Microsoft Windows

Advances in Memory Management for Windows

October 12, 2007

Abstract
This paper provides information about enhancements in memory management for Windows Vista® and Windows Server® 2008. It describes the changes that Microsoft has implemented internally in the operating system and provides guidelines for application developers, driver writers, and hardware vendors to take advantage of these advances.

Readers should be familiar with the basics of Windows memory management as described in *Windows Internals* by Mark Russinovich and David Solomon.

This information applies for the following operating systems:
Windows Server 2008
Windows Vista

The current version of this paper is maintained on the Web at:
http://www.microsoft.com/whdc/system/cec/MemMgt.mspx

Feedback: Please tell us whether this paper is useful to you. Give us your comments at:
http://connect.microsoft.com/Survey/Survey.aspx?SurveyID=4925&SiteID=221

References and resources discussed here are listed at the end of this paper.

Contents

Introduction 4
About the Memory Manager 4
Virtual Address Space 5
Dynamic Allocation of Kernel Virtual Address Space 5
Details for x86 Architectures 6
Details for 64-bit Architectures 7
Kernel-Mode Stack Jumping in x86 Architectures 7
Use of Excess Pool Memory 8
Security: Address Space Layout Randomization 9
Effect of ASLR on Image Load Addresses 9
Benefits of ASLR 11
How to Create Dynamically Based Images 11
I/O Bandwidth 11
Microsoft SuperFetch 12
Page-File Writes 12
Coordination of Memory Manager and Cache Manager 13
Prefetch-Style Clustering 14
Large File Management 15
Hibernate and Standby 16
Advanced Video Model 16

NUMA Support....17
Resource Allocation....17
Default Node and Affinity....18
Interrupt Affinity....19
NUMA-Aware System Functions for Applications....19
NUMA-Aware System Functions for Drivers....19
Paging....20
Scalability....20
Efficiency and Parallelism....20
Page-Frame Number and PFN Database....20
Large Pages....21
Cache-Aligned Pool Allocation....21
Virtual Machines....22
Load Balancing....22
Additional Optimizations....23
System Integrity....23
Diagnosis of Hardware Errors....23
Code Integrity and Driver Signing....24
Data Preservation during Bug Checks....24
What You Should Do....24
For Hardware Manufacturers....24
For Driver Developers....24
For Application Developers....25
For System Administrators....25
Resources....25

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Introduction

Microsoft has implemented major enhancements in memory management for Windows Vista® and Windows Server® 2008. These changes add features and improve performance in the following areas:

- More efficient use of virtual address (VA) space.
- Stronger security.
- Better usage of I/O bandwidth.
- Faster hibernate/standby and resume.
- Support for the Windows Vista advanced video model.
- Support for nonuniform memory access (NUMA) architectures.
- Better scalability for server hardware and applications.
- Greater system integrity.

Many of the memory management changes are transparent to applications and drivers, so existing code runs without modification. To benefit from some of the changes, however, developers should modify or relink their applications as described in this paper.

About the Memory Manager

The memory manager handles the allocation and management of physical and virtual memory for the operating system. The following are the most important services that it provides:

- Managing key system resources, such as the paged and nonpaged memory pools and system cache.
- Mapping the VA space into physical memory.
- Paging.
- Protecting the address space of processes from each other and from the operating system itself.

The memory manager works with the I/O manager and the cache manager to ensure that processes can quickly access required data, as Figure 1 shows.

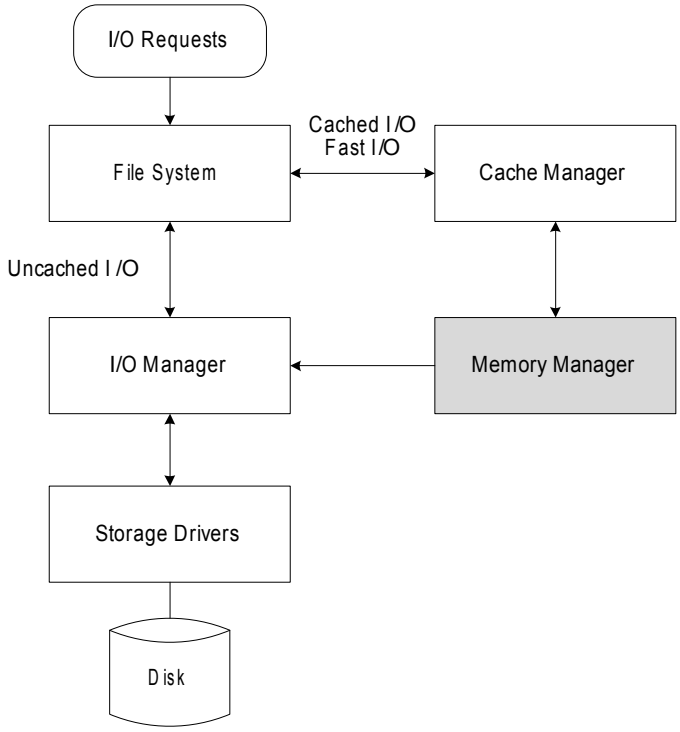

Figure 1. Memory Manager, I/O Manager, and Cache Manager

As Figure 1 shows, the file system receives I/O requests from applications and calls the I/O manager or cache manager to handle them. The I/O manager handles the interactions among devices and applications. Both the I/O manager and applications call the memory manager to map files on behalf of drivers and to allocate memory for internal uses. The memory manager handles page faults as required for any subsequent access to the mapped files. The cache manager provides an interface between the file system and the memory manager for both fast I/O and cached I/O. The cache manager allocates part of the kernel VA space to map views of files based on cached I/O access patterns.

Virtual Address Space

Window Vista implements significant changes to use VA space more efficiently, simplify administration, and improve scalability for greater numbers of processors and larger memory configurations. These changes largely eliminate differences based on registry size, configuration, and stock keeping unit (SKU). The VA space changes include:

- Dynamic allocation of kernel virtual address space
- Kernel-mode stack jumping in x86 architectures
- Use of excess pool memory

Dynamic Allocation of Kernel Virtual Address Space

In Windows Vista and later Windows releases, kernel VA space is dynamically allocated. The sizes and locations of important system resources—including the paged and nonpaged memory pools—are no longer fixed, but instead are

dynamically adjusted according to operational requirements. As a result, system tuning is automatic. Administrators are not typically required to manually reconfigure systems to prevent resource imbalances.

Figure 2 shows the effect of the changes in kernel VA space allocation.

