for extending the process private area. Support for PAE is transparent to these server processes, allowing both SQL Server 2000 and IIS 6.0 to scale sideways. Both SQL Server and Oracle can also use AWE to gain access to additional RAM beyond their 4 GB limit on virtual addresses.

Scaling sideways

SQL Server 2000 can scale sideways, allowing you to run multiple named instances of the sqlserver process, as illustrated in Figure 13. A white paper entitled "Microsoft SQL Server 2000 Scalability Project-Server Consolidation" available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql2k/html/sql_asphosting.asp documents the use of SQL Server and PAE to support multiple instances of the database process address space, sqlservr.exe, on a machine configured with 32 GB of RAM. Booting with PAE, this server consolidation project defined and ran 16 separate instances of MS SQL Server. With multiple database instances defined, it was no longer necessary to use the ***/3GB*** switch to extend the addressing capability of a single SQL Server address space.

IIS 6.0. The IIS 6.0 application server architecture supplies perhaps the most flexible set of configuration options available in any Windows server application in the face of 32-bit virtual memory constraints. Figure 15 illustrates the most salient features of the Microsoft Web server architecture that is designed to make the best use possible of potentially scarce virtual memory resources.

IIS 6.0 features a kernel mode driver, http.sys, which is invoked directly whenever conventional HTTP Method calls are received and processed by the TCP/IP networking protocol stack. The HTTP kernel-mode driver eschews the standard system file cache, which is virtual-memory constrained in 32-bit Windows, in favor of a dedicated cache that can be addressed only using the physical memory addresses of the HHTP Response objects stored there. Using physical addressing mode exclusively allows the HTTP Response Cache to grow in size unconstrained by the system's virtual memory addressing limitations. Not only can the HTTP Response Cache grow larger than the 960 MB limit on the system file cache's virtual addressing range, it can do so without utilizing any of the limited range of virtual addresses reserved for other system memory pools. In cases where a fully rendered HTTP Response object is already available in the HTTP Response Cache, IIS 6.0 can respond to a Get Request without ever having to leave kernel mode or suffer a context switch to a User mode processing thread.

Web applications written using either Microsoft's Active Server Pages scripting extensions or the newer ASP.NET runtime facilities cannot be trusted to execute safely in kernel mode. IIS 6.0 supports a new feature called Application Pools (also known as ***Web Gardens***) that queues ASP and ASP.NET requests to a User mode server process called w3wp.exe for execution. Figure 15 illustrates the ASP.NET ISAPI filter that initiates ASP.NET processing inside a w3wp runtime container process. The aspnet_isapi.dll, which is loaded by the ISAPI interface that IIS has supported for several generations, is also shown inside w3wp, along with the .NET Framework Common Language Runtime, mscoree.dll, that links to the extensive set of runtime services the .NET Framework provides. The aspx page containing

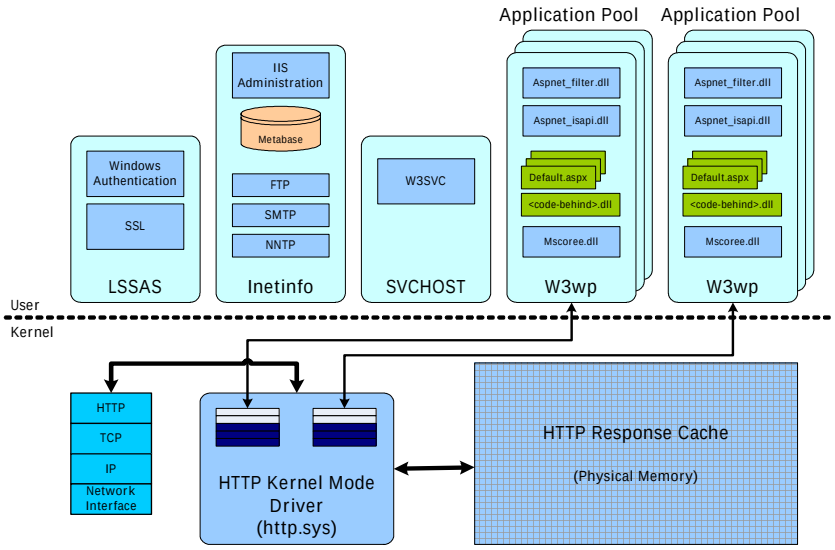

FIGURE 15. THE APPLICATION SERVER ARCHITECTURE OF IIS 6.0.

the HTML that is necessary to render the page properly and its corresponding code-behind executable are also illustrated. (The .NET Framework 2.0 uses a slightly different mechanism than the .NET Framework version 1 that merges the aspx page and the code file at runtime.)