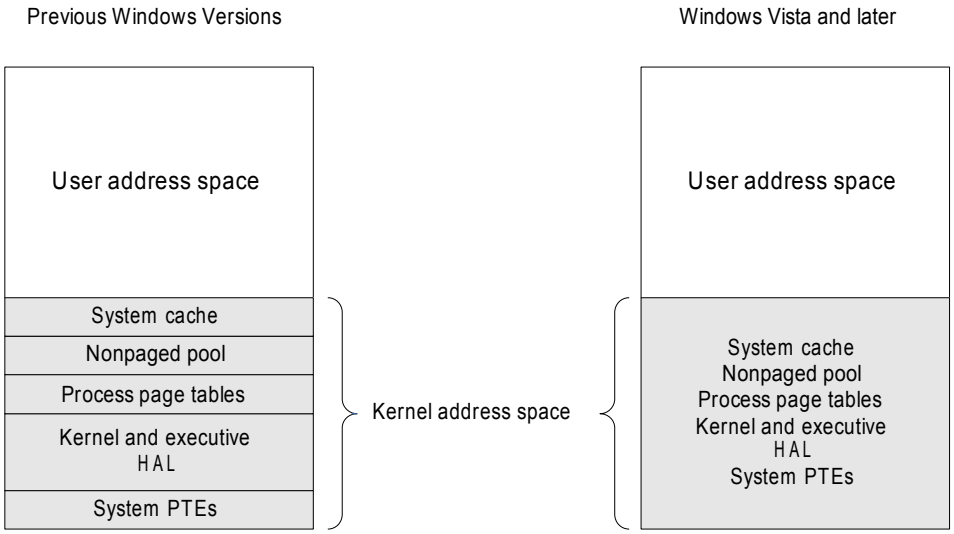

Figure 2. Kernel Virtual Address Space

As Figure 2 shows, in earlier Windows releases, the kernel VA space was allocated statically. Resources were allocated at fixed sizes and locations. Static allocation imposed artificial limits on the sizes of the memory pools, system page tables, and other system resources. The sizes of some resources (such as the memory pools) were set by registry keys or by the SKU.

In Windows Vista and later Windows releases, kernel VA space is allocated dynamically, similar to the user address space. Kernel VA space is limited only by the amount of virtual memory that is available on the architecture. The sizes and locations of individual resources can change according to current system requirements. Thus, system resources no longer reside at fixed locations in the VA space. Windows Vista ignores the values of registry keys that earlier Windows versions used at boot time to determine resource sizes. As a result of dynamic allocation, all resources are available to any requestor.

Features such as special pool and Driver Verifier can be enabled without reboot because the system can dynamically allocate the VA space that they require instead of preallocating it at boot time. In Windows Vista and later Windows releases, such features do not require any memory when they are not in use. Thus, features are "free" if not used, whereas in earlier Windows versions, kernel VA space was allocated at boot time if such features were enabled—even if they were not being used.

Details for x86 Architectures

In 32-bit Windows, the full kernel VA space is shared by all resources that the system is using. Individual resources have no preset size limits, except for nonpaged pool, which is limited to 75 percent of physical memory. A 32-bit system that is booted with the standard configuration has a full 2 gigabytes (GBs) of kernel VA space for the resources to share. An administrator can use the **bcdedit** command to increase the amount of user VA space, thus causing an equivalent reduction in the amount of kernel VA space.

Because system resource allocation is now determined dynamically instead of at boot time, 32-bit systems that are booted with 3 GB of user VA space and physical address extension (PAE) can now use a full 64 GB of memory. Earlier 32-bit Windows versions imposed a limit of 16 GB of memory when booted with 3 GB of user VA space and PAE.

By using a kernel-mode debugger, developers and administrators can obtain information about the kernel address space usage in x86 architectures. The **!vm** kernel-mode debugger extension with the flag value 0x21 displays information about kernel VA space usage without process-specific information. This command can be helpful in inspecting fragmentation of the address space, which can have a greater impact on 32-bit systems than on 64-bit systems because 32-bit systems have significantly less kernel VA space.

In rare circumstances in 32-bit architectures, severe fragmentation can cause exhaustion of the kernel VA space. The 32-bit versions of Windows Vista SP1 and Windows Server 2008 support a set of registry keys with which administrators can limit resource sizes in such systems. These keys are ignored in 64-bit systems. For more information, see “Memory Management Registry Keys” on MSDN®.

Details for 64-bit Architectures

In 64-bit Windows, 128 GB is available for each resource (except for nonpaged pool) along with a 1terabyte (TB) system cache, independent of the amount of physical memory in the machine. Nonpaged pool is limited to 75 percent of physical memory in Windows Server 2008 and 40 percent in Windows Vista and earlier Windows releases.

As an example, consider a 64-bit system with 512 megabytes (MBs) of RAM. In Windows XP, the paged pool in such a system is relatively small, maybe 1 GB. In a system with 1 TB of RAM, however, the paged pool would be much larger—up to 128 GB. In Windows Vista, the paged pool is 128 GB in both the 512MB and 1TB systems, regardless of the physical memory size. Administrators are no longer required to reconfigure the system or change registry settings to run an application that uses a large amount of paged pool.

Kernel-Mode Stack Jumping in x86 Architectures

Windows Vista further increases the availability of kernel VA space in x86 architectures by more efficiently using space for kernel-mode stacks in recursive calls into the kernel. When a thread executes a Windows API system call and thus transitions into the 32-bit kernel, the Windows Vista memory manager provides kernel-mode stack space by generating a new 16-kilobyte (KB) stack that is chained to the first stack. When the thread unwinds from the system call, the memory manager unchains and deletes the stack. This feature, called "kernel-mode stack jumping," reduces the amount of VA space that the thread requires and thus lessens the size of the thread’s memory footprint. Kernel-mode stack jumping results in more efficient memory use by individual Terminal Server clients and therefore typically enables two to four times more clients to run on each machine.

Figure 3 shows how Windows Vista differs from earlier Windows versions in the use of kernel-mode stack space in x86 architectures.

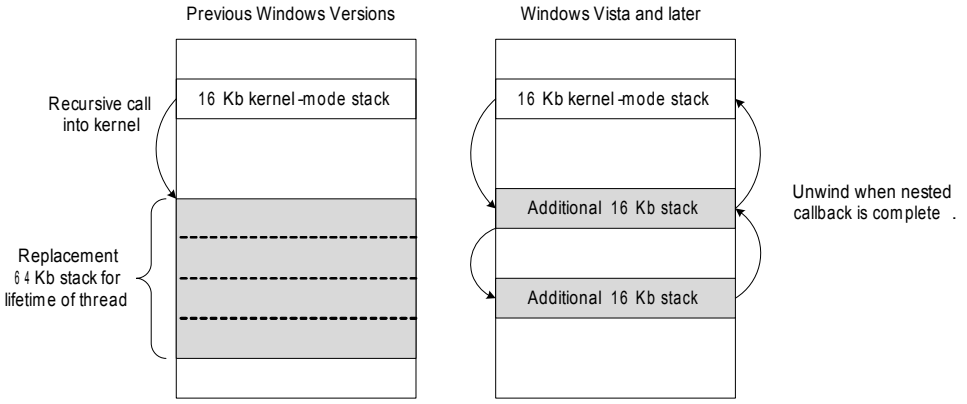

Figure 3. Kernel-Mode Stack Jumping in 32-bit Windows Vista

In earlier Windows versions, a thread that executes a system call receives an expanded, 64KB virtual stack (although only 12 KB, with a 4KB guard page, is usable for each system call). The thread retains this stack until it terminates, regardless of whether it continues to use the stack space. For such threads, the additional space is not used when the thread is not running kernel-mode code.

In 32-bit versions of Windows Vista, a thread receives an additional 16KB stack each time Win32k.sys issues a callback that results in a transition to the kernel. When each nested system callback is complete, the memory manager deletes the associated stack.

Drivers that require additional kernel-mode stack space can take advantage of this feature by using the **KeExpandKernelStackAndCallout** function. This function enables a driver to specify the amount of stack space—up to the value defined in Ntddk.h as MAXIMUM_EXPANSION_SIZE—with which the system calls a particular callback function. If the current stack does not have enough space, the system allocates one or more additional stacks, as required, and deletes them when the specified driver callback returns. **KeExpandKernelStackAndCallout** is supported in Windows Server 2003 for 64-bit architectures and in Windows Vista for all architectures.

Use of Excess Pool Memory

When a kernel-mode component allocates more than a page of memory, the memory manager now uses the pool memory between the end of that allocation and the next page boundary to satisfy other memory requests.

Drivers must not access memory beyond the end of any allocation. Driver Verifier has checked for this error for many years, so existing drivers that have been properly tested should not encounter any problems. Driver writers should nevertheless be aware of the change. A driver that uses memory beyond the end of its allocation can corrupt the contents of memory that is allocated to another component—or even to itself.

Figure 4 shows how the Windows Vista allocation of pool memory differs from that of earlier Windows releases.

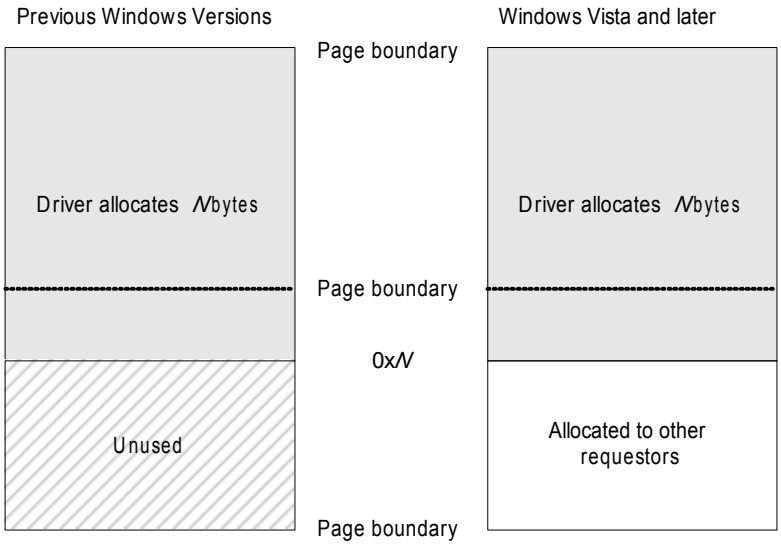

Figure 4. Excess Pool Allocation

In Figure 4, a driver allocates *N* bytes, where *N* is more than a page but less than a multiple of the page size. In earlier Windows releases, the memory beyond that allocation to the next page boundary is wasted. In Windows Vista and later releases, however, the memory manager can allocate such memory to other requestors.

Security: Address Space Layout Randomization

To improve system security and limit the damage that a buffer overrun exploit could cause, Windows Vista loads DLLs and executable images at a different address each time they are loaded. This technique, called address space layout randomization (ASLR), makes it much more difficult for rogue applications to predict the location of certain system APIs.

Developers must opt in to ASLR support for DLLs and executables by using the /**DYNAMICBASE** linker option. This option sets a flag in the image header that indicates that the image can be loaded at a randomly chosen address. Independent software vendors (ISVs) are strongly encouraged to opt in to ASLR for their products.

Effect of ASLR on Image Load Addresses

In Windows Vista, the memory manager at startup randomly selects an image-load bias from one of 256 64KB-aligned addresses at the top of the user-mode address space. The image-load bias is the starting address of a region into which the memory manager loads DLLs that support ASLR—that is, DLLs that are flagged for dynamic-base loading. The same image-load bias applies to all user-mode processes on the system, so that processes can share dynamically-based DLLs.

In earlier Windows versions, the memory manager tried to load DLLs at the same location each time in all processes, using the load address specified in the DLL header (assuming no address space collisions occur within the process). Consequently, when hackers discovered buffer overrun bugs in Microsoft or third-

party products, they were able to use the known linked address of a particular function in one of these images to launch an attack.

Figure 5 shows the effects of ASLR on load addresses in a hypothetical 32-bit Windows Vista system.

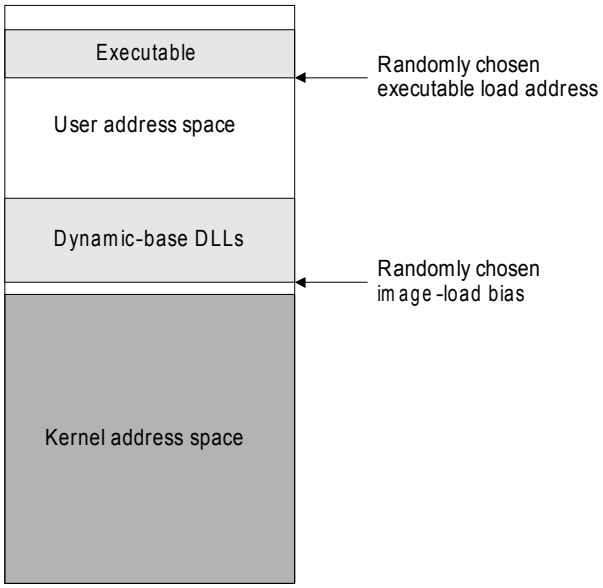

Figure 5. Effect of ASLR on Load Addresses

In Figure 5, the memory manager loads the first dynamic-based DLL at the randomly chosen image-load bias and then works its way up through the address space to assign addresses for the remaining dynamic-based DLLs. DLLs that are linked with /**DYNAMICBASE** are loaded at the same address across all processes that use them, thus enabling code sharing among those processes, assuming that adequate VA space is available and that no VA space collisions occur in the processes. For executable images, the memory manager follows a similar pattern, selecting a randomly chosen 64KB-aligned address that is near the base load address that is stored in the image header. Processes can share executables in the same way that they share DLLs.

If a DLL or executable image is unloaded by all the processes that are using it, the memory manager does not necessarily load it at the same address the next time it is required. Instead, the memory manager again randomly chooses a load address.

Windows Vista loads DLLs that do not support ASLR in the same way as earlier Windows versions did. Specifically, the system attempts to load the image at the address specified in the image header. If another DLL is already loaded at that location, thus causing a collision, the system starts at the highest available address in the process and works down until it finds a place at which it can load the image. If additional processes load the same DLL, the system loads the image at the same address in those processes. If collisions occur in the subsequent processes, the system relocates the preloaded DLL, thus potentially causing a "domino effect" as DLLs load.

Benefits of ASLR

ASLR increases security by eliminating the predictability of load addresses. In addition, it packs DLLs into a smaller range of VAs than in earlier Windows versions, thus avoiding collisions, saving page table space, and leaving more contiguous VA space for applications. Dynamic-based DLLs continue to be shared among the processes that use them.

ASLR applies to user-mode executables and Microsoft-supplied DLLs in Windows Vista and Windows Server 2008 and extends to the kernel, hardware abstract layer (HAL), and driver components in Windows Server 2008. Drivers are automatically dynamically relocated, but user executables and DLLs must explicitly opt in by using the **/DYNAMICBASE** linker option.

ASLR, when combined with no-execute protection, provides strong security against malware that attempts to exploit application vulnerabilities such as buffer overflows. By itself, ASLR is a relatively weak security measure against buffer overruns. A hacker who cannot determine the address of a system routine can inject and execute code off the stack. No-execute protection alone is similarly weak: it prevents code execution but does not prevent a hacker from using the known address of a system function. Used together, however, ASLR and no-execute are very strong. A hacker can neither execute rogue code off the stack nor off a known address in a DLL.