Web sites defined to IIS 6.0 can be assigned to run in separate Application Pools, which correspond to separate copies of the w3wp.exe container process. Additional options are also available at the Application Pool level to devote multiple worker processes to execute the web site programs, as illustrated in Figure 16. This set of multi-process options permits great flexibility in allowing the web application to expand sideways.

Exchange. Microsoft recommends that Exchange 2000 and 2003 use both the ***/3GB*** and ***/userva*** switches in boot.ini to allocate up to 3 GB of RAM for the Exchange Information Store application (store.exe). The store.exe process in Exchange is a database application that maintains the Exchange messaging data store. However, in Exchange 2000 store.exe will not allocate much more than about 1GB of RAM unless you make registry setting changes because the database cache will allocate only 900 MB by default. This value can be adjusted upward using the ADSI Edit tool to set higher values for the msExchESEParamCacheSizeMin and msExchESEParamCacheSizeMax runtime parameters.

Both Exchange 2000 and 2003 will run on PAE-enabled machines. However, Exchange 2000 doesn't make any calls to the AWE APIs to utilize virtual memory beyond its 4GB address space. Exchange 2003 checks the memory management configuration when the store process starts. If the virtual memory settings are not optimal for Exchange, an event 9665 Warning is written to the Application Event log. The logic behind this virtual memory check is discussed in KB article 815372 entitled, "How to optimize memory usage in Exchange Server 2003," along with suggestions for tuning Exchange to run well on machines with 2 GB or more of RAM installed.

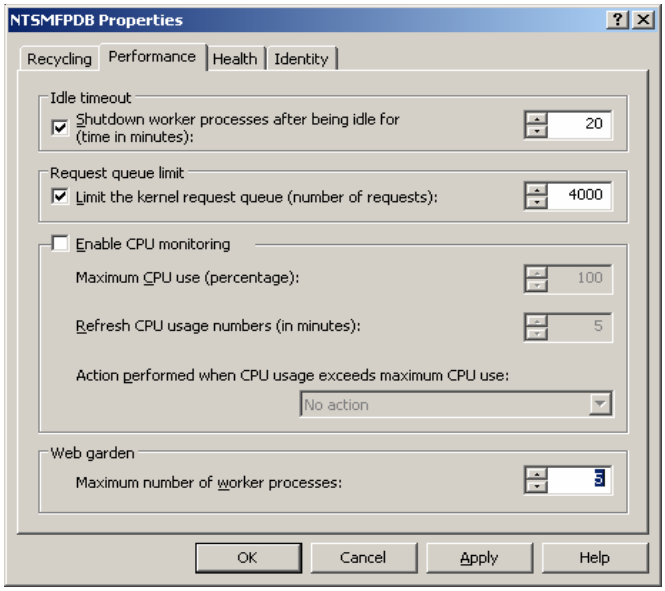

FIGURE 16. MULTIPLE PROCESSES CAN SERVICE ASP AND ASP.NET WEB SITES IN IIS 6.0.

AWE Support

MS SQL Server. Using SQL Server 2000 with AWE brings a load of special considerations. You must specifically enable the use of AWE memory by an instance of SQL Server 2000 Enterprise Edition by setting the ***awe enabled*** option using the ***sp_configure*** stored procedure. When an instance of SQL Server 2000 Enterprise Edition is run with ***awe enabled*** set to 1, The instance does not dynamically manage the working set of the address space. Instead, the database instance acquires nonpageable memory & holds all virtual memory acquired at startup until it is shut down.

Because the virtual memory the SQL Server process acquires when it is configured to use AWE is held in RAM for the entire time the process is active, the ***max server memory*** configuration setting should also be used to control how much memory is used by each instance of SQL Server that uses AWE memory.

Oracle. AWE support in Oracle is enabled by setting the AWE_WINDOW_MEMORY Registry parameter. Oracle recommends that AWE be used along with the ***/3GB*** extended User area addressing boot switch. The AWE_WINDOW_MEMORY parameter controls how much of the 3 GB address space to reserve for the AWE region, which is used to map database buffers. This parameter is specified in bytes and has a default value of 1 GB for the size of the AWE region inside the Oracle process address space.

If AWE_WINDOW_MEMORY is set too high, there might not be enough virtual memory left for other Oracle database processing functions - including storage for buffer headers for the database buffers, the rest of the SGA, PGAs, stacks for all the executing program threads, etc. As Process(Oracle)\Virtual Bytes approaches 3GB, then out of memory errors can occur, and the Oracle process can fail. If this happens, you then need to reduce db_block_buffers and the size of the AWE_WINDOW_MEMORY.

Oracle recommends using the ***/3GB*** option on machines with only 4GB of RAM installed. With Oracle allocating 3 GB of private area virtual memory, the Windows operating system and any other applications on the system are squeezed into the remaining 1GB. However, according to Oracle, the OS normally does not need 1GB of physical RAM on a dedicated Oracle machine, so there are typically several hundred MB of RAM available on the server. Enabling AWE support allows Oracle to access that unused RAM, perhaps grabbing as much as an extra 500MB of buffers can be allocated. On a machine with 4 GB of RAM, bumping up db_block_buffers and

turning on the AWE_WINDOW_MEMORY setting, Oracle virtual memory allocation may reach 3.5 GB.

SAS. SAS support for PAE includes an option to place the Work library in extended memory to reduce the number of physical disk I/O requests. SAS is also enabled to run with the ***/3GB*** and can use the extra private area addresses for SORT work

Fine-tuning System Virtual memory

The upper half of the 32-bit 4 GB virtual address range is earmarked for System virtual memory addresses. The system virtual address range is limited to 2 GB, and using the ***/userva*** boot option it can be limited even further to as little as 1 GB. On a large 32-bit machine, it is not uncommon to run out of virtual memory in the system address range. The culprit could be a program that is leaking virtual memory from the Paged Pool. Alternatively, it could be caused by active usage of the system address range by a multitude of important system functions, including kernel threads, TCP session data, the file cache, and many other normal functions. When the number of free System PTEs reaches zero, no function can allocate virtual memory in the system range. Unfortunately, you can sometimes run out of virtual addressing space in the Paged or Nonpaged Pools before all the System PTEs are used up. When you run out of system virtual memory addresses, whether due to a memory leak or virtual memory creep, the results are usually catastrophic.

The system virtual memory range, 2 GB wide, is divided into four major pools: the System PTEs, the Nonpaged Pool, the Paged Pool, and the File Cache. The size of the four main system area virtual memory pools is determined initially based on the amount of RAM. There are pre-determined maximum sizes of the Nonpaged and Paged Pool, but there is no guarantee that they will reach their predetermined limits before the system runs out of virtual addresses. This is because a substantial chunk of system virtual memory remains in reserve to be allocated on demand - it depends on which memory allocation functions requisition it first. Nonpaged and Paged Pool maximum extents are defined at system start-up, based on the amount of RAM that is installed, up 256 MB for the NonPaged Pool and 512 MB for the Paged Pool. In addition, a 900 MB region is earmarked for the pool where System PTEs are allocation. Meanwhile, the file cache is granted an initial virtual address range of 512 MB.

Using the ***/userva*** or ***/3GB*** boot options that shrink the system virtual address range in favor of a larger process private address range substantially increases the risk of running out of system virtual memory. For example, the ***/3GB*** boot option that reduces the system virtual memory range to 1 GB and cuts the default size of the Nonpaged and Paged Pools in ½ for a given size RAM.

The operating system's initial pool sizing decisions can also be influenced by a series of settings in the HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management key, as illustrated in Figure 17.

Three settings are available, NonpagedPoolSize, PagedPoolSize, and SystemPages, to specify explicitly the size of the Nonpaged Pool, the Paged Pool, and the System PTE pool, respectively. Using the Memory performance monitoring counters, if you are able to determine that the system is encountering a critical shortage of virtual memory in any of these three pools, the appropriate Registry setting can be used to boost the initial size of the pool. Alternatively, you might base your analysis on running the !vm debugger command