In general, ASLR has no impact on performance because it is implemented at the kernel memory-management level. In 32-bit systems, a small improvement in performance may occur because of code sharing and more efficient use of the address space. However, degradation could occur in highly congested 32-bit systems that are loaded with many random images, depending on the quantity and size of the images.

How to Create Dynamically Based Images

By default, ASLR applies to all system DLLs and EXEs, which are all linked with /DYNAMICBASE. ISVs are strongly encouraged to opt into support for ASLR. To create a dynamically based image, developers should relink their applications with the **/DYNAMICBASE** option, which is available in the Microsoft Linker (version 8.00.50727.161 or later) and supported in Microsoft® Visual Studio® 2005 SP1. This switch is ignored by earlier Windows versions, so no backwards compatibility problems arise.

To use both ASLR and no-execute, developers should use the /**NXCOMPAT** flag in addition to /**DYNAMICBASE**.

For more information about ASLR and no-execute protection, see "Windows Vista ISV Security" on MSDN.

I/O Bandwidth

Processor speeds and memory sizes have increased by several orders of magnitude over the last few years, but I/O and disk speed have perhaps only doubled (although recent improvements such as solid-state drives are beginning to change that). Many of the processor and memory improvements are not fully utilized because the I/O subsystem simply cannot keep up. Improving I/O speed was an important challenge for Windows Vista and involved changes in the following areas:

- Microsoft SuperFetch™
- Page-file writes

- Coordination of memory manager and cache manager
- Prefetch-style clustering
- Large file management
- Hibernate and standby

Microsoft SuperFetch

SuperFetch is an adaptive page prefetch technique in Windows Vista that analyzes data usage patterns and preloads data into memory according to those patterns.

As end users go about their business, the memory manager logs information about their activities. A user-mode SuperFetch service analyzes the log to find patterns in usage and then prioritizes the commonly used pages on a system-wide basis. The service then calls the memory manager to preload those pages at the appropriate time based on their regular use. The memory manager preloads at low priority when the processor is not being used for other work and the disk is not busy.

For example, consider a system that is regularly used to run payroll at noon on Fridays. The memory manager logs information about the pages that are required for the payroll application, and the SuperFetch service recognizes that these pages are used regularly and thus prioritizes them for loading around noon. The result is more efficient use of virtual memory and better performance for the end user.

For more information on SuperFetch, see "Windows PC Accelerators." SuperFetch is not supported in Windows Server 2008.

Page-File Writes

Windows Vista includes numerous enhancements that make writing to the page file faster and more efficient. In Windows Vista, the memory manager writes more data in each operation, aligns pages with their neighbors, and does not write pages that are completely zero.

In earlier Windows versions, the size of write operations was limited to 64 KB. Consequently, the memory manager could write a maximum of 64 KB to the page file in a single operation. Windows Vista removes the 64KB limit, so the memory manager can write the page file in much larger clusters. Write operations are now typically 1 MB.

Earlier algorithms were designed to ensure that data was written to disk as quickly as possible. In Windows Vista, however, the goal is to write smart as well as fast. Moving the disk heads to read and write data is a relatively time-consuming operation, so reducing the number of such operations can help improve I/O bandwidth. When the memory manager prepares to write a modified page to its backing store, it also inspects nearby pages to determine whether any of them also need to be written, regardless of whether they have been unmapped or trimmed from the working set. A write operation thus contains a cluster of neighboring pages—some of which are still in a working set and some of which are not. By clustering writes in this way, the memory manager helps to increase the contiguity of data on disk and thus speed up subsequent page-in requests for the same data.

In addition, the memory manager now checks for zeroed pages before it reads or writes. Internal traces showed that in some situations 7 to 8 percent of write operations involved pages that were completely zero. Instead of using valuable I/O bandwidth to write and potentially read such pages, the memory manager simply puts the page on the zero-page list and marks the page as demand-zero in the page table. If the corresponding VA is subsequently accessed, the memory

manager uses a zero page from memory instead of reading in a zero page from disk.

Windows Server 2008 and Windows Vista SP1 conserve additional I/O bandwidth by delaying modified page writes on systems with adequate physical memory. Instead of using a predetermined modified ("dirty") page threshold, these Windows releases write modified pages based on an algorithm that considers the amount of physical memory that is available and the current power state of the disk. This approach requires fewer disk I/O operations and can extend laptop battery life by powering up the disks on laptops less often.

Coordination of Memory Manager and Cache Manager

Windows Vista supports greater coordination between the memory manager and the cache manager for write operations. The memory manager and the cache manager write modified pages to their backing store by using worker threads. The memory manager uses two such threads: the modified page writer thread and the mapped page writer thread. The cache manager uses several lazy-writer threads, which periodically queue dirty pages from the system cache to be written to their backing store. Windows Vista coordinates the actions of these threads to optimize seek operations, reduce latency, and minimize duplication of I/O operations. Additional enhancements provide parallel operation and avoid the use of locks whenever possible.

The modified page writer thread writes dirty pages from memory to the paging files when required by a combination of the number of modified pages, the number of available pages, and the reuse of existing cached pages.

The mapped page writer thread writes dirty pages from mapped files out to their backing store at timed intervals. In Windows Vista and later Windows releases, the mapped page writer sweeps through the dirty page list at regular intervals and flushes all the pages out to their backing store. If the number of free pages is low, the system accelerates the sweep by using a shorter interval. In earlier Windows releases, the mapped page writer flushed everything at absolute 5-minute intervals. Windows Vista writes dirty pages sooner than earlier Windows releases and the write operations typically involve less data.

The cache manager’s lazy-writer threads periodically queue dirty pages from the system cache to be written to their backing store. The lazy-writer threads are coordinated with the mapped page writer thread to issue write requests to the file system in an orderly way. Specifically, the system writes pages to their backing store in linear order whenever possible, from the first page to the last page, instead of writing the dirty pages based on their order in the modified-page list. This approach is markedly more efficient for large, sparse files.

Consider a disk file that has been extended in length from 1 MB to 200 MB. Some of the new pages contain data, and some do not. In earlier Windows versions, a page near the end of the file—for example, at 180 MB—might be written first. The file system would zero-fill the disk blocks from 1 MB to 180 MB to prevent file corruption in the event that the system crashed before data from additional modified pages could be written to the intervening blocks. Then the file system would write additional blocks in whatever order the requests arrived from the mapped page writer thread and the lazy-writer threads, working independently. In Windows Vista, the mapped page writer and lazy-writer threads coordinate to write all the modified pages in order from 1 MB to 180 MB. All of these pages would have to be written eventually anyway. Writing them in linear order moves the read/write heads on the disk more efficiently and avoids the time-consuming action of zero-filling the intervening blocks for which modified pages are already available.

The memory manager now also supports multiple asynchronous flushes to improve performance. The writer threads issue multiple overlapping write requests for all the data and then wait for all such requests to complete. In earlier Windows versions, the threads issued the requests serially, one at a time. The use of asynchronous requests enables the write operations to occur in parallel where possible. Meanwhile, the writer threads continue to run at processor and memory speed instead of being tied to I/O speeds for each individual write request.