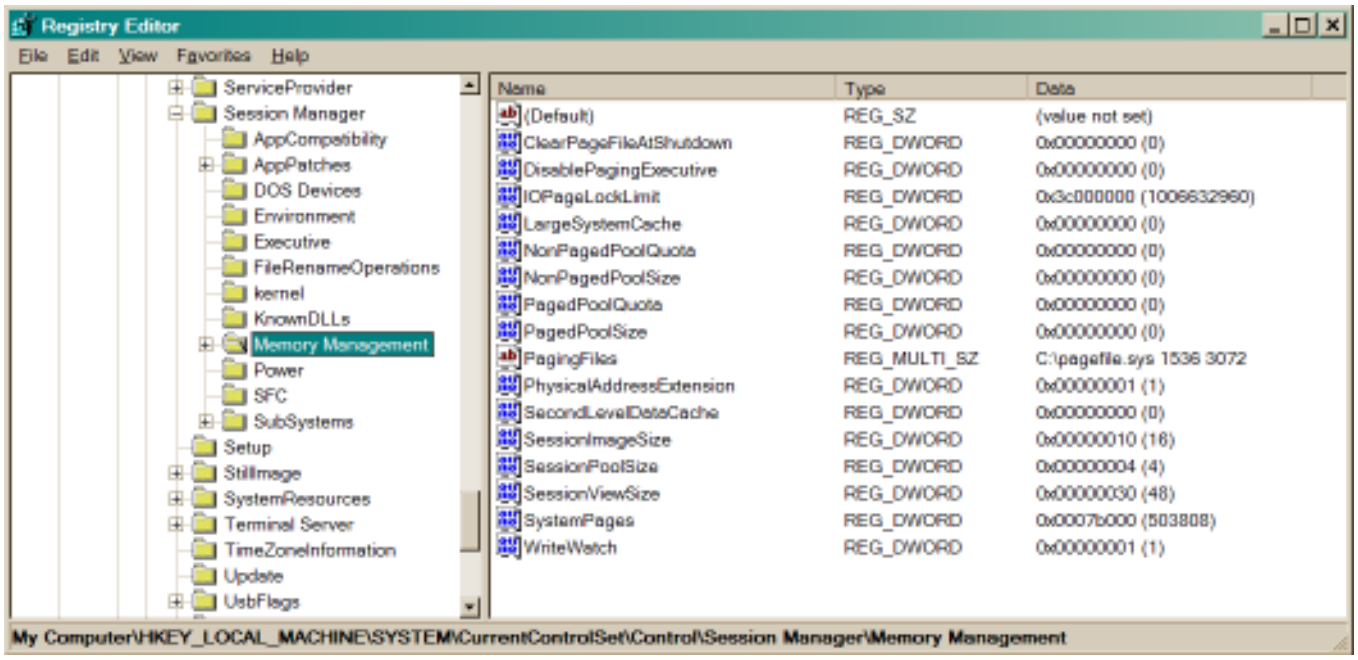

FIGURE 17. MEMORY MANAGEMENT POOL SIZING SETTINGS.

HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management settings		
NonpagedPoolSize	Defaults based on the size of RAM, up to 256 MB.	Can be set explicitly. -1 extends NonpagedPoolSize to its maximum
PagedPoolSize	Defaults based on the size of RAM, up to 650 MB.	Can be set explicitly. -1 extends PagedPoolSize to its maximum
SystemPages	Defaults based on the size of RAM	Can be set explicitly. -1 extends SystemPages to its maximum
LargeSystemCache	960 MB	Zero value allocates minimum sized file cache

TABLE 1. MEMORY MANAGEMENT REGISTRY SETTINGS

against the crash dump of a system stricken with a critical shortage of virtual memory.

Rather than force you to partition the system area precisely, you can also set the SystemPages, NonpagedPoolSize, or PagedPoolSize settings to a -1 value (or 0xffffffff, since the Registry Editor does not allow for signed integers), which instructs the operating system to allow the designated pool to grow as large as possible. Note that setting more than one of the three system memory pool sizing parameters to -1 will backfire because the OS will no longer understand which one of the pools needs to be the largest.

Component	64-bit	32-bit
Process Virtual Address Space	16 TB	4 GB
User Address Space	8 TB	2 GB
Nonpaged Pool	128 GB	256 MB*
Paged Pool	128 GB	512 MB*
System File Cache	1 TB	1 GB
System PTEs	128 GB	660 MB*
Hyperspace	8 GB	4 MB
Paging File Size	512 TB	4 GB

* These are initial sizing requests, not hard limits on the size of the Paged, Nonpaged Pools, and System PTEs in Windows Server 2003.

TABLE 2. A COMPARISON OF VIRTUAL MEMORY ALLOCATIONS IN 64-BIT TO 32-BIT SYSTEMS

A fourth setting, LargeSystemCache, is enabled by default on Windows Server 2003 machines. When LargeSystemCache is enabled, a 512 MB region of virtual memory is initially allocated for the system file cache, which is then allowed to extend into an overflow area of virtual memory to a maximum size of 960 MB, assuming virtual memory is available. Server applications like SQL Server, Oracle, Exchange and even Active Directory rely on their own internal caches, utilizing the system file cache very little. IIS 6.0 relies mainly on its dedicated kernel-mode physical memory cache, but may supply some load on the system file cache. If the main server applications are not using the system file cache, then LargeSystemCache can be set to 0, which minimizes the initial allocation of the system file cache. This allows the OS greater discretion in maximizing one of the other system memory pools.