Prefetch-Style Clustering

The memory manager prefetches large clusters of pages to satisfy page faults and populate the system cache. The prefetch operations read data directly into the system's page cache instead of into a working set in virtual memory, so the prefetched data does not consume VA space and the size of the fetch operation is not limited to the amount of VA space that is available. The prefetched pages are put on the standby list and marked as in transition in the page table entry (PTE). If a prefetched page is subsequently referenced, the memory manager adds it to the working set. However, if it is never referenced, no system resources are required to release it.

If any pages in the prefetched cluster are already in memory, the memory manager does not read them again. Instead, it uses a dummy page to represent them, as Figure 6 shows.

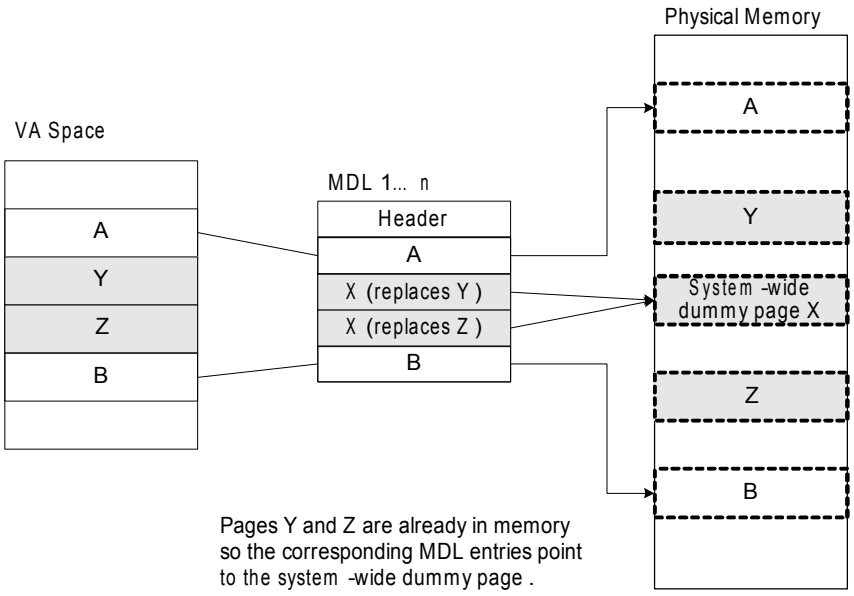

Figure 6. Prefetch-style clustering

In Figure 6, the file offsets and VAs that correspond to pages A, Y, Z, and B are logically contiguous although the physical pages themselves are not necessarily contiguous. Pages A and B are nonresident, so the memory manager must read them. Pages Y and Z are already resident in memory, so it is not necessary to read them. (In fact, they might already have been modified since they were last read in from their backing store, in which case it would be a serious error to overwrite their contents.) However, reading pages A and B in a single operation is more efficient than performing one read for page A and a second read for page B. Therefore, the memory manager issues a single read request that comprises all four pages (A, Y, Z, and B) from the backing store. Such a read request includes as many pages as

make sense to read, based on the amount of available memory, the current system usage, and so on.

When the memory manager builds the memory descriptor list (MDL) that describes the request, it supplies valid pointers to pages A and B. However, the entries for pages Y and Z point to a single system-wide dummy page X. The memory manager can fill the dummy page X with the potentially stale data from the backing store because it does not make X visible. However, if a component accesses the Y and Z offsets in the MDL, it sees the dummy page X instead of Y and Z.

The memory manager can represent any number of discarded pages as a single dummy page, and that page can be embedded multiple times in the same MDL or even in multiple concurrent MDLs that are being used for different drivers. Consequently, the contents of the locations that represent the discarded pages can change at any time.

Prefetch-style clustering can affect the operation of those few drivers that directly reference pointers in the MDL to read data. Driver writers should make no assumptions about the order or contents of pages that are described by an MDL and must not rely on the value of the data at any location that is referenced by an MDL. In general, most drivers never directly reference a memory location in an MDL to get the data, so this restriction affects only a few drivers.

Drivers that perform decryption or calculate checksums that are based on the value of data in the pages that the MDL maps must not reference pointers from the system-supplied MDL to access the data. Instead, to ensure correct operation, such a driver should create a temporary MDL that is based on the system-supplied MDL that the driver received from the I/O manager. The driver tip "What Is Really in That MDL?" outlines the procedure that such drivers should follow.

If the MDL describes a buffer for a direct I/O write operation, the application that issued the I/O request might also have mapped a view of the same pages into its address space. If so, the application could modify the data at the same time that a driver examines it. Drivers must handle this situation appropriately, by creating a temporary MDL through which to double-buffer the contents and see a snapshot of the data.

Drivers that use MDLs as part of a typical I/O operation without accessing the data in the underlying MDL pages are not required to create a temporary MDL. Internally, the memory manager keeps track of all the pages that are resident and how each is mapped. When a driver passes an MDL to a system service routine to perform I/O, the memory manager ensures that the correct data is used.

Prefetch-style clustering was originally introduced in Windows XP, where it was used in a few places. Windows Vista uses prefetch-style clustering pervasively throughout the operating system, providing greater performance benefits.

Large File Management

Windows Vista provides better I/O performance for operations on large and sparse files. The Windows Vista memory manager describes file ranges by using an Adelson-Velsky/Landis (AVL) tree to describe the disk blocks that a file spans. An AVL tree is a self-balancing binary tree, which provides for more efficient operations than a linked list, which earlier Windows releases used. The linked list required a linear walk through all the sections in the file.

An AVL tree greatly increases the speed of system functions that use file offsets to map, flush, and purge large files. Consequently, backups are now typically twice as fast as they were in Windows XP.

Hibernate and Standby

For Windows Vista and later Windows releases, hibernation and standby are faster and more efficient than in earlier Windows versions. The system now performs hibernation and standby in the following two steps:

1. Copy the contents of physical memory to the hibernate file on disk. All device stacks are active.
2. Shut down all device stacks except those on the hibernation path. Copy only changed data to the hibernate file.

Hibernation and standby now use the same memory management mirroring technique that is used in fault-tolerant systems. The memory manager copies the contents of physical memory to disk while all device stacks are active. Therefore, the copy operation can take advantage of larger I/O sizes, scatter/gather direct memory access (DMA), and other advanced, efficient I/O techniques to save data for hibernation. Such techniques include prefetch support, so that the pages required to resume quickly are read into memory if needed and then included in the hibernation file.

After the initial copy operation is complete, the system shuts down all the device stacks except those on the hibernation path. It then copies to the hibernate file only the data that has changed since the first copy operation. Hibernation no longer purges the page cache, as in earlier Windows releases. Instead, the system writes cached data to the hibernate file intelligently based on what is being used. It also clusters the data so that each write request is about twice as large as in earlier Windows releases. Consequently, hibernation and standby are much faster and users are no longer required to understand how they differ. For equivalent system snapshots, overall hibernation time is about twice as fast as in earlier Windows releases and the hibernation file is about half the size.

Advanced Video Model

The new video architecture in Windows Vista and later Windows releases more fully uses modern graphics processing units (GPUs) and virtual memory to provide more realistic shading, texturing, fading, and other video effects for gaming and simulations.

To support the video architecture, the memory manager provides a new mapping type called rotate virtual address descriptors (VADs). Rotate VADs enable the video drivers to quickly switch user views from regular application memory into cached, noncached, or write-combined accelerated graphics port (AGP) or video RAM on a per-page basis, with full support for all cache attributes. In this way, the video architecture can transfer data directly by using the GPU for higher performance and rotate unneeded pages in and out on demand. Figure 7 shows how a VA can map to a page in either regular physical memory or graphics memory.

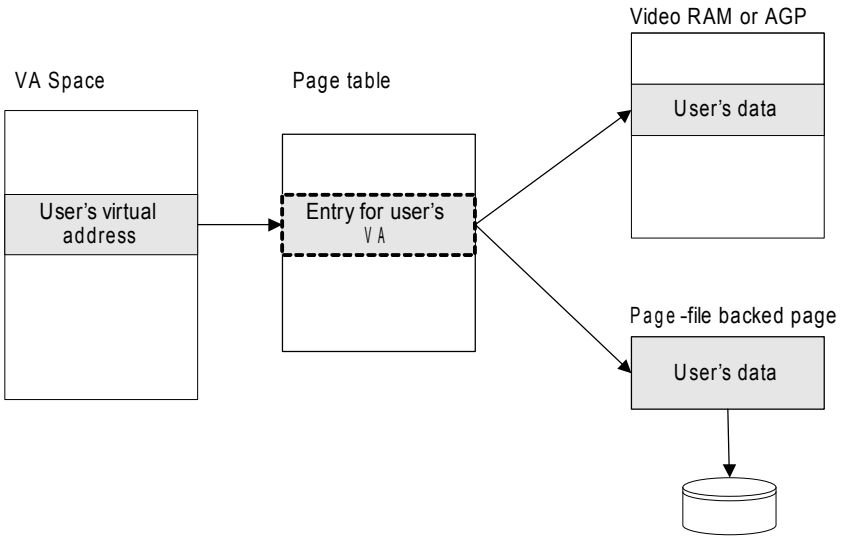

Figure 7. Rotate Virtual Address Descriptors

In Figure 7, the page table entry for a location in the user's VA space can reference either a page that is backed by a page in video RAM or AGP or by a page in a file. To switch views, the video driver simply supplies the new address. This technology enables video drivers to use the GPU for direct transfers and thus can improve performance by 100 times over the previous video model.

NUMA Support

Windows Vista more fully uses the capabilities of NUMA architectures than any earlier Windows release. The basic philosophy behind the NUMA support is to build as much intelligence as possible into the memory manager and operating system so that applications are insulated from the details of the individual machine hardware.

NUMA support in Windows Vista and Windows Server 2008 includes changes in the following areas:

- Resource allocation
- Default node and affinity
- Interrupt affinity
- NUMA-aware system functions for applications
- NUMA-aware system functions for drivers
- Paging

Resource Allocation

At startup, Windows Vista builds a graph of the NUMA node access costs—that is, the distance from each node to the other nodes. The system uses this graph to determine the optimal node from which to obtain resources such as pages during operation. If adequate resources are not available on the optimal node, the system consults the graph to determine the next best choice.

Applications can optionally specify an ideal processor, but otherwise do not require any knowledge about the architecture of a particular machine. Windows ensures that whenever possible the application runs on the ideal processor and that any

memory that the application allocates comes from the ideal processor's node—that is, the ideal node. If the ideal node is unavailable or its resources are exhausted, the system uses the next best choice.

Earlier Windows releases allocated the memory from the current processor's node, and if its resources were exhausted, from any node. Thus, in earlier Windows releases, a process that ran briefly on a non-optimal node could have long-lasting memory allocations on that node, causing slower, less-efficient operation.

With the Windows Vista defaults, the system allocates the memory on the ideal node even if the process is currently running on some other node. If memory is not available on the ideal node, the system chooses the node that is closest to the ideal node instead of whichever node happens to have memory available. Overall, the Windows Vista defaults lead to more intelligent system-wide resource allocation by increasing the likelihood that a process and its resources are on the same node or on the most optimal alternatives.

The same defaults apply to kernel-mode drivers, which can run in the process context of the calling thread, the system thread, or an arbitrary thread. If a driver allocates memory while running in the context of the calling thread—as often occurs with I/O requests—the system uses the ideal node for the thread's process as the default. If the driver is running in the process of the system thread—as is typical for **DriverEntry**, *AddDriver*, *EvtDriverDeviceAdd*, and related start-up functions—the system uses the ideal node of the system process. If the driver is running in an arbitrary thread context, the system uses the ideal node for that process. Drivers can override these defaults by using the **MmXxxx** system functions that are described in "NUMA-Aware System Functions for Drivers," later in this paper. For example, a driver might allocate memory on a particular node if its device interrupts on that node.

The system has a single nonpaged memory pool that includes physical memory from all nodes. The initial nonpaged pool is mapped into a continuous range of VAs. When a component requests nonpaged pool, the memory manager uses the thread's ideal node as an index into the pool, so that the memory is allocated from the ideal node. The paged memory pools were made NUMA aware in Windows Server 2003.

Internally, the system PTEs and system cache are now allocated evenly across nodes. Formerly, such memory was allocated on the boot node and in rare situations could exhaust the free pages on that node. The memory manager's own internal look-aside lists are similarly NUMA aware.

Default Node and Affinity

As mentioned in the previous section, Windows Vista uses the node that contains the ideal processor as the default node for memory allocation. In Windows XP and earlier Windows releases, the default is the node that contains the processor on which the thread is currently running.

Applications can specify NUMA affinity based on any of the following, in order of preference:

- VAD, by using **VirtualAllocExNuma** or **MapViewOfFileExNuma**.
- Section, by using **CreateFileMappingNuma**.
- Thread, by using **SetThreadAffinityMask** or **SetThreadIdealProcessor**.
- Process, by using **SetProcessAffinityMask**.

The system uses the application-specified affinity whenever possible. Windows attempts to satisfy all such requests, as described in the previous section, but does not guarantee that a given request will be completely satisfied from the requested node. If adequate resources are not available on the requested node, the system uses the most optimal node that has adequate resources. This approach satisfies the request quickly rather than waiting indefinitely for memory to become available on the ideal node.

Interrupt Affinity

Drivers for PCI devices that support message-signaled interrupts (MSI or MSI-X) can specify an interrupt affinity—that is, the set of processors on which the device's interrupt service routine (ISR) runs—for each MSI message that the device generates. This feature can significantly improve performance, especially on network interface cards (NICs) that support receive-side scaling (RSS).

A driver can specify an affinity for a particular MSI message when it connects the interrupt. A driver can also set default affinity and affinity policy by setting values for the **Interrupt Management\Affinity Policy** registry key in the *DDInstall*.HW section of its INF. An administrator can also set these values in the registry.

For more information, see the WinHEC presentation "NUMA I/O Optimizations," the white paper "Interrupt Architecture Enhancements in Windows," and "Interrupt Affinity and Priority" in the WDK. For NIC-specific information, see "NDIS MSI-X" in the WDK.

NUMA-Aware System Functions for Applications

Windows Vista supports the following new NUMA-aware system functions for applications:

- **VirtualAllocExNuma** reserves or commits a range of virtual memory and requests memory on a particular node.
- **CreateFileMappingNuma** creates or opens a file-mapping object and requests memory on a particular node.
- **MapViewOfFileExNuma** maps a view of a file-mapping object and requests memory on a particular node.
- **AllocateUserPhysicalPagesNuma** allocates physical memory from a particular node.
- **QueryWorkingSetEx** can be used to obtain the node on which a particular VA is currently allocated.

These functions are NUMA-aware versions of the existing, similarly named functions. For more information on these functions, see the MSDN Web site.

NUMA-Aware System Functions for Drivers

Drivers can use two new system functions to specify an affinity for memory allocation on a particular node:

- **MmAllocateContiguousMemorySpecifyCacheNode**
- **MmAllocatePagesForMdlEx**

MmAllocateContiguousMemorySpecifyCacheNode is similar to the existing **MmAllocateContiguousMemorySpecifyCache** function, except that a driver can request memory from a particular node on a machine that supports the NUMA architecture.

MmAllocatePagesForMdlEx is similar to **MmAllocatePagesForMdl**, but allows a driver to optionally request pages only on the current thread's ideal node, to skip zeroing of the pages upon allocation, and to specify the cache type that is used to map the pages.

Paging

Windows Server 2008 incorporates further NUMA enhancements for paging. Server 2008 prefetches pages to the application's ideal node and migrates pages to the ideal node when a soft page fault occurs. A soft page fault occurs when the system can find the requested page elsewhere in memory, whereas a hard page fault requires reading the page in from disk.

Scalability

As Windows runs on larger and more powerful machines, Microsoft continues to enhance the system's ability to scale up to service more and faster processors and RAM.

Efficiency and Parallelism

As a result of numerous internal improvements to the memory manager, memory allocation now requires fewer I/O operations and fewer locks for optimal throughput.

Internally, the memory manager now uses a bitmap instead of a linked list to track the free pages in the nonpaged pool. Unlike a linked list, a bitmap can be searched without a lock, thus reducing contention for the associated lock by more than 50 percent. Furthermore, bitmaps provide automatic coalescing of contiguous free pages. Windows Server 2008 also uses bitmaps to describe system PTEs.

Large shared sets are now directly mapped instead of hashed. When the number of entries in a hash table is very large, frequent collisions typically occur unless the hash table can be dynamically resized. However, resizing is expensive to perform on large sets. Direct mapping is therefore more efficient than hashing in this situation.

In Windows Server 2008, the allocation of physically contiguous memory is greatly enhanced. Requests to allocate contiguous memory are much more likely to succeed because the memory manager now dynamically replaces pages, typically without trimming the working set or performing I/O operations. In addition, many more types of pages—such as kernel stacks and file system metadata pages, among others—are now candidates for replacement. Consequently, more contiguous memory is generally available at any given time. In addition, the cost to obtain such allocations is greatly reduced.

Page-Frame Number and PFN Database

The page-frame number (PFN) database contains information about all of the physical memory in the machine. In 64-bit editions of Windows Vista SP1 and Windows Server 2008, page-frame numbers are 64 bits long to support large amounts of memory and NUMA architectures, on which the physical address space is sometimes sparsely populated with memory.

In earlier Windows releases, whenever a new page was needed, the memory manager acquired the PFN spinlock and removed a new page from the appropriate list chained through the PFN database. Instead, Windows Vista maintains short lists of immediately available zero pages and free pages for each NUMA node and page color. (Page coloring is a technique that the memory manager uses to reduce the possibility of cache-line collisions among pages.) In many cases—particularly

demand-zero faults and copy-on-write faults—the system can now get a single page without acquiring the PFN lock. Reducing the number of spin lock acquisitions eliminates potential spins on other processors and thus improves parallelism.

Large Pages

Windows Server 2003 introduced large pages for user-mode applications. Windows Vista and Windows Server 2008 use large pages more extensively internally and provide enhanced support for them. Windows Vista and Windows Server 2008 use large pages for the following:

- Initial nonpaged pool
- PFN database
- User application and driver images
- Page file-backed shared memory
- User-mode **VirtualAlloc** allocations
- Driver I/O space mappings

A user-mode application can allocate pages as large as 4 MB on x86-based systems by using the **VirtualAlloc** function with the MEM_LARGE_PAGES flag. Table 1 lists the large page sizes that are supported in Windows hardware platforms.

Table 1. Large Page Sizes

Architecture	Large page size
x86	4 MB
x86 with PAE enabled	2 MB
x64	2 MB
Itanium	16 MB

An application can call **GetLargePageMinimum** to determine the current large page size.

The Windows Vista memory manager allocates ranges of large pages more quickly than earlier Windows releases did. The entire range is no longer required to be contiguous, so that attempts to allocate large pages are more likely to succeed and less likely to cause page thrashing. For example, if an application requests 10 MB of large pages, Windows Vista and later Windows releases can allocate five large pages of 2 MB each (if large pages are 2 MB on the individual hardware platform) instead of trying to find 10 MB of physically contiguous memory.

The Windows Vista memory manager also keeps track of which NUMA nodes the allocated memory belongs to and can zero large pages in parallel by dispatching threads to the appropriate nodes to zero them locally.

Cache-Aligned Pool Allocation

Windows Vista implements support for cache-aligned pool allocation. Drivers can specify the following flags in the **ExAllocatePoolXxx** functions to request cache-aligned memory:

- **NonPagedPoolCacheAligned**
- **PagedPoolCacheAligned**

These flags were defined in earlier Windows releases, but they were ignored.

Virtual Machines

Efficiency and scalability are not only important for good server performance, they are also critical for Windows to run as a guest operating system in a virtualized system. Windows Vista incorporates several changes that improve its performance in virtual machine scenarios.

The translation look-aside buffer (TLB) caches the translation from VAs to physical addresses so that the processor can quickly access this information. If an address is not available in the TLB, the processor typically must make several memory references, which are quite time consuming. Consequently, overall system performance decreases as the TLB hit rate drops.

One way to increase the likelihood that an address will be in the TLB is to flush the TLB less often. Each time a page has been made invalid, its entry must be flushed from the buffer. A page becomes invalid when it is unmapped, freed, trimmed from the working set, or modified by a copy-on-write operation, among others. The entry must also be flushed if changes are made to the protection or cache attributes of the page.

Flushing the entire translation buffer across all processors is a relatively expensive operation that requires significant operating system overhead. Furthermore, after the buffers are flushed, they must be repopulated. Windows Vista rarely flushes the entire buffer. As a result, virtual machines can operate much more efficiently.

If a virtualized system hosts several guest operating systems, the size of the guests can constrain hypervisor performance and limit scalability. To use a smaller memory footprint and thus be a better guest system, Windows Server 2008 frees unneeded memory that it has speculatively allocated. In particular, the system reclaims memory from the initial nonpaged pool if it is not being used.

Load Balancing

Windows Vista exports the following new events to help in load balancing:

- **LowCommitConditionNotification**
- **HighCommitConditionNotification**
- **MaximumCommitConditionNotification**

The **LowCommitConditionNotification** event is set when the operating system's commit charge is low, relative to the current commit limit. In other words, memory usage is low and a lot of space is available for allocations.

The **HighCommitConditionNotification** event is set when the operating system's commit charge is high, relative to the current commit limit. In other words, memory usage is high and very little space is available. If adequate disk space is available, the system obtains more memory by automatically increasing the page file size up to the limit imposed by the administrator. A short-term option is to reduce the current system load. Long-term solutions are to increase the minimum page file size or add RAM.

The **MaximumCommitConditionNotification** event is set when the operating system's commit charge is near the maximum commit limit. In other words, memory usage is very high, very little space is available, and the system cannot increase the size of its paging files because of the current limits imposed by the administrator. A system administrator can always increase the size or number of paging files, without restarting the computer, if adequate disk space is available. Other alternatives are to increase the minimum or maximum page file sizes, add RAM, or reduce the load.

These events supplement the pool notification events that were added in Windows Server 2003. Drivers and other kernel-mode components can register for these events. For more information on memory-related notification events, see "Standard Event Objects" in the WDK.

Additional Optimizations

Additional memory manager optimizations involve the following areas:

- **VirtualAlloc** and address windowing extensions (AWE) allocations.
- **VirtualProtect** function.
- Windows on Windows (WOW) on 64-bit systems.

Windows acquires a per-process address space lock to synchronize changes to the user address space. In Windows Vista, this lock supports both shared and exclusive access, whereas in earlier Windows versions, the lock supported exclusive access only. Consequently, many operations such as **VirtualAlloc** and **VirtualQuery** can now run in parallel. Overall, changes within **VirtualAlloc** reduce the time required for AWE allocations by over 2500 percent in some scenarios.

The **VirtualProtect** function changes the access protection on a region of pages in virtual memory. When a page's access protection attribute changes, processors must flush the corresponding TLB entry. Windows Server 2008 issues a single flush request to all processors whose TLB might contain the entry instead of multiple single requests to each individual processor. As a result, **VirtualProtect** can change access protection on large regions 60 times faster than in earlier Windows versions.

On 64-bit architectures, Windows Vista uses demand-zeroed memory instead of pool memory to allocate the page-table bitmaps for 32-bit binary emulation. This change enables 32-bit binaries to run much more efficiently because they require a smaller system memory footprint and perform fewer I/O operations.

System Integrity

Through online crash analysis (OCA), users can upload data about system crashes to Microsoft. This data has provided useful information about the causes of common system crashes and has led to several system enhancements to detect and handle potential system corruption. Windows Vista and Windows Server 2008 incorporate advances to improve system integrity in the following areas:

- Diagnosis of hardware errors
- Code integrity and driver signing
- Data preservation during bug checks

Diagnosis of Hardware Errors

As mentioned in "Page-File Writes" earlier in this paper, Windows maintains a list of zero pages. Hardware errors such as DMA transfer errors and single bit errors can corrupt memory after the pages have initially been zeroed, so Windows Vista checks the list to ensure that these pages actually are zero. If the system finds an error, it records the physical address at which the error occurred and the nature of the error in the event log. This information helps to pinpoint single-bit errors that are caused by hardware failures.

Machines that frequently encounter such errors are often prone to application hangs and crashes that are extremely difficult to track down. OCA data indicates that such failures are much more common than previously suspected. Independent hardware

vendors (IHVs) can help diagnose and prevent such errors by using error-correcting code (ECC) memory.

Code Integrity and Driver Signing

The memory manager implements a simple, high-speed technique to validate images for code integrity. This feature enforces the mandatory code signing for kernel-mode drivers on x64-based systems.

For more information on code signing, see "Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista," "Kernel-Mode Code Signing Walkthrough," and "Summary of Windows Driver Signing Requirements."

Windows Vista supports hot-patching for both system-wide and session drivers, so that patches can be installed without rebooting the user's system. Thus, users can take advantage of security patches as soon as they become available without waiting for reboot.

Data Preservation during Bug Checks

Windows Vista preserves more data than earlier Windows versions when certain nondestructive bug checks occur. For example, if a bug check occurs when the system is paging in part of a kernel-mode component, the system cannot proceed because the component is missing required information, but the data that is already in the system cache is not affected. To prevent data loss, the memory manager writes out all the modified data from the system cache to its backing store (typically a disk file) and then issues a bug check. Only failures to page in kernel-mode code or data are fatal; failures to page in user process code or data merely cause an exception in the application.

To further protect system data, Windows Vista supports the ability to mark views of the system cache as read only. The registry uses this feature to protect its views from inadvertent driver corruption. Thus, registry data is read only except when it is actively being modified.

Driver writers can use the new **.pagein** debugger command to view the contents of kernel-mode memory addresses that have been paged out to disk. For more information about this command, see Debugging Tools for Windows.

What You Should Do

Most of the memory management enhancements that are described in this paper are internal and are transparent to administrators, software developers, and hardware manufacturers. However, a few of the changes require awareness or action to gain maximum benefit and to contribute to an improved user experience.

Here are the most important effects for hardware manufacturers, driver developers, application developers, and system administrators.

For Hardware Manufacturers

- Use ECC.

For Driver Developers

- Never attempt to access memory beyond what the driver has allocated. Use Driver Verifier to catch this error.
- Handle dummy pages correctly in drivers that directly access the contents of MDLs.

- Use **KeExpandKernelStackAndCallout** as necessary to gain additional kernel stack space.
- Use the new NUMA-aware system functions in drivers that are sensitive to NUMA architectures and specify interrupt affinity if this is important for your device.
- Use new events for notification about system load if your driver allocates memory that it can release during operation.
- Be aware of driver signing requirements for Windows Vista, particularly for x64 architectures.
- Use the .**pagein** debugger command to inspect kernel-mode data that was paged out.

For Application Developers

- Relink with the **/DYNAMICBASE** and **/NXCOMPAT** options to enable ASLR with no-execute protection for Windows Vista.
- Be aware that the default NUMA node for memory allocation is now the ideal node instead of the current node.
- Use the new NUMA-aware system functions to control memory allocation and query page locations on NUMA architectures.

For System Administrators

- Understand the dynamic kernel VA allocation so that you can modify system tuning—or avoid tuning altogether.
- Use a debugger with the **!vm** 21 command to inspect details of kernel VA space use on 32-bit systems.
- Check the system event log for zero-page corruption errors. Upload crash data to OCA whenever possible.

Resources

MSDN:

Windows Vista ISV Security
http://msdn2.microsoft.com/en-us/library/bb430720.aspx

Memory Management Registry Keys
http://msdn2.microsoft.com/en-us/library/bb870880.aspx

Windows Driver Kit:
http://msdn2.microsoft.com/en-us/library/aa972908.aspx

Kernel-Mode Driver Architecture Design Guide
Memory Management
Interrupt Affinity and Priority
Standard Event Objects

Kernel-Mode Driver Architecture Reference
Standard Driver Routines
Driver Support Routines

Driver Development Tools
Boot Options for Driver Testing and Debugging

Network Design Guide
NDIS MSI-X

Windows Hardware and Driver Central:

Driver Signing Requirements for Windows [home page]
http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx

Digital Signatures for Kernel Modules on Systems Running Windows Vista
Summary of Windows Kernel-Mode Driver Signing Requirements

Windows PC Accelerators: Performance Technology for Windows Vista
http://www.microsoft.com/whdc/system/sysperf/accelerator.mspx

What Is Really in That MDL?
http://www.microsoft.com/whdc/driver/tips/mdl.mspx

Interrupt Architecture Enhancements in Windows
http://www.microsoft.com/whdc/system/bus/PCI/MSI.mspx

NUMA I/O Optimizations
http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/SVR-T332_WH07.pptx

Microsoft TechNet:

Inside the Windows Vista Kernel: Part 3 (April 2007)
http://www.microsoft.com/technet/technetmag/issues/2007/04/VistaKernel/

Book:

Windows Internals, Fourth Edition,
Russinovich, Mark, and David A. Solomon. Redmond, WA: Microsoft Press, 2005
http://www.microsoft.com/MSPress/books/6710.aspx