64-bit addressing

The problems discussed above where 32-bit Windows applications are virtual memory-constrained vanish when 64-bit processors running 64-bit applications are deployed. The 64-bit architectures available today allow massive amounts of virtual memory to be allocated. Windows Server 2003 supports a 16 TB virtual address space for User mode processes running in 64-bit mode. As in 32-bit mode, this 16 TB address space is divided in half, with User mode private memory locations occupying the lower 8 TB and the operating system occupying the upper 8 TB. Table 2 compares the virtual memory addressing provided in 64-bit Windows to 32-bit machines in their default configuration.

On 64-bit systems, all operating system functions are provided in native 64-bit code. The massive amount of virtual memory available to operating system functions eliminates the architectural con-

straints that lead to critical virtual memory shortages in the situations discussed above. The concerns enumerated above simply evaporate. For example, in the Terminal Server example illustrated in Figure 9, space for Free System PTEs dropped to less than 100 when the benchmark workload reached 240 User sessions. When the same Terminal Server workload is moved to an x64 machine, the virtual memory constraint is removed. As a result, the number of Terminal Server User sessions could be increased dramatically, in effect, until some other bottlenecked resource in the configuration was manifest. In the case of this specific workload, Microsoft reported that processor limitations start to become evident when ***double*** the number of Terminal Server User sessions are executed.

The 64-bit machines arriving on the scene offer a modicum of relief to virtual memory-constrained 32-bit applications that are LARGE_ADDRESS_AWARE. Processes running 32-bit code use the thin WOW64 (Windows on Windows) translation layer on 64-bit systems to communicate to operating system services. 32-bit User mode applications that are linked with the IMAGE_FILE_LARGE_ADDRESS_AWARE switch enabled so they could take advantage of the ***/3GB*** switch in 32-bit machines can allocate a private address space as large as 4 GB when running on 64-bit Windows. Note that the ***/3GB*** boot switch is not supported in 64-bit Windows. The larger potential 32-bit User process address space is possible because operating system function no longer need to occupy the upper half of the total 4 GB process address space. This allows User mode applications to allocate almost the full 4 GB 32-bit addressing range.

Running virtual memory-constrained 32-bit applications on 64-bit machines to supply additional virtual memory headroom is particularly attractive when the 64-bit machines can run native 32-bit instructions without exacting a severe performance penalty. That was not the case with the 64-bit Itanium IA-64 machines that were introduced several years ago that are supported by the Microsoft Server 2003 operating system. The IA-64, a radically new processor architecture featuring a new instruction set, executes the IA-32 instruction set using emulation, which results in a severe performance penalty. IA-64 machines are designed to run native 64-bit applications without compromising their performance, with backwards compatibility provided to run 32-bit applications as a secondary consideration. To date, however, there are a very limited number of native 64-bit server applications that could run on these machines. The short list of native 64-bit server applications currently includes IIS 6.0, Microsoft SQL Server, Oracle, and SAP, among others. (Microsoft maintains a far from complete list at http://www.microsoft.com/windowsserver2003/64bit/x64/app64catalog.aspx.)

The need to run 32-bit server-based applications during a long period of transition to 64-bit Windows makes the AMD-64 and Intel x64 architecture an attractive interim choice. (The Intel x64 is a clone of the original AMD-64 design.) Both AMD-64 and x64 machines are capable of executing native 32-bit instructions with little or no performance penalty. Processor-constrained 32-bit applications will suffer a slight performance hit, but virtual memory-constrained server applications like MS Exchange receive significant relief.

References.

[1] Hennessey and Patterson, ***Computer Architecture: A Quantitative Approach***. Morgan Kaufmann, 2002, 3rd edition.

[2] Tracy Kidder, ***The Soul of a New Machine***. Boston: Back Bay Books, 2000.

[3] Mark Friedman and Odysseas Pentakalos, ***Windows 2000 Performance Guide***. O'Reilly Associates, 2002.

[4] Mark Friedman, "Virtual memory constraints in 32-bit Windows," ***Proceedings***, CMG 2003.

[5] Sreenivas Shetty, "Terminal Server Performance: Tuning, Methodologies and Windows Server 2003 x64 Impact." Powerpoint presentation. Microsoft TechEd 2005 conference